Decision Making under Uncertainty: Scalability and Applications

Peng Dai

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

University of Washington

2011

Program Authorized to Offer Degree: Computer Science and Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Peng Dai

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Co-Chairs of the Supervisory Committee:

_____
Daniel S. Weld

_____
Mausam

Reading Committee:

_____
Daniel S. Weld

_____
Mausam

_____
James A. Fogarty

Date: _____

Signature_____

Date_____

University of Washington

Abstract

Decision Making under Uncertainty: Scalability and Applications

Peng Dai

Co-Chairs of the Supervisory Committee:
Professor Daniel S. Weld
Computer Science and Engineering

Research Assistant Professor Mausam
Computer Science and Engineering

Almost every decision problem in the world involves uncertainty, thus falling in the category of decision making under uncertainty. Markov decision processes (MDPs) are a powerful and widely-adopted formulation for modeling decision making under uncertainty problems. Exact solutions to MDPs are commonly found using dynamic programming techniques. While the time complexity of dynamic programming algorithms is polynomial in the number of states, the algorithms can become very slow if the state space is large. Additionally, the entire MDP model needs to be loaded in memory before dynamic programming can be applied. This prohibitive use of memory is the major bottleneck in scaling MDP algorithms to real-world problems. To speed up optimal dynamic programming, we develop the solver, *focused topological value iteration* (FTVI), that combines using the graphical information of a problem with a heuristic-guided search. In most cases, FTVI's convergence speed outperforms state-of-the-art MDP solvers by an order of magnitude while keeping the same level of accuracy. We characterize the type of domains where FTVI excels. To overcome the memory bottleneck of dynamic programming, we developed the solver, *partitioned external-memory value iteration* (PEMVI), that utilizes external memory. PEMVI divides the state space into partition blocks and performs (one or more) backups on all states in a piecemeal fashion. Our

automatic, domain-independent partitioning algorithm for PEMVI uses static problem analysis to identify candidates for partitions, and chooses a suitable partition using heuristic search. Surprisingly, the automatic partitioning engine, without using any domain-specific information, constructs more effective partitions than manually constructed partitions, which further improves scalability.

While decision making under uncertainty is well studied, it can be applied to current problems. We find it especially interesting to apply it in quality control in crowdsourcing. Crowdsourcing refers to outsourcing tasks to a crowd of unknown people (workers) as an open call. It has become immensely popular with hoards of employers (requesters), who use it to solve a wide variety of jobs, such as dictation transcription and content screening. While the labor are easy to find and usually cheap, it is costly to track the quality of individual work. To do so, requesters often subdivide a large task into a chain of small-sized subtasks. Those subtasks are then often combined into a complex, iterative workflow, in which workers check and improve each others' results. We model the crowdsourced workflow control problem as a partially-observable MDP (POMDP), and propose a method to perform quality control automatically. Specifically, we design and implement an agent, TurKontrol, which learns a proposed mathematical model on workers' competency and uses it to dynamically control the workflow. Our model and agent are demonstrated to be useful in practice — the dynamic workflow computed by TurKontrol generates statistically-significant, better results than a non-adaptive workflow, while incurring the same amount of cost.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

This dissertation is not only a comprehensive report of my four-year PhD work at the University of Washington, but also the result of help from numerous people.

I am luckier than most PhD students to have two superb supervisors, Dr. Daniel Weld and Dr. Mausam. They are both great researchers and advisors. Dan is knowledgeable in almost all aspects of Computer Science and therefore a terrific source of enlightenment. He is prescient, inspiring and patient. He is keen on technology as he showed great enthusiasm once I ran into technical difficulties in perl programming. He offers me continuous guidance in research, professional activities, teaching and advising. He is a terrific model for researchers as he is sensitive to any potential problem of the work. I never remember he overlooked a thing where he is unsure about. Mausam is super encouraging and supportive. He understands all the technical details of my work, much of which contains his contributions. I always feel energetic and full of ideas after brainstorming with him. He is modest, sensitive and cares about students. I still remember he generously took over part of my mission during a conference deadline once I was overwhelmed by a mid-term exam. He is very approachable and consistently ready to help.

I am equally fortunate to be surrounded by many devoted and intelligent colleagues. I could not remember how much I benefit from discussions with them, Andrey Kolobov, Raphael Hoffman, Michael Toomim, Dr. James Fogarty, Lydia Chilton, Todd Shiller, and many people from the AI group. My work in crowdsourcing is much enlightened by Greg Little, who invented TurKit, a user-friendly toolkit to post and collect information on MTurk, which my crowdsourcing work uses intensively. He offers countless, timely technical support for TurKit. I thank my undergraduate advisee, Mimi Fung, who helped me a lot in building the server for my crowdsourcing agent, TurKontrol, and setting up its connection with TurKit.

I am thankful for several researchers, who influenced me deeply throughout my growth. Dr. Judy Goldsmith, my Masters' thesis advisor at the University of Kentucky, brought me to the area of Artificial Intelligence and inspired my interests when I was a young researcher. Much of my automated planning work was built on top of the planner released by Dr. Eric Hansen. His strict and diligent altitude in research and implementation have gained my respect for him. I also thank Dr. Shlomo Zilberstein for his encouragement during our interactions. I can still remember his positive altitude for life after many years.

I wish to thank my advisor at NASA Ames Research Center, Javier Barreiro, for his help and guidance. I benefited a lot from discussion with researchers there, Dr. Nicholas Meuleau, Dr. David Smith and Dr. Tristan Smith.

Many thanks to my friends, Fei Wu, Hoifung Poon, Xiaoyu Chen, Hao Lu, Shulin Yang, Xu Miao, Xiao Ling, Hao Du, Congle Zhang, Sai Zhang, Xiao Wang, Zhe Xu, Qi Shan, Yanping Huang, Shengliang Xu, Jue Wang, Yingyi Bu, and Ke Zheng, who share their joy and bear my depression, which contributed to my four colorful years at UW.

Last but not least, I want to thank my wife, Jieling Han, my parents, Xia Yin and Hesong Dai, for their love, support, and patience.

DEDICATION

To Jieling

and

in loving memory of my grandfather

Chapter 1

INTRODUCTION

Decision making under uncertainty, a key area of Artificial Intelligence, is widely used to model decision-making problems in the real world. Planning and reasoning serve as the core module for many intelligent agents and real-world applications. Depending on whether the decision agent has full sensing power of the environment, a planning problem is typically modeled as a fully-observable Markov decision process (MDP), or a partially-observable Markov decision process (POMDP). For instance, NASA researchers use MDPs to model the next-generation Mars rover planning problems [23, 53, 90, 95]. MDPs are also used to formulate military operations planning [3] and coordinated multi-agent planning [101]. The problems of automatically controlling a helicopter [1] or a car [2] can also be modeled as MDPs. POMDPs [72] bring more flexibility when the agent has limited knowledge of the environment. Recent POMDP applications include robot navigation [106, 116] and dialog management systems [83, 132]. [26] provides a comprehensive a list of POMDP applications.

While a POMDP is more general and thus more powerful in representation abilities than an MDP, it is typically solved by transforming into an equivalent, yet sometimes infinitely larger MDP. Therefore, many MDP solvers can be applied to POMDP problems. Despite the general representation power of (PO)MDPs, the poor scalability of their solvers prevents their application in a broader range of problems. The scalability issues come from two orthogonal dimensions: time and space.

The classical algorithms of solving MDPs, such as value iteration [13] or policy iteration [65], need to load the model into memory, initialize a random yet sub-optimal solution, and iteratively update the solution until an optimal value function is reached. While algorithms using dynamic programming consume time that is polynomial in the number of states [85, 18], they are typically slow when the state space is large. Value iteration, for example, conducts blind updates of the value of every state in every iteration. This can be

very inefficient when the solution of only a small subset of the state space is sub-optimal.

Another obstacle to scaling optimal probabilistic planning is that the common MDP solvers, *e.g.*, value iteration [13], LAO* [58], and RTDP [8], are based on dynamic programming and thus require the amount of memory that is polynomial in the number of states, $|\mathcal{S}|$, *i.e.*, exponential in the number of problem features. This prohibitive use of memory is a major bottleneck in scaling MDP algorithms to real-world problems. Even relatively compact symbolic representations, like algebraic decision diagrams, tend to grow large in size, exhausting memory on moderate sized MDPs [62].

Our work is motivated by the scalability issue of MDP solvers. Approaching the time and space aspects individually, we propose two state-of-the-art optimal MDP solvers, *focused topological value iteration* that computes an equally optimal solution using much less time than previous solvers, and *partitioned external-memory value iteration* that solves much larger problems than previously attempted. To popularize decision making under uncertainty, another piece of work is applying its techniques to perform quality control for crowdsourcing. We propose TurKontrol, a tool for dynamically controlling a crowdsourced workflow that achieves statistically-significant better results than a non-adaptive workflow.

1.1 Faster Optimal Planning

The first contribution of our work is in generating an equally optimal solution with much less time by combining using the graphical information of a problem with a heuristic-guided search. Our new algorithm, *focused topological value iteration* (FTVI) [38], first performs a phase of heuristic search and eliminates provably sub-optimal actions found during the search. Then it builds an informative graphical structure based on the remaining actions, namely the topological order of the strongly connected components, and solves components sequentially by the topological order. We find that a very short phase of heuristic search is often able to eliminate many actions leading to an MDP structure that is amenable to efficient, topology-based solutions. We evaluate FTVI across several benchmark domains and find that FTVI outperforms the topological value iteration (TVI) algorithm [32], its precursor, by significant margins. Surprisingly, we also find that FTVI outperforms other

state-of-the-art heuristic search algorithms in most of the domains. This is unexpected, since common wisdom dictates that heuristic-guided search is much faster than all-state dynamic programming. To better understand this big improvement, we study the convergence speed of algorithms on a few problem features. We discover two important features of problems that are hard for heuristic search algorithms: smaller number of goal states and long search distance to the goal. These features are commonly found in many domains. We show that, in such domains, FTVI outperforms heuristic search in convergence speed by an order of magnitude on average, and sometimes by even two orders of magnitude.

1.2 Optimal Planning on Large Problems

The second contribution of our work is on enlarging the set of optimally solvable problems by using external memory. We propose an algorithm named *partitioned external-memory value iteration* (PEMVI) [36]. PEMVI partitions the state space into disjoint blocks, most of which are stored on disk at any time. In order to backup the states in block p, the values corresponding to neighboring blocks are also loaded; several backups are done on the states in p before moving to the next. By choosing a good partition and selecting a good order in which to back them up, PEMVI achieves an efficient flow of information between states with reduced disk activity. Our experiments show that PEMVI solves problems much larger than what internal memory techniques like value iteration and RTDP can handle. Moreover, it outperforms the external-memory value iteration (EMVI) algorithm [48], a contemporary external-memory MDP algorithm, by an order of magnitude. For example, we solve one problem in two months which would require several years with EMVI.

Another feature of PEMVI is its complete automation. To provide PEMVI with the essential input, the partitioning of the state space, we develop and implement an automatic partitioning engine for PEMVI. We first build a theory of *XOR groups* that extends previous work [47] and constructs necessary and sufficient conditions for a set of fluents to be a valid XOR group. We use static domain analysis to identify XOR groups present in the domain and search over a partition space created by these XOR groups yielding a suitable partition. We evaluate three criteria (*viz.* coherence, locality and balance) for this search, and conclude that a combination of coherence and balance yields the best results. We observe that our

automatic partitioning engine constructs better partitions than manual partitioning; as a result, PEMVI is able to solve problems larger than ones that were previously reported.

1.3 Case Study: Decision-Theoretic Quality Control for Crowdsourcing

To popularize decision making under uncertainty, we seek applications from real-world problems and conduct a case study of quality control for *crowdsourcing.* Crowdsourcing [131] is a recent framework in which tasks are outsourced to a crowd of unknown people ("workers") as an open call. It has become immensely popular with hoards of employers ("requesters"), who use it to solve a wide variety of jobs, such as natural language translation, dictation transcription, content screening, review writing, *etc.* Its usefulness has been demonstrated by several successful, international markets, of which *Amazon's Mechanical Turk* (AMT) is perhaps the most famous. Workers exchange their contributions to the tasks posted by requesters for monetary rewards, in a contract-free manner. Due to high variances in the workers' capabilities, motivations, and the current fragile credit system, requesters often subdivide a large task into a chain of small-sized subtasks to achieve quality results. Those subtasks are then often combined into a complex, iterative workflow, in which workers check and improve each others' results. Managing a dynamic crowd-sourced workflow is an important but challenging problem. A successful workflow enables requesters to maximize their utilities given the money they spend, while additionally avoids a large amount of administrative and logistic work. However, there are often too many factors influencing the whole process, making the problem of controlling the workflow too complicated to tackle. The problem naturally fits in a POMDP paradigm when we regard the quality of an answer as a partially-observable variable, and represent workers' competencies in generating quality answers as a probabilistic transition model.

We design and implement an agent, TurKontrol [40], which implements the proposed mathematical framework, solves it using a limited-step lookahead algorithm, and uses the solution to optimize and control the workflow. Figure 1.1 shows the sequence of decisions made by TurKontrol, for the task of iteratively writing and improving a description of a photograph. The agent automatically decides, given observations (the history of previous tasks, participating workers and answers), which description (artifact) is the best,

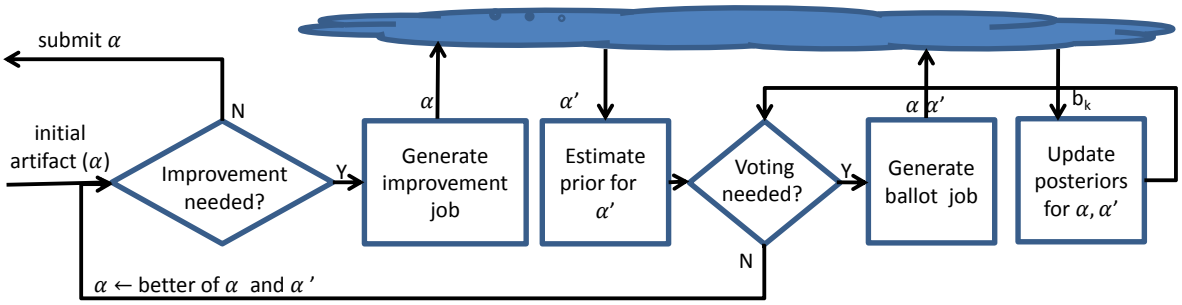


Figure 1.1: Computations needed by TurKontrol for control of an iterative-improvement workflow.

when to improve a description or vote on candidate descriptions, and when to stop the process. Extensive simulations show that TurKontrol is robust in a variety of scenarios and parameter settings, and results in higher utilities than previous, fixed policies.

To more rigorously validate this work, we deploy our system on Mechanical Turk. As a bootstrapping step, we learn the model parameters from limited, noisy data. We transform the learning problem into several optimization problems and apply statistical learning techniques to find the most accurate model parameters. Then we plug in the model to TurKontrol and demonstrate that the model is useful in practice — the dynamic workflow computed by TurKontrol generates statistically-significant, higher-quality results than a non-adaptive workflow, when the two systems consume the same amount of money. The model can be refined when new data is available, as TurKontrol is able to perform automatic, off-line model updates.

1.4 Outline

The outline of the rest of the dissertation is as follows: Chapter 2 formally defines MDPs and POMDPs, and reviews algorithms that solve them. Chapter 3 describes FTVI, the new, optimal algorithms of solving MDPs by using graphical information and heuristic search. Chapter 4 introduces PEMVI, a new external-memory MDP algorithm and an automatic system of partitioning the MDP model for PEMVI. Chapter 5 is a case study on decision making under uncertainty application for crowdsourcing. We leverage the framework to

automatically control a crowdsourced workflow based on tradeoffs between quality and cost. We present related work in Chapter 6, propose future work in Chapter 7 and conclude in Chapter 8.

Chapter 2

BACKGROUND

2.1 Markov Decision Processes

AI researchers typically use Markov decision processes (MDPs) to formulate fully-observable decision making under uncertainty problems.

Definition 1 *An MDP is a five-tuple* $\langle \mathcal{S}, \mathcal{A}, Ap, T, C \rangle$*, where*

- $\mathcal{S}$ *is a finite set of discrete states.*
- $\mathcal{A}$ *is a finite set of all applicable actions.*
- $Ap : \mathcal{S} \rightarrow \mathcal{P}(\mathcal{A})$ *is the applicability function.* $Ap(s)$ *denotes the set of actions that can be applied in state* s*.* $\mathcal{P}(\mathcal{A})$ *is the power set of* $\mathcal{A}$*.*
- $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ *is the transition function describing the effect of an action execution.*
- $C : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}^+$ *is the cost of executing an action in a state.*

The agent executes its actions in discrete time steps. At each step, the system is at one distinct state $\mathbf{s} \in \mathcal{S}$. The agent can execute any action $\mathbf{a}$ from a set of *applicable actions* $Ap(\mathbf{s}) \subseteq \mathcal{A}$, incurring a cost of $C(\mathbf{s}, \mathbf{a})$. The action takes the system to a new state $\mathbf{s}'$ stochastically. The transition process has the *Markov property*, *i.e.*, the new state $\mathbf{s}'$ is independent of all previous states given the current state $\mathbf{s}$. The transition probability is defined by $T_{\mathbf{a}}(\mathbf{s}'|\mathbf{s})$. The model assumes *full observability*, *i.e.*, after executing an action and transitioning stochastically to a next state as governed by T, the agent has full knowledge of the state.

We define two relationships *successor* and the *predecessor* both on $(\mathcal{S}, \mathcal{S})$ as follows:

Definition 2 *If a state* $\mathbf{s}'$ *is the transitioning result of applying some action* $\mathbf{a}$ *at state* $\mathbf{s}$*, or* $T_{\mathbf{a}}(\mathbf{s}'|\mathbf{s}) > 0$*, then* **s'** *is a* successor *of* $\mathbf{s}$*. Contrarily,* $\mathbf{s}$ *is a* predecessor *of* **s'**.

The *horizon* of an MDP is the number of steps for which costs are accumulated. We concentrate on a special set of MDPs called *stochastic shortest path* (SSP) problems. Despite its simplicity, SSP is a general MDP representation. Any infinite-horizon, discounted-reward MDP can be easily converted to an undiscounted SSP [14]. The horizon in such an MDP is *indefinite*, *i.e.*, finite but unbounded, and the costs are accumulated with *no* discounting. There are two more components of a SSP:

- $\mathbf{s_0}$ is the initial state.
- $\mathcal{G} \subseteq \mathcal{S}$ is the set of sink goal states. Reaching any one of $\mathbf{g} \in \mathcal{G}$ terminates execution.

The cost of an execution is the sum of all costs along the path from $\mathbf{s_0}$ to the first goal state encountered.

Any solution to an MDP problem is in the form of a *policy*.

Definition 3 *A policy,* $\pi : \mathcal{S} \rightarrow \mathcal{A}$*, of an MDP is a mapping from the state space to the action space.*

A policy is *static* if the action taken at each state is the same for every step. $\pi(\mathbf{s})$ indicates which action to execute when the system is at state $\mathbf{s}$. To solve an MDP we need to find an *optimal policy* ($\pi^* : \mathcal{S} \rightarrow \mathcal{A}$), a probabilistic execution plan that reaches a goal state with the minimum expected cost. We evaluate any policy π by its *value function*, the set of values that satisfy the following equation:

$$V^{\pi}(\mathbf{s}) = C(\mathbf{s}, \pi(\mathbf{s})) + \sum_{\mathbf{s}' \in \mathcal{S}} T_{\pi(\mathbf{s})}(\mathbf{s}'|\mathbf{s}) V^{\pi}(\mathbf{s}'). \tag{2.1}$$

Any optimal policy must satisfy the following system of *Bellman equations*:

$$\begin{aligned} V^*(\mathbf{s}) &= 0 \quad \text{if } \mathbf{s} \in \mathcal{G}, \text{ else} \\ V^*(\mathbf{s}) &= \min_{\mathbf{a} \in Ap(\mathbf{s})} \left[C(\mathbf{s}, \mathbf{a}) + \sum_{\mathbf{s}' \in \mathcal{S}} T_{\mathbf{a}}(\mathbf{s}'|\mathbf{s}) V^*(\mathbf{s}') \right]. \end{aligned} \tag{2.2}$$

The corresponding optimal policy can be extracted from the value function:

$$\pi^*(\mathbf{s}) = argmin_{\mathbf{a}\in Ap(\mathbf{s})}\left[C(\mathbf{s},\mathbf{a}) + \sum_{\mathbf{s}'\in\mathcal{S}} T_{\mathbf{a}}(\mathbf{s}'|\mathbf{s})V^*(\mathbf{s}')\right], \forall \mathbf{s}\in\mathcal{S}-\mathcal{G}. \tag{2.3}$$

Given an implicit optimal policy π^* in the form of its optimal value function $V^*(\cdot)$, we measure the superiority of an action by a *Q-function*: $\mathcal{S}\times\ \mathcal{A}\rightarrow\mathbb{R}$.

Definition 4 *The* Q-value *of a state-action pair* $(\mathbf{s},\mathbf{a})$ *is the value of state* $\mathbf{s}$*, if an immediate action* $\mathbf{a}$ *is performed, followed by* π^* *afterwards. More concretely,*

$$Q^*(\mathbf{s},\mathbf{a}) = C(\mathbf{s},\mathbf{a}) + \sum_{\mathbf{s}'\in\mathcal{S}} T_{\mathbf{a}}(\mathbf{s}'|\mathbf{s})V^*(\mathbf{s}'). \tag{2.4}$$

Therefore, the optimal value function can be expressed by:

$$V^*(\mathbf{s}) = min_{\mathbf{a}\in Ap(s)}Q^*(\mathbf{s},\mathbf{a}). \tag{2.5}$$

2.1.1 Dynamic Programming

Most optimal MDP algorithms are based on dynamic programming, whose utility was first proved by a simple yet powerful algorithm named *value iteration* [9]. Value iteration first initializes the value function arbitrarily, for example all zero. Then, the values are updated iteratively using an operator called the *Bellman backup* (Line 7 of Algorithm 1) to create successively better approximations for each state per iteration. We define the *Bellman residual* of a state to be the absolute difference of a state value before and after a Bellman backup. Value iteration stops when the value function converges. In implementation, it is typically signaled by when the *Bellman error*, the largest Bellman residual of all states, becomes less than a pre-defined threshold, δ. We call a Bellman backup a *contraction operation* [12], if for every state, its Bellman residual never increase with the iteration number.

Value iteration converges to the optimal value function in time polynomial in $|\mathcal{S}|$ [85, 18], yet in practice it is usually inefficient, since it blindly performs backups over the state space iteratively, often introducing many unnecessary backups.

Algorithm 1 Value Iteration

1: **Input:** an MDP $M = \langle \mathcal{S}, \mathcal{A}, Ap, T, C \rangle$, δ: the threshold value
2: initialize V arbitrarily
3: **while** true **do**
4: $\quad Bellman\_error \leftarrow 0$
5: $\quad$ **for** each state $\mathbf{s} \in S$ **do**
6: $\quad\quad oldV \leftarrow V(\mathbf{s})$
7: $\quad\quad V(s) \leftarrow min_{a \in Ap(\mathbf{s})} \left[C(\mathbf{s}, \mathbf{a}) + \sum_{\mathbf{s}' \in \mathcal{S}} T_{\mathbf{a}}(\mathbf{s}'|\mathbf{s}) V(\mathbf{s}') \right]$
8: $\quad\quad Bellman\_residual(\mathbf{s}) \leftarrow |V(\mathbf{s}) - oldV|$
9: $\quad\quad Bellman\_error \leftarrow max(Bellman\_error, Bellman\_residual(\mathbf{s}))$
10: $\quad$ **if** $Bellman\_error < \delta$ **then**
11: $\quad\quad$ **return** V

Heuristic Search

To improve the efficiency of dynamic programming, researchers have explored various ideas from traditional heuristic-guided search, and have consistently demonstrated their usefulness for MDPs [8, 58, 17, 20, 93, 118, 111]. The basic idea of heuristic search is to consider an action only when necessary, which leads to a more conservative backup strategy. This strategy helps to save a lot of unnecessary backups.

Definition 5 *A heuristic function $h : \mathcal{S} \rightarrow \mathbb{R}^+$ is a mapping from the state space to non-negative real values, where $h(\mathbf{s})$ is an estimate of $V^*(\mathbf{s})$.*

Definition 6 *A heuristic function h is* admissible *if it never over-estimates the value of a state,*

$$h(s) \leq V^*(\mathbf{s}), \forall \mathbf{s} \in \mathcal{S}. \tag{2.6}$$

We interchangeably write an admissible heuristic function as V_l, to emphasize that $V_l(s)$ is a lower bound of $V^*(s)$.

Definition 7 *A* greedy policy *π is the best policy by one-step lookahead given the current*

value function, V:

$$\pi(s) = argmin_{\mathbf{a} \in Ap(\mathbf{s})} \left[C(\mathbf{s}, \mathbf{a}) + \sum_{\mathbf{s}' \in \mathcal{S}} T_{\mathbf{a}}(\mathbf{s}'|\mathbf{s}) V(\mathbf{s}') \right], \forall \mathbf{s} \in \mathcal{S} - \mathcal{G}. \tag{2.7}$$

Definition 8 *A* policy graph*, $G_\pi = (\mathcal{V}, \mathcal{E})$, for an MDP with the set of states $\mathcal{S}$ and policy π is a directed, connected graph with vertices $\mathcal{V} \subseteq \mathcal{S}$, where $\mathbf{s_0} \in \mathcal{V}$, and for any $\mathbf{s} \in \mathcal{S}$, $\mathbf{s} \in \mathcal{V}$ iff $\mathbf{s}$ is reachable from $\mathbf{s_0}$ under policy π. Furthermore, $\forall \mathbf{s}, \mathbf{s}' \in \mathcal{V}$, $\langle \mathbf{s}, \mathbf{s}' \rangle \in \mathcal{E}$ (the set of edges of G_π) iff $T_{\pi(\mathbf{s})}(\mathbf{s}'|\mathbf{s}) > 0$.*

Heuristic search algorithms have two main features: (1) The search is limited to states that are reachable from the initial state. Given the heuristic value, a heuristic search algorithm generates a running greedy policy, as well as its policy graph. The algorithm performs a series of heuristic searches, until all states on the greedy policy graph converge. A search typically starts from the initial state, with successor states explored in a best-first manner. Visited states have their values backed up during the search. (2) Since heuristic search algorithms do fewer backups than value iteration, they require special care to guarantee final optimality. So values of the state space have to be initialized by an admissible heuristic function. Note that value iteration can also take advantage of initial heuristic values as an informative starting point, but does not require the heuristics to be admissible to guarantee optimality.

Different heuristic search algorithms use different search strategies and therefore perform Bellman backups in different orders.

The AO* algorithm [102] solves acyclic MDPs, so it is not applicable to general MDPs. LAO* [58] is an extension to the AO* algorithm that can handle MDPs with loops. Improved LAO* (ILAO*) [58] is an efficient variant of LAO*. It iteratively performs complete searches that discover a running greedy policy graph. In detail, the greedy policy graph only contains the initial state $\mathbf{s_0}$ when a search starts. New states are added to the graph by means of *expansions* over a *frontier state* in the depth-first manner, until no more states can be added. In a state expansion, one of its greedy actions is chosen, and all its successor states are added into the graph. States that are not expanded yet contain successors are called frontier states. Later, states of the greedy policy graph are backed up *only once* in the

post-order when they are visited. Each search iteration performs at most $|\mathcal{S}|$ backups, but in practice this number is typically much smaller. ILAO* terminates when all states of the current greedy policy graph have a Bellman residual less than a given δ.

Real-time dynamic programming (RTDP) [8] is another popular algorithm for MDPs. It interleaves dynamic programming with search through plan execution trials. An execution trial is a path that originates from $\mathbf{s_0}$ and ends at any goal state or by a bounded-step cutoff. Each execution step simulates the result of one-step plan execution. The agent greedily picks an action $\mathbf{a}$ of the current state $\mathbf{s}$, and mimics the state transition to a new current state $\mathbf{s}'$, chosen stochastically based on the transition probabilities of the action, *i.e.*, $\mathbf{s}' \sim T_{\mathbf{a}}(\mathbf{s}'|\mathbf{s})$. Dynamic programming happens when states are backed up immediately when they are visited. RTDP is good at finding a good sub-optimal policy relatively quickly. However, in order for RTDP to converge, states on the optimal policy have to be backed up sufficiently, so its convergence is usually slow. To overcome the slow convergence problem of RTDP, researchers later proposed several heuristic search variants of the algorithm.

Bonet and Geffner [17] introduced a smart *labeling* technique in a RTDP extension named labeled RTDP (LRTDP). They label a state $\mathbf{s}$ *solved* if every state reachable from $\mathbf{s}$ by applying the greedy policy is either a goal state, or is solved, or has a Bellman residual no greater than the threshold δ. States that are labeled as solved no longer get backed up in any future search. Labeling helps speed up convergence as it avoids many unnecessary backups over states that have already converged. After an execution trial, LRTDP tries to label every unsolved state in the reverse order of the visit. To label a state $\mathbf{s}$, LRTDP initiates a DFS from $\mathbf{s_0}$ and checks if all states reachable under the greedy policy rooted at $\mathbf{s}$ are solved, and back up them otherwise. LRTDP terminates when all states of the current policy graph are solved. Bonet and Geffner also applied the labeling technique in another algorithm called HDP [16]. HDP uses Tarjan's algorithm to find all the strongly connected component of an MDP to help label solved states and implicitly control the order in which states are backed up in a search trial.

McMahan et al. [93] proposed another extension named bounded RTDP (BRTDP), not only uses a lower bound heuristic of the value function V_l, but also an upper bound V_u. BRTDP has two key differences from the original RTDP algorithm. First, once BRTDP

backs up a state $\mathbf{s}$, it updates both the lower bound and the upper bound. Second, when choosing the next state $\mathbf{s}'$, the difference of its two bounds, $V_u(\mathbf{s}') - V_l(\mathbf{s}')$, is also taken into consideration. More concretely, $\mathbf{s}' \sim T_{\mathbf{a}}(\mathbf{s}'|\mathbf{s})[V_u(\mathbf{s}') - V_l(\mathbf{s}')]$, which focuses search on states that are less likely to be converged. Smith and Simmons [118] introduced a similar algorithm named focused RTDP (FRTDP). Similar to BRTDP, FRTDP also keeps two bounds for a state, but uses slightly different metrics for picking the next state. Also, FRTDP assumes a discounted cost setting, so it is not immediately applicable to SSP problems.

Recently Sanner et al. [111] described another advanced RTDP variant named Bayesian RTDP, which also uses two value bounds. Like BRTDP, the basic idea of Bayesian RTDP is to also focus search/execution trials on unconverged states. Its key assumption is that the true value function of a state $\mathbf{s}$, $V^*(\mathbf{s})$, is uniformly distributed on the interval $[V_l(\mathbf{s}), V_u(\mathbf{s})]$. Therefore, the probability density function of $V^*(\mathbf{s})$ is $\mathbf{1}_{v \in [V_l(\mathbf{s}), V_u(\mathbf{s})]}[\frac{1}{V_u(\mathbf{s}) - V_l(\mathbf{s})}]$, and $E[V^*(\mathbf{s})] = \frac{1}{2}[V_l(\mathbf{s}) + V_u(\mathbf{s})]$. To evaluate how important it is to pick state $\mathbf{s}'$ as the next state, it refers to the notion of *value of perfect information* (VPI), which intuitively tells the expected Q-value difference of the current state-action pair, $Q(\mathbf{s}, \mathbf{a})$, with and without the knowledge of $V^*(\mathbf{s}')$. To choose $\mathbf{s}'$, Bayesian RTDP uses a metric that combines the BRTDP metric and the VPI value.

Exploration and Exploitation

Recently a general Monte-Carlo (MC) simulation algorithm named UCT (short for upper confidence bounds applied on trees) [77] was proposed to approximately solve large MDP problems. Its power has been demonstrated by its application to several challenging problems such as Go [56] and Real-time Strategy Games [7]. Applying the MC simulation algorithm to an MDP M, the pesudo-code is shown in Algorithm 2. The algorithm performs simulation trials originating from the initial states $\mathbf{s_0}$ continuously until running out of time. During each simulation, one action $\mathbf{a}$ is chosen for the current state $\mathbf{s}$, based on some heuristic function h. Same as RTDP, the current state is chosen stochastically based on the transition probabilities of the action. One simulation trial terminates when a goal state is encountered, and then the Q-values of all visited state action pairs are backed up

Algorithm 2 Monte-Carlo simulation for MDPs

1: **input:** an MDP M with state space $\mathcal{S}$, initial state $\mathbf{s_0}$, goal states $\mathcal{G}$, a heuristic function h, and the learning rate θ.
2: **for** each state **s do**
3: **for** each action **a do**
4: initialize $Q(\mathbf{s}, \mathbf{a})$ arbitrarily
5: **while** not TIMEOUT **do**
6: search($\mathbf{s_0}$)
7:
8: **search(s)**
9: $visited \leftarrow$ empty stack
10: **if** $s \notin \mathcal{G}$ **then**
11: $\mathbf{a} \leftarrow argmin_{\mathbf{a} \in Ap(\mathbf{s})} h(\mathbf{s}, \mathbf{a})$
12: choose $\mathbf{s}'$ stochastically from $T_{\mathbf{a}}(\cdot|\mathbf{s})$
13: $visited$.push$(\mathbf{s}, \mathbf{a})$
14: $\mathbf{s} \leftarrow \mathbf{s}'$
15: $r \leftarrow V^*(\mathbf{s})$
16: **while** $visited \neq$ empty stack **do**
17: $(\mathbf{s}, \mathbf{a}) \leftarrow$ pop$(visited)$
18: $Q(\mathbf{s}, \mathbf{a}) \leftarrow \theta Q(\mathbf{s}, \mathbf{a}) + (1 - \theta)[C(\mathbf{s}, \mathbf{a}) - r]$
19: $r \leftarrow Q(\mathbf{s}, \mathbf{a})$

in the reverse order they are visited during the trial.

Different heuristics can be applied in choosing actions for the current state, as a tradeoff between exploration versus exploitation. One can pick a greedy action, as an example of pure exploitation. Oppositely, a random action can be chosen, as an example of exploration. UCT is a combination of both. It gives credits to already proved, better actions, but at the same time pays attention to some under-explored branches, in case better policies can be found. It keeps the total number of times a state $\mathbf{s}$ is visited, $n_{\mathbf{s}}$, and the times action $\mathbf{a}$ is

picked when $\mathbf{s}$ is visited, $n_{(\mathbf{s},\mathbf{a})}$, where $\sum_{\mathbf{a}\in\mathcal{A}} n_{(\mathbf{s},\mathbf{a})} = n_{\mathbf{s}}$. The heuristic function $h_{UCT}(\mathbf{s})$ is defined as follows:

$$h_{UCT}(\mathbf{s}) = Q(\mathbf{s},\mathbf{a}) - \kappa\sqrt{\frac{2ln\ n_{\mathbf{s}}}{n_{(\mathbf{s},\mathbf{a})}}} \tag{2.8}$$

The first term of the evaluation function is the Q-function of the state action pair. The second term shows a bias on actions that are visited less frequently, as the value $n_{(\mathbf{s},\mathbf{a})}$ is on the numerator. The *exploration parameter* κ is used to balance between exploration and exploitation. Theoretical error bounds given sufficient number of trials have been proved (see [77] for detail).

2.2 Partially-Observable Markov Decision Processes

Partially-observable MDPs (POMDPs) [72] bring more flexibility in modeling problems by adding the layer of uncertainty in knowing the exact state the agent is currently in. Due to its generality almost all single-agent, real-world problems can be modeled as a POMDP.

Definition 9 *The paradigm of a POMDP extends an MDP by adding a finite set of observations $\mathcal{O}$ and the corresponding observation model $p(\mathbf{o}|\mathbf{s})$, the probability that the agent observes $\mathbf{o}$ at state $\mathbf{s}$.*

Typically the agent does not have complete information of the state it is currently in, but instead has a *belief state* $\mathbf{b}$:

$$\mathbf{b}(\mathbf{s}) \in [0,1],\ \sum_{\mathbf{s}\in\mathcal{S}} \mathbf{b}(\mathbf{s}) = 1, \tag{2.9}$$

where $\mathbf{b}(\mathbf{s})$ is the probability that the current state is $\mathbf{s}$, inferred from the history of the previous actions and observations.

To solve an POMDP problem, one typically transforms the problem by building a model that contains all the belief states of the POMDP. As the transition between two belief states is known, the resulting model is an MDP. Therefore, solving the POMDP problem maps to finding the optimal policy of the transformed, belief-state MDP. However, without reachability test, the belief state space is infinite even when a POMDP has only two states $\mathbf{s_1}$ and $\mathbf{s_2}$, because a belief state can be an arbitrary combination of $\mathbf{b}(\mathbf{s_1}) = p$ and $\mathbf{b}(\mathbf{s_2}) = 1-p$ where p can be any real number in $[0,1]$.

Due to the infinite state-space size, solving a POMDP is a hard problem. For the simplest case where the set of $\mathcal{A}$, $\mathcal{S}$ and $\mathcal{O}$ are finite, Sondik [120] proved that the optimal value function for finite horizon is piecewise linear and convex (PWLC), so the value of a belief state $\mathbf{b}$ can be represented by a finite set of $|\mathcal{S}|$ dimensional *α-vectors*. They use a α-vector of dimension $|\mathcal{S}|$ to represent the value of a policy graph rooted at state $\mathbf{s}$. Therefore, the value of a belief state $\mathbf{b}$ can be represented by:

$$\langle \mathbf{b}, \alpha \rangle = \int_{\mathbf{s} \in \mathcal{S}} \alpha(\mathbf{s}) \mathbf{b}(\mathbf{s}). \tag{2.10}$$

Porta *et al.* [107] are the first to prove that continuous Bellman backups are contracting and isotonic, which guarantee that value iteration type of algorithms converge on α-vector backups. However, the number of reachable belief states on a policy graph grows exponentially in the number of observations $|\mathcal{O}|$. To solve a POMDP problem the complexity of one update iteration is exponential in the total number of observations [72]. For an indefinite horizon POMDP one needs to perform an infinite number of iterations in the worst case, as the total number of reachable belief states can be infinite. Therefore, the problem is undecidable [87] in the worst case.

Researchers seek various approximation methods to solve POMDPs efficiently. One simple method is through discretization. The idea is to transfer continuous variables into discrete variables, whereby decrease the problem space from infinite to finite. It is typically implemented by bucketing the range of a continuous variable into a finite number of intervals and regarding all variable assignments falling into the same interval as a same value. However, even with discretization, some problems are still too big to solve.

Point-based value iteration [122, 105, 114] is a very successful approach. The basic idea is to approximate the value function of the belief space $\mathcal{B}$ by only estimating the values of a fixed, sampled subset, $\bar{\mathcal{B}} \subseteq \mathcal{B}$. For all the other belief states, the values are approximately estimated according to the values of the sampled space, *e.g.*, inferred from integration [107] or mixture Gaussian representation [24]. For a tractable, finite-horizon problem, the algorithm iteratively computes an approximately optimal t-horizon value function V_t from V_{t-1}

by the following backup operator:

$$\begin{aligned} V_t(\mathbf{b}) &= \min_{\mathbf{a}\in A} \sum_{\mathbf{s}\in\mathcal{S}} [C(\mathbf{s},\mathbf{a})\mathbf{b}(\mathbf{s}) \\ &\quad + \sum_{\mathbf{o}\in\mathcal{O}} min_{\alpha_{t-1}\in V_{t-1}} \sum_{\mathbf{s}\in\mathcal{S}} \sum_{\mathbf{s}'\in\mathcal{S}} T(\mathbf{s}'|\mathbf{s},\mathbf{a}) p(\mathbf{o}|\mathbf{s}') \alpha_{t-1}(\mathbf{s}')\mathbf{b}(\mathbf{s})]. \end{aligned} \tag{2.11}$$

Due to the PWLC properties of a discrete POMDP, the values computed by the point-based algorithms are an upper bound of the optimal value function of the POMDP. However, no known bounds on how close to optimal have been established.

Chapter 3

FASTER OPTIMAL PLANNING

The invention of heuristic search algorithms, such as RTDP [8] variants and LAO* [58], significantly speeds up the convergence of optimal MDP algorithms. However, they sometimes perform poorly, especially on large and complex problems. This chapter discusses a thread of new algorithms that solve an MDP much faster by using its graphical information.

3.1 A Limitation of Previous Solvers

Value iteration backs up states iteratively based on some fixed order. Heuristic search backs up states in a dynamic, informed order, implied by when they are visited in the search. A state can be backed up in the pre-order (when it is first visited, *e.g.*, variants of RTDP), or the post-order (when searches back track, *e.g.*, ILAO*). None of the algorithms use an MDP's graphical structure, an intrinsic property that governs the complexity of solving a problem [85], in a way to decide the order in which states are solved.

Consider a PhD program in some Finance department. Figure 3.1 shows an MDP that describes the progress of a PhD student. For simplicity reasons, we omit the action nodes, the transition probabilities, and the cost functions. The goal state set is a singleton $\mathcal{G} = \{\mathbf{g}\}$, which indicates a student gets her PhD degree. A directed edge between two states means the head state is one successor state of the tail state under at least one action. The initial state, $\mathbf{s_0}$, describes the status of an entry-level student. She has to first pass the qualifying exam, which consists of finding a supervisor and passing an exam. Before passing the exam one can choose to work with a different supervisor (back to state $\mathbf{s_0}$ in the figure). State $\mathbf{s_1}$ indicates the student has found a supervisor. Then she works on her proposal, which consists of a written document and an oral exam. She has to pass both in two consecutive quarters; otherwise back to state $\mathbf{s_2}$. After passing the proposal, at state $\mathbf{s_4}$, she needs to defend her thesis, passing which reaches the goal state $\mathbf{g}$.

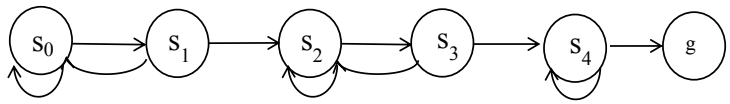


Figure 3.1: A simple MDP example. The action nodes, the transition probabilities, and the cost functions are omitted. The goal state set is a singleton $\mathcal{G} = \{\mathbf{g}\}$. A directed edge between two states means the head state is one successor state of the tail state under some action.

Observing the MDP, we find the optimal order to back up states is $\mathbf{s_4}$, then $\mathbf{s_2}$ and $\mathbf{s_3}$, till they converge, followed by $\mathbf{s_0}$ and $\mathbf{s_1}$. The reason is that the value of $\mathbf{s_4}$ does not depend on the values of other non-goal states. Similarly, the values of $\mathbf{s_2}$ and $\mathbf{s_3}$ do not depend on the values of either $\mathbf{s_0}$ or $\mathbf{s_1}$. Value iteration as well as heuristic search algorithms do not take advantage of the graphical structure and apply this backup order, as they do not contain an "intelligent" subroutine that discovers the graphical structure, nor use this information in the dynamic programming step. The intuition of our new approaches is to discover the intrinsic complexity of solving an MDP by studying its graphical structure, which later contributes to a more intelligent backup order.

3.2 Topological Value Iteration

First observe that the value of a state depends on the values of its successors. For example, suppose state $\mathbf{s_2}$ is a successor state of $\mathbf{s_1}$ under action $\mathbf{a}$ ($T_{\mathbf{a}}(\mathbf{s_2}|\mathbf{s_1}) > 0$). By the Bellman equations $V^*(\mathbf{s_1})$ is dependent on $V^*(\mathbf{s_2})$. In this case, we define state $\mathbf{s_1}$ *causally depends* on state $\mathbf{s_2}$. Note that the causal dependence relationship is transitive. We can find out all causally dependent states implicitly by building a reachability graph G_R of the MDP. The set of vertices of G_R equals the set of states that are reachable from $\mathbf{s_0}$. A directed edge from vertex $\mathbf{s_1}$ to $\mathbf{s_2}$ means that there exists at least one action $\mathbf{a} \in Ap(\mathbf{s_1})$, such that $T_{\mathbf{a}}(\mathbf{s_2}|\mathbf{s_1}) > 0$. As the causal relationship is transitive, a directed path from state $\mathbf{s_1}$ to $\mathbf{s_k}$ in G_R means $\mathbf{s_1}$ is causally dependent on $\mathbf{s_k}$, or $V^*(\mathbf{s_1})$ depends on $V^*(\mathbf{s_k})$. Also note that two vertices can be causally dependent on each other, which we call *mutual causal dependence*, if they are successors of each other.

Due to causal dependence, it is usually more efficient to back up $\mathbf{s_2}$ ahead of $\mathbf{s_1}$. With

this observation, we have the following theorem.

Theorem 10 *If an MDP is acyclic, then there exists an optimal backup order. By applying the optimal order, the optimal value function can be found with each state needing only one backup.*

The theorem is easy to prove and, furthermore, the optimal backup order is a topological order of the vertices in G_R. However, in general, MDPs contain cycles and it is common for one state to mutually causally depend on another.

If two states are mutually causally dependent, the best order to back up them is unclear. On the other hand, if neither state is causally dependent on the other, the order of backup does not matter. Finally, if one state is causally dependent on the other (and not vice versa), it is better to order the backups so that the state which is causally dependent is updated later. To apply this idea we then group together states that are mutually causally dependent and make them a *meta-state*. We make a new directed graph G_M where a directed edge between two meta-states $\mathcal{X}$ and $\mathcal{Y}$ exists if and only if there exists two states $\mathbf{s_1}$ and $\mathbf{s_2}$ and an action $\mathbf{a} \in Ap(\mathbf{s_1})$ such that $\mathbf{s_1} \in \mathcal{X}$, $\mathbf{s_2} \in \mathcal{Y}$ and $T_{\mathbf{a}}(\mathbf{s_2}|\mathbf{s_1}) > 0$. It is clear that G_M is acyclic, otherwise all states on such a cycle are mutually causally dependent, and by our construction rule they should belong to the same meta-state. In this case, we can back up states in G_M in their topological order. By Theorem 10, each such state only requires one *meta-backup*. It is called a meta-backup since a meta-state may contain multiple states. To perform a meta-backup, we can apply any dynamic programming algorithm, such as value iteration, on all states belonging to the corresponding meta-state.

The pseudo-code of TVI is shown in Algorithm 3. We first apply Kosaraju's algorithm [29] to find the set of strongly connected components (SCCs, or meta-states) in the causality graph G_R, and its topological order. ($id[\mathbf{s}]$ indicates the topological order of the SCC that state $\mathbf{s}$ belongs to.) It is based on the fact that by reversing all the edges in G_R, the resulting graph, G'_R, has the same strongly connected components as the original. From using that, we can get the SCCs by doing a forward traversal to find an ordering of vertices, followed by a traversal of the reverse of the graph in the order generated by the first traversal. Kosaraju's algorithm is efficient, as its time complexity is linear in the number

of states. When the state space is large, running the algorithm leads to unavoidable yet acceptable overhead. In many cases the overhead is well compensated by the computational gain. We then use value iteration to solve each SCC $\mathcal{C}$ (as a meta-backup) in its topological order.

Algorithm 3 Topological Value Iteration

1: **Input:** an MDP $M = \langle \mathcal{S}, \mathcal{A}, Ap, T, C \rangle$, δ: the threshold value
2: **SCC**(M)
3: **for** $i \leftarrow 1$ to *cpntnum* **do**
4: $\quad \mathcal{S}' \leftarrow$ the set of states $\mathbf{s}$ where $id[\mathbf{s}] = i$
5: $\quad M' \leftarrow \langle \mathcal{S}', \mathcal{A}, Ap, T, C \rangle$
6: $\quad$ **VI**(M', δ)
7:
8: **Function** $SCC(M)$ (Kosaraju's algorithm)
9: construct G_R of M
10: construct a graph G'_R which reverses the head and tail vertices of every edge in G_R
11: {call Kosaraju's algorithm [29]. It inputs G_R and G'_R and outputs *cpntnum*, the total number of SCCs, and $id : \mathcal{S} \rightarrow [1, cpntnum]$, the id of the SCC each state belongs to, by topological order.}
12: **return** $(cpntnum, id)$

3.2.1 Optimality

When the Bellman operator is a *contraction operation* [12], we have:

Theorem 11 *Topological Value Iteration is guaranteed to converge to a value function with a Bellman error that is no greater than δ.*

Proof We first prove that TVI is guaranteed to terminate in finite time. Since each MDP contains a finite number of states, it contains a finite number of connected components. In solving each of these components, TVI uses value iteration. Because value iteration is guaranteed to converge in finite time (given a finite δ), TVI, which is essentially a finite number of value iterations, terminates in finite time.

We then prove TVI is guaranteed to converge to an optimal value function with Bellman error at most δ. We prove by induction.

First, if an MDP contains only one SCC, then TVI coincides with VI, an optimal algorithm. By the contraction property of Bellman backups, when VI converges, the Bellman error of the state space is at most δ.

Now, consider the case where an MDP contains multiple SCCs. At any point, TVI is working on one component $\mathcal{C}$. We know that the optimal value of every state $\mathbf{s} \in \mathcal{C}$, $V^*(\mathbf{s})$, depends only on the optimal values of the states that are descendants of $\mathbf{s}$. We also know that any descendant $\mathbf{s}'$ of $\mathbf{s}$ must belong either to $\mathcal{C}$, or a component $\mathcal{C}'$ that is topologically no later than $\mathcal{C}$. This means either its value is computed by VI in the same batch as $\mathbf{s}$ ($\mathbf{s}' \in \mathcal{C}$), or state $\mathbf{s}'$ is already converged ($\mathbf{s}' \in \mathcal{C}'$). Therefore, when VI finishes solving $\mathcal{C}$, the value of $\mathbf{s}$ must converge with Bellman residual at most δ. We also know that the values of all states that belong to a component that is earlier than $\mathcal{C}$ does not depend on those of states in component $\mathcal{C}$. Therefore, after component $\mathcal{C}$ converges, the Bellman residual of states in those components remain unchanged and thus are at most δ. From above, when TVI terminates, the Bellman residuals of all states are at most δ. This means the Bellman error of the state space is at most δ.

From the high-level perspective, TVI decomposes an MDP into sub-problems and finds the value of the state space in a batch manner, component by component. When a component is converged, all its states will be safely treated as sink states, as their values do not depend on values of states belonging to later components.

3.2.2 Implementation

We made two optimizations in implementing TVI. The first one is an uninformed reachability analysis. TVI does not depend on any initial state information. However, once given that information, TVI is able to mark the reachable components and later ignore the unreachable ones in the dynamic programming step. The reachable state space can be found by a depth-first search starting from s_0, with an overhead that is linear in $|\mathcal{S}|$ and $|\mathcal{A}|$. It is extremely useful when only a small portion of the state space is reachable (*e.g.*, most

domains from the International Planning Competition 2006 [67]).

The second optimization is to use heuristic values $V_l(\cdot)$ as a starting point. We used the h_{min} [17], an admissible heuristic:

$$\begin{aligned} h_{min}(s) &= 0 \quad \text{if } \mathbf{s} \in \mathcal{G}, \text{ else} \\ h_{min}(\mathbf{s}) &= \min_{\mathbf{a} \in Ap(\mathbf{s})} \left[C(\mathbf{s}, \mathbf{a}) + min_{\mathbf{s}':T_{\mathbf{a}}(\mathbf{s}'|\mathbf{s})>0} h_{min}(\mathbf{s}') \right]. \end{aligned} \tag{3.1}$$

To implement it, we first construct a new deterministic problem. For each action and successor pair of the original MDP, we add to the new problem a deterministic action with the same cost and the same, deterministic successor. We then solve this new problem by a single, backward, breadth-first search from the set of goal states. Values of the deterministic problems are h_{min}.

3.2.3 Experiments

We address the following questions in our experiments: (1) How does TVI compare with VI and heuristic search algorithms on MDPs that contain multiple SCCs? (2) What are the most favorable problem features for TVI?

We compared TVI with several other optimal algorithms, including VI [9], ILAO* [58], LRTDP [17], BRTDP [93], Bayesian RTDP [111] (BaRTDP), and HDP [16][1]. We used the fully optimized C code of ILAO* provided by Eric A. Hansen and additionally implemented the rest of the algorithms over the same framework. We performed all experiments on a 2.5GHz Dual-Core AMD Opteron(tm) Processor with 2GB memory. Recall that BRTDP and BaRTDP use upper bounds. We used the upper bounds as described in Chapter 3.3.2. We used $\alpha = 2 \times 10^{-6}$ and $\tau = 10$ for BRTDP and BaRTDP.[2] For BaRTDP, we used the probabilistic termination condition in Algorithm 3 of [111]. [3]

---

[1]Notice that this comparison is somewhat unfair to TVI, since heuristic search algorithms may not expand portions of the state space, if their sub-optimality can be proved. Still, we make this comparison to understand the practical benefits of TVI v.s. all other known optimal MDP algorithms

[2]α is the termination threshold of BRTDP (it terminates when $v_u(\mathbf{s_0}) - V_l(\mathbf{s_0}) < \alpha$). τ indicates the stopping condition of each heuristic search trial. For more detailed discussions on the two parameters, please refer to [93].

[3]This termination condition may result in sub-optimal policies, so the reported times of BaRTDP in

We compared all algorithms on *running time*, time between an algorithm starts solving a problem until generating a policy with a Bellman error of at most $\delta(= 10^{-6})$. We terminated an algorithm if it did not find such a policy within five minutes. Note that there are other performance measures such as anytime performance (the original motivation of BaRTDP) and space consumption, but the main motivation of TVI is to decrease convergence time. We expect TVI to have a very steep anytime performance curve, because it postpones backing up the initial state till it starts working on the SCC where the initial state belongs to. Space, on the other hand, is less interesting because in-memory MDPs algorithms requires that the MDP model stored in the main memory before dynamic programming can apply. Therefore, they all share the same space limit. For work on overcoming space limitation, see the next Chapter.

We tested all algorithms on a set of artificially-generated "layered" MDPs. For each such MDP of state size $|\mathcal{S}|$, we partition the state space evenly into a number n_l of *layers*, labeled by integers $1, \ldots, n_l$. We allow states in higher numbered layers to be the successors of states in lower numbered layers, but not vice versa, so each state $\mathbf{s}$ only has a limited set of allowable successor states, denoted by $succ(\mathbf{s})$. A layered MDP is parameterized by two other variables: the number of actions per state, n_a, and the maximum number of successor states per action, n_s. When generating the transition function of a state-action pair $(\mathbf{s}, \mathbf{a})$, we draw an integer k uniformly from $[1, n_s]$. Then k distinct successors are uniformly sampled from $succ(\mathbf{s})$ with random transition probabilities. We pick one state from layer n_l as the only goal state. One property of a layered MDP is that it contains at least n_l connected components.

There are several planning domains that lead to multi-layered MDPs. An example is the game Bejeweled, or any game with difficulty levels: each level is at least one layer. Or consider a chess variant without pawn promotions, played against a stochastic opponent. Each set of pieces that could appear on the board together leads to at least one strongly connected component. But we know of no multi-layered standard MDP benchmarks. Therefore, we

---

this paper are lower bounds. Note that BaRTDP mainly aims at improving the anytime performance of RTDP, which is *orthogonal* to convergence time. We report its convergence speed for thorough investigation purposes.

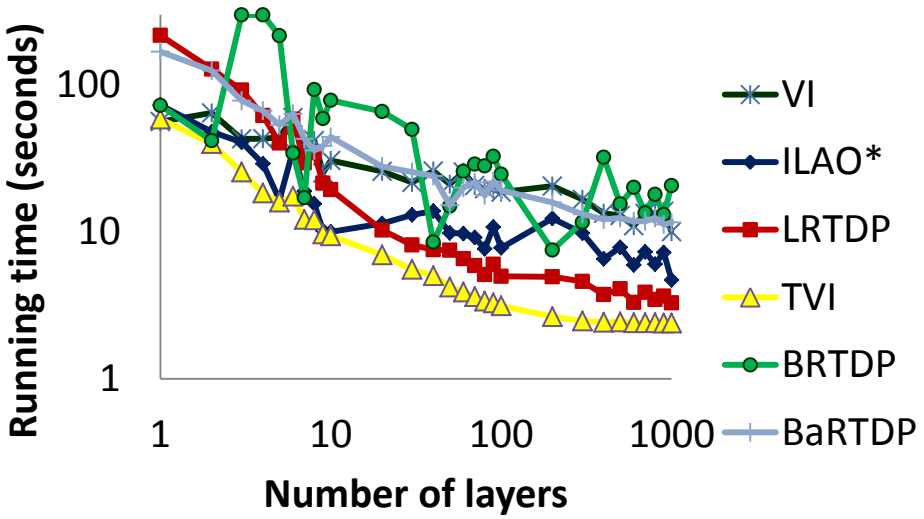


Figure 3.2: Running times of algorithms with different number of layers n_l on random layered MDPs with $|\mathcal{S}| = 50000$, $n_a = 10$, and $n_s = 10$. Note that the two coordinates are both log-scaled. When $n_l > 10$ TVI not only outperforms VI, but also other state-of-the-art heuristic search algorithms.

compare, in this section, on artificial problems to study TVI's performance across controlled parameters, such as n_l and $|\mathcal{S}|$. Next section contains more comprehensive experiments on benchmark problems.

We generated problems with different parameter configurations and ran all algorithms on the same set of problems. The running times, if the process converged within the cut-off, are reported in Figures 3.2 and 3.3. Each element of the table represents the median convergence time of running 10 MDPs with the same configuration.[4] Note that varying $|\mathcal{S}|$, n_l, n_a, and n_s yields many MDP configurations. We tried more combinations than the representative ones reported. We found HDP much slower than the other algorithms, so did not include its performance.

For the first experiment, we fixed $|\mathcal{S}|$ to be 50,000 and varied n_l from 1 to 1,000. Observing Figure 3.2 we first find that, when there is only one layer, the performance of TVI

[4]We picked median instead of mean just to avoid an unexpected hard problem, which takes a long time to solve, thereby dominating the performance.

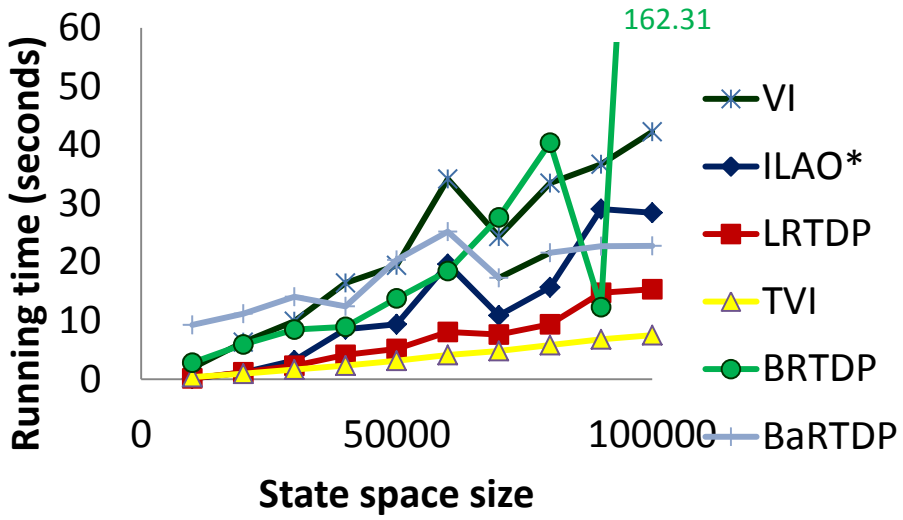


Figure 3.3: Running times of algorithms with different state space size $|\mathcal{S}|$ with fixed $n_l = 100$, $n_a = 10$, and $n_s = 10$. TVI not only outperforms VI, but also other state-of-the-art heuristic search algorithms. The relative performance of TVI improves as $|\mathcal{S}|$ increases.

is slightly worse than VI, as such an MDP probably contains an SCC that contains the majority of the state space, which defeats the benefit of TVI. But TVI consistently outperforms VI if $n_l > 1$. When $n_l \leq 10$, TVI equals or beats ILAO*, the fastest heuristic search algorithm for this set of problems. When $n_l > 10$, TVI outperforms all the other algorithms in all cases by a visible margin. Also note that, as the number of layers increases the running times of all algorithms decrease. This is because the MDPs become more structured, therefore simpler to solve. The running time of TVI decreases second fastest to that of LRTDP. LRTDP is very slow when $n_l = 1$ and its running time drops dramatically when n_l increases from 1 to 20. As TVI spends constant time in generating the topological order of the SCCs, its fast convergence is mainly due to the fact that VI is much more efficient in solving many small (and roughly equal-sized) problems than a large problem whose size is the same as the sum of the small ones. This experiment shows TVI is good at solving MDPs with many SCCs.

For the second experiment, we fixed n_l to be 100 and varied $|\mathcal{S}|$ from 10,000 to 100,000.

We find that, when the state space is 10,000 TVI outperforms VI, BRTDP and BaRTDP, but slightly underperforms ILAO* and LRTDP. However, as the problem size grows TVI soon takes the lead. It outperforms all the other algorithms when the state space is 20,000 or larger. When the state space grows to 100,000, TVI solves a problem 6 times as fast as VI, 4 times as fast as ILAO*, 2 times as fast as LRTDP, 21 times as fast as BRTDP, and 3 times as fast as BaRTDP. This experiment shows that TVI is even more efficient when the problem space is larger.

3.3 Focused Topological Value Iteration

Topological value iteration improves the performance of value iteration most significantly when an MDP has many equal-sized strongly connected components. However, we also observe that many MDPs do not have evenly distributed connected components. This is due to the following reason: a state can have many actions, most of which are sub-optimal. These sub-optimal actions, although not part of an optimal policy, may lead to connectivity between a lot of states. For example, domains like Blocksworld have reversible actions. Due to these actions most states are mutually causally dependent. As a result, states connected by reversible actions end up forming a large connected component, making TVI slow.

On the other hand, heuristic search is a powerful solution technique, which successfully concentrates computation, in the form of backups, on states and transitions that are more likely to be part of an optimal policy. However, heuristic search uses the same backup strategy on all problems, thus missing out on the potential savings from knowing the graphical structure information.

If we knew about the existence of an action in the optimal policy, we could eliminate the rest actions for its outgoing state, thus breaking some connectivity. Of course, such information is never available. However, with a little help from heuristic search, we can eliminate sub-optimal actions from a problem leading to a reduced connectivity and hopefully, smaller sizes of strongly connected components.

Figure 3.4 shows the graphical representation of a part of one simple MDP that has 7 states and 12 actions. In the figure, successors of probabilistic actions are connected by an arc. For simplicity, transition probabilities $T_{\mathbf{a}}$, costs $C(\mathbf{s}, \mathbf{a})$, initial state and goal states are

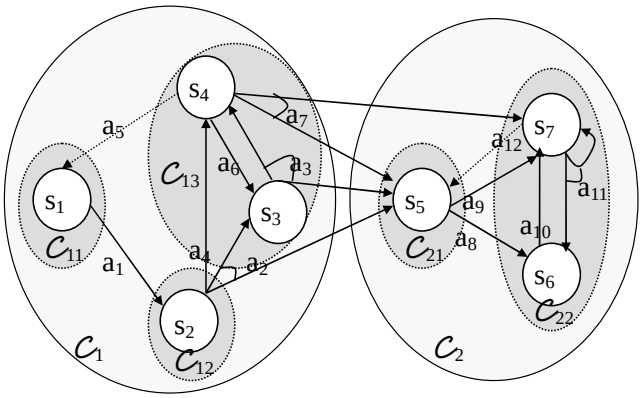


Figure 3.4: The graphical representation of an MDP and its set of strongly connected components (before and after the knowledge of some sub-optimal actions). Arcs represent probabilistic transitions, *e.g.*, $\mathbf{a_7}$ has two probabilistic successors – $\mathbf{s_5}$ and $\mathbf{s_7}$.

omitted. Using TVI, we can divide the MDP into two SCCs $\mathcal{C}_1$ and $\mathcal{C}_2$. However, suppose we are given some additional information that $\mathbf{a_5}$ and $\mathbf{a_{12}}$ are sub-optimal. Based on the remaining actions, $\mathcal{C}_1$ and $\mathcal{C}_2$ can be sub-divided into three and two smaller components respectively (as shown in the figure). Dynamic programming will greatly benefit from the new graphical structure, since solving smaller components can be much easier than a large one.

3.3.1 The FTVI Algorithm

The key insight of our novel algorithm is to break the big components into smaller parts, by removing actions that can be proven to be suboptimal for the current problem at hand. This exploits the knowledge of the current initial state and goal, which TVI mostly ignores. We call our new algorithm *focused topological value iteration* (FTVI) [39]. The pseudo-code is shown in Algorithm 4.

At its core, FTVI makes use of the *action elimination* theorem, which states:

Theorem 12 Action Elimination *[12]: If a lower bound of* $Q^*(\mathbf{s}, \mathbf{a})$ *is greater than an upper bound of* $V^*(\mathbf{s})$ *then action* $\mathbf{a}$ *cannot be an optimal action for state* $\mathbf{s}$.

This gives us a template to eliminate actions, except that we need to compute a lower bound for Q^* and an upper bound for V^*. FTVI keeps two bounds of V^* simultaneously: the lower bound $V_l(\cdot)$ and the upper bound $V_u(\cdot)$. $V_l(\cdot)$ is initialized via the admissible heuristic. We note two properties of V_l: (1) $Q_l(\mathbf{s}, \mathbf{a})$ computed by a one-step lookahead given the current lower bound value $V_l(\cdot)$ (Line 30, Algorithm 4) is a lower bound of $Q^*(\mathbf{s}, \mathbf{a})$, and (2) all the V values remain lower bounds throughout the algorithm execution process, if they were initialized by an admissible heuristic. So, this lets us easily compute a lower bound of Q^*, which also improves as more backups are performed.

Similar properties hold for V_u, the upper bound of V^*, *i.e.*, if we initialize V_u by an upper bound and perform backups based on V_u then each successive value estimate remains an upper bound. The later implementation section lists our exact procedure to compute the lower and upper bounds in a domain-independent manner. We note that to employ action elimination we can use any lower and upper bounds, so if a domain has informative, domain-dependent bounds available, that can be easily plugged into FTVI.

FTVI contains two sequential steps. In the first step, which we call the *search* step, FTVI performs a small number of heuristic searches similar to ILAO*, *i.e.*, backs up a state at most once per iteration. This makes the searches in FTVI fast, but still useful enough to eliminate sub-optimal actions. There are two main differences in common heuristic search and the search phase of FTVI. First, in each backup, we update the upper bound in the same manner as the lower bound. This is reminiscent of backups in BRTDP [93]. Second, we also check and eliminate sub-optimal actions using action elimination (Lines 30–33).

In the second step, the *computation* step, FTVI generates a directed graph G_{SR} in the same manner as TVI generates G_R, but only based on the remaining actions. More concretely, a directed edge from vertex $\mathbf{s_1}$ to $\mathbf{s_2}$ exists if there is an *uneliminated* action $\mathbf{a}$ such that $T_{\mathbf{a}}(\mathbf{s_2}|\mathbf{s_1}) > 0$. It is easy to see that the graph G_{SR} generated is always a sub-graph of G_R. FTVI then finds all connected components of G_{SR}, their topological order, and solves each component sequentially in the topological order.

We can state the following theorem for FTVI.

Theorem 13 *FTVI is guaranteed to converge to the optimal value function.*

Algorithm 4 Focused Topological Value Iteration

Input: an MDP $\langle \mathcal{S}, \mathcal{A}, Ap, T, C \rangle$, x: the number of search iterations in a batch, y: the lower bound of the percentage of change in the initial state value for a new batch of search iterations, δ: the threshold value
{step 1: search}
while true **do**
    $old\_value \leftarrow V_l(\mathbf{s_0})$
    for $iter \leftarrow 1$ to x **do**
        $Bellman\_error \leftarrow 0$
        for every state s **do**
            mark every state as unvisited
        $\mathbf{s} \leftarrow \mathbf{s_0}$
        $Search(\mathbf{s})$
        if $Bellman\_error < \delta$ **then** {The value function converges}
            return V_l
    if $old\_value/V_l(s_0) > (100 - y)\%$ **then**
        break

{step 2: computation}
$M' \leftarrow \langle \mathcal{S}', \mathcal{A}, Ap, T, C \rangle$
TVI(M', δ) {by applying the backup operator with action elimination}

Function $Search$ **Input:** a state $\mathbf{s}$
if $\mathbf{s} \notin \mathcal{G}$ **then**
    mark $\mathbf{s}$ as visited
    $\mathbf{a} \leftarrow argmin_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a})$
    for every unvisited successor $\mathbf{s}'$ of action $\mathbf{a}$ **do**
        $Search(\mathbf{s}')$
    $Bellman\_error \leftarrow max(Bellman\_error, Back - up(s))$

Function $Back - up$ **Input:** a state $\mathbf{s}$
for each action $\mathbf{a}$ of $Ap(\mathbf{s})$ **do**
    $Q(\mathbf{s}, \mathbf{a}) \leftarrow C(\mathbf{s}, \mathbf{a}) + \sum_{\mathbf{s}' \in \mathcal{S}} T_{\mathbf{a}'}(\mathbf{s}'|\mathbf{s}) V_l(\mathbf{s}')$
    if $Q_l(\mathbf{s}, \mathbf{a}) > V_u(\mathbf{s})$ **then**
        {$\mathbf{a}$ is sub-optimal}
        eliminate $\mathbf{a}$ from $Ap(\mathbf{s})$
$oldV_l \leftarrow V_l(\mathbf{s})$
$V_l(\mathbf{s}) \leftarrow min_{\mathbf{a} \in Ap(\mathbf{s})} Q(\mathbf{s}, \mathbf{a})$
$V_u(\mathbf{s}) \leftarrow min_{\mathbf{a} \in Ap(\mathbf{s})} [C(\mathbf{s}, \mathbf{a}) + \sum_{\mathbf{s}' \in \mathcal{S}} T_{\mathbf{a}'}(\mathbf{s}'|\mathbf{s}) V_u(\mathbf{s}')]$
return $|V_l(\mathbf{s}) - oldV_l|$

The correctness of the theorem is based on two facts: (1) action elimination preserves soundness, and (2) TVI is an optimal planning algorithm (Theorem 11).

3.3.2 Implementation

There are several interesting questions to answer in implementation. How to calculate the initial upper and lower bounds? How many search iterations do we need to perform in the search step? Is it possible that FTVI converges in the search step? What if there still remains a large component even after action elimination?

We used the same lower bound V_l as in TVI (see Section 3.2.2). For the upper bound, we started with a simple upper bound:

$$V_u(\mathbf{s}) = 0 \quad \text{if } \mathbf{s} \in \mathcal{G}, \text{ else} \quad V_u(\mathbf{s}) = \infty. \tag{3.2}$$

This initialization gives us a global yet very imprecise upper bound. To improve its tightness, we performed a backward best-first search from the set of goal states. States visited have their V_u values updated as in Algorithm 4, Line 35. We can iteratively get tighter and tighter bounds when more backward searches are performed.

The time spent on search can have a significant impact on FTVI. Very few search iterations might not eliminate enough sub-optimal actions. However, too many search iterations will turn FTVI into a heuristic search algorithm and trade off the advantage of FTVI. We did a control experiment by varying the total number of heuristic search trials on two problems. Figure 3.5 shows that the performance on a Wet-floor problem matches our hypothesis perfectly. For the Drive problem, the number of search trials does not affect the convergence speed too much, but too many search trials turn out to be harmful.

Considering the tradeoff, we let the algorithm automatically determine the number of search iterations. FTVI incrementally performs a batch of x search iterations. After the batch, it computes the amount of change to the $V_l(s_0)$ value. If the change is greater than $y\%$, a new batch of search is performed. Otherwise, the search phase is considered complete. In our implementation, we use $x = 100$, and $y = 3$.

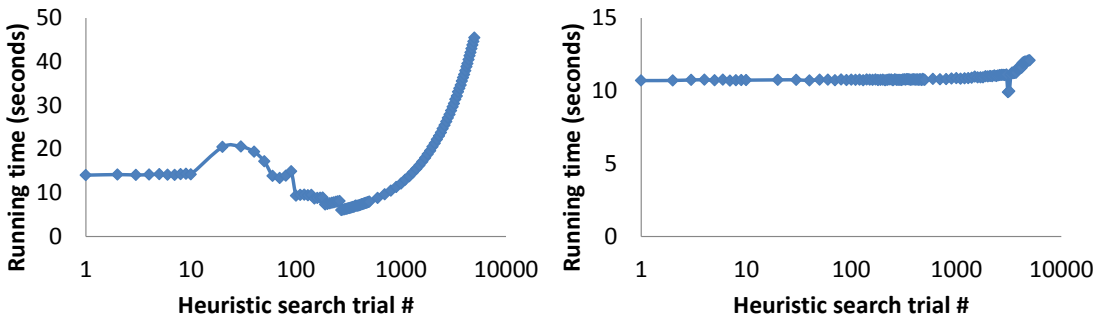


Figure 3.5: Running times of FTVI with different number of initial search trials on (left) a Wet-floor problem and (right) a Drive problem. Too few trials are sometimes less helpful for eliminating enough sup-optimal actions, and too many trials are harmful.

An interesting case occurs when the optimal value is found during the search step. Although FTVI performs a limited number of search iterations, it is possible that a problem is optimally solved within the search step. It is helpful to keep track of optimality information during the search step, so that FTVI can potentially skip some unnecessary search iterations and the entire computation step. To do this, we only need to maintain a Bellman error of the current search iteration, and terminate FTVI if the error is smaller than the threshold (Lines 11–12). In our experiment, we find this simple optimization to be extremely helpful in promoting the performance of FTVI.

Sometimes there are cases where G_{SR} still contains large connected components. This can be caused by two reasons (1) An optimal policy indeed has large components, or (2) the connectivity caused by many suboptimal actions is not successfully eliminated by search. To try to further decompose these large components, we let FTVI perform additional *intra-component heuristic searches*. An intra-component heuristic search is a search that takes place only inside a particular component. Its purpose is to find new, sub-optimal actions, which might help decompose the component. Given a component $\mathcal{C}$ of G_{SR}, we define $Source_{\mathcal{C}}$ to be the set of states where none of its incoming transitions are from states in $\mathcal{C}$. In other words, states in $Source_{C}$ are the incoming bridge states between $\mathcal{C}$ and rest of the MDP. An intra-component heuristic search of $\mathcal{C}$ originates from a state in $Source_{\mathcal{C}}$. A

search branch terminates when a state outside $\mathcal{C}$ is encountered.

We did some experiments and compared the performance of FTVI with and without additional intra-component search on problems from four domains, namely Wet-floor [20], Single-arm pendulum [133], Drive, and Elevator [67]. Our results show that additional intra-component search only provided limited gains in Wet-floor problems, in which it helped decrease the size of the largest components by approximately 50% on average, and sped up the convergence by 10% at best. However, intra-component search turned out to be harmful for the other domains, as it did not provide any new graphical information (no smaller components were generated). On the contrary, the search itself introduced a lot of unnecessary overhead. So we used the version that does not perform additional intra-component search throughout the rest of the experiments.

3.3.3 Experiments

We address the following two questions in our experiments: (1) How does FTVI compare with other algorithms on a broad range of domain problems? (2) What are the specific kind of domains on which FTVI should be preferred over heuristic search?

We implemented FTVI on the same framework as in Section 3.2.3, and used the same cut-off time of 5 minutes for each algorithm per problem. To investigate the helpfulness of action elimination, we also implemented a VI variant that applies action elimination in backups. We used the same threshold value $\delta = 10^{-6}$, and ran BRTDP and BaRTDP on the same upper bound as FTVI.

Relative Speed of FTVI

We evaluated the various algorithms on problems from eight domains — Mountain Car, Single and Double Arm Pendulum [133], Wet-floor [20][5], and four domains from International Planning Competition 2006 — Drive, Elevators, TireWorld and Blocksworld. A mountain car problem usually has many source states.[6] We chose each source state as an initial state,

[5]Note that we used the probability of wet cells, $p = 0.5$.

[6]A source state is a state with no incoming transitions.

Problem	VI	VI (w/ a.e.)	ILAO*	LRTDP	BRTDP	BaRTDP	TVI	FTVI
MCar100	1.40	0.74	1.91	1.23	2.81	63.55 (*)	0.68	**0.22**
MCar300	26.12	13.40	11.91	229.70	117.23	180.64 (*)	23.22	**2.35**
MCar700	278.16	124.34	101.65	-	216.01	262.92 (*)	233.98	**13.06**
SAP100	2.30	1.06	1.81	2.58	9.39	111.59 (*)	2.37	**0.17**
SAP300	42.61	19.90	32.40	-	-	-	44.2	**2.96**
SAP500	174.71	77.99	131.17	-	-	-	-	**9.56**
WF200	19.95	13.71	11.22	-	22.08	1.99 (*)	20.58	**8.81**
WF400	105.79	98.97	**73.88**	-	97.73	103.87 (*)	100.78	74.24
DAP10	0.77	0.67	1.01	51.45	3.04	222.33 (*)	0.75	**0.59**
DAP20	21.41	17.62	32.68	-	144.12	-	21.95	**17.49**
Drive	2.00	1.39	1.60	**0.69**	7.85	4.17 (*)	1.23	1.07
Drive	20.58	14.20	96.09	273.37	163.91	4.17 (*)	13.03	**10.63**
Drive	-	-	-	-	-	3.94 (*)	74.70	**41.93**
Elevator (IPPC p13)	-	-	227.53	-	-	-	58.46	**54.11**
Elevator (IPPC p15)	236.91	133.80	27.35	-	-	-	14.59	**12.11**
Tireworld (IPPC p5)	33.88	16.46	**0.00**	0.14	0.01	0.03	2.26	**0.00**
Tireworld (IPPC p6)	47.88	23.04	**0.00**	0.16	0.01	0.04	48.81	**0.00**
Blocksworld (IPPC p4)	17.69	17.69	**0.02**	0.26	1.93	-	54.35	**0.02**
Blocksworld (IPPC p5)	14.19	14.19	**0.00**	0.11	0.66	-	54.34	**0.00**

Table 3.1: Total running times of the different algorithms on problems in various domains. FTVI outperforms all algorithms by vast margins. (Fastest times are bolded. '-' in Time means that the algorithm failed to solve the problem within 5 minutes. The *'s mean the algorithm terminated with sub-optimal solutions.)

and averaged the statistics per problem. Table 3.1 lists the running times for the various algorithms. For FTVI, we additionally report (in Table 3.2) the time used by the searches (T_{search}), and the time spent in generating the graphical structure (T_{gen}), if a problem is not solved during the search phase, where the leftover is the time spent in solving the SCCs. We also compared the size of the biggest component (BC size) generated by TVI and FTVI.

Overall we find that FTVI outperforms the other five algorithms on most of these domains. FTVI outperforms TVI in all domains. Notice that on the MCar problems, FTVI establishes very favorable graphical structures (strongly connected components of size one)

Problem	Reachable $\|\mathcal{S}\|$	TVI		FTVI			
		BC size	$Time$	BC size	T_{search}	T_{gen}	$Time$
MCar100	10,000	7,799	0.68	1	0.20	0.01	**0.22**
MCar300	90,000	71,751	23.22	1	2.22	0.13	**2.35**
MCar700	490,000	390,191	233.98	1	12.29	0.76	**13.06**
SAP100	10,000	9,999	2.37	n/a	0.17	n/a	**0.17**
SAP300	90,000	89,999	44.2	n/a	2.96	n/a	**2.96**
SAP500	250,000	-	-	n/a	9.56	n/a	**9.56**
WF200	40,000	39,999	20.58	15,039	3.30	0.12	**8.81**
WF400	160,000	159,999	100.78	141,671	14.27	0.36	74.24
DAP10	10,000	9,454	0.75	n/a	0.59	n/a	**0.59**
DAP20	160,000	150,489	21.95	n/a	17.49	n/a	**17.49**
Drive	4,563	4,560	1.23	4,560	0.11	0.02	1.07
Drive	29,403	29,400	13.03	29,400	0.15	0.15	**10.63**
Drive	75,840	75,840	74.70	75,840	0.18	0.40	**41.93**
Elevator (IPPC p13)	539,136	1,053	58.46	1,053	0.01	1.73	**54.11**
Elevator (IPPC p15)	539,136	1,053	14.59	1,053	0.01	1.60	**12.11**
Tireworld (IPPC p5)	671,687	23	2.26	n/a	0.00	n/a	**0.00**
Tireworld (IPPC p6)	724,933	618,448	48.81	n/a	0.00	n/a	**0.00**
Blocksworld (IPPC p4)	103,121	103,104	54.35	n/a	0.02	n/a	**0.02**
Blocksworld (IPPC p5)	103,121	103,204	54.34	n/a	0.00	n/a	**0.00**

Table 3.2: Detailed performance statistics for TVI and FTVI. (BC size means the size of the biggest connected component. 'n/a' means FTVI converged in the search step and skipped the computation step. All running times are in seconds. Fastest times are bolded. '-' in Time means that the algorithm failed to solve the problem within 5 minutes.)

during the search step.[7] This graphical structure makes the second step of FTVI trivial. But TVI has to solve much bigger components, so it runs much slower. For the Drive domain, even if it does not find a more informed graphical structure, the advanced backup with action elimination enables FTVI converge faster. FTVI outperforms heuristic search algorithms most significantly in domains such as MCar, SAP and Drive. It is faster than ILAO*

[7] If we allow FTVI to perform the computation step as opposed to stop at the search step when a problem is solved, it will find similar structures in the Tireworld and Blocksworld problems.

by an order of magnitude. This shows the extreme effectiveness of FTVI's decomposing a problem into small sub-problems using advanced graphical information and solving these sub-problems sequentially. The three RTDP algorithms are not competitive with the other algorithms in these domains, and fail to return a solution by the cutoff time for many problems. FTVI shows limited speedup against heuristic search in domains such as Wet-floor, DAP, and Elevator. FTVI is on par with ILAO*, and vastly outperforms TVI in Tireworld and Blocksworld domains, as it converges within the search step. The convergence speed of value iteration is typically slow, as it backs up states iteratively by a fixed order. Adding action elimination to Bellman backups increases the convergence speed of VI up to two times, especially in the Mountain Car, Single Arm Pendulum, and Elevator domains, but its convergence speed is usually at least one magnitude slower than those of FTVI.

Factors Determining Performance

We have shown that FTVI is faster than heuristic search in many domains, but its relative speedup is domain-dependent. Can we find any domain features that are particularly beneficial for FTVI or worse for heuristic search? In this evaluation we performed control experiments by varying the domains across different features and study the effect on planning time of various algorithms.

We make an initial prediction of three features.

1. The number of goals in the domain: If the number of goal states is small, search may take a long time before it discovers a path to a goal. Therefore, many sub-optimal policies might be evaluated by a heuristic search algorithm.

2. Search depth from the initial state to a goal state: This depth is a lower bound of the length of an execution trial and also of the *size* of any policy graph. A greater depth implies more search steps per iteration, which might make evaluating a policy time-consuming.

3. Heuristic informativeness: The performance of a heuristic search algorithm depends a lot on the quality of the initial heuristic function. We expect the win from FTVI to

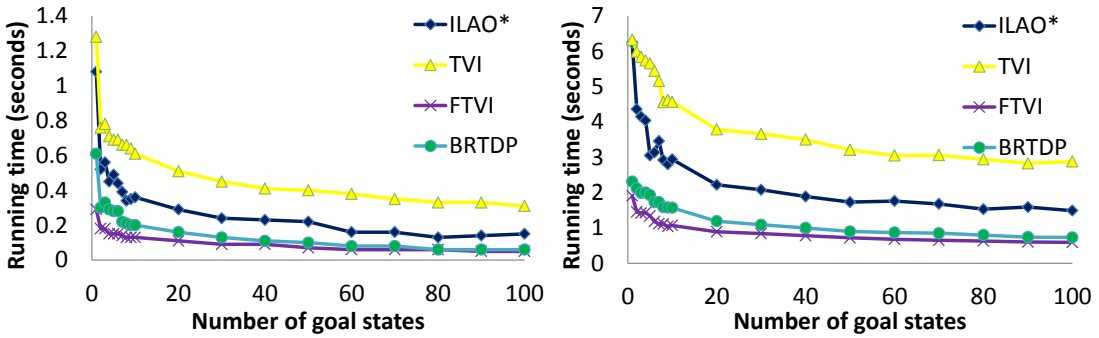


Figure 3.6: Running times of algorithms with different number of goal states and problem size (left) $|\mathcal{S}| = 10,000$ (right) $|\mathcal{S}| = 50,000$ in random MDPs. FTVI and TVI slow down the least significantly when the number of goal states is small.

increase when heuristic is less informed.

The Number of Goals. As far as we know, there is no suitable domain where we can specify the total number of goal states arbitrarily, so we used an artificial domain. In this domain each state has two applicable actions, and each action has at most two random successors. We tested all algorithms on domains of two sizes, 10,000 (Figure 3.6(left)) and 50,000 (Figure 3.6(right)). For each problem size, we fixed the shortest goal distance but varied the number of goal states $|\mathcal{G}|$. More concretely, after generating the state transitions, we performed a BFS from the initial state, and randomly picked goal states on a same search depth. For each $|\mathcal{G}|$ value, we generated 10 problems, and reported the median running time of four algorithms (LRTDP and BaRTDP were slow in this domain). We observe that all algorithms take more time to solve a problem with a smaller number of goal states than with a larger number. However, beyond a point ($|\mathcal{G}| > 20$ in our experiments), the running times become stable. FTVI runs only marginally slower when $|\mathcal{G}|$ is small, suggesting that its performance is less dependent on the number of goal states. BRTDP is the second best in handling small goal sets, and it runs nearly as fast as FTVI when the goal set is large. Even though TVI runs the slowest among the four algorithms, its performance shows less severe dependence on the number of goal states. It runs almost as fast as ILAO* when the goal set size is 1. In contrast, ILAO* runs twice as fast as TVI when the goal set size is

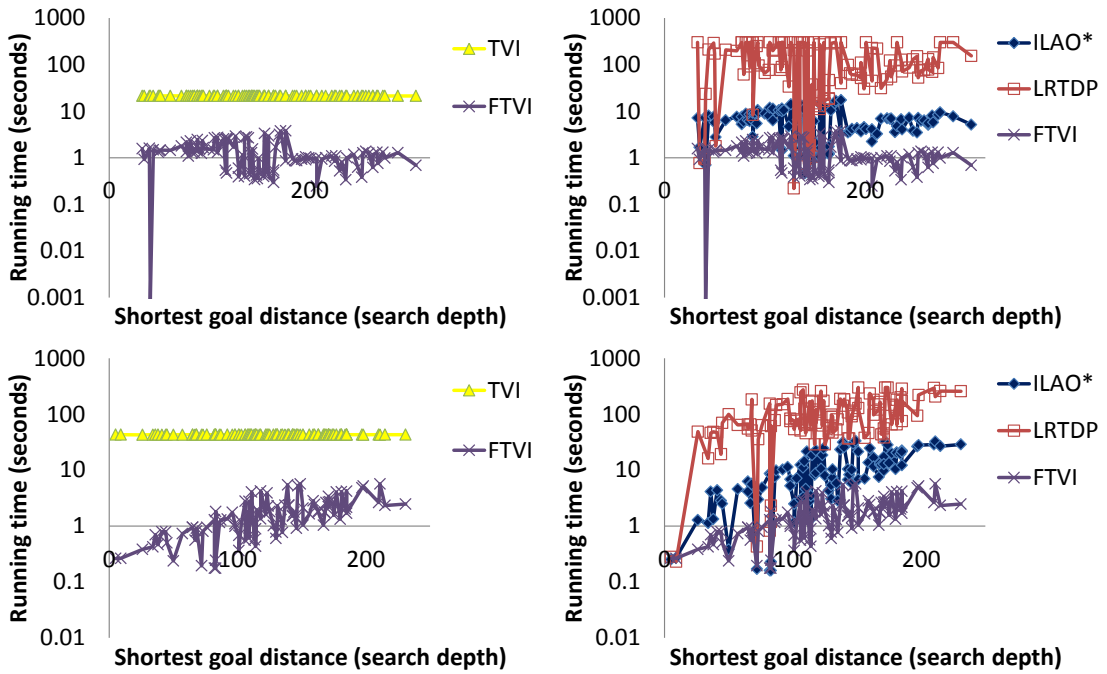


Figure 3.7: Running times of algorithms with different shortest distance to the goal on (top) mountain car 300 × 300 (MCar300), (bottom) single-arm pendulum 300 × 300 (SAP300) problems, (left) comparison of FTVI and TVI, and (right) comparison of FTVI and heuristic search algorithms. Heuristic search algorithms slow down massively (note the log scale) when the search depth is large.

greater than 20.

Search Depth. In this experiment, we studied how the search depth of a goal from the initial state influences the performance of various algorithms. We chose a Mountain car problem and a Single-arm pendulum problem. We randomly picked 100 initial states from the state space[8] and measured the shallowest search depth, or, the shortest distance to a goal state, d. The running times in Figure 3.7 are ordered by d. BaRTDP does not terminate with an optimal policy for many instances, so its performance is not shown. BRTDP has the biggest variance so its performance is not included for clarity purposes.

As we can see, FTVI is the fastest algorithm in this suite of experiments. It converges

[8]Note that these problems have well-defined initial states. Here we picked initial states arbitrarily from $\mathcal{S}$.

very quickly for all initial states (usually around one or two seconds on Mcar300, and less than 10 seconds on SAP300). TVI's performance is unaffected by the search depth, which is expected, since it is a variant of value iteration and has no search component. In the MCar300 problem, we do not find strong evidence that the running time of any algorithm depends on the search depth. FTVI runs an order of magnitude faster than TVI, ILAO*, and BRTDP and two orders of magnitude faster than LRTDP. In the SAP300 problems, the running times of all algorithms except TVI increase as search depth increases. LRTDP runs fast when d is relatively small, but it slows down considerably and is unable to solve many problems when d becomes larger. ILAO*'s convergence speed varies a bit when the distance is small. As d increases, its running time also increases. BRTDP's performance (not included) is close to that of ILAO* when d is small, but becomes slower and performs similar to LRTDP when d is large. In this problem, heuristic search algorithms unanimously suffer significantly from the increase in the search depth, as their running times increase by at least two orders of magnitude from small to large d values. On the other hand, FTVI slows down by only one order of magnitude, which makes it converge one order of magnitude faster than ILAO*, one to two orders of magnitude faster than BRTDP and TVI, and two orders of magnitude faster than LRTDP for large depths.

Heuristic Quality. Finally we studied the effect of the heuristic informativeness on the algorithms. We conducted two sets of experiments, based on two sets of consistent heuristics. We found BRTDP slower than other algorithms in all problems and BaRTDP to be comparable (about 50% slower than LRTDP) only on the Wet100 problem, so did not include their running times. In the first experiment, we pre-computed the optimal value function of a problem using value iteration, and used a fraction of the optimal value as an initial heuristic. Given a fraction $f \in (0, 1]$, we calculated $h(s) = f \times V^*(s)$. Figure 3.8 plots the running times of different algorithms against f for three problems. Note that $f = 1$ means the initial heuristic is already optimal, so a problem is trivial for all algorithms, but TVI has the overhead of building a topological structure. FTVI, however, is able to detect convergence in the search step and circumvent this overhead, so it is fast. LRTDP is slow in the Wet100 problem, so its running times in the Wet100 problem are omitted from the figure. The figure shows that as f increases (i.e. as the heuristic becomes more informative)

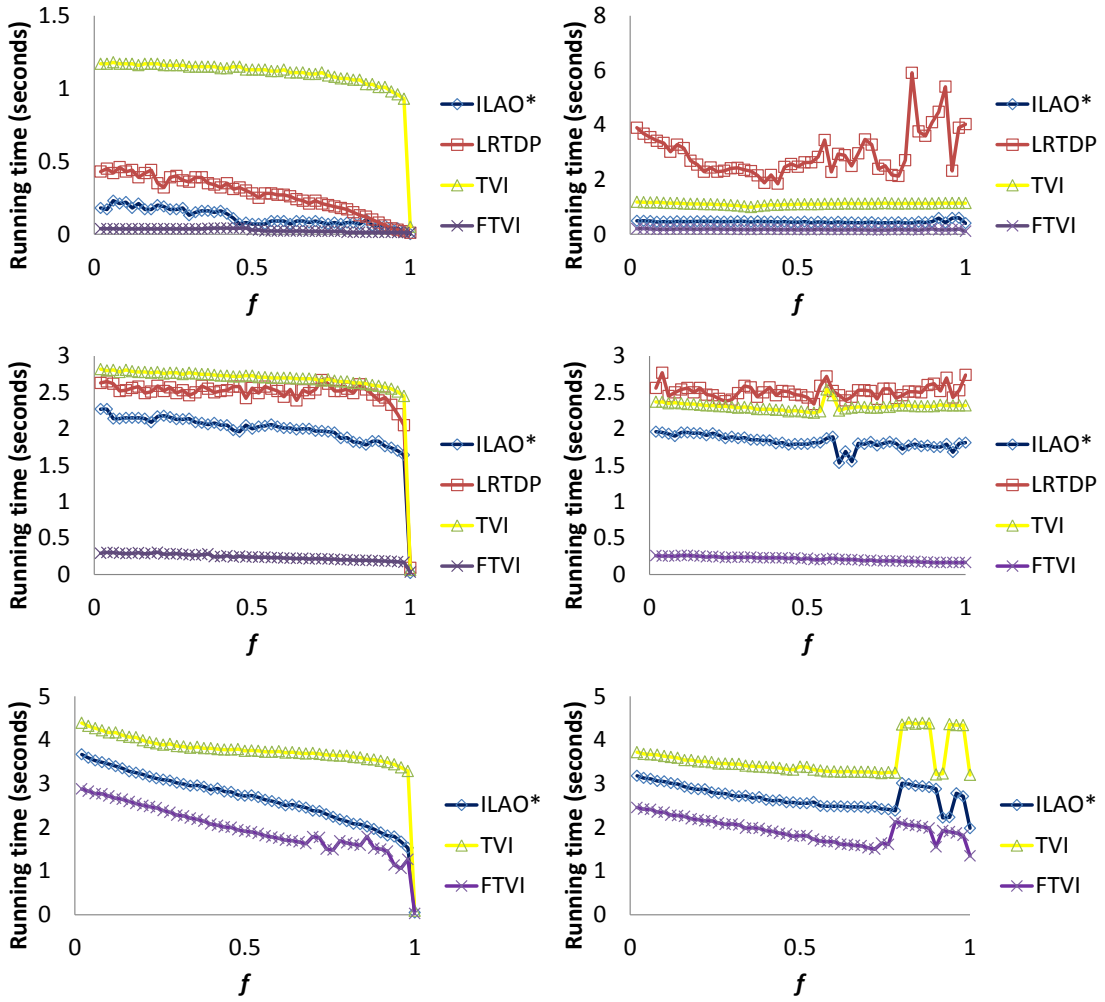


Figure 3.8: Running times of algorithms with different initial heuristic on (top) mountain car 100×100 (MCar100), (middle) single-arm pendulum 100×100 (SAP100), and (bottom) wet-floor 100×100 (WF100) problems. All algorithms are equally sensitive to the heuristic informativeness. (left) $f = \sum_{bfs \in \mathcal{S}} h(\mathbf{s}) / \sum_{\mathbf{s} \in \mathcal{S}} V^*(\mathbf{s})$ (right) $f = \sum_{\mathbf{s} \in \mathcal{S}} h(\mathbf{s}) / \sum_{\mathbf{s} \in \mathcal{S}} V_l(\mathbf{s})$.

the running times of all algorithms decrease almost linearly. This is true even for TVI, which is not a heuristic-guided algorithm, but takes less time, probably because the initial values affect the number of iterations required until convergence.

To thoroughly study the influence of the heuristics, we conducted a second set of experiments. In this experiment, we used a fractional V_l value as our initial heuristic. Recall that V_l is a lower bound of V^* computed by the value of a deterministic problem. We calculated the initial heuristic by $h(\mathbf{s}) = f \times V_l(\mathbf{s})$. All included algorithms show a similar smooth decrease in running time when f increases. BRTDP, however, shows strong dependence on the heuristics in the Wet100 problem. Its running time decreases sharply from 96.91 seconds to 0.54 seconds and from 99.81 seconds to 6.21 seconds from when $f = 0.02$ to when $f = 1$ in the two experiments. Stable changes in the two experiments suggests the following for algorithms except BRTDP. (1) No algorithm is particularly vulnerable to a less informed heuristic function; (2) extremely informative heuristics (when f is very close to 1) do not necessarily lead to extra fast convergence. This result is in-line with recent results in [60] for deterministic domains.

Discussion

From the experiments, we learn that FTVI is vastly better in domains whose problems have a small number of goal states and a long search depth from the initial state to a goal (such as MCar, SAP and Drive). But the convergence control module of FTVI helps in successfully matching the performance of FTVI with the fastest heuristic search algorithm. In addition, FTVI displays limited advantage over heuristic search in the two intermediate cases where a problem has (1) many goal states but long search depth (Elevator), or (2) a short depth but fewer goal states (DAP). In conclusion, FTVI is our algorithm of choice whenever a problem has either a small number of goal states or a long search depth.

3.4 Summary

We propose an optimal algorithm for solving MDPs, *focused topological value iteration* (FTVI), which first performs heuristic search trials in order to eliminates provably sub-optimal actions and later solves the remaining problem with the topological value itera-

tion algorithm. We evaluate FTVI across several benchmark domains and find that FTVI outperforms TVI, by significant margins. Surprisingly, FTVI also outperforms other state-of-the-art heuristic search algorithms in most of the domains. This is unexpected, since common wisdom dictates that heuristic-guided search is much faster than all-state dynamic programming. We discover two important features of problems that are hard for heuristic search algorithms: smaller number of goal states and long search distance to the goal. These features are commonly found in many domains. We show that, in such domains, FTVI outperforms heuristic search in convergence speed by an order of magnitude on average, and sometimes by even two orders of magnitude.

Chapter 4

THE OPTIMAL PLANNING ON LARGE PROBLEMS

Previous section discusses new algorithms on speeding up the convergence of optimal planning by using graphical information in dynamic programming.

Unfortunately, the common algorithms for solving MDPs, including (F)TVI, are based on dynamic programming and thus require amount of memory that is polynomial in the number of states, $|\mathcal{S}|$, *i.e.*, exponential in the number of domain features. This prohibitive use of memory is a major bottleneck in scaling MDP algorithms to real-world problems. Even relatively compact symbolic representations, like algebraic decision diagrams, tend to grow large in size, exhausting memory on moderate sized MDPs [62]. This chapter reviews an orthogonal effort in scaling up optimal probabilistic planning: solving large MDPs using external memory.

4.1 Previous Work

To solve the scalability problem, Edelkamp et al. proposed a variant of value iteration called external memory value iteration (EMVI) [48]. It is the first MDP algorithm that makes use of disk space to store MDP models. When backups are performed, relevant portions of the model are moved from the disk to the main memory. Note from Equation 2.2 that to back up a state $\mathbf{s}$ the transition function of $\mathbf{s}$ and the values of all its successors need to be in memory. If one performs backups in an arbitrary order, one would thrash with inefficient transfers to and from disk. To overcome this problem, Edelcamp et al. used a clever idea: before every iteration, the algorithm sorts the transition functions $T_{\mathbf{a}}(\mathbf{s}'|\mathbf{s})$ by the successor state $\mathbf{s}'$, and attaches the current value function $V(\mathbf{s}')$ to the corresponding edge. To do so, EMVI builds a directed graph $G = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V} = \mathcal{S}$ and an edge $e = (\mathbf{s}, \mathbf{a}, \mathbf{s}') \in \mathcal{E}$ exists if there is a probabilistic transition between $\mathbf{s}$ and $\mathbf{s}'$ along action $\mathbf{a}$ ($T_{\mathbf{a}}(\mathbf{s}'|\mathbf{s}) > 0$). It further attaches the value of the head state, $V(\mathbf{s}')$, as well as the transition probability,

Algorithm 5 External Memory Value Iteration

1: **Input:** an MDP $M = \langle \mathcal{S}, \mathcal{A}, Ap, T, C \rangle$, δ: the threshold value
2: **for** each state $\mathbf{s} \in \mathcal{S}$ **do**
3: **for** each action $\mathbf{a} \in Ap(\mathbf{s})$ **do**
4: initialize $Q(\mathbf{s}, \mathbf{a})$ arbitrarily
5: $V(\mathbf{s}) \leftarrow min_{\mathbf{a} \in Ap(\mathbf{s})} Q(\mathbf{s}, \mathbf{a})$
6: attach $V(\mathbf{s}')$ and $T_a(\mathbf{s}'|\mathbf{s})$ to each edge $\mathcal{E} = \{e | e = (\mathbf{s}, \mathbf{a}, \mathbf{s}')\}$ and sort $\mathcal{E}$ by the head state $\mathbf{s}'$
7: **while** true **do**
8: $Bellman\_error \leftarrow 0$
9: duplicate $\mathcal{E}$ to $\mathcal{E}'$
10: sort $\mathcal{E}'$ by the tail state on external memory
11: **for** each state $\mathbf{s} \in \mathcal{E}$ (by scanning $\mathcal{E}$) **do**
12: **for** each state $\mathbf{s}' \in \mathcal{E}'$ (by scanning $\mathcal{E}'$) **do**
13: $V(\mathbf{s}) \leftarrow min_{\mathbf{a} \in Ap(\mathbf{s})} \left[C(\mathbf{s}, \mathbf{a}) + \sum_{\mathbf{s}' \in \mathcal{S}} T_{\mathbf{a}}(\mathbf{s}'|\mathbf{s}) V(\mathbf{s}') \right]$
14: $Bellman\_residual(\mathbf{s}) \leftarrow |V(\mathbf{s}) - oldV|$
15: $Bellman\_error \leftarrow max(Bellman\_error, Bellman\_residual(\mathbf{s}))$
16: **if** $Bellman\_error < \delta$ **then**
17: **return** V

$T_{\mathbf{a}}(\mathbf{s}'|\mathbf{s})$, to each edge and keep an external-memory hash table of $\mathcal{E}$. At the beginning of each iteration, EMVI duplicates the hash table $\mathcal{E}$ by creating $\mathcal{E}'$ and re-sorts the edges in $\mathcal{E}'$ based on the tail state. In performing one backup iteration, EMVI scans the two hash tables in parallel, matching the head state in $\mathcal{E}$ with the tail state in $\mathcal{E}'$ for an multiplication operation (namely, $T_{\mathbf{a}}(\mathbf{s}'|\mathbf{s})V(\mathbf{s}')$). When all the successor states of an action $\mathbf{a}$ in $\mathcal{E}'$ have been completely scanned, the Q-value of that transition is calculated. The minimization operation is performed when all actions of $\mathbf{s}$ (edges with tail state $\mathbf{s}$ in $\mathcal{E}$) has been scanned. As states in both hash tables are properly sorted, the two hash tables only need to be scanned once for one backup iteration.

4.2 Partitioned External-Memory Value Iteration

Similar to EMVI we propose to overcome the memory bottleneck in value iteration using additional storage in the external memory. However, in contrast to the previous approach, we exploit the benefits of partitioning the state space. Partitioning of an MDP has been studied previously [133], but not in the context of external-memory algorithm. Our algorithm, partitioned external-memory value iteration (PEMVI) [36], demonstrates its applicability in solving large MDPs much faster.

PEMVI runs in two phases: (1) partition generation, and (2) asynchronous dynamic programming that partitions the state space into blocks and then performs backups on a block by block basis by successively loading all the information necessary to back up each block in the internal memory.

In this section we focus our discussion on the second phase. We will elaborate partition generation more in later sections.

4.2.1 Definitions

We make a few definitions before explaining PEMVI.

Definition 14 *A* partition *of an MDP is the division of the state space, $\mathcal{S}$, into disjoint* partition blocks $\mathcal{P} = \{p_0, \ldots, p_n\}$ *such that* $\mathcal{S} = \cup_i p_i$.

Each block in the partition contains at least one state and each state belongs to exactly one partition block[1]. The function $part(s) : \mathcal{S} \rightarrow \mathcal{P}$, a surjection from the state space to the partition space, specifies which partition block a given state $\mathbf{s}$ is in.

Definition 15 *A* partition block p' *is a successor block of another partition block* p *if there exist two states* $\mathbf{s}$ *and* $\mathbf{s}'$ *s.t.* $part(\mathbf{s}) = p$, $part(\mathbf{s}') = p'$ *and* s' *is a successor of* s. *Symmetrically,* p' *is the* predecessor block *of* p. *The set of all successor blocks of a partition block* p *is denoted* $succpart(p)$. *Similarly, the set of all predecessor blocks of a partition block* p *is denoted* $predpart(p)$.

[1]Note that our algorithm retains its properties even if a state belongs to multiple partition blocks.

For each partition block p, we are interested in two components: a *transition component* and a *value component*. We are interested in partitions which are refined enough that every block in $\mathcal{P}$ can be backed up in the main memory. Let M denote the size of main memory.

Definition 16 *The transition component of a partition p contains the transition functions and the cost functions of all states $\mathbf{s}$ in the partition block, $T_p = \{T_{\mathbf{a}}(\cdot|\mathbf{s}), C(\mathbf{s},\mathbf{a})|\mathbf{s} \in p\ , \mathbf{a} \in Ap(\mathbf{s})\}$. The value component contains the current value function $V_p = \{V(\mathbf{s})|\mathbf{s} \in p\}$ of all its states.*

Definition 17 *A partition block, $p \in \mathcal{P}$, is* valid *if the its transition component plus the value components of its successor blocks fit in the main memory:*

$$|T_p| + \sum_{q \in succpart(p)} |V_q| \leq M \tag{4.1}$$

Additionally, a partition $\mathcal{P}$ is valid *if $\forall p \in \mathcal{P}$, p is valid.*

4.2.2 External Asynchronous Dynamic Programming

Asynchronous dynamic programming is a version of dynamic programming in which the value updates occur in an arbitrary order instead of a systematic pass through the state space. Value iteration through asynchronous dynamic programming preserves the convergence properties under the conditions that all states are eventually backed up enough times.

PEMVI can be seen as a version of asynchronous value iteration in which the backups occur in an order dictated by the partition blocks. The use of external memory comes about in storing the information relevant to each partition block.

Given a valid partition, PEMVI first computes an ordering of partition blocks. In each iteration PEMVI loads the transition components of the partition block p, one at a time in the order. It additionally loads the value components of all the successor blocks, $succpart(p)$. Thus after this I/O operation the internal memory contains all the information necessary to perform a Bellman backup for all states in p. PEMVI computes one (or more) backups for this partition block. It then updates the value component of p in the external memory and moves on to the next partition block in the order.

In each iteration we track the maximum Bellman error, e, of the system. When e is within our threshold limit (δ) we know that the algorithm has converged and we can terminate. Algorithm 6 presents the pseudo-code for PEMVI.

Because our algorithm is a special case of asynchronous value iteration it has properties similar to value iteration.

Theorem 18 PEMVI *terminates in finite time and converges to a δ-optimal value function as long as the optimal value function is finite.*

Recall that the number of states is $|\mathcal{S}|$ and the total number of state transitions is $|T|$. Let $max_p = \max_{p \in P} |predpart(p)|$. We have the following theorem regarding the I/O complexity of PEMVI:

Theorem 19 *Partitioned External Value Iteration performs at most $O(scan(|T|) + max_p \times scan(|\mathcal{S}|))$ I/Os per iteration.*

Proof In each iteration, PEMVI loads the transition component of each partition block at most once, which requires $O(scan|T|)$ I/Os. To back up a partition block, PEMVI needs to load all its successor blocks. Thus the value component of a particular partition block p, is loaded up to $|predpart(p)|$ times. This incurs a total of $\sum_{p \in \mathcal{P}} |predpart(p)| \times scan(|V_p|)$ I/Os, which is upper-bounded by $max_p \times scan(|\mathcal{S}|)$.

$|T|$ is usually significantly larger than $|\mathcal{S}|$, as it is a function of three-dimensional domain $\mathcal{S} \times \mathcal{A} \times \mathcal{S}$. Moreover, for structured problems, such as ours, the number of successors and predecessors is small, hence an effective partitioning can keep max_p down. Thus, the dominating I/O cost of PEMVI is in the scan of $|T|$. EMVI, on the other hand, requires an external sort of $|T|$ at every iteration. Therefore, PEMVI improves the I/O complexity of EMVI by a logarithmic factor per iteration.

4.2.3 Single v.s. Multiple Backups

Our initial implementation of PEMVI computed a single backup per state in each iteration. However, early experiments confirmed that PEMVI spent most of its time on I/O

Algorithm 6 Partitioned External-Memory Value Iteration

1: **Input:** $\mathcal{S}$, $\mathcal{A}$, λ, δ
2: partition $\mathcal{S}$ into $\mathcal{P}$
3: generate a partition order for $\mathcal{P}$
4: **repeat**
5: **for** every partition block $p \in \mathcal{P}$ in the order **do**
6: load the transition component of p
7: load the value components of every $p' \in succpart(p)$
8: **repeat**
9: $num\_runs \leftarrow 0$
10: **for** every state $\mathbf{s} \in p$ **do**
11: $oldV \leftarrow V(\mathbf{s})$
12: $V(\mathbf{s}) \leftarrow min_{\mathbf{a} \in Ap(\mathbf{s})} \left[C(\mathbf{s}, \mathbf{a}) + \sum_{\mathbf{s}' \in \mathcal{S}} T_{\mathbf{a}}(\mathbf{s}'|\mathbf{s}) V(\mathbf{s}') \right]$
13: $Bellman\_residual(\mathbf{s}) \leftarrow |V(\mathbf{s}) - oldV|$
14: $num\_runs \leftarrow num\_runs + 1$
15: $Bellman\_error(p) \leftarrow \max_{\mathbf{s} \in p}(Bellman\_residual(\mathbf{s}))$
16: **until** $num\_runs > \lambda$ or $Bellman\_error(p) < \delta$
17: write back the value component of p
18: release used memory
19: **until** $(\max_{p \in \mathcal{P}} Bellman\_error(p) < \delta)$

operations. As a natural extension we started performing multiple backups per iteration — thus increasing message passing, improving amortization of I/O, and speeding convergence in most cases. Note that external memory value iteration is unable to perform multiple backups without I/O, since it has to sort the complete, tabular transition function before it is ready to perform each iteration of dynamic programming.

Our mechanism of doing multiple backups is the following: We set an upper bound, λ, on the number of backup runs. In each run, we back up every state in the partition block, and measure the Bellman error. We finish backing up a partition block either when the

maximum Bellman error is sufficiently small, *i.e.*, the states of the block have converged given the fixed values of their successors, or the maximum number of runs are completed. Line 16 of Algorithm 6 describes this in the pseudo-code.

4.2.4 Backup Order

In each iteration of PEMVI every partition block is backed up once. For some problems, the order of backing up partition blocks is important, since a good order might make the algorithm converge faster, such as applying the strategies used by the (F)TVI algorithms (see the previous chapter for more details). We explore different heuristics for the backup order.

Our first heuristic (Algorithm 7) attempts to maximize information flow by choosing a back-up order which processes states which are close to the goal before those which are far. We call the partition block which contains the goal state the *goal block*. We compute a "best flow" order that places the partition blocks in the increasing order of their distance from the goal block. We name the reverse of this order as the "worst flow" order. A random order chooses the partition order randomly. For example, consider the partitioning scheme illustrated in Figure 4.2. If the goal state is in the partition block p_0 then a "best flow" order could be p_0, p_1, p_4, p_2, p_5, p_8, p_3, p_6, p_9, p_{12}, p_7, p_{10}, p_{13}, p_{11}, p_{14}, p_{15}. We conjecture the performance of the random order to be between the "best flow" and the "worst flow" order.

Our second heuristic for computing the backup order attempts to minimize the total disk I/O. The next partition block in the order is chosen greedily to be the one that minimizes the I/O at the current step. We start from the goal block and then simulate the I/O operations and the memory contents. Whenever to choose the next partition block, we look at all the value components and transition components already in memory. We choose the next partition to be the one that requires minimum I/O. In effect, we will favor partition blocks that have high overlap with the part of the MDP model already in memory, in essence minimizing the I/O required in the current step. Algorithm 8 describes the details of this computation.

To get the benefits of the I/O efficient order we modify line 18 of Algorithm 6 by not

Algorithm 7 Backup Order: Maximize Information Flow

1: **Input:** p_g (goal block)
2: $O \leftarrow \langle p_g \rangle$
3: $L \leftarrow \{p | (p \in O) \wedge (predpart(p) \not\subseteq O)\}$
4: **while** $L \neq \emptyset$ **do**
5: $\quad p \leftarrow$ first block of L
6: $\quad N \leftarrow$ partition blocks in $predpart(p)$ not in O
7: $\quad$ append each $p' \in N$ at the end of O
8: $\quad L \leftarrow \{p | (p \in O) \wedge (predpart(p) \not\subseteq O)\}$
9: **return** O as the backup order

releasing a value component when it is needed by the next partition block. Similarly, in line 7, we only load the value components that are not already in memory.

Also note that Algorithm 7 is able to detect the partition blocks that cannot reach the goal block, thus whose states must have infinite values, so filter them out of the dynamic programming phase. Algorithm 8 itself cannot detect these blocks, but a pre-scan can be enforced to make sure its input $\mathcal{P}$ only contains blocks that can reach the goal block.

Algorithm 8 Backup Order: Minimize I/O

1: **Input:** $\mathcal{P}$ (set of partition blocks), p_g (goal block)
2: $O \leftarrow \langle p_g \rangle$
3: $\mathcal{P} \leftarrow \mathcal{P} - \{p_g\}$
4: **while** $\mathcal{P} \neq \emptyset$ **do**
5: $\quad p \leftarrow$ the last element in O
6: $\quad L \leftarrow \cup_{p' \in succpart(p)} \{V_{p'}\}$
7: $\quad p^* \leftarrow min_{p' \in \mathcal{P}} \left(|T_{p'}| + | \cup_{p'' \in succpart(p')} \ V_{p''} - L| \right)$
8: $\quad \mathcal{P} \leftarrow \mathcal{P} - \{p^*\}$
9: $\quad$ append p^* at the end of O
10: **return** O as the backup order

4.3 A Theory of XOR Groups

Before discussing our automatic partition generation system, we provide a theory of *XOR groups*, the building blocks of generating a partition [37].

4.3.1 Previous Work: State Abstraction in Deterministic Domains

To create a high quality partition of the state space we build on the prior research in deterministic planning. Edelkamp and Helmert [47] study the problem of *encoding* — explicitly representing a state in terms of the state variables and their values. A planning problem is not immediately solvable before it is changed from its logic representation into an explicit state representation. The action that does this transition is called *encoding*. A trivial encoding is to use a bit array[2], where the value of a bit represents the truth value of its designated state variable. To obtain a compact encoding, Edelkamp and Helmert find a group of *balanced* predicates, where (1) exactly k of them are true in the initial state, and (2) the total number of true predicates cannot be increased by any action. In fact, the nomenclature of 'balanced' is inappropriate, since the number of true predicates can decrease. We name them *monotone* in this paper. In cases when $k = 1$, these groups of predicates can map to a single multi-valued fluent, and thus saving on the encoding length. These monotone predicates were used by Zhou and Hansen [136] as *XOR groups* to define state abstractions for structured duplicate detection in external-memory search algorithms. Zhou and Hansen also defined the *locality* heuristic as a criterion for choosing a suitable state abstraction. This work generalizes the notion of XOR groups, adapts them to probabilistic domains, and investigates additional partitioning heuristics, which are more relevant to probabilistic domains.

4.3.2 Our Generalization

While there is an exponential number of ways to partition the state space, we focus on structurally meaningful partitions, which can be represented compactly. We use the key notion of XOR groups developed by Zhou and Hansen, but, generalize it as follows: *A*

[2]We assume variables are Boolean. If a variable is multi-valued, then one should use a character array.

set of formulae forms an XOR group if exactly one formula in the set is true in any state reachable from the initial state. For example, for any formula φ, the set $\{\varphi, \neg\varphi\}$ represents an XOR group. For another example consider the Explosive-Blocksworld domain with two block constants, b_1 and b_2. The fact that either a block (say b_1) is *clear* or else it must have a block *on* it can be represented as an XOR group: $\{clear(b_1), on(b_1, b_1), on(b_2, b_1)\}$. Sometimes it is clearer to think of an XOR group as a logical formula. We can write this XOR group logically as $clear(b_1) \oplus on(b_1, b_1) \oplus on(b_2, b_1)$. It is easy to see that an XOR group compactly represents a partition of the state space, where each *partition block* contains all the states satisfying a single formula (exclusive disjunct) in the group.

Our definition of an XOR group is more general than the original one by Zhou and Hansen — they only allowed positive literals in an XOR group. On the other hand we accept any complex formula in our definition of XOR-groups.

In order to achieve a compact representation, it is convenient to specify XOR groups using first-order logic. This is facilitated by introducing some syntactic sugar, an *exclusive quantifier* $\exists_1$, to the logic. Intuitively, $\exists_1 x\ \varphi(x)$ means that there exists exactly one constant that satisfies $\varphi(x)$, that is, *exclusively quantified* is represented as $\exists_1 x$. With this notation, a large XOR group can be represented by a short logical formula. For example, if there were many blocks in the domain, then the XOR group shown previously would get quite long, but this notation allows a compact specification:

$$\exists_1 b\ [clear(b_1) \oplus on(b, b_1)]. \tag{4.2}$$

Note that this group is specific to the block, b_1, but there is a similar group for b_2: $\{clear(b_2), on(b_1, b_2), on(b_2, b_2)\}$. Similarly, it can be compactly represented by

$$\exists_1 b\ [clear(b_2) \oplus on(b, b_2)]. \tag{4.3}$$

These two XOR groups (Equations 4.2 and 4.3) can be represented, by adding a new, universally quantified, variable b':

$$\forall b' \exists_1 b\ [clear(b') \oplus on(b, b')].$$

If we ground this formula (assuming a universe of two blocks), we get a formula, whose

conjuncts corresponds to an XOR group each.

$$(clear(b_1) \oplus on(b_1, b_1) \oplus on(b_2, b_1)) \quad \wedge \quad (clear(b_2) \oplus on(b_1, b_2) \oplus on(b_2, b_2))$$

Definition 20 *A formulae $\mathcal{F}$ is* balanced under *an action* **a** *if in each of its probabilistic outcomes,* **a***'s effects either (i) do not change the truth value of any formula in $\mathcal{F}$ or (ii) make exactly one of the formulae in $\mathcal{F}$ true, while making another (previously true) formula in $\mathcal{F}$ false.*

Definition 21 Necessary and Sufficient Conditions*: A set of grounded formulae $\mathcal{F}$ forms a XOR group if and only if $\mathcal{F}$ satisfies the following two conditions:*

1. *Exactly one $\varphi \in \mathcal{F}$ is satisfied in the initial state.*

2. *For every ground action* **a** *that is executable in the reachable part of the state space, $\mathcal{F}$ is balanced under* **a***.*

These constraints help us efficiently evaluate whether a candidate set of formulae is indeed an XOR group or not.

Since we plan to use XOR groups as building blocks for computing a partition, we wish to identify XOR groups that are interesting to us. As an example consider a *trivial XOR group*, an XOR group that contains only a formula and its negation, $\mathcal{F} = \{\varphi, \neg\varphi\}$, where φ is never true in the reachable state space. In that case we can simply add φ as an additional formula in any other XOR group, but any such XOR group won't be helpful. Similarly, if we have two XOR groups $\mathcal{F}_1 = \{\varphi, \chi\}$ and $\mathcal{F}_2 = \{\psi, \phi\}$, we can clearly generate a new XOR group $\mathcal{F} = \{\varphi \wedge \psi, \varphi \wedge \phi, \chi \wedge \psi, \chi \wedge \phi\}$. In this case the two groups $\mathcal{F}_1$ and $\mathcal{F}_2$ are of the most interest, as we can always generate $\mathcal{F}$ on demand. We elaborate how to find such XOR groups in the following section.

4.4 Automated Partition Generation

Our partitioning system contains several pieces, shown in Figure 4.1. First, it performs static analysis of the domain and identifies the various XOR groups present in it. The external-memory reachability analysis filters out irrelevant parts of the state space. Additionally, it

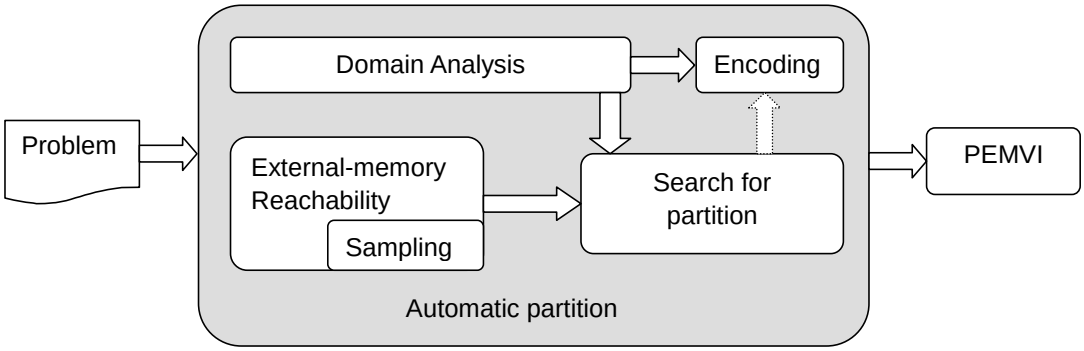


Figure 4.1: Flow chart of our automated partition generation system

samples a subset of the reachable space, which facilitates the next phase (searching for a partition) by estimating the block sizes of various partitions. Then it searches in the space of partitions, represented using the XOR groups, to construct a valid partition. Next, the encoding module scans the reachable state space and computes a much more compact string representation for each state based on the XOR groups. It also prepares the data structures necessary for PEMVI, such as distributing transition and value components into disk files, one for each partition block.

4.4.1 Domain Analysis

Our original definition of XOR groups is very general, but, searching over such a general space of formulae may not be practical, so we add two constraints in our search for XOR groups: (1) All formulae in an XOR group must be literals (instead of a general Boolean formula). (2) All terms in a first-order XOR formula must either be universally quantified or exclusively quantified (*i.e.*, they can't be constants). In theory, these assumptions may result in missing some XOR groups, but, our experiments demonstrate that our method yields useful XOR groups for all planning benchmarks. Similar results were reported in [136].

We identify XOR groups by performing static domain analysis — first finding *first-order* XOR formulae and later grounding them to construct the final set belonging to a *ground* XOR group. In the process we use a new term: We say a set of literals, $\mathcal{L}$, is *fully balanced*

(balanced for short) if it is balanced under *all* ground actions. This is faster to check than the necessary XOR conditions which quantify over *executable* actions. A balanced set of literals is *minimal* if no proper subset is also balanced. We denote the set of arguments of a predicate *pred* by *args*(*pred*) and the set of all arguments of a predicate list $Pred$ by *args*($Pred$).

Finding First-Order XOR Groups: At the highest level, our algorithm searches through syntactic XOR formulae in first-order logic and looks for candidates that satisfy the XOR conditions of the previous section. We iterate through all predicate set, and find if we can construct a first-order XOR formula out of each set.

Suppose we want to construct an XOR formula such that each literal corresponds to a predicate from the set $Pred$. Note that each predicate $\in Pred$ may be in its positive or negated form and two predicates $\in Pred$ may be the same[3]. Such a $Pred$ is the input to the Algorithm 9.

Algorithm 9 first computes all possible variable bindings (Lines 3-7). For example, suppose $Pred = \langle clear(\cdot),\ on(\cdot,\cdot)\rangle$. Then the three arguments in $Pred$ could be bound to the same variable name: $clear(x) \oplus on(x,x)$; all different variable names: $clear(x) \oplus on(y,z)$ or the six intermediate cases. All such cases will lead to different final XOR formulae. To compute this we consider all possible *type-homogeneous equivalence relations* for the set of arguments and assign the same variable name if two arguments lie in the same equivalence class (line 6). We define a type-homogeneous equivalence relations of a set of arguments $Args$ to be a set of non-empty, non-overlapping subsets of arguments whose union is $Args$, with the constraint that elements in one equivalence class must have the same type. For example, if $Args = \{b_1, b_2, c_1, c_2\}$ (two blocks and two colors) then type-homogeneous equivalence relation of $Args$ is $\{\{b_1, b_2\}, \{c_1, c_2\}\}, \{\{b_1\}, \{b_2\}, \{c_1, c_2\}\}, \{\{b_1, b_2\}, \{c_1\}, \{c_2\}\}, \{\{b_1\}, \{b_2\}, \{c_1\}, \{c_2\}\}$. Algorithm 10 outlines a method to compute this set.

Given this candidate XOR formula, φ, whose variables are not quantified so far, we wish to find a set, $\mathcal{X}$ ($\subseteq$ *args*(φ)), such that (1) when all arguments in $\mathcal{X}$ are universally quantified and all arguments in *args*(φ) $- \mathcal{X}$ are exclusively quantified, a correct XOR group

[3]This is necessary in searching for XOR groups of the form $\forall b[clear(b) \oplus \neg clear(b)]$

Algorithm 9 Xorify

```
1: Input: Pred = {pred_1, pred_2, ..., pred_k} (a set of predicates)
2: // compute candidate XORs
3: Xorifier ← ∅
4: R ← set of all possible type-homogeneous equivalence relations of args(Pred) (Algorithm 10)
5: for all equivalence relations r ∈ R do
6:     φ ← XOR of all predicates in Pred such that two arguments are represented by the same
       variable name iff they lie in the same equivalence class ∈ r
7:     add φ to Xorifier
8:
9: for all φ ∈ Xorifier do
10:    Open ← {∅} (a singleton with element ∅)
11:    while Open ≠ ∅ do
12:        delete the first set, X, from Open
13:        if is_XOR((∀_{x∈X} x)(∃1_{y∈args(φ)−X} y)[φ]) then
14:            output (∀_{x∈X} x)(∃1_{y∈args(φ)−X} y)[φ]
15:            return success
16:        X' ← X
17:        for all arg ∈ args(φ) − X do
18:            X' ← X' ∪ {arg}
19:            if X' ∉ Open then
20:                append X' at the end of Open
21: return failure
22:
23: is_XOR(F)
24: ground F to a set of predicates L
25: for each ground action a do
26:    if L is not balanced under a then
27:        return false
28: return true
```

results; and (2) no proper subset of $\mathcal{X}$ meets condition 1. For all $\mathcal{X}$ that meet condition 1 we say $\mathcal{X}$ *xorifies* ϕ. We call a set that meets both condition 1 and 2 a *minimal xorifier*.

Algorithm 10 Compute Type-Homogeneous Equivalence Relation

```
1: Input: Args (a set of arguments), k (the number of types)
2: distribute arguments by type into sets Args_1, ..., Args_k.
3: for each type t_i do
4:    ℝ_i ← EquivalenceRelation(Args_i)
5: return the Cartesian product of ℝ_1, ..., ℝ_k
6:
7: Function EquivalenceRelation(B)
8: if B = ∅ then
9:    return {∅} (a singleton that contains emptyset)
10: else
11:    x ← last element in B
12:    ℝ ← EquivalenceRelation(B − {x})
13:    ℝ_new ← ∅
14:    for every relation 𝓡 ∈ ℝ do
15:       for every equivalence class D ∈ 𝓡 do
16:          D_new ← D ∪ {x}
17:          add ((𝓡 ∪ {D_new}) − D) to ℝ_new
18:       add 𝓡 ∪ {{x}} to ℝ_new
19:    return ℝ_new
```

To find the minimal xorifier we perform a breadth-first search over all subsets of $\mathit{args}(\varphi)$, starting from the empty set (lines 10-20 in Algorithm 9). Expanding a node adds another argument to the current node. The goal is a minimal set $\mathcal{X}$ that xorifies φ. Running this algorithm on different predicate sets Pred yields several first-order XOR formulae. Finally, we ground these to get the concrete XOR groups.

As an illustration, consider a modified Explosive-Blocksworld, where blocks are colored. Consider a world with two pigment constants, *red* and *blue*, and an initial state, which specifies that initially b_1 and b_2 are *red* and *blue* respectively. The only action to modify a block's color is paint:

```
(:action paint
  :parameters (?b - block ?c ?nc - pigment)
  :precondition (color ?b ?c)
  :effect (and (color ?b ?nc)
               (not (color ?b ?c))))
```

To find the minimal set using *just* the predicate $color(b, c)$ we start the search with $\mathcal{X} = \emptyset$ (Lines 10 and 12). $\emptyset$ does not xorify *color*, since the XOR formula $\exists_1 b \exists_1 c\ [color(b, c)]$ does not hold, because, both blocks have some color in the initial state. The algorithm then looks for xorifiers of size 1. Consider $\mathcal{X} = \{c\}$, or equivalently an XOR group $\forall c \exists_1 b\ [color(b, c)]$. This means there exists exactly one block of each color. Although the initial state satisfies this condition, the *paint* action may violate it, by coloring two blocks into the same color. So it does not xorify *color*. But $\mathcal{X} = \{b\}$ xorifies *color*, since it means every block must have exactly one color. This minimal xorifier leads to two ground XOR groups: $color(b_1, red) \oplus color(b_1, blue)$ and $color(b_2, red) \oplus color(b_2, blue)$.

Our algorithm for computing XOR groups has marked differences from the previous version by Edelkamp & Helmert. We call their version the *EH algorithm*. In particular, EH algorithm (1) did not handle any XOR formulae with negated predicates, and (2) required exactly one argument in the whole formula to be exclusively quantified, hence they cannot infer properties like $\forall x[loves(x, x) \oplus depressed(x)]$, or $\forall pkg \exists_1 truck \exists_1 city[on(pkg, truck) \oplus in(pkg, city)]$. We implemented an extension of our algorithm that additionally finds *simple negated XOR groups* like $\forall b[clear(b) \oplus \neg clear(b)]$.

Grounding XOR Formulae: After we have the set of first-order XOR formulae, we ground them to get the concrete XOR groups. The grounding works in two steps. For each first-order XOR formula, in the first step we iterate over all possible assignments of the universally quantified variables. Each assignment maps to one ground XOR group where the assigned variables are treated as constants. In the second step, for each such XOR group we ground each predicate as follows: if the predicate does not contain an exclusively quantifed variable (in other words, it is a literal), we copy it; otherwise we ground the predicate with all possible assignments of the set of exclusively quantified variables, which results in multiple literals. In the end, we connect all literals using the XOR relation.

For example, suppose we want to ground the XOR formula $\forall pkg \exists_1 truck \exists_1 city[on(pkg, truck) \oplus in(pkg, city)]$ for a package-delivery domain (a package is either on a truck or in some city) that has two package constants pkg_1 and pkg_2, two truck constants $truck_1$ and $truck_2$, and two city constants $city_1$ and $city_2$. Applying the first step, as package is the only universally quantified variable, we map the first-order XOR group into two ground groups:

$$\exists_1 truck \exists_1 city[on(pkg_1, truck) \oplus in(pkg_1, city)], \quad \text{and}$$
$$\exists_1 truck \exists_1 city[on(pkg_2, truck) \oplus in(pkg_2, city)].$$

Now we illustrate how to apply the second step on the first XOR group. For the first predicate $on(pkg_1, truck)$, as truck is the only exclusively quantified variable, it is grounded into two literals $on(pkg_1, truck_1)$ and $on(pkg_1, truck_2)$. Similarly the second predicate $on(pkg_1, city)$ is ground into two literals $on(pkg_1, city_1)$ and $on(pkg_1, city_2)$. The final ground XOR group is:

$$on(pkg_1, truck_1) \oplus on(pkg_1, truck_2) \oplus on(pkg_1, city_1) \oplus on(pkg_1, city_2).$$

4.4.2 External-Memory Reachability Analysis

Often the state space reachable from the initial state and the state space expressed by the PPDDL description are of extremely varied sizes. For example, the Blocksworld problems have orders of magnitude more unreachable states than reachable ones, because the PPDDL description allows for several blocks on top of the same block and one block on top of several blocks, whereas the semantics of the domain given the typical initial states does not. We perform reachability analysis to focus computation only on the relevant subset of the states. Since the state spaces can be huge we implement an external-memory version, which is a layered BFS that uses delayed duplicate detection [73]. Our optimizations enable the search algorithm to switch automatically between the in-memory algorithm (when a search frontier fits in the memory), and the external-memory version.

Sampling: Additionally, we build a random subset of the reachable state space by sampling each state with a uniform probability x. This subset helps us in estimating the sizes of partition blocks and their transitions in the next phase.

4.4.3 Search for Valid Partitions

Recall that a *valid partition* is one in which any partition block can be backed up in memory, *i.e.*, its transition components as well as the value components of its successor blocks fit in memory. In this phase we search for a valid partition by successively applying the XOR groups computed by domain analysis.

Choosing a good partition is vital for PEMVI algorithm, since it can massively save on the computation time, as well as memory requirements. Previously, Zhou and Hansen [136] proposed one heuristic called *locality* as a criterion for choosing a suitable state abstraction. Locality prefers a partition that has the lower number of successor blocks, so in our case it makes loading successor blocks less I/O costly. We additional propose two new heuristics, *coherence* and *balance*. The coherence heuristic favors a partition whose total percentage of intra-block state transitions is the greatest among all transitions, and therefore optimizes the information flow to achieve faster convergence. The balance heuristic prioritizes a partition that shrinks the memory requirement of PEMVI most significantly, thus it can help reach a valid partition as early as possible.

However, evaluating these heuristics requires computing these criteria, *e.g.*, percentage of intra-block transitions, for the reachable space, an endeavor too ambitious to undertake. We exploit our sampled state space to make this tractable. With the (unknown) partition block size l, the size of the sampled space in that block follows a Binomial distribution(l, x) if a state is sampled uniformly with probability x. If sampling generates a subspace of size r, then l follows a negative binomial distribution(r, x), with mean $\mu = r/x$ and standard deviation $\sigma = \sqrt{r(1-x)}/x$. This distribution, according to the *central limit theorem* [109], is approximately *Normal*. The size l has a very slight probability ($\approx 0.1\%$) of being greater than $\mu + 3\sigma$. We use this number to be the upper bound while estimating l. We estimate the number of transitions into a block and other criteria in a similar fashion.

Given a heuristic (any of these above), ideally, we will enumerate all partitions and evaluate them based on the heuristic value to get the best partition. Unfortunately, if there are n XOR groups in the domain, the total number of possible partitions is 2^n, as apply any subset of the XOR groups results in one partition. Moreover, evaluating a partition,

even though made feasible by the use of sampled state space, is still quite time-consuming (see Section 4.5 for details). For this reason, we trade off optimality for time and use greedy search by picking the XOR group that has the best heuristic estimate at each partitioning step. In the experiments section we empirically compare these three heuristics and develop a strategy to maximize the algorithm performance.

Sometimes the partitioning process itself exhausts the available memory. The *block transition table* is a table that represents the successor block relations between all partition blocks. In our implementation, we represent the table as an adjacency list, and store it in memory. When the algorithm generates too many partition blocks then the block transition table overflows and we terminate the algorithm with failure.

4.4.4 Encoding

Encoding refers to changing the explicit logic state representation into an implicit string representation, which ideally requires less storage. Based on the set of found XOR groups we can encode the state space into a compact representation. For ease of illustration we describe the case when all domain features are Boolean. In such a case a naive encoding will use a bit-string whose length equals the number of the features. To reduce this further we can define a single multi-valued feature for an XOR group of k literals and the total number of bits needed for that feature will be $\log_2 k$ and we will save on $k - \log_2 k$ bits. In our encoding algorithm, we define these features greedily for each XOR group and remove all the participating ground predicates. We continue this process until all XOR groups are exhausted. We are left with a set of multi-valued features that we newly defined and some Boolean features (for the original ground predicates that didn't participate in any XOR group). We use this representation for a compact encoding of a state.

4.5 Experiments

In our empirical evaluation we wish to answer two sets of questions. One regarding the performance of PEMVI and the comparison with EMVI, the other regarding the effectiveness of automatic partitioning.

We implemented PEMVI, value iteration and the automatic partitioning algorithm in C/C++, based on the framework of miniGPT [19]. Similar to Edelkamp et al. the threshold value δ we used for our experiments is 10^{-4}. All experiments were run on a dual-core AMD 2.4GHz with 4GB RAM and a 60GB SATA II (300MB/s) hard disk.

4.5.1 Performance of PEMVI

We aim at answering the following questions: Does PEMVI scale up to problems unsolvable by value iteration? Is PEMVI more efficient than EMVI? What are the best settings for PEMVI? For example, do multiple backups per iteration outperform single backup, how important is choosing an optimal backup order?

Numerical Domains

We semi-automatically partitioned the two probabilistic, numerical domains that are used in [48]: Racetrack [8] and Wetfloor [20].

The Wet-Floor Domain. Each Wet-Floor problem represents a navigation grid where the goal is to move from a starting location to a goal location. Each cell (state) in the grid has a probability of 0.4 to be slippery – the actions of moving out of the cell have probabilistic effects of reaching unintended states. The only successors of a cell are its neighboring cells, but one doesn't always go in the intended direction due to slippery. One possible XOR group stems from whether or not a cell is slippery, but this has low coherence. Recursively splitting on the value of the x and y coordinates, on the other hand, yields high coherence and low locality.

Figure 4.2 illustrates our partition schema. It shows a 16×16 Wet-Floor problem, which has been grouped into 16 partition blocks, based on x and y coordinates. Each partition block has the same size and thus, the same number of states. Under this partition, the cardinality of $succpart(p)$ for a particular partition block p equals the number of neighboring partition blocks plus one (since p is a successor of itself). So the locality of this partition schema is 5. For example, in order to back up partition block $p5$, we need to load T_{p5}, as well as V_{p1}, V_{p4}, V_{p5}, V_{p6} and V_{p9}. While for value iteration, all 16 transition components

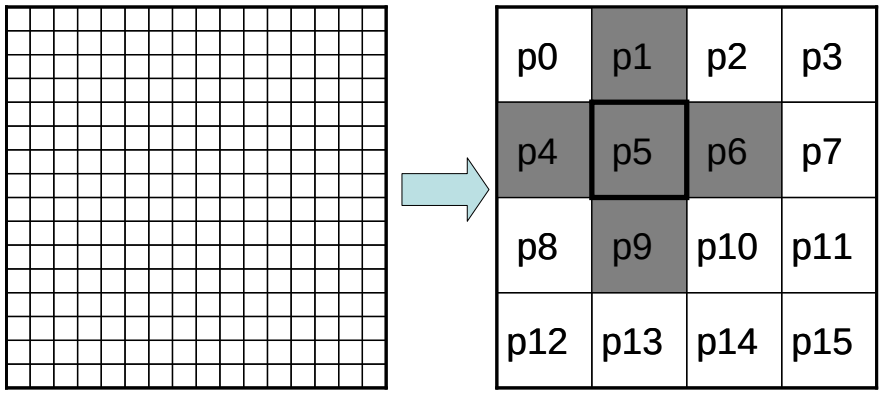


Figure 4.2: Partitioned MDP for one Wet-Floor problem. The grayed partition blocks on the right side need to be stored in memory in order to back up partition block p_5: the transition component of p_5 and the value components of its successors.

and value components must be in memory at all times. Using PEMVI, 15/16 of the space required to store transition components and 11/16 of the space required to store value components can be saved. If memory were so tight that this partition was still invalid, one could achieve additional savings by further increasing the number of partition blocks — the locality of this partitioning scheme does not change with granularity. Of course, for efficiency one generally seeks a valid partition with the smallest number of blocks.

The Racetrack Domain. Racetrack problems are another grid world, which is a popular testbed in reinforcement learning. The car starts stochastically from one of a fixed set of $\langle x, y \rangle$ points with speed 0. At each state, the car can accelerate or decelerate, changing its speed by at most one unit. Clearly, we may partition this domain using the same XOR constraint (x and y coordinates) as in Wet-Floor, but this constraint yields bad locality and bad coherence, because the x and y coordinates can change greatly when the car is moving fast. Instead, we partition on the instantaneous x and y components of velocity. The locality of this partition method is 9 since both v_x and v_y can change by at most one unit per action. By imposing a speed limit of 4 in every directional component, we bound the number of partition blocks by $(4 * 2 + 1)^2 = 81$.

Problem	States	Partition Blocks	I/O time	CPU time
Wet-Floor	1,000,000	100	2,182	508
Racetrack	3,413,725	81	18,848	3,370
Racetrack	6,980,425	81	31,415	5,334

Table 4.1: PEMVI Running time (in seconds) on some large problems in two numerical domains. VI and LRTDP failed to solve these problems.

Against Value Iteration

As expected, we verified that VI quickly exhausts available memory on our suite of problems. In all domains PEMVI easily solved problems too big for VI. We also tried labeled RTDP (LRTDP), another optimal algorithm, yet one which is more scalable due to reachability methods [17]. However, LRTDP also failed to solve the large problems. For the largest problems, an external memory algorithm like PEMVI is a must. Table 4.1 reports some of the large problems that PEMVI could solve. We also solved a Wet-Floor problem with 100 million states.

Single v.s. Multiple Backups

To find the best parameters for PEMVI we compared the single-backup versus multiple-backup settings. We tested eight Wet-Floor problems, whose grid sizes range from 100×100 to 800×800. For small problems we constrained the memory available to the algorithms. The convergence time of the two versions of PEMVI on these problems is plotted in Figure 4.3(left). We noticed that the multiple-backup version converged an order of magnitude faster than the single-backup version. On average, the single-backup version was 11.83 times slower than the multiple-backup version. We conclude that the ability to perform multiple backups in the same iteration without performing additional I/Os is an important feature of PEMVI that can result in significant time savings.

To better characterize this behavior, we looked at the running time spent on each iteration for the case of multiple backups on the 100×100 Wet-Floor problem. We ran PEMVI

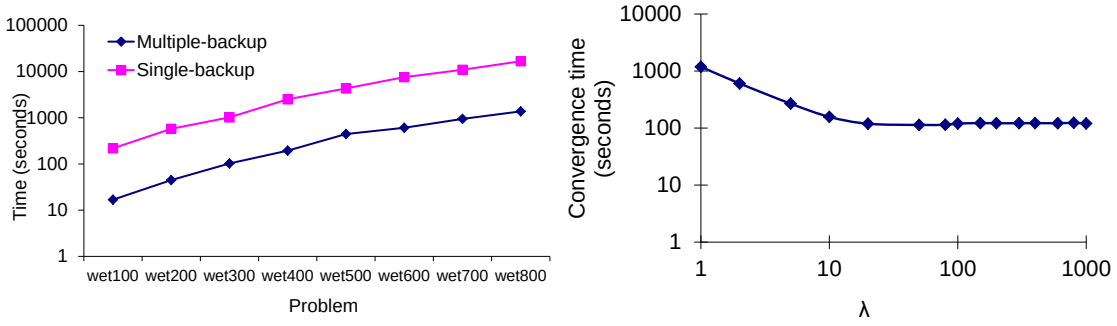


Figure 4.3: (left) Convergence time for single v.s. multiple backups. (right) Convergence time of different λ values on a Wet-Floor 300×300 problem..

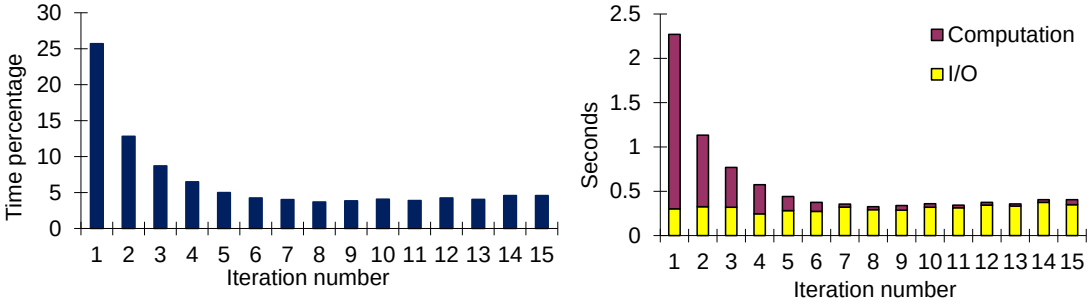


Figure 4.4: (a, left) Percentage running times for each iteration on the Wet-Floor 100×100 problem. (b, right) Actual computation time and I/O time for PEMVI. PEMVI spends more computation time at the first few iterations and almost constant time on I/O per iteration.

10 times and averaged its running time for each iteration. Observe in Figure 4.4(left), the first iteration took over 25% of the computation time with the first five iterations using 60% of the cumulative CPU time. In subsequent iterations, partition blocks converged after very few backups. I/O costs, naturally, were constant across iterations (from Figure 4.4(right)). We also performed a control experiment varying λ, the maximum number of backups per state in an iteration. The convergence times on a Wet-Floor problem with grid size 300×300 are plotted in Figure 4.3(right). We noticed that when λ changed from 1 to 20, the convergence time dropped significantly, but when λ was greater than 20, running times were not that distinct. So PEMVI is not overly sensitive to λ as long as it is not too small.

While multiple backups got us substantial savings in Wet-Floor problems, the Racetrack domain did not benefit from it, since its partitioning is one of low coherence, *i.e.*, partition blocks contain many more external transitions than internal transitions. Often, multiple-backups converged about a factor of 2 slower than single-backup on racetrack problems, which mirrors the findings in [133], where some partitions are more effective than the others. This also suggests that different heuristics for the automated partitioning might have significantly different results, and we plan to explore this further in subsequent experiments.

In all further experiments with multiple backups, we set $\lambda = 100$.

Backup Order

We also investigated the importance of the order in which the various partition blocks are backed up. We first evaluated the order produced by Algorithm 7, which aims to maximize information flow. For the eight Wet-Floor problems we computed the ratio of running time for "worst flow" ordering divided by that of "best flow" ordering. For single backups the ratio was only 1.021 with standard deviation of 0.007 — almost no difference. For multiple-backup version, the ratio was 1.320 with standard deviation of 0.155 — a larger difference. For both versions, the running time for random ordering lay consistently between the two extreme orderings.

We next evaluated the order generated by Algorithm 8, which tries to minimize I/O.

This order on a multiple-backup version gave us a maximum speedup of 10% on the eight Wet-Floor problems, but a larger 15% on the single-backup one. Similarly, for the single-backup version in a mid-sized Racetrack problem we got a speedup of 10% compared to 20% in the single-backups.

Not surprisingly we find that results of backup order are correlated with the ratio between the I/O and computational time for the algorithm. Typically, the single-backup algorithm's I/O time is significantly higher than the computation time, and thus backup order maximizing the information flow is less significant; minimizing I/Os yields better savings. On the other hand, the multiple-backup algorithm is more positively affected by the order which maximizes information flow.

For consistency we use the "best flow" order on all subsequent experiments.

Comparison with EMVI

For the EMVI algorithm, we used the implementation that was provided by Shahid Jabbar. After careful walk through of the program, we find that it depends on an input – a domain-dependent problem encoding. We encountered technical difficulty in applying the encoding on logical domains or re-implementing the whole planning system, therefore failed to test EMVI on logical domains such as the ones that are used in the International Planning Competition [67].

We compared PEMVI and EMVI on the two probabilistic domains used by Edelkamp et al. in their evaluations — Wet-Floor and Racetrack. We first compared the two algorithms on relatively small problems (under 1 M states). Figure 4.5 compares the number of iterations (and thus, implicitly the amount of I/O required) for both algorithms. In the Wet-Floor domain, EMVI's number of iterations increased linearly (R^2 value 0.986) with the grid perimeter of the problems. In contrast, the number of iterations taken by PEMVI remained stable, well under 20; a vast difference between the algorithms, which we credit to the power of multiple backups. We know from Theorem 19 that PEMVI improves the I/O complexity of EMVI by a logarithmic factor per iteration. Also from previous experiments, PEMVI spends majority of its running time on disk I/O. While PEMVI runs fewer itera-

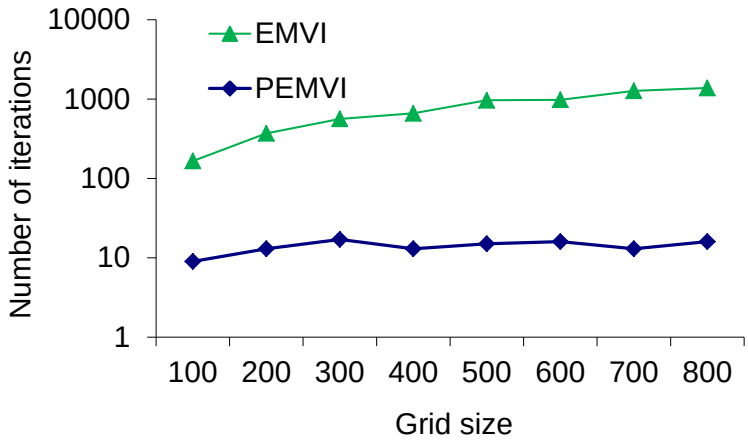


Figure 4.5: Comparison of number of iterations required by EMVI and PEMVI for problems in the Wet-Floor domain. PEMVI ran almost constant number of iterations, while the number of iterations EMVI ran increases linearly in the grid perimeter.

tions than EMVI, we conclude that PEMVI outperforms EMVI on the Wet-Floor domain, more dramatically when problems are larger.

The results on these relatively small problems were very encouraging so we attempted larger problems. We ran PEMVI on the largest Wet-Floor problem (grid size $10,000 \times 10,000$) mentioned in [48]. EMVI did not actually solve the problem, but their paper reported an expected running time to convergence of 2 years. PEMVI managed to solve this problem after 62 iterations, taking just under 2 months — an order of magnitude faster than EMVI.

We also ran EMVI and PEMVI on two middle-sized Racetrack problems with grid sizes of 75×75 and 50×50. EMVI took 2.5 times as long as PEMVI using single backups and 1.28 times as long as PEMVI, when it used multiple backups ($\lambda = 100$). At least with the current partitioning scheme, PEMVI is not able to derive great benefit from locality in this domain.

4.5.2 The Effectiveness of Automatic Partitioning

Through the control experiments we figured out a setting that is best for PEMVI: multiple-backup with $\lambda = 100$, and backing up partition blocks by the order that maximizes information flow. Now we investigate the usefulness of automatic partitioning.

We address the following questions: (1) How do the different partitioning heuristics (*viz.* locality, coherence and balance) compare with each other? (2) Does our general algorithm for finding XOR groups produce better XOR groups compare to the EH algorithm? Does our algorithm scale? (3) How does the quality of automatically-generated partitions compare to those that are manually generated? (4) Does automatic partitioning scale to large problems in the IPC domains? (5) Does the use of simple-negated XOR groups benefit the partitioning process?

To perform domain-independent experiments, we used eight of the ten domains from the International Planning Competition [67]. We dropped the Random domain due to its lack of logical semantics. We had to drop the Pitch-Catch domain as miniGPT failed to parse it.

Comparing Heuristics

We first performed control experiments in order to determine which of locality, coherence and balance is the best heuristic. Using a suite of six problems from three domains, we ran PEMVI with partitions generated by following each heuristic's guidance. We measured overall running time (partition generation + policy construction) with a cut-off of 10 hours. Figure 4.6 shows the time taken on a log scale. Apart from the smallest problem, coherence outperformed the other heuristics, which often failed to generate partitions that could solve the problem within the time limit.

Moreover, when we tested on even larger Explosive-Blocksworld problems, we observed that none of the heuristics enabled PEMVI to generate a solution within 10 hours. However, balance seemed to get closest in the sense that it was able to generate a valid partition (just not one that could be solved within the limit). In contrast, the other heuristics exhausted memory while just trying to find a valid partition.

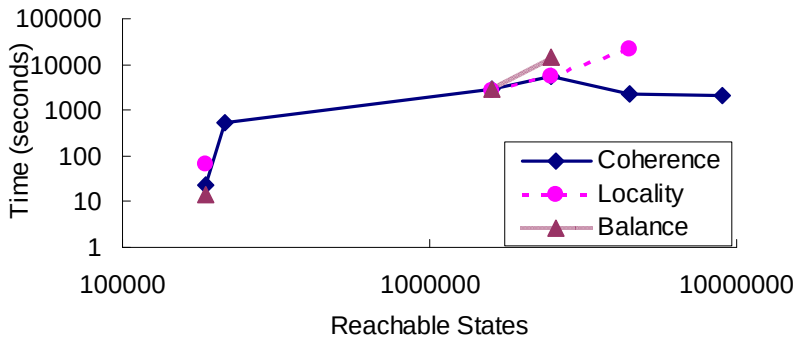


Figure 4.6: Running time of PEMVI using partitions created by different heuristics (missing points means problem not solved).

Algo	Blocksworld	Drive	Elevator	Schedule	Ex-Blocksworld	Tireworld	U-Drive	Zeno
EH	1	2	1	3	3	1	2	3
Ours	1	3	2	3	3	1	3	5

Table 4.2: Number of XOR formulae found by Edelkamp & Helmert (EH) and ours in IPC-06 domains. Our algorithm finds more XOR formulae in 50% of the domains.

These observations suggest a hybrid strategy. We partition greedily using the coherence heuristic, until either we find a valid partition or we reach a state where further partitioning will cause overflow of the block transition table. In this latter case we backtrack, trying the balance heuristic at the previous decision points. We use this hybrid scheme in all further experiments (Algorithm 11).

Comparing domain analysis algorithms

We now compare the performance of our new algorithm for finding XOR formulae (Algorithm 1) with the EH algorithm [47]. For all the problems that we tried, both algorithm ran very fast (finished in less than 1 second). As expected, the new algorithm consistently found a superset of XOR formulae. Table 1 shows that our algorithm finds additional XOR formulae in 50% of the domains. With the new set of XOR formulae, PEMVI managed to solve Zeno p5 in 23% of the time, with 53% of memory compared to using the smaller set

Algorithm 11 Search for Partition

```
1: input: S (state space), Z (the set of XOR groups)
2: // P means a partition, p means a partition block
3: P ← {p|p = S}
4: backtrack ← false
5: d ← 0
6: while P is not valid do
7:    if backtrack = false then
8:       pick a greedy z ∈ Z using coherence heuristic
9:       d ← d + 1
10:   else
11:      pick a greedy z ∈ Z using balance heuristic
12:   modify P by partitioning P using z
13:   if block transition table overflows then
14:      backtrack ← true
15:      if d ≥ 0 then
16:         d ← d − 1
17:         backtrack to partition depth d AND goto 6
18:      else
19:         return fail
20: return P
```

of XOR formulae found by the EH algorithm.

Quality of Automatic Partitioning

We now compare the partition generated by our algorithm with the Explosive-Blocksworld partition devised manually in our previous work [36]. That partition recursively applied a grounding of the $\exists_1 b\ [clear(b') \oplus on(b, b')]$ XOR group, using a new block b' at each level.

In contrast, automatic partitioning starts by picking the same XOR group, but at the next level it picks a group that differentiates different locations of b', namely $\exists_1 b\ [holding(b') \oplus on(b', b)]$ (whether b' is in hand, or on top of another block). By considering these two related XOR groups as a pair, the resulting partition gets a better performance, as we show

below.

We compare the two partition schemes by averaging over four 7-block Explosive-Blocksworld problems, whose average number of reachable states is 9,649,979. Table 4.3 compares the performance of the two partitioning approaches, which each found a valid partition at level 2 on every problem. We observe that on average automatic partitioning solves problems slightly faster (5.5%) and with slight less memory (6.4%). When tested on even larger (8- and 9-block) problems, manual partitioning failed completely, overflowing its block transition table. In contrast, automatic partitioning solved these problems handily (Table 4.4).

	CPU (s)	I/O (s)	Memory (MB)
Manual partition	2,894	1,489	1,681
Automatic partition	2,414	1,729	1,574

Table 4.3: Average PEMVI running time and memory usage on four 7-block Explosive-Blocksworld problems using different partitioning schema. Manual partition failed on larger (8- and 9-block) problems whereas automatic partitioning solved them.

Scalability

We evaluated our system on problems from eight domains from IPC-06 [67]. Since problems in Drive and Unrolled-Drive are extremely small, we do not report those results. Table 4.4 lists some of the large problems solved by PEMVI.[4] For each problem, we also report the size of its *reachable* state space, time spent performing reachability and partitioning, I/O time and CPU time consumed by PEMVI, as well as the planner's peak memory usage. As an external memory algorithm, the convergence of PEMVI on these problems is relatively fast (usually within a couple of hours). This shows the great power of our automated techniques in solving large probabilistic planning problems.

[4]The ones with * are not original IPC problems

Problem	Reachable $\|\mathcal{S}\|$	Automatic Partitioning		PEMVI		
		Reachability	Partition	I/O	CPU	Memory (MB)
Blocksworld p5	103,096	47	40	120	872	148
Elevator p15	538,316	990	893	220	898	877
Schedule p5	5,062,359	1,984	1,893	2,413	1,161	1,980
Zeno p5	5,223,057	3,456	3,187	1,340	13,918	448
Ex-Blocksworld α*	21,492,777	8,215	8,574	5,267	806	1,212
Tireworld p15	29,664,846	9,908	8,534	27,423	6,197	1,359
Ex-Blocksworld β*	42,899,585	16,656	16,059	10,926	1,527	580

Table 4.4: Running times of automatic partitioning, including reachability analysis and the search for a valid partition, and PEMVI, including I/O and CPU (in seconds) and memory usage (in mega-bytes) on some large problems in IPC-06 domains.

Simple-Negated XOR groups

We also find that very few XOR groups are present in some domains. For them we decided to use *simple-negated XOR groups*, of the form $\{l, \neg l\}$, where l is a ground literal. This modification especially helps the Blocksworld and the Tireworld domains, since they have very few XOR groups with complex structure.

Results show that, for the four Tireworld problems we tried, using simple-negated XOR groups lead to faster convergence (average 50.0% speedup, but most useful on small problems). For the Blocksworld domain, failure to consider simple-negated groups does not slow convergence on small problems (e.g., 5-block ones). However, on larger problems (e.g. 6-block ones and larger), we were unable to find a solution without using simple-negated groups. We conclude that simple-negated groups are very important in generating a partition, especially when the non-trivial XOR groups are too sparse. So, overall their use makes the algorithm more robust to a wide variety of domains.

4.6 Summary

We propose an optimal algorithm named *partitioned external-memory value iteration* (PEMVI) that overcomes the memory limitation of dynamic programming. It automatically divides the state space into partition blocks and performs (one or more) backups on all states in a piecemeal fashion. The experiments show that PEMVI solves problems much larger than what internal memory techniques can handle. Moreover, it outperforms a state-of-the-art external-memory MDP algorithm by an order of magnitude in convergence speed. The automatic, domain-independent partitioning algorithm for PEMVI uses static problem analysis to identify candidates for partitions, and chooses a suitable partition using heuristic search. Surprisingly, the automatic partitioning engine, without using any domain-specific information, constructs more useful partitions than the manually constructed partitions, which further improves scalability.

Chapter 5

CASE STUDY: DECISION-THEORETIC QUALITY CONTROL FOR CROWDSOURCING

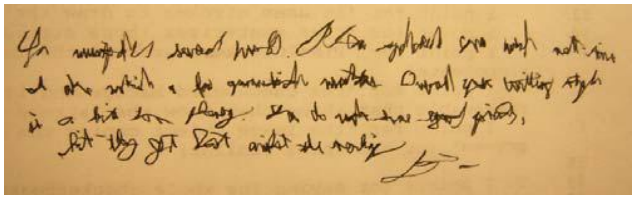
Previous two chapters elaborate our contributions on the theoretical aspects of decision-theoretic planning. In this chapter, we talk about a case study of decision making under uncertainty – quality control for crowdsourcing.

5.1 Introduction

In today's rapidly accelerating economy an efficient workflow for achieving one's complex business task is often the key to business competitiveness. With the development of US economy, its structure sees a trend in labor-centric tasks gradually moved to developing countries such as India and China. The *out-sourcing* market is ever-growing and worth $373 Billion [25] in the year 2009. In recent years, a new, revolutionary labor structure has been evolved. People from all over the world gather on-line forming a virtual labor market, thus enabling a contract-free market place.

Crowd-sourcing, "the act of taking tasks traditionally performed by an employee or contractor, and outsourcing them to a group (crowd) of people or community in the form of an open call" [131], has the potential to revolutionize information-processing services by quickly coupling human workers with software automation in productive workflows [63].

While the phrase "crowd-sourcing" was only termed in the year 2006, the area has grown rapidly in economic significance with the growth of general-purpose platforms such a Amazon's *Mechanical Turk* [94] and task-specific sites for call centers [86], programming jobs [126] and more. It has become immensely popular with hoards of employers (requesters), who use it to solve a wide variety of jobs, such as dictation transcription, content screening, etc. Workers exchange their contributions to the tasks posted by requesters for monetary reward, in a contract-free manner. With the high volume of available workforces, in the magnitude of hundreds of thousands, requesters can quickly find workers with matching



"You (misspelled) (several) (words). Please spellcheck your work next time. I also notice a few grammatical mistakes. Overall your writing style is a bit too phoney. You do make some good (points), but they got lost amidst the (writing). (signature)"

According to our ground truth, the highlighted words should be "flowery", "get", "verbiage" and "B-" respectively.

Figure 5.1: A handwriting recognition task (almost) successfully solved at Mechanical Turk using an iterative workflow. Workers were shown the text written by a human and in a few iterations they deduced the message (with errors highlighted). Figure adapted from [84].

expertise for various sorts of work. The two most desirable features of Mechanical Turk are its flexible, contract-free relationship between requesters and the workers as well as the cheap yet abundant labor resource. The flexible requester-worker relationship overlooks laborious paperwork, which helps achieves real-time responses on some tasks [15]. Most tasks on Mechanical Turk are typically worth one to a few cents, and the effective wage of workers is only a few dollars per hour [55].

Despite its cheap labor, researches show that work done through crowd-sourcing can achieve high quality in annotation tasks [119, 121]. Recent research [84] has shown surprising success in solving difficult tasks using the strategy of incremental improvement in an iterative workflow; similar workflows are used commercially to automate dictation transcription and screening of posted content. See Figure 5.1 for a successful example of a complex task solved using Mechanical Turk — this challenging handwriting was deciphered step by step, with output of one worker feeding as the input to the next. Additional voting jobs were used to assess whether a worker actually improved the transcription compared to the prior effort.

Due to high variances in the workers capabilities and motivations, requesters often manually subdivide a large task into a chain of small-sized subtasks to achieve quality results.

Even with mechanisms such as duplication and evaluation, as well as the more advanced iterative improvement technique, quality is not well guaranteed. This partly explains why Mechanical Turk is rarely used for very quality demanding jobs. This raises an interesting yet challenging question for AI: can statistical modeling and advanced AI techniques be helpful for crowd-sourced quality control?

From an AI perspective, crowd-sourced workflows offer a new, exciting and impactful application area for intelligent control. Although there is a vast literature on decision-theoretic planning and execution (*e.g.*, [110, 11, 72]), it appears that these techniques have yet to be applied to control a crowd-sourcing platform. While the handwriting example shows the power of collaborative workflows, we still do not know answers to many questions: (1) what is the optimal number of iterations for such a task? (2) how many ballots should be used for voting? (3) how do these answers change if the workers are skilled (or very error prone)?

This case study offers initial answers to these questions by presenting a decision-theoretic planner, which dynamically optimizes iterative workflows to achieve the best quality/cost tradeoff. The rest of the chapter is structured as follows. Chapter 5.2 discusses the motivation example. Chapter 5.3 illustrates the model details of our agent, TurKontrol. Chapter 5.4 introduces the planning algorithms of TurKontrol. Chapter 5.5 presents some simulation-based investigation of the performance of TurKontrol. Chapters 5.6 illustrates how to learn the model parameters. Chapter 5.7 demonstrates the usefulness of TurKontrol on Mechanical Turk, with real tasks.

5.2 Motivation Example

While the ideas in our paper are applicable to different workflows, for our case study we choose the iterative workflow introduced by Little *et al.* [84] depicted in Figure 5.2. This particular workflow is representative of a number of flows in commercial use today; at the same time, it is moderately complex making it ideal for first investigation.

Little's chosen task is iterative text improvement. There is an initial job, which presents the worker with an image and requests an English description of the picture's contents. A subsequent iterative process consists of an *improvement job* and *voting jobs*. In the

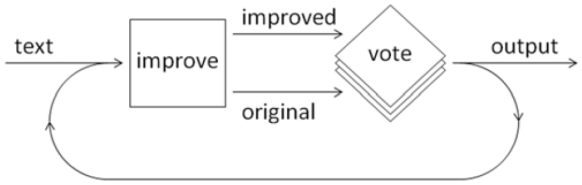


Figure 5.2: Flowchart for the iterative text improvement task, reprinted from [84].

improvement job, a (different) worker is shown this same image as well as the current description and is requested to generate an improved English description (see Figure A.1). Next $n(\geq 1)$ ballot jobs are posted ("Which text best describes the picture?"). See Figures A.2, A.3, A.4, and A.5 for our user interface design. Based on a majority opinion the best description is selected and the loop continues. Little *et al.* have shown that this iterative process generates better descriptions for a fixed amount than allocating the total reward to a single author.

Little *et al.*, support an open-source toolkit, TurKit, that provides a high-level mechanism for defining moderately complex, iterative workflows with voting-controlled conditionals. However, TurKit doesn't have built-in methods for monitoring the accuracy of workers; nor does TurKit automatically determine the ideal number of voters or estimate the appropriate number of iterations before returns diminish.

5.3 Decision Theoretic Optimization

The agent's control problem for a workflow like iterative text improvement is defined as follows. As input the agent is given an initial artifact (or a job description for requesting one), and the agent is asked to return an artifact which maximizes some payoff based on the quality of the submission. Intuitively, something is high-quality if it is better than most things of the same type. For engineered artifacts (including English descriptions) one may say that something is high quality if it is difficult to improve. Therefore, we define the *quality* of an artifact by $q \in [0, 1]$. An artifact with quality q means an average dedicated worker has probability $1 - q$ of improving the artifact. In our initial model, we assume that

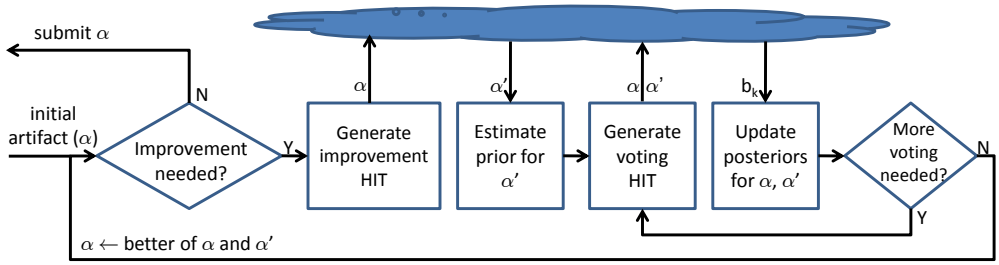


Figure 5.3: Computations needed by TurKontrol for control of an iterative-improvement workflow.

requesters will express their utility as a function U from quality to dollars.

The quality of an artifact is never exactly known – it is at best estimated based on domain dynamics and observations (like vote results). Thus, it is a POMDP problem. Moreover, since quality is a real number, it is a POMDP in continuous state space [24]. These kind of POMDPs are especially hard to solve for realistic problems. We overcome the computational bottleneck by performing limited lookahead search to make planning more tractable.

Figure 5.3 summarizes a high-level flow for our planner's decisions. At each step we track our belief in qualities (q and q') of the previous (α) and the current artifact (α') respectively. Each decision or observation gives us new information, which is reflected in the quality posteriors. These distributions also depend on the accuracy of workers, which we also incrementally estimate based on their previous work.

Based on these distributions we estimate expected utilities for each action. This lets us answer questions like (1) when to terminate the voting phase (thus switching attention to artifact improvement), (2) which of the two artifacts is the best basis for subsequent improvements, and (3) when to stop the whole iterative process and submit the result to the requester.

Below, we present our mathematical analysis in detail.

5.3.1 The POMDP Model

We first lay out the POMDP of the crowd-sourced workflow control problem. For the POMDP definition, refer to Chapter 2.

Definition 22 *The POMDP is a seven-tuple* $\langle \mathcal{S}, \mathcal{A}, T, C, R, \mathcal{O}, P \rangle$*, where*

- $\mathcal{S} = \{s(q, q') | q, q' \in [0, 1]\}$ *is a set of states on the interval-ed, two-dimensional, continuous space.*

- $\mathcal{A} = \{\text{do a ballot HIT}, \text{do an improvement HIT}, \text{stop}\}$ *is a finite set of all applicable actions. The stop action terminates an execution of the system.*

- $R(q) : [0, 1] \rightarrow R^+$ *is the reward received from submitting an artifact with quality* $q \in [0, 1]$.

- $T : \mathcal{S} \times \mathcal{A} \rightarrow \mathcal{S}$ *is the transition function.*

- $C : \mathcal{A} \rightarrow \mathbb{R}^+$ *is the cost function (amount of money paid for each HIT). We have fixed cost for each HIT type,* c_{imp} *for the cost of an improvement HIT and* c_b *for the cost of a ballot HIT.*

- $\mathcal{O} = \{\text{true}, \text{false}\}$ *is a Boolean answer received for a ballot question.*

- $P : \mathcal{S} \rightarrow \mathcal{O}$ *is the probability of receiving an observation.*

Our goal of solving the problem is to maximize the long-term expected utility (expected reward minus the total cost). The optimal policy of the underlying MDP should satisfy the following system of Bellman equations.[1]

$$\begin{aligned} V^*(\mathbf{s}) &= -R(q) \quad \text{if } \mathbf{s} \in \mathcal{G}, \text{ else} \\ V^*(\mathbf{s}) &= \min_{\mathbf{a} \in Ap(\mathbf{s})} \left[C(\mathbf{a}) + \sum_{\mathbf{s}' \in \mathcal{S}} T_{\mathbf{a}}(\mathbf{s}' | \mathbf{s}) V^*(\mathbf{s}') \right]. \end{aligned} \tag{5.1}$$

[1]We formalize it as a minimization problem for consistency. It can be trivially transformed into a equivalent maximization problem.

The belief state of the POMDP is the probability density functions of two quality distributions f_Q and $f_{Q'}$. We elaborate its components T and P.

Transition Function

Suppose we have an artifact α, with an unknown quality q and a prior[2] density function $f_Q(q)$. Suppose a worker x takes an improvement job and submits another artifact α', whose quality is denoted by q'. Since α' is a suggested improvement of α, q' depends on the initial quality q. Moreover, a higher accuracy worker x may improve it much more so q' depends on x. We define $f_{Q'|q,x}$ as the conditional quality distribution of q' when worker x improved an artifact of quality q. This function describes the dynamics of the domain. With a known $f_{Q'|q,x}$ we can easily compute the prior on q' from the law of total probability:

$$f_{Q'}(q') = \int_0^1 f_{Q'|q,x}(q') f_Q(q) dq. \tag{5.2}$$

Observation Function

While we do have priors on the qualities of both the new and the old artifacts, we do not know for sure whether the new artifact is an improvement over the old or not. The worker may have done a good job or a bad job. Even if it is an improvement we need to assess how good of an improvement it is. Our workflow at this point tries to gather evidence to answer these questions by generating ballots and asking new workers a question: "Is α' a better answer than α for the original question?". Say n workers take the ballot HIT and their votes are $\overrightarrow{\mathbf{b}^n} = b_1, \ldots, b_n$, where $b_i \in \{1, 0\}$.

Based on these votes we compute the posteriors in quality, $f_{Q|\overrightarrow{\mathbf{b}^n}}$ and $f_{Q'|\overrightarrow{\mathbf{b}^n}}$. These posteriors have three roles to play. First, more accurate beliefs lead to a higher probability of keeping the better artifact for subsequent iterations. Second, within the voting phase confident beliefs help us decide when to stop voting. Third, a high quality belief also helps us decide when to quit the iterative process and submit.

[2]We will discuss how to get such distributions in Chapter 5.6.

In order to accomplish this we make some assumptions. First, we assume each worker x is diligent, so she answers all ballots to the best of her ability. Still she may make mistakes, and we have full knowledge of her accuracy. Second, we assume that several workers will not collaborate adversarially to defeat the system.

These assumptions might lead one to believe that the probability distributions for worker responses, $P(b_i)$, are independent of each other. Unfortunately, this independence is violated due to a subtlety. The reason is that even though the different workers are not collaborating a mistake by one worker changes the error probability of others. This happens because a mistake gives evidence that the question may be intrinsically hard and hence, difficult for others to get it right also. To get around this we introduce *intrinsic difficulty*, d, of our question ($d \in [0, 1]$). It depends on whether the two quality s are very close or not. Closer the two artifacts the more difficult it is to judge whether one is better or not. We define the relationship between the difficulty and quality s as

$$d(q, q') = 1 - |q - q'|^{\mathcal{M}}, \tag{5.3}$$

where $\mathcal{M}$ is the *difficulty constant*. We can safely assume that given d the probability distributions will be independent of each other.

Moreover, each worker's accuracy will vary with the problem's difficulty. We define $a_x(d)$ as the accuracy of the worker x on a question of difficulty d. One will expect everyone's accuracy to be monotonically decreasing with d. It will approach random behavior as questions get really hard, *i.e.*, $a_x(d) \to 0.5$ as $d \to 1$. Similarly, as $d \to 0$, $a_x(d) \to 1$. We use a group of polynomial functions $\frac{1}{2}[1 + (1 - d)^{\gamma_x}]$ for $\gamma_x > 0$ to model $a_x(d)$ under these constraints. It is easy to check that this polynomial function satisfies all the conditions when $d \in [0, 1]$. Note that smaller the γ_x the more concave the accuracy curve, and thus greater the expected accuracy for a fixed d. See Figure 5.4 for an illustration.

Note that given knowledge of d we can compute the likelihood of a worker answering "Yes". We consider the ith worker x_i who has accuracy $a_{x_i}(d)$. We calculate $P(b_i = 1 \mid q, q')$

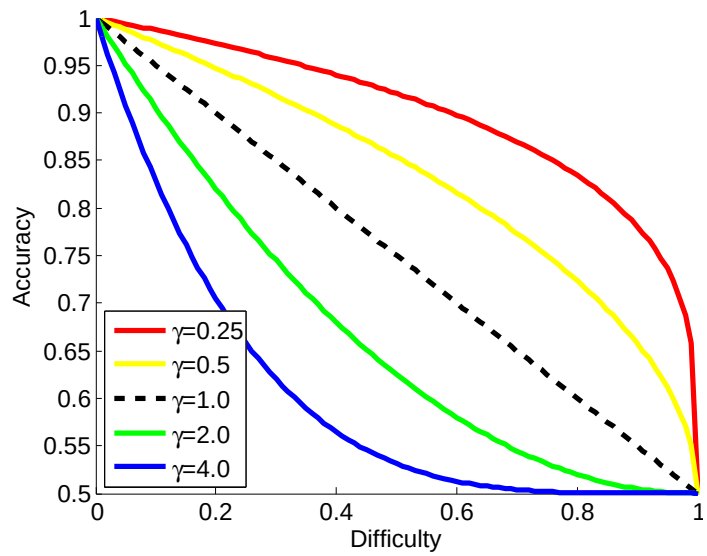


Figure 5.4: Workers' ballot answer accuracies given the intrinsic difficulty under various error parameters γ's. With the same difficulty, the greater a worker's γ value, the more likely that worker makes a mistake.

as:

$$\begin{aligned} \text{If } q' > q \; P(b_i = 1|q, q') &= a_{x_i}(d(q, q')), \\ \text{If } q' \leq q \; P(b_i = 1|q, q') &= 1 - a_{x_i}(d(q, q')). \end{aligned} \tag{5.4}$$

Belief Update

Recall that the belief state of the POMDP is the probability density functions of two quality distributions f_Q and $f_{Q'}$. Given a new observation, one needs to update the belief state of the POMDP. We first derive the posterior distribution given one more ballot b_{n+1}, $f_{Q|\overrightarrow{\mathbf{b}^{n}+1}}(q)$ based on existing distributions $f_{Q|\overrightarrow{\mathbf{b}^{n}}}(q)$ and $f_{Q'|\overrightarrow{\mathbf{b}^{n}}}(q)$. We abuse notation slightly, using $\overrightarrow{\mathbf{b}^{n}+1}$ to symbolically denote that n ballots are known and we will receive another ballot (value currently unknown) in the future. By applying the Bayes rule we get

$$f_{Q|\overrightarrow{\mathbf{b}^n+1}}(q) \propto P(b_{n+1} \mid q, \overrightarrow{\mathbf{b}^n}) f_{Q|\overrightarrow{\mathbf{b}^n}}(q) \tag{5.5}$$

$$= P(b_{n+1} \mid q) f_{Q|\overrightarrow{\mathbf{b}^n}}(q) \tag{5.6}$$

Equation 5.6 is based on the independence of workers. Now we apply the law of total probability on $P(b_{n+1} \mid q)$:

$$P(b_{n+1} \mid q) = \int_0^1 P(b_{n+1} \mid q, q') f_{Q'|\overrightarrow{\mathbf{b}^n}}(q') dq' \tag{5.7}$$

The same sequence of steps can be used to compute the posterior of α'.

$$f_{Q'|\overrightarrow{\mathbf{b}^n+1}}(q') \propto P(b_{n+1} \mid q', \overrightarrow{\mathbf{b}^n}) f_{Q'|\overrightarrow{\mathbf{b}^n}}(q') \tag{5.8}$$

$$= P(b_{n+1} \mid q') f_{Q'|\overrightarrow{\mathbf{b}^n}}(q') \tag{5.9}$$

$$= \left[\int_0^1 P(b_{n+1}|q, q') f_{Q|\overrightarrow{\mathbf{b}^n}}(q) dq \right] f_{Q'}(q')$$

Discussion

Why should our belief in the quality of the previous artifact change (posterior of α) based on ballots comparing it with the new artifact? This is a subtle, but important point. If the improvement worker (who has a good accuracy) was unable to create a much better α' in the improvement phase that must be because α already has a high quality and is no longer easily improvable. Under such evidence we should increase quality of α, which is reflected by the posterior of α, $f_{Q|\overrightarrow{\mathbf{b}}}$. Similarly, if all voting workers unanimously thought that α' is much better than α, it means the ballot was very easy, *i.e.*, α' incorporates significant improvements over α and the quality s should reflect that.

This computation helps us determine the prior quality for the artifact in the the next iteration. It will be either $f_{Q|\overrightarrow{\mathbf{b}}}$ or $f_{Q'|\overrightarrow{\mathbf{b}}}$ (Equations 5.6 and 5.9), depending on whether we decide to keep α or α'.

5.3.2 Utility Estimations

We now discuss the computation for the utility of an additional ballot. At this point, say, we have already received n ballots ($\overrightarrow{\mathbf{b}^n}$) and we have posteriors of the two artifacts $f_{Q|\overrightarrow{\mathbf{b}^n}}$ and $f_{Q'|\overrightarrow{\mathbf{b}^n}}$ available to us. We use $U_{\overrightarrow{\mathbf{b}^n}}$ to denote the expected utility of stopping now, *i.e.*, without another ballot and $U_{\overrightarrow{\mathbf{b}^n+1}}$ to denote the utility after another ballot. $U_{\overrightarrow{\mathbf{b}^n}}$ can be easily computed as the maximum expected utility we get from the two artifacts α and α':

$$U_{\overrightarrow{\mathbf{b}^n}} = max\{E[U(Q|\overrightarrow{\mathbf{b}^n})], E[U(Q'|\overrightarrow{\mathbf{b}^n})]\}, \text{ where} \tag{5.10}$$

$$E[U(Q|\overrightarrow{\mathbf{b}^n})] = \int_0^1 U(q) f_{Q|\overrightarrow{\mathbf{b}^n}}(q)dq \tag{5.11}$$

$$E[U(Q'|\overrightarrow{\mathbf{b}^n})] = \int_0^1 U(q') f_{Q'|\overrightarrow{\mathbf{b}^n}}(q')dq' \tag{5.12}$$

Using $U_{\overrightarrow{\mathbf{b}^n}}$ we need to compute the utility of taking an additional ballot, $U_{\overrightarrow{\mathbf{b}^n+1}}$. The $n+1^{th}$ ballot, b_{n+1}, could be either "Yes" or "No". The probability distribution $P(b_{n+1} \mid q, q')$ governs this, which also depends on the accuracy of the worker (see Equation 5.4). However, since we do not know which worker will take our ballot job, we assume anonymity and expect an average worker $\overline{x}$ with the accuracy function $a_{\overline{x}}(d)$. Recall from Equation 5.3 that difficulty, d, is a function of the similarity in quality s. Because q and q' are not exactly known, probability of getting the next ballot is computed by applying law of total probability on the joint probability $f_{Q,Q'}(q, q')$:

$$P(b_{n+1}) = \int_0^1 \left[\int_0^1 P(b_{n+1}|q, q') f_{Q'|\overrightarrow{\mathbf{b}^n}}(q')dq' \right] f_{Q|\overrightarrow{\mathbf{b}^n}}(q)dq. \tag{5.13}$$

These allow us to compute $U_{\overrightarrow{\mathbf{b}^n+1}}$ as follows (c_b is the cost of a ballot)

$$U_{\overrightarrow{\mathbf{b}^n+1}} = max\{E[U(Q|\overrightarrow{\mathbf{b}^n+1})], E[U(Q' \mid \overrightarrow{\mathbf{b}^n+1})]\} - c_b, \text{ where} \tag{5.14}$$

$$E[U(Q \mid \overrightarrow{\mathbf{b}^n+1})] = \int_0^1 \left(\sum_{b_{n+1}} U(q) f_{Q|\overrightarrow{\mathbf{b}^{n+1}}}(q) P(b_{n+1}) \right) dq. \tag{5.15}$$

We can write a similar equation for $E[U(Q' \mid \overrightarrow{\mathbf{b}^n+1})]$:

$$E[U(Q' \mid \overrightarrow{\mathbf{b}^n+1})] = \int_0^1 \left(\sum_{b_{n+1}} U(q') f_{Q'|\overrightarrow{\mathbf{b}^{n+1}}}(q') P(b_{n+1}) \right) dq'. \tag{5.16}$$

Similarly, we can compute the utility of an improvement step. We already have access to current beliefs on the quality of α and α'. Based on those and Equation 5.10 we can choose α or α' as the better artifact. The belief of the chosen artifact acts as f_Q for Equation 5.2 and we can estimate a new prior $f_{Q'}$ after an improvement step. Expected utility of improvement will be

$$\max \left\{ \int_0^1 U(q) f_Q(q) d(q), \int_0^1 U(q') f_{Q'}(q') d(q') \right\} - c_{imp}. \tag{5.17}$$

Here c_{imp} is the cost an improvement HIT.

5.4 Planning Algorithms

At any step we can either choose to do an additional vote, choose the better artifact and attempt another improvement or submit the artifact. We already described computations for utilities for each option. For a greedy 1-step lookahead policy we can simply pick the best of the three options.

5.4.1 Limited Lookahead

Of course, a greedy policy may be much worse than the optimal. Our first decision-making algorithm is l-step lookahead. It is similar to greedy search where we instead evaluate all sequences upto l decisions, find the best sequence based on the expected utility estimation and then execute the first action of the sequence and repeat. For example, when $l = 2$, our algorithm considers the following set of action sequences $\{\langle stop \rangle, \langle ballot, stop \rangle,$

$\langle improvement, stop\rangle$, $\langle ballot, ballot\rangle$, $\langle ballot, improvement\rangle$, $\langle improvement, ballot\rangle$, $\langle improveemnt, improvement\rangle\}$.

5.4.2 A discretized POMDP Algorithm

We also try a discretization-based POMDP algorithm. We first approximate and discretize the belief space into a finite-state MDP and later solve the MDP using value iteration. We call it the ADBS method, short for approximation and discretization of belief space. Our approximation method features a distribution by the values of its mean and standard deviation. Therefore, the belief space is a four-tuple $\langle \mu, \sigma, \mu', \sigma' \rangle$. Next we discretize the four variables into small, equal-sized intervals. As quality is a real number in $[0, 1]$, it is easy to show $\mu \in [0, 1]$ and $\sigma \in [0, 1]$. With discretization and the known bounds the belief space becomes finite. ADBS first performs a reachability search from the initial state s_0[3] to build an MDP model of the approximated belief space. It then optimally solves the discretized MDP by value iteration.

5.4.3 A UCT Variant

We apply the UCT algorithm to the agent control problem. First note that UCT is not immediately applicable, since we are unsure about the exact quality of the submitted artifact at a terminal state. One key requirement of UCT is the fully-observable environment, as it needs the concrete reward value at the terminal states in the end of a trial to back up visited states. To handle this, we run UCT on the belief space, which is fully-observable. Next, as the belief space is infinite, UCT might run out of memory without approximation. One can apply the simulation method used by the ADBS method. However, a better approximation is by the history of the workflow. One example of a history can be: (step 1) In the improvement HIT, worker x submits an improvement. (step 2) In the ballot HIT, worker y thinks x have made an improvement. (step 3) The agent submits the artifact submitted by x. Ignoring the identities and accuracies of workers, we can encode this history into a string i, t, s where i stands for someone finishes an improvement HIT, t means in a

[3]The initial state of a problem is immediately after we get a first artifact.

ballot HIT, a worker thinks the new artifact is better than the old artifact, and s means the agent submits the artifact with higher expected quality. This is an approximation since the identities of participating workers are ignored in order to reduce the belief space size. But this approximation works well from simulation results (see Chapter 5.5.4 for more details).

Since two different histories can map to the same belief state with small probability, this approximation method may generate duplicates. However, this approach has several advantages. First is straightforward policy mapping. The policy generated from history is intuitive and can be directly used in decision making. Second is simple encoding. We know in most cases the distributions f_Q and $f_{Q'}$ do not have closed forms. To get a decent approximation, we need to set up domain features (such as the moments of a distribution) and implement a fine-grained discretization method for the variables. To achieve this, we either use human knowledge (as we did in the ADBS algorithm), or use some sophisticated technique such as [91, 50, 82]. Third is easy evaluation. One difficulty in evaluating the ADBS method is that there is no systematic way to quantify how much the sub-optimality comes from approximation errors (with details in the following section). However, it is quite simple to perform this type of evaluation for the history approximation.

5.4.4 Updating Difficulty and Worker Accuracy

The agent updates its estimated d before each decision point based on its estimates of quality s as follows:

$$\begin{aligned} d^* &= \int_0^1 \int_0^1 d(q, q') f_Q(q) f_{Q'}(q') dq dq' \\ &= \int_0^1 \int_0^1 (1 - |q - q'|^{\mathcal{M}}) f_Q(q) f_{Q'}(q') dq dq' \end{aligned} \tag{5.18}$$

After completing this ballot we have access to estimates for d^* and the believed answer. We can use this information to update our record on the quality of each worker. In particular, if someone answered a question correctly then she is a good worker (and her γ_x should decrease) and if someone made an error in a question her γ_x should increase. Moreover the

increase/decrease amounts should depend on the difficulty of the question. The following simple update strategy may work:

1. If a worker answered a question of difficulty d correctly then $\gamma_x \leftarrow \gamma_x - d^*\delta$.

2. If a worker made an error when answering a question then $\gamma_x \leftarrow \gamma_x + (1 - d^*)\delta$.

We use δ to represent the learning rate, which we could slowly reduce over time so that the accuracy of a worker approaches an asymptotic distribution.

5.4.5 Implementation

In a general model such as ours maintaining a closed form representation for all these continuous functions may not be possible. Uniform discretization is the simplest way to approximate these general functions. However, for efficient storage and computation TurKontrol could employ the piecewise constant/piecewise linear value function representations or use particle filters. Even though approximate both these techniques are very popular in the literature for efficiently maintaining continuous distributions [91, 46] and can provide arbitrarily close approximations. Because some of our equations require double integrals and can be time consuming (*e.g.*, Equation 5.18) these compact representations will help in overall efficiency of the implementation.

5.5 Simulations

This section aims to empirically answer the following questions: (1) How deep should be an agent's lookahead to best tradeoff between computation time and utility? (2) How well do the ABDS algorithm perform and scale? (3) What is the error of our history approximation for the UCT algorithm? (4) How well does the UCT algorithm perform? (5) What is the best planner for TurKontrol? (6) Does TurKontrol make better decisions compared to TurKit? (7) Can our planner outperform an agent following a well-informed, fixed policy? (8) How well our planner handles workers that are worse in ballot jobs than in improvement jobs?

5.5.1 Experimental Setup

We set the maximum utility to be 1000 and use a convex utility function $U(q) = 1000\frac{e^q-1}{e-1}$ with $U(0) = 0$ and $U(1) = 1000$. We assume the quality of the initial artifact follows a Beta distribution $Beta(1, 9)$, which implies that the mean quality of the first artifact is 0.1. Suppose the quality of the current artifact is q, we assume the conditional distribution $f_{Q'|q,x}$ is Beta distributed, with mean $\mu_{Q'|q,x}$ where

$$\mu_{Q'|q,x} = q + 0.5\,[\ (1-q) \times (a_x(q) - 0.5) + q \times (a_x(q) - 1)\ ], \tag{5.19}$$

and the conditional distribution is $Beta(10\mu_{Q'|q,x}, 10(1-\mu_{Q'|q,x}))$. We know a higher quality means it is less likely the artifact can be improved. We model results of an improvement task, in a manner akin to ballot tasks; the resulting distribution of qualities is influenced by the worker's accuracy and the improvement difficulty, $d = q$.

We fix the ratio of the costs of improvements and ballots, $c_{imp}/c_b = 3$, because ballots take less time. We set the difficulty constant $\mathcal{M} = 0.5$. In each of the simulation runs, we build a pool of 1000 workers, whose error coefficients, γ_x, follow a bell shaped distribution with a fixed mean γ. We also distinguish the accuracies of performing an improvement and answering a ballot by using one half of γ_x when worker x is answering a ballot, since answering a ballot is an easier task, and therefore a worker should have higher accuracy.

5.5.2 The l-Step Lookahead Algorithm

We first run 10,000 simulation trials with average error coefficient γ=1 on three pairs of improvement and ballot costs — (30,10), (3,1), and (0.3,0.1) — trying to find the best lookahead depth l for the l-step lookahead algorithm. Figure 5.5 shows the average *utility*, the reward of the submitted artifact minus the payment to the workers, with different lookahead depths, denoted by TurKontrol(l). Note that there is always a performance gap between TurKontrol(1) and TurKontrol(2), but the curves of TurKontrol(3) and TurKontrol(4) generally overlap. We also observe that when the costs are high, such that the process usually finishes in a few iterations, the performance difference between TurKontrol(2) and deeper step lookaheads is negligible. Since each additional step of lookahead increases the compu-

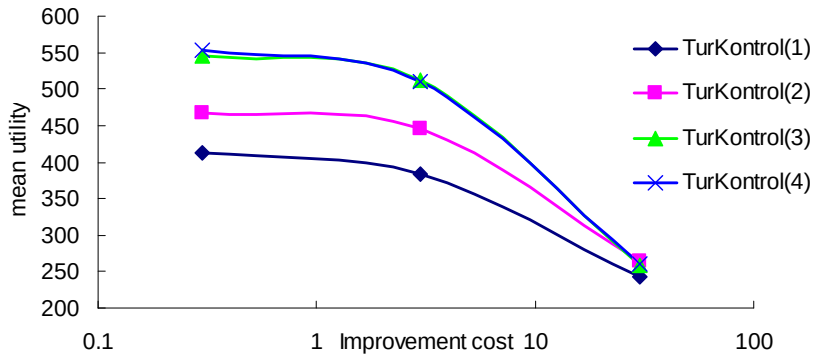


Figure 5.5: Average utility of TurKontrol with various lookahead depths calculated using 10,000 simulation trials on three sets of (improvement, ballot) costs: (30,10), (3,1), and (0.3,0.1). Longer lookahead produces better results, but 2-step lookahead is good enough when costs are relatively high: (30,10).

tational overhead by an order of magnitude, we limit TurKontrol' lookahead to depth 2 in subsequent experiments.

5.5.3 The ADBS Algorithm

We try different resolutions of discretization, *i.e.* various interval lengths, and run with average error coefficient γ=1 on three pairs of improvement and ballot costs — (30,10), (3,1), and (0.3,0.1). Table 5.1 lists the average utility of TurKontrol(2) on 10,000 simulations (results incorporated from Figure 5.5). For the ADBS algorithm, we report the size of the reachable belief states under each resolution[4], $|\mathcal{S}|$, and the value of the initial state, $V^*(\mathbf{s_0})$, calculated by value iteration.

We find TurKontrol(2) outperforms ADBS in all settings. Moreover, the smaller the costs, the bigger the performance gap. The bad performance of ADBS is probably due to the errors generated during approximation and discretization. Also notice that with more refined discretization, the reachable state space grows very quickly (at approximately a rate of an order of magnitude per doubly refined interval), yet the optimal value of the

[4]Recall that we represent a belief state with four variables ranged in $[0, 1]$. The discretization algorithm buckets the $[0, 1]$ interval into equal-lengthed sub-intervals and regards two real values the same if they fall into the same sub-interval. The smaller the interval length, the more refined a resolution is.

Costs	TurKontrol(2)	ADBS			
		Interval Length	0.02	0.01	0.005
		$\lvert\mathcal{S}\rvert$	3,577	41,072	809,420
(30,10)	264.906	$V^*(\mathbf{s_0})$	253.951	253.951	253.951
(3,1)	445.721	$V^*(\mathbf{s_0})$	365.866	366.798	367.674
(0.3,0.1)	466.820	$V^*(\mathbf{s_0})$	382.066	385.698	387.366

Table 5.1: Average utility of TurKontrol(2) and performance of the ADBS algorithms with different resolution (discrete interval lengths) on three sets of (improvement, ballot) costs: (30,10), (3,1), and (0.3,0.1). TurKontrol(2) outperforms the ADBS algorithms in all settings.

initial state increases very slowly. This indicates that the error basically comes from the approximation as opposed to discretization. We also try an interval length of 0.0025, but the algorithm terminates prematurely when the reachability search runs out of memory – indicating poor scalability of the ADBS algorithm. This experiment shows that a simple POMDP method does not work as good as the limited lookahead algorithm, yet it is too early to draw the conclusion that POMDP algorithms do not work on this planning problem. In the future we plan to try other more sophisticated POMDP algorithms.

5.5.4 The UCT Algorithm

We apply the following method to evaluate how much error is generated from averaging workers accuracies. In a new version of TurKontrol(2), we do everything the same as TurKontrol(2) except regarding every worker as an anonymous worker of the same, average accuracy. We denote this new variant TurKontrol(2,fixed). Note in this case, the workflow suggested is non-adaptive, as there is no uncertainty in workers' accuracies. We perform 10,000 simulations on the two favoring configurations (more afterwards):

1. $\gamma = 2$ and ballots accuracies are higher than improvement accuracies: TurKontrol(2) gets 4.5% more utility than TurKontrol(2,fixed).

2. $\gamma = 1$ and ballots accuracies are the same as improvement accuracies: TurKontrol(2)

gets 3.7% more utility than TurKontrol(2,fixed).

This shows using average γ is not a bad approximation, because it only downgrades the expected utility of TurKontrol(2) by less than 5% in the two most unfavorable cases. We expect it to produce even better results if approximation is refined, *e.g.*, grouping the workers by accuracy.

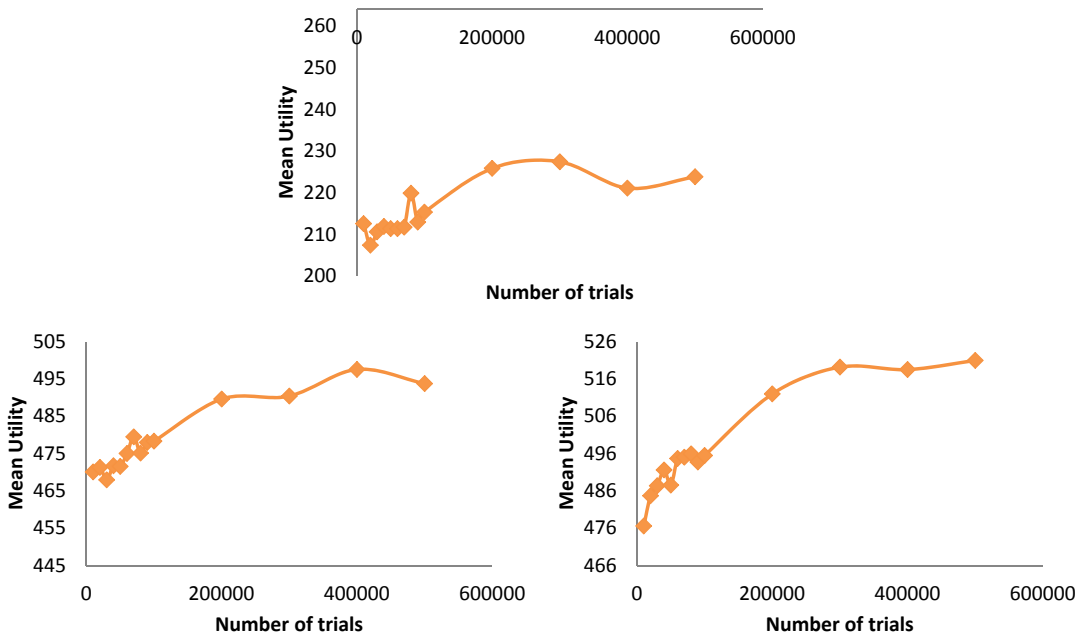


Figure 5.6: Average utility of the UCT algorithm after different number of trials averaged over 10,000 simulation trials compared to TurKontrol(2) (the horizontal line) with improvement and ballot costs (top) (30,10) (bottom left) (3,1) (bottom right) (0.3,0.1). UCT achieves higher average utility when the costs are low.

In investigating the UCT algorithm, we use a fixed $\kappa = 50$ and a dynamically decreasing learning parameter $\theta = \frac{0.2 \times log_2 10}{n_{(s,a)}}$. As an action has been visited more frequently, the effect of randomness decreases, so less weight should be given to the new observation. Figure 5.6 plots the average utility of UCT algorithms as a function of the number of trials (10,000 to 500,000) completed. The horizontal line stands for the average utility of TurKontrol(2) (data used in Figure 5.5). We first find that the performance of UCT improves with the increase in the number of completed trials. We also observe that UCT

outperforms TurKontrol(2) after very few (10,000) trials on the two problems where the costs are small, but underperforms TurKontrol(2) on the problem where the costs are high. This is probably because TurKontrol(2) already performs close to optimal on the high cost problem. We also find UCT consistently underperforms TurKontrol(3) (from comparing against Figure 5.5). This experiment shows that UCT is useful for quickly finding a good sub-optimal policy but has limited power in finding a close to optimal policy in the long run.

5.5.5 Choosing the Best Planning Algorithm

Comparing the results of the three algorithms: the l-step lookahead, ADBS and UCT, we conclude that the l-step lookahead is the most efficient planning algorithm for the domain. From now on, we use use the 2-step lookahead algorithm and TurKontrol interchangeably.

5.5.6 The Effect of Poor Workers

We now consider the effect of worker accuracy on the effectiveness of agent control policies. Using fixed costs of (30,10), we compare the average net utility of three control policies. The first is TurKontrol(2). The second, TurKit, is a non-adaptive policy from the literature [84]; it performs as many iterations as possible until its fixed allowance (400 in our experiment) is depleted and on each iteration it does at least two ballots, invoking a third only if the first two disagree. Our third policy, TurKontrol(fixed), combines elements from decision theory with a fixed policy. After simulating the behavior of TurKontrol(2), we compute the integer mean number of iterations, μ_{imp} and mean number of ballots, μ_b, and use these values to drive a fixed control policy (μ_{imp} iterations each with μ_b ballots). Thus this represents non-adaptive policy whose parameters are tuned to worker fees and accuracies.

Figure 5.7 (top) shows that both decision-theoretic methods work better than the TurKit policy, partly because TurKit runs more iterations than needed. A Student's t-test shows all differences are statistically significant with $p < 0.01$. We also note that the performance of TurKontrol(fixed) is very similar to that of TurKontrol(2), when workers are very inaccurate, γ=4. Indeed, in this case TurKontrol(2) executes a nearly fixed policy itself. In all other

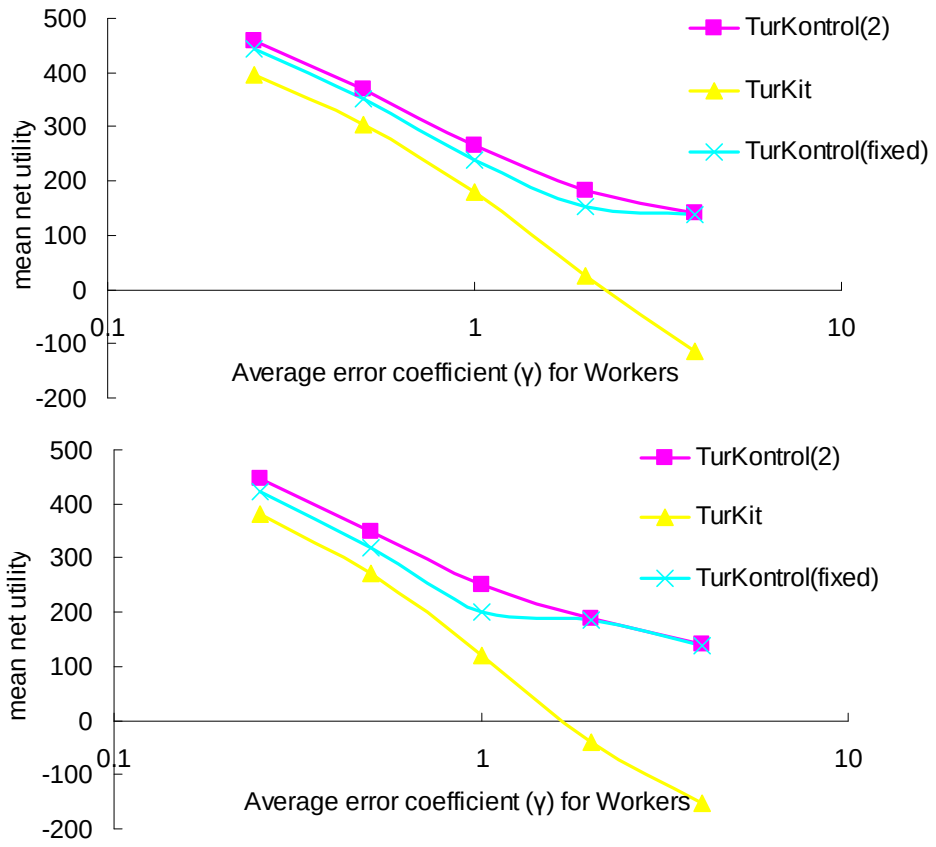


Figure 5.7: Average utility of three control policies averaged over 10,000 simulation trials, varying mean error coefficient, γ. (top) workers have better accuracies in ballot HITs than improvement HITs (bottom) workers are equally accurate in both ballot and improvement HITs. TurKontrol(2) produces the best policy in every cases.

cases, however, TurKontrol(fixed) consistently underperforms TurKontrol(2). A Student's t-test results confirm the differences are all statistically significant for $\gamma < 4$. We attribute this difference to the fact that the dynamic policy makes better use of ballots, *e.g.*, it requests more ballots in late iterations, when the (harder) improvement tasks are more error-prone. The biggest performance gap between the two policies manifests when γ=2, where TurKontrol(2) generates 19.7% more utility than TurKontrol(fixed).

5.5.7 Robustness in the Face of Bad Voters

As a final study, we considered the sensitivity of the previous three policies to increasingly noisy voters. Specifically, we repeated the previous experiment using the same error coefficient, γ_x, for each worker's improvement *and ballot* behavior (Figure 5.7 (bottom)). (Recall, that we previously set the error coefficient for ballots to one half γ_x to model the fact that voting is easier.) Figure 5.7(bottom) has the same shape as that of Figure 5.7(top) but with lower overall utility. Once again, TurKontrol(2) continues to achieve the highest average utility across all settings. Interestingly, the utility gap between the two TurKontrol variants and TurKit is consistently bigger for all γ than in the previous experiment. In addition, when γ=1, TurKontrol(2) generates 25% more utility than TurKontrol(fixed) — a bigger gap seen in the previous experiment. A Student's t-test shows all that the differences between TurKontrol(2) and TurKontrol(fixed) are significant when $\gamma < 2$ and the differences between both TurKontrol variants and TurKit are significant at all settings.

5.6 Model Learning

In the previous sections, we introduce the POMDP model for the quality control problem, and present extensive simulation results taking for granted a given model. This section addresses a practical problem – learning the model from real data from Mechanical Turk [41]. This learning problem is challenging due to a large number of parameters and sparse and noisy training data.

In order to estimate TurKontrol's POMDP model, one must learn two probabilistic transition functions. The first function is the probability of a worker x answering a ballot question correctly, which is controlled by the error parameter γ_x of the worker. The second function estimates the quality of an improvement result, the new artifact returned by a worker.

5.6.1 Learning the Ballot Model

Figure 5.8 presents our generative model of ballot jobs; shaded variables are observed. We seek to learn the error parameters $\vec{\gamma}$ where γ_x is parameter for the xth worker and use the

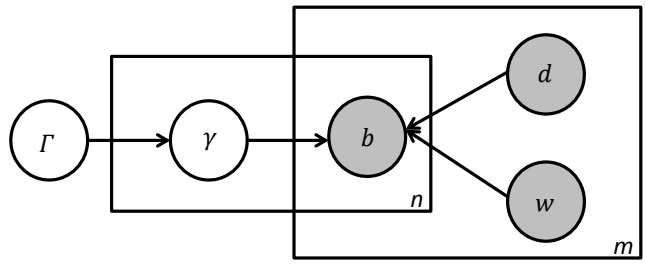


Figure 5.8: A plate model of ballot jobs under a supervised learning setting; b represents the ballot outcome; γ, a worker's individual error parameter; d, the difficulty of the job and w, truth value of the job. Γ is the prior on workers' errors. Shaded nodes represent observed variables.

mean $\bar{\gamma}$ as an estimate for future, unseen workers. To generate training data for our task we select m pairs of artifacts and post n copies of a ballot job, which asks the workers to choose between these pairs. We use $b_{i,x}$ to denote x^{th} worker's ballot on the ith question. Let $w_i = true(false)$ if the first artifact of the ith pair is (not) better than the second, and d_i denote the difficulty of answering such a question.

We assume the error parameters are generated by a random variable Γ. The ballot answer of each worker directly depends on her error parameter, as well as the difficulty of the job, d, and its real truth value, w. For our learning problem, we collect w and d for the m ballot questions from the consensus of three human experts and treat these values as observed. In our experiments we assume a uniform prior of Γ, though our model can incorporate more informed priors[5]. Our aim is to estimate γ_x parameters – we use the standard maximum likelihood approach. We use vector notation with $b_{i,x}$ denoting x^{th} worker's ballot on the ith question and $\vec{\vec{b}}$ denotes all ballots.

$$P(\vec{\gamma}|\vec{\vec{b}}, \vec{w}, \vec{d}) \propto P(\vec{\gamma})P(\vec{\vec{b}}|\vec{\gamma}, \vec{w}, \vec{d}) \tag{5.20}$$

Under the uniform prior of Γ and conditional independence of different workers given

[5]We also tried priors that penalized extreme values but that did not help in our experiments.

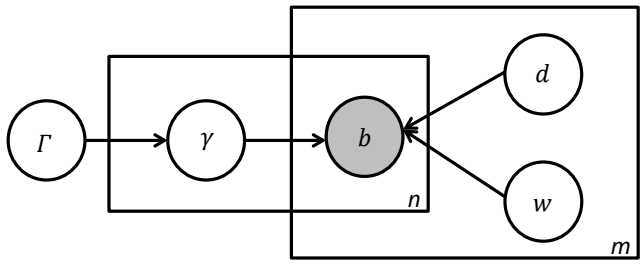


Figure 5.9: A plate model of ballot jobs under an unsupervised learning setting; b represents the ballot outcome; γ, a worker's individual error parameter; d, the difficulty of the job and w, truth value of the job. Γ is the prior on workers' errors. Shaded nodes represent observed variables.

difficulty and truth value of the task, Equation 5.20 can be simplified to:

$$\begin{aligned} P(\vec{\gamma}|\vec{\vec{b}}, \vec{w}, \vec{d}) &\propto P(\vec{\vec{b}}|\vec{\gamma}, \vec{w}, \vec{d}) \\ &= \Pi_{i=1}^{m}\Pi_{x=1}^{n} P(b_{i,x}|\gamma_x, d_i, w_i). \end{aligned} \tag{5.21}$$

Taking the log, the Maximum likelihood problem is:

$$\begin{aligned} \text{Constants}: &\quad d_1, \ldots, d_m, w_1, \ldots, w_m, b_{11}, \ldots, b_{m,n} \\ \text{Variables}: &\quad \gamma_1, \ldots, \gamma_n \\ \text{Maximize}: &\quad \textstyle\sum_{i=1}^{m}\sum_{x=1}^{n} log[P(b_{i,x}|\gamma_x, d_i, w_i))] \\ \text{Subject to}: &\quad \emptyset \end{aligned}$$

We also try an unsupervised learning algorithm. The plate notation is shown in Figure 5.9. We don't assume the true values or the difficulties are provided. Instead, we adopt an EM-style algorithm based on Whitehill *et al.*'s learning mechanism [129]. We initialize the difficulty values and error parameters with the corresponding average values learned from the supervised learning algorithm. We find applying prior distributions on the parameters does not help, so do not use a prior. During the expectation step, we compute the probability of the truth values given workers' answers, difficulty values and the error parameters. During the maximization step, we update the difficulty and error parameters based on the likelihood function (Equation 5.21).

Experiments on Ballot Model

We evaluate the effectiveness of our learning procedures on the image description task. We select 20 pairs of descriptions ($m = 20$) and collect sets of ballots from 50 workers. Spammers were detected manually and dropped from learning ($n = 45$). The supervised learning process has an overall cost of $4.50. We solve the optimization problem using the NLopt package.[6]. We implement the unsupervised learning algorithm based on Whitehill's framework [129].

Once the error parameters are learned they can be evaluated in a five-fold cross-validation experiment as follows: take 4/5th of the image pairs and learn error parameters over them; use these parameters to estimate the true ballot answer ($\tilde{w}_i$) for the images in the fifth fold. Our cross-validation experiment obtains an accuracy of 80.01%, which is barely different from the unsupervised learning algorithm and the majority baseline (with 80% accuracy). Indeed, we doublecheck that the four ballots frequently missed by the models are those in which the mass opinion differs from our expert labels.

We also compare the confidence, degree of belief in the correctness of an answer, for the two approaches. For the majority vote, we calculate the confidence by taking the ratio of the votes with the correct answer and the total number of votes. For our model, we use the average posterior probability of the correct answer. The average confidence values of using our supervised ballot model is much higher than the majority vote (82.2% against 63.6%). This shows that even though the two approaches achieve the same accuracy on all 45 votes, the ballot model has superior belief in its answer.

While the confidence values are different the ballot models (learned from both supervised and unsupervised learning) seem to offer no distinct advantage over the simple majority baseline given a large number of votes. In hindsight, this is not surprising, since we are using a large number of workers. In other work researchers have shown that a simple average of a large number of non-experts often beats even the expert opinion [119].

However, one will rarely have the resources to doublecheck each question by 45 voters, so we study this further by varying the number of available voters. For each image pair,

[6] *http://ab-initio.mit.edu/wiki/index.php/NLopt*

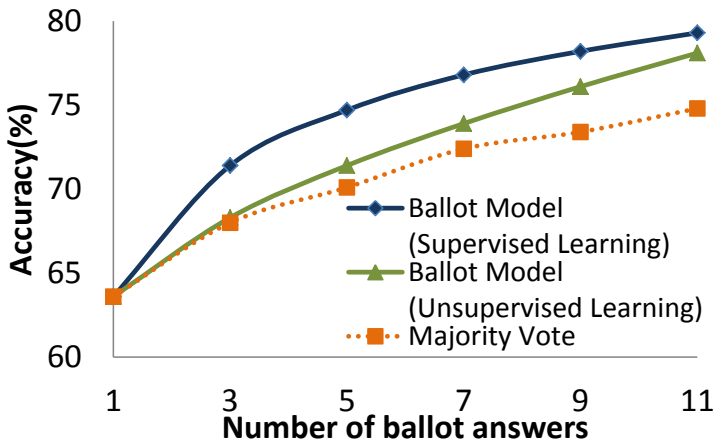


Figure 5.10: Accuracies of using our ballot model (by applying both supervised and unsupervised learning) and majority vote on random voting sets with different size, averaged over 50,000 random sample sets for each size. The models generated by the two learning algorithms both achieve higher accuracy than the majority vote. Using supervised learning, our ballot model achieves significantly higher accuracy than the majority vote ($p < 0.01$).

we randomly sample 50,000 sets of 3-11 ballots and compute the average accuracies of the two approaches. Figure 5.10 shows that our model (learned from both the supervised and the unsupervised learning algorithm) consistently outperforms the majority vote baseline. Furthermore, applying the supervised learning algorithm consistently achieves higher accuracy than applying the unsupervised learning algorithm, which shows the usefulness of using expert labeling. The unsupervised learning algorithm gradually catches up as the number of votes increases, which distinguishes itself from majority voting. In contrast, the model learned from supervised learning always outperforms majority voting by a margin. With just 11 votes, it is able to achieve an accuracy of 79.3%, which is very close to that using all 45 votes. Also, the supervised ballot model with only 5 votes achieves similar accuracy as a majority vote with 11. This shows the value of the ballot model – it significantly reduces the number of votes and thus the amount of money needed for the same desired accuracy. We use the model learned from the supervised learning algorithm for the remaining experiments. However, as the unsupervised learning algorithm does not require any labeled data, it can be a good complement when data labeling is expensive.

5.6.2 Estimating Artifact Quality

In order to learn the effect of a worker trying to improve an artifact (next section), we need labeled training data, and this means determining the quality of an arbitrary artifact. Since quality is a partially-observable statistical measure, we consider three ways to approximate it: simulating the definition, direct expert estimation, and averaged worker estimation.

Our first technique simply *simulates the definition*. We ask k workers to improve an artifact α and as before use multiple ballots, say l, to judge each improvement. We define quality of α to be 1 minus the fraction of workers that are able to improve it. Unfortunately, this method requires $k + kl$ jobs in order to estimate the quality of a single artifact; thus, it is both slow and expensive in practice. As an alternative, *direct expert estimation* is less complex. We teach a statistically-sophisticated computer scientist the definition of quality and ask her to estimate the quality to the nearest decile.[7] Our final method, *averaged worker estimation*, is similar, but averages the judgments from several Mechanical Turk workers via *scoring jobs*. These scoring jobs provide a definition of quality along with a few examples, mapped to a 0-10 integer scale; the workers are then asked to score several more artifacts. See Figures A.6, A.7, and A.8 for our user interface design.

Experimental Observations. We collect data on 10 images from the Web and use Mechanical Turk to generate multiple descriptions for each. We then select one description for each image, carefully ensuring that the chosen descriptions span a wide range of detail and language fluency. We also modified a description to obtain one that, we felt, was very hard to improve, thereby accounting for the high quality region. When simulating the definition, we average over $k = 22$ workers.[8] We use a single expert for direct expert estimation and an average of 10 worker scores for averaged worker estimation.

Our hope, following [119], was that averaged worker estimation, definitely the cheapest method, would prove comparable to expert estimates and especially to the simulated definition. Indeed, we find that all three methods produce similar results. They agree on the two

[7]The consistency of this type of subjective rating has been carefully evaluated in the literature; see *e.g.* [30].

[8]We collected 24 sets of improvements, but two workers improved less than 3 artifacts, so they were tagged as spammers and dropped from analysis.

	α_1	α_2	α_3	α_4	α_5	α_6	α_7	α_8	α_9	α_{10}
Definition Simulation	0.05	0.53	0.48	0.62	0.33	0.41	0.10	0.4	0.74	0.32
Averaged Worker Estimation	0.12	0.51	0.43	0.45	0.44	0.36	0.27	0.56	0.63	0.44
Direct Expert Estimation	0.1	0.5	0.4	0.6	0.4	0.2	0.3	0.5	0.9	0.4

Table 5.2: Qualities of 10 artifacts, collected through three methods: definition simulation, direct expert estimation and averaged worker estimation. The averaged worker estimation is the cheapest method that provides comparable results to the other two method.

best and worst artifacts, and on average both expert and worker estimates are within 0.1 of the score produced by simulating the definition. We conclude that averaged worker estimation is equally effective and additionally easier and more economical (1 cent per scoring job); so we adopt this method to assess qualities in subsequent experiments.

5.6.3 Learning the Improvement Model

Finally, we describe our approach for learning a model for the improvement phase. Our objective is to estimate the quality q' of a new artifact, α', when worker x improves artifact α of quality q. We represent this using a conditional probability density function $f_{Q'_x|q}$. Moreover, we also learn a prior distribution, $f_{Q'|q}$, to model work by a previously unseen worker.

There are two main challenges in learning this model: first, these functions are over a two-dimensional continuous space, and second, the training data is scant and noisy. To alleviate the difficulties, we break the task into two learning steps: (1) learn a mean value for quality using regression, and (2) fit a conditional density function given the mean. We make the second learning task tractable by choosing parametric representations for these functions. Our full solution follows the following steps:

1. Generate an improvement job that contains u original artifacts $\alpha_1, \ldots, \alpha_u$.

2. Crowd-source v workers to improve each artifact to generate uv new artifacts.

3. Estimate the qualities q_i and $q'_{i,x}$ for all artifacts in the set (see previous section). q_i is the quality of α_i and $q'_{i,x}$ denotes the quality of the new artifact produced by worker x. These act as our training data.

4. Learn a worker-dependent distribution $f_{Q'_x|q}$ for every participating worker x.

5. Learn a worker-independent distribution $f_{Q'|q}$ to act as a prior on unseen workers.

We now describe the last two steps in detail: the learning algorithms. We first estimate the mean of worker x's improvement distribution, denoted by $\mu_{Q'_x}(q)$.

We assume that $\mu_{Q'_x}$ is a linear function of the quality of the original artifact, *i.e.*, the mean quality of the new artifact linearly changes with the quality of the original one.[9] By introducing $\mu_{Q'_x}$, we separate the variance in a worker's ability in improving all artifacts of the same quality from the variance in our training data, which is due to her starting out from artifacts of different qualities. To learn this we perform linear regression on the training data $(q_i, q'_{i,x})$. This yields $q'_x = a_x q + b_x$ as the line of regression with standard error e_x, which we truncate for values outside $[0, 1]$.

To model a worker's variance when improving artifacts with the same quality, we consider three parametric representations for $f_{Q'_x|q}$: Triangular, Beta, and Truncated Normal. While clearly making an approximation, restricting attention to these distributions significantly reduces the parameter space and makes our learning problem tractable. Note that we assume the mean, $\hat{\mu}_{Q'_x}(q)$, of each of these distributions is given by the line of regression, $a_x q + b_x$. We consider each distribution in turn.

Triangular: The triangular-shaped probability density function has two fixed vertices $(0, 0)$ and $(1, 0)$. We set the third vertex to $\hat{\mu}_{Q'_x}(q)$, yielding the following density function:

$$f_{Q'_x|q}(q'_x) = \begin{cases} \frac{2q'_x}{\hat{\mu}_{Q'_x}(q)} & \text{if } q'_x < \hat{\mu}_{Q'_x}(q) \\ \frac{2(1-q'_x)}{1-\hat{\mu}_{Q'_x}(q)} & \text{if } q'_x \geq \hat{\mu}_{Q'_x}(q). \end{cases} \tag{5.22}$$

[9] While this is obviously an approximation, we find it is surprisingly close; $R^2 = 0.82$ for the worker-independent model.

Beta: We wish the Beta distribution's mean to be $\hat{\mu}_{Q'_x}$ and its standard deviation to be proportional to e_x. Therefore, we train a constant, c_1, using gradient descent that maximizes the log-likelihood of observing the training data for worker x.[10] This results in $f_{Q'_x|q} = Beta(\frac{c_1}{e_x} \times \hat{\mu}_{Q'_x}(q), \frac{c_1}{e_x} \times (1 - \hat{\mu}_{Q'_x}(q)))$. The error e_x appears in the denominator because the two parameters for the Beta distribution are approximately inversely related to its standard deviation.

Truncated Normal: As before we set the mean to $\hat{\mu}_{Q'_x}$ and the standard deviation to be $c_2 \times e_x$ where c_2 is a constant, trained to maximize the log likelihood of the training data. This yields $f_{Q'_x|q} = \textit{Truncated Normal}(\hat{\mu}_{Q'_x}(q), c_2^2 e_x^2)$ where the truncated interval is $[0, 1]$.

We use similar approaches to learn the worker-independent model $f_{Q'|q}$, except that training data is of the form $(q_i, \bar{q'_i})$ where $\bar{q'_i}$ is the *average* improved quality for i^{th} artifact, *i.e.*, the mean of $q'_{i,x}$ (over all workers). The standard deviation of this set is $\sigma_{Q'_i|q_i}$. As before, we start with linear regression, $q' = aq + b$. The Triangular distribution is defined exactly as before. For the other two distributions, we have their standard deviations depend on the conditional standard deviations, $\sigma_{Q'_i|q_i}$. We assume that the conditional standard deviation $\sigma_{Q'|q}$ is quadratic in q, therefore an unknown conditional standard deviation given any quality $q \in [0, 1]$ can be inferred from existing ones $\sigma_{Q'_1|q_1}, \ldots, \sigma_{Q'_v|q_v}$ using quadratic regression. As before, we use gradient descent to train variables c_3 and c_4 for Beta and Truncated Normal respectively.

Experimental Observations

We seek to determine which of the three distributions best models the data, and we employ leave-one-out cross validation. We set the number of original artifacts and number of workers to be ten each ($u = v = 10$). It costs a total of \$16.50 for this data collection. The algorithm iteratively trains on nine training examples, *e.g.* $\{(q_i, \bar{q'_i})\}$ for the worker-independent case, and measures the probability density of observing the tenth. We score a model by summing the ten log probability densities.

Our results show that Beta distribution with $c_1 = 3.76$ is the best conditional distribu-

[10] We use Newton's method with 1,000 random restarts. Initial values are chosen uniformly from the real interval $(0, 100.0)$.

tion for worker-dependent models. For the worker-independent model, with an intercept of 0.35 and slope of 0.34 from the linear regression, Truncated Normal with $c_4 = 1.00$ performs the best. We suspect this is the case because most workers have average performance and Truncated Normal has a thinner tail than the Beta. In all cases, the Triangular distribution performs worst. This is probably because Triangular assumes a linear probability density, whereas, in reality, workers tend to provide reasonably consistent results, which translates to higher probabilities around the conditional mean. We use these best performing distributions in all subsequent experiments.

5.7 TurKontrol on Mechanical Turk

Having learned the POMDP parameters, our final evaluation assesses the benefits of the dynamic workflow controlled by TurKontrol versus a non-adaptive workflow (as originally used in TurKit [84]) under similar settings, specifically using the same monetary consumption. We aim at answering the following questions: (1) Is there a significant quality difference between artifacts produced using TurKontrol and TurKit? (2) What are the qualitative differences between the two workflows?

As before, we evaluate on the image description task, in particular, we use 40 fresh pictures from the Web and employ iterative improvement to generate descriptions for these. For each picture, we restrict a worker to take part in at most one iteration in each setting (*i.e.*, non-adaptive or dynamic). We set the user interfaces to be identical for both settings and randomize the order in which the two conditions are presented to workers in order to eliminate human learning effects. Altogether there are 655 participating workers, of which 57 take part in both settings.

We devise automated rules to detect spammers. We reject an improvement job if the new artifact is identical to the original. We reject ballot and scoring jobs if they are returned so quickly that the worker could not have made a reasonable judgment.

Note that our system does not need to learn a model for a new worker before assigning them jobs; instead, it uses the worker-independent parameters $\bar{\gamma}$ and $f_{Q'|q}$ as a prior. These parameters get incrementally updated as TurKontrol obtains more information about their accuracy.

Recall that TurKontrol performs decision-theoretic control based on a user-defined utility function. We use $U(q) = \$25q$ for our experiments. We set the cost of an improvement job to be 5 cents and a ballot job to be 1 cent. We use the 3-step lookahead algorithm for the controller. Under these parameters, TurKontrol-workflows run an average of 6.25 iterations with an average of 2.32 ballots per iteration, costing about 46 cents per image description on average.

We use TurKit's original fixed policy for ballots, which requests a third ballot if the first two voters disagree. We compute the number of iterations for TurKit so that the total money spent matches TurKontrol's. Since this number comes to be 6.47 we compare against three cases: TurKit_6 with 6 iterations, TurKit_7 with 7 iterations and TurKit_{67} a weighted average of the two that equalizes monetary consumption.

For each final description we create a scoring job in which multiple workers score the descriptions. Figure 5.11 compares the artifact qualities generated by TurKontrol and by TurKit_{67} for the 40 images. We note that most points are below the $y = x$ line, indicating that the dynamic workflow produces superior descriptions. Furthermore, the quality produced by TurKontrol is greater on average than TurKit's, and the difference is statistically significant: $p < 0.01$ for TurKit_6, $p < 0.01$ for TurKit_{67} and $p < 0.05$ for TurKit_7, using the student's t-test.

Using our parameters, TurKontrol generates some of the highest-quality descriptions with an average quality of 0.67. TurKit_{67}'s average quality is 0.60; furthermore, it generates the two worst descriptions with qualities below 0.3. Finally, the standard deviation for TurKontrol is lower (0.09) than TurKit's (0.12). These results demonstrate overall superior performance of decision-theoretic control on live, crowd-sourced workflows.

While the 11% average quality promotion brought by TurKontrolis statistically significant, some wonder if it is *material*. To better illustrate the importance of quality, we include another experiment. We run the nonadaptive, TurKit policy for additional improvement iterations, until it produces artifacts with an average quality equal to that produced by TurKontrol. Fixing the quality threshold, the TurKit policy has to run an average of 8.76 improvements, compared to the 6.25 improvement iterations used by TurKontrol. As a result the nonadaptive policy spends 28.7% more money than TurKontrol to achieve

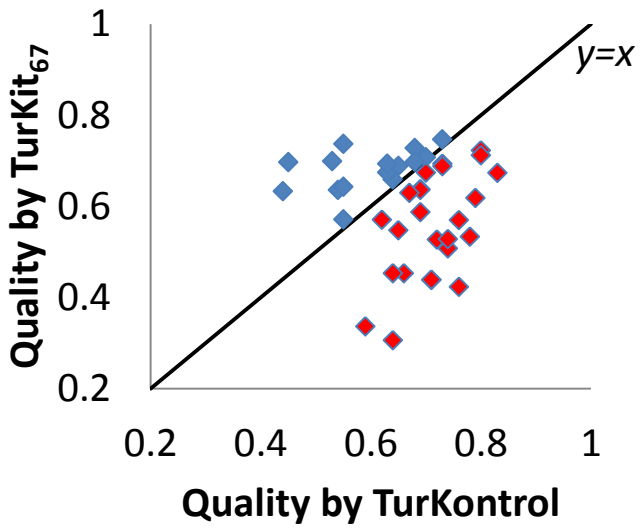


Figure 5.11: Average qualities of 40 descriptions generated by TurKontrol and by TurKit respectively, under the same monetary consumption. TurKontrol generates statistically-significant higher-quality descriptions than TurKit.

the same quality results. Note that final artifact quality is neither linear in the number of iterations nor total cost. Intuitively, it is much easier to improve an artifact when its quality is low than when it is high.

We also qualitatively study TurKontrol's behavior compared to TurKit's and find an interesting difference in the use of ballots. Figure 5.12 plots the average number of ballots per iteration number. Since TurKit's ballot policy is fixed, it always uses about 2.45 ballots per iteration. TurKontrol, on the other hand, uses ballots much more intelligently. In the first two improvement iterations TurKontrol does not bother with ballots because it expects that most workers will improve the artifact. As iterations increase, TurKontrol increases its use of ballot jobs, because the artifacts are harder to improve in later iterations, and hence TurKontrol needs more information before deciding which artifact to promote to the next iteration. The eighth iteration is an interesting exception; at this point improvements have become so rare that if even the first voter rates the new artifact as a loser, then TurKontrol often believes the verdict.

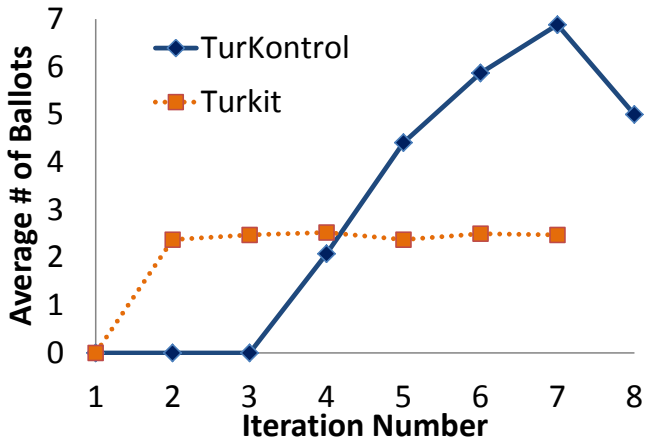


Figure 5.12: Average number of ballots for the non-adaptive and dynamic workflows. TurKontrol makes and intelligent use of ballots.

Besides using ballots intelligently we believe that TurKontrol adds two other kinds of reasoning. First, six of the seven pictures that TurKontrol finished in 5 iterations have higher qualities than TurKit's. This suggests that its quality tracking is working well. Perhaps due to the agreement among various voters, TurKontrol is able to infer that a description already has quality high enough to warrant termination. Secondly, TurKontrol has the ability to track individual workers, and this also affects its posterior calculations. For example, in one instance TurKontrol decided to trust the first vote because that worker had superior accuracy as reflected in a low error parameter. We expect that for repetitive tasks this will be an enormously valuable ability, since TurKontrol will be able to construct more informed worker models and take much superior decisions.

We present one image description example in Figure 5.13. It is interesting to note that both processes managed to find out the origin of the image. However, the TurKontrol version is consistently better in language, factuality and level of detail. In retrospect, we find the nonadaptive workflow probably made a wrong ballot decision in the sixth iteration, where a decision was critical yet only three voters were consulted. TurKontrol on the other hand, reached a decision after 6 unanimous votes at the same stage.

Figure 5.13: An image description example. It took TurKontrol 6 improvement HITs and 14 ballot HITs to reach the final version: "*This is Gene Hackman, in a scene from his film "The Conversation," in which he plays a man paid to secretly record people's private conversations. He is squatting in a bathroom gazing at tape recorder which he has concealed in a blue toolbox that is now placed on a hotel or motel commode (see paper strip on toilet seat). He is on the left side of the image in a gray jacket while the commode is on the right side of the picture. His fingertips rest on the lid of the commode. He is wearing a black court and a white shirt. He has put on glasses also.*." It took the nonadaptive workflow 6 improvement HITs and 13 ballot HITs to reach a version: "*A thought about repairing : Image shows a person named Gene Hackman is thinking about how to repair the toilet of a hotel room. He has opened his tool box which contains plier, screw diver, wires etc. He looks seriously in his tool box & thinking which tool he willuse. WearING a grey coat he sits in front of the toilet seat resting gently on the toilet seat*"

5.8 Summary

Our work is one of the first investigations into the applications of modern decision-theoretic planning techniques to quality control for crowdsourcing. We use decision theory to model a popular class of iterative workflows as a partially-observable Markov decision process (POMDP) and define equations (the model of the POMDP) that govern the various steps of the process. Our agent, TurKontrol, implements the proposed mathematical framework and applies advanced planning techniques to optimize and control the workflow. To bootstrap the mathematical model from limited amount of noisy data, we transform the learning problem into several optimization problems and apply statistical learning techniques to find the most accurate model parameters. We plug in the model to TurKontrol and demonstrate that the model is useful in practice — the dynamic workflow computed by TurKontrol generates statistically-significant higher-quality results than a traditional, nonadaptive workflow, where both systems consume the same amount of money. Moreover, the nonadaptive policy spends 28.7% more money than TurKontrol to achieve the same quality results.

Chapter 6

RELATED WORK

We categorize related work into two main topics: decision making under uncertainty and applications for crowdsourcing.

6.1 Decision Making under Uncertainty

6.1.1 Problem Decomposition

Besides TVI several other pieces of work share the idea of decomposing an MDP into sub-problems and combining their solutions for the final policy, *e.g.*, [59, 103]. However, these approaches typically assume some additional structure in the problem, either known hierarchies, or known decomposition into weakly coupled sub-MDPs, *etc.*, whereas FTVI assumes no additional structure. Moreover, FTVI is optimal whereas other algorithms are approximate.

The HDP algorithm [16] is similar to TVI in the sense that it uses the Tarjan's algorithm (slightly different from the Kosaraju's algorithm) to find the strongly connected components of a greedy graph. It computes the SCCs multiple times and dynamically during the depth-first searches when HDP tries to label solved states. But it does not find the topological order of the SCCs nor decompose a problem and use the topological order to sequentially solve each SCC.

The idea of finding the topological order of strongly connected components of an MDP has been extended to solving POMDPs. The topological order-based planner (POT) [43] uses the topological order information of the underlying MDPs to help solve a POMDP problem faster. We believe the idea can be extended to help solve even harder problems, such as decentralized POMDP [10], in the future.

6.1.2 Using Double Bounds

Like FTVI, BRTDP [93], Bayesian RTDP [111] and Focused RTDP [118] (FRTDP) also keep an upper bound for the value function. However, none of these algorithms use upper bounds for action elimination. Instead, they use the upper bound information to judge how close a state is to convergence, by calculating the difference between the upper and lower bound values. For example, BRTDP tries to focus search trials on states whose two bounds have larger differences, or intuitively, states whose values are less converged. Unlike FTVI, none of the three algorithms computes the strongly connected components of an MDP. Further, the performance of BRTDP (and similarly Bayesian RTDP) is highly dependent on the quality of the heuristics. FRTDP only works in the discounted setting. Therefore, it is not immediately applicable for stochastic shortest path problems.

6.1.3 Prioritization Methods

Prioritized sweeping [97] and its extensions, focussed dynamic programming [54] and improved prioritized sweeping [92], order backups intelligently with the help of a priority queue. Each state in the queue is prioritized based on the potential improvement in value of a backup over that state. In [35], such algorithms have been demonstrated to have large overhead in maintaining a priority queue. They are outperformed by a simple backward search algorithm, which implicitly prioritizes backups without a priority queue. Moreover, prioritized sweeping and improved prioritized sweeping find the optimal value of the entire state space of an MDP, as they do not use the initial state information. Focussed dynamic programming, however, is able to make use of the initial state information, but it is not an optimal algorithm. All three algorithms are massively outperformed by an LAO* variant [35].

6.1.4 State Abstraction

When an MDP maintains a logical representation, another type of algorithm aggregates groups of states of an MDP by features, represents them as a factored MDP using algebraic and Boolean decision diagrams (ADDs and BDDs) and solves the factored MDP using ADD

and BDD operations; SPUDD [62], sLAO* [51], sRTDP [52] are examples. The factored representation can be exponentially simpler than a flat MDP, but the computation efficiency is problem-dependent. The idea of these algorithms are orthogonal to those of (F)TVI. Exploring ways of combining the ideas of (F)TVI with compact logical representation to achieve further performance improvements remains future work.

6.1.5 Action Elimination

Action elimination was originally proposed by Bertsekas [12]. It has been proved to be helpful for RTDP in the factored MDP setting [81], when the cost of an action depends on only a few state variables. Action elimination is also very useful in temporal planning [89]. It has been extended to *combo-elimination*, a rule to prune irrelevant action combinations in a setting when multiple actions can be executed at the same time.

6.1.6 Approximate Algorithms

When an MDP is too large to be solved optimally, another thread of work solves MDPs approximately. The typical way to do this is to use deterministic relaxations of the MDP and/or basis functions [57, 108, 104, 135, 78, 79, 80]. The techniques of these algorithms are orthogonal to the ones by FTVI, and an interesting future direction is approximate FTVI by applying basis functions.

6.1.7 External-Memory Algorithms

EMVI [48] is the first optimal, external-memory MDP algorithm. We have compared PEMVI both theoretically and empirically with EMVI and showed that PEMVI outperforms EMVI. External-memory algorithms for non-deterministic problems are pervasive. See examples in [138, 137]. For external-memory graph algorithms, see [28].

6.1.8 Systematic Partitioning

Wingate and Seppi [133] propose an algorithm, called prioritized partitioned value iteration (PPVI). PPVI statically partitions the state space of non-logical domains into a number

of partition blocks, and backs up states on a block basis. They show that, by using some clever prioritization metrics, the process of value iteration can be greatly sped up. The focus of PPVI, however, is on faster convergence of problems small enough to fit in the main memory. They also share the scalability bottleneck due to limited memory similar to other dynamic programming approaches.

Automatic partitioning classical planning domains was deeply studied by Zhou and Hansen [136], inspired by previous work [47]. Their aim of generating a partition is to reduce the number of the states that are checked in order to find duplicate states during systematic, external-memory searches.

Our automatic partitioning idea is related to *variable resolution* [98, 42, 99, 100], whose motivation is to automatically and dynamically group similar states as an abstract state, solve the abstract MDP, and use the solution to the abstract MDP as an approximation. However, we aim to solve the original MDP optimally, and thus our approach differs from them in a fundamental way. Our emphasis is on creating a partition that minimizes the total time (I/O + backup) for optimal policy construction whereas they aim at partitions that achieves better approximation. However, dynamic partitioning is not I/O efficient, and does not guarantee that the abstract MDP fits in memory.

6.2 Applications for Crowdsourcing

6.2.1 Decision Making

Since crowdsourcing is a recent development, few people have tried using AI or machine learning to control such platforms. Ipeirotis *et al.* [68] observe that workers tend to have bias on multiple-choice annotation tasks. They propose a cost function and apply it to the confusion matrix to filter out spammers, and thereby separate annotation bias from annotation errors. Their model works most effectively on discrete-type classification tasks when there are more than two classes. In contrast, their model assumes workers' errors are completely independent, whereas, our TurKontrol model handles situations where workers make correlated errors due to the intrinsic difficulty of the task.

Huang *et al.* [66] look at the problem of designing HITs under constraints. They concen-

trate their investigation on tagging key words of images. Given budget and time constraints, the objective is to maximize the number of useful tagged labels for a pool of images. They leverage several variables in designing a group of HITs: reward per HIT, number of images per HIT, number of labels requested per image, and the number of HIT, which is constrained by the budget. They experiment on a set of fixed variable configurations. Using the results from the experiments, they learn two types of model between (1) the number of labels and the design variables, and (2) the completion time and the design variables. By wisely setting variables, such as reward per task and the number of labels requested per image, from the learned model, they increase the number of useful tags acquired.

Donmez *et al.* [45] observe that workers' accuracy often changes over time (*e.g.*, due to fatigue, mood, task similarity, *etc.*). They assume answers are completely determined by the (unknown) accuracy, through an HMM model. They use particle filtering to make posterior belief updates of an accuracy model. Simulations show that the new method can effectively predict workers' accuracy, which helps the system choose the most suitable workers and obtain more accurate results. As these approaches are orthogonal to ours, we would like to integrate the methods in the future.

Shahaf and Horvitz [112] develop an HTN-planner style decomposition algorithm to find a coalition of workers, each with different skill sets, to solve a task. Some members of the coalition may be machines and others humans; different skills may command different prices. For example, translating a document from a source language to a destination language sometimes involves translating into an intermediate language or requires an additional iteration of improvement. Therefore, a translation task can be dissembled into a sequence of pipe-lined sub-tasks, where the output of one sub-task becomes the input of its successor, each of which is completed by a human or a program. In a real-world setting, errors are propagated and human resource are limited. Given a set of translation tasks, the problem is to find solutions for all tasks with minimal error. They show the NP-hardness of the problem and solve it with a polynomial, constant-factor approximation algorithm. Their decision engine, *Lingua Mechanica*, achieves better results than using greedy heuristics. In contrast to our work, Shahaf and Horvitz do not consider methods for learning models of their workers.

6.2.2 Incorporating Votes

Whitehill *et al.* [129] study the problem of integrating image annotations by Mechanical Turk workers. Their proposed model is very similar to ours in that they explicitly model each worker's accuracy as well as the difficulty of a task. Their worker model is more general than ours in representing adversarial workers in an elegant way. Given labels, they apply an Expectation-Maximization algorithm to learn the (unknown) parameters. Similar to TurKontrol, their agent, GLAD, integrates human labels from Bayesian updates, based on the learned parameters. Their results align well with our findings that integrating labels in an intelligent way achieves higher accuracy than the majority vote. However, all their parameters are inferred from a black box, with little labeled data. We empirically show that our supervised learning algorithm outperforms their unsupervised learning algorithm. Unlike TurKontrol, which incorporates a principled way to trade off cost versus quality, their system does not make theoretic suggestions in an online decision-making setting.

Welinder *et al.* [128] extend Whitehill *et al.*'s work by modeling images and workers as multi-dimensional entities. As for each worker, their model is able to capture more useful factors, such as personal bias and error tolerance level. They show their new model outperforms previous work. As they solve their model with little supervision, their model may suffer from the same problem as GLAD.

Kern *et al.* [74] study the problem of integrating votes from workers. Their system, CSP-1, integrates workers' results by a weighted majority vote mechanism. Given a confidence target for each task, their system is able to make automatic decisions on whether to elicit one additional label or to stop, if the confidence level is not predicted to be achievable when the upper limit of labels is reached. They demonstrate that the weighted majority vote mechanism achieve confidence threshold faster than plain majority vote. However, they assume an error rate per worker "the same for all tasks of the same type", which does not take into account the difficulties of various tasks of the same type.

6.2.3 User Behavior

To circumvent spamming, Kittur *et al.* [75] study the problem of designing HITs so that workers are more likely to answer them seriously. They find that making a task easily verifiable and setting a task such that spamming is equally hard as answering seriously are the most effective heuristics.

Studies [31, 69] show that financial reward is a primary motivation of Mechanical Turk workers. However, Manson and Watts [88] find that higher payment does not necessarily leads to high-quality answers. One reason is that, as pointed out by Heyman and Ariely [61], volunteers take jobs more out of altruism, so are usually more diligent than workers who take jobs out of reciprocity.

6.2.4 Bonus

Bonus is not a new term form Mechanical Turk, as many requesters have already adopted it. For example, Castingwords [27], a company that recruits workers for audio transcription tasks, pays bonus for excellent work. Castingwords assigns a qualification score (known as the PPT score) to each worker, which indicates how well she performs in the past. The reward of a worker is usually related to her score. With a score of 99 (very high), a worker is allowed to take the most well-paid HITs at a rate of one dollar per audio minute. Unfortunately, the formulas for the PPT score computation are anonymous. As another example, workers on Askville [5], a user-driven crowdsourcing site where users answer questions posted by others, earn bonuses based on how useful their answers are deemed by the askers.

To the best of our knowledge, a bonus decision and its magnitude of existing commercial systems are typically based on evaluations from peer workers. However, no mathematics on calculating the amount of bonus have been published. Further, no known study has been conducted on whether a bonus leads to improvement of overall result quality or recruitment of more competent workers.

Chapter 7
FUTURE WORK

To date, my work on decision making under uncertainty has ranged from designing fundamental new algorithms for large-scale, probabilistic planning problems to applying the latest techniques to challenging, real-world problems. There are many, interesting directions to pursue in the future.

7.1 Optimal Planning

7.1.1 Prioritization methods

To speed up convergence, another thread of work concentrates on prioritizing backups. The priority usually relates to the potential value change of a state, and therefore reflects the urgency to back up a state. However, maintaining a priority queue creates a lot of overhead in many cases [35]. When using a priority queue for external-memory algorithms, an element in the priority queue can refer to a group of states, such as a partition block in the PEMVI algorithm. Applying an intelligent backup order will accelerate convergence, the biggest problem of external-memory algorithms. Two interesting questions are: (1) What should the granularity of prioritization be? and (2) When and how to dynamically adjust the granularity?

7.1.2 Heuristic Search

Heuristic search is a useful tool in expediting the convergence speed of planning algorithms. Using external memory helps enlarge the scope of problems that can be solved by dynamic programming. One direction that appears straightforward is to combine these two approaches. However, the main difficulty is that the two approaches are, by nature, in conflict with each other. A heuristic search algorithm performs state value backups in an intelligence order, suggested by a problem's graphical information. On the other hand, the model of the

problem is usually stored in a hash table, and states that are graphically connected are not always closely located. This can be a problem if states on the disk have to be frequently loaded in memory. To overcome this problem, one might borrow ideas from my work on automated partitioning, trying to group together states that are graphically related.

7.1.3 Crowd-sourced Partitioning

We have demonstrated the usefulness of automatic partitioning for PEMVI. With the availability of human workers, one can utilize human knowledge and consolidate domain-specific knowledge in generating a crowd-sourced partition for PEMVI. We choose to try a crowd-sourced partition because it will bring several benefits where automatic, computer-generated partitions are not able to achieve.

- Human workers have an intuitive access to abundant domain knowledge, good partitioning examples, and can easily construct human understandings of complex concepts, *e.g.*, a valid partition, information flow, *etc.*, and use all information to carry out human reasoning.

- Humans can be more helpful for partitioning numerical domains. In a navigation grid domain such as Wet-Floor, there are usually too many candidate partitions, *e.g.*, any value of x_0 or y_0 can be a candidate partitioning by the XOR formula $\{x \leq x_0 \oplus x > x_0\}$ or $\{y \leq y_0 \oplus y > y_0\}$. However, the most useful ones are those where the resulting partition divides the state space along the x or y axis into equal-sized blocks. This reasoning is typically hard for computers but straightforward for human.

- Crowd-sourced partitioning can be evaluated by human experts or computers, so can be a good complement to automatic partitioning.

There are many other topics about probabilistic planning worth further exploring in the future, such as multi-criteria planning vs. ranking policies with respect to a single criterion [33, 34] and over-subscription planning.

7.2 Crowdsourcing

7.2.1 Long-Term Goal

Our long-term goal is to build a self-adaptive quality control system. The system can benefit from the following components: (1) a dynamic and temporal worker model, (2) an effective incentive system, (3) an automatic way of characterizing, detecting and predicting spammers, and (4) a useful way of transferring the model from one type of task to other types of task. We will elaborate some of the topics in this section.

7.2.2 Handling A Changing Model

We have discussed, in Chapter 5.6, a bootstrapping method of learning an initial worker model for TurKontrol. However, keeping a dynamic model is necessary. One reason is due to the errors in the learning process, caused by biased and insufficient training data, or by local optima in the learning process. Another reason lies in the changes of workers' abilities. As more HITs are taken, workers gain experiences in working with similar HITs, so their competencies may have increased. The key problem is how to dynamically change the control policy to adjust model changes. Our current strategy is to perform offline model updates given new data, *e.g.*, once per week or per batch of tasks. Similar work has been done in learning the user preference model of a recommender system for an online bookstore [115].

Several other approaches can be applied to handle a dynamic model. Reinforcement learning [123] is an on-line learning technique that keeps updating the optimal policy of a planning problem with new information. Model-based learning is one branch of reinforcement learning where the agent keeps an approximate model in memory. Algorithms such as RMax [21] keep refining the model with new information. The optimal policy is inferred by solving the approximate model by using a planning algorithm such as value iteration. Model-based learning for POMDPs has also been studied [113]. To apply a model-based learning algorithm, we plan to start from the bootstrapped model and continuously update it. The other branch of reinforcement learning is model-free learning, where the agent does not keep any model in memory, but tries to infer the best policy from exploration. Policy

learning [124] is one such example. It has been successfully applied in NLP problems [22]. Model-free learning helps get a good policy relatively quickly, but the quality of the policy is usually not as good as model-based learning in the long run. We plan to try both methods in parallel and investigate which one works better in practice.

7.2.3 Implementing Incentives

To build a more powerful quality control system, we plan to study the effects of incentives in the future. An incentive [130] refers to "any factor (financial or non-financial) that enables or motivates a particular course of action, or counts as a reason for preferring one choice to the alternatives. It is an expectation that encourages people to behave in a certain way". As Mechanical Turk has provided us with an ample workforce, our objectives in providing an incentive are less about attracting a larger amount of workers, but more about hiring qualified and talented workers.

Paying A Bonus

We first explore a potential bonus system. *Mechanism design* is a research area that aims at setting rules, such as determining the price amount for participating players, for game-like situations. Consider each worker as a game player who is smart and has complete knowledge of the game. She acts strategically, and is able to take advantage of the available information. We, the requesters, set our anticipation of the game. A mechanism designer is an agent whose job is to design a game to achieve the anticipated goal.

To validate the applicability of mechanism design, we first note workers indeed act *strategically* in handling paid jobs [6, 71] and in reacting to all-pay auctions such as Witkey websites [44, 134]. One user behavior study on Taskcn [125], a Witkey website, shows that a large proportion of winning submissions are completed by a small group of users (winners) and the winners' behaviors are different from other workers. For example, Yang et al. [134] point out that the winners tend to take part in jobs whose winning rate is higher (where less people compete). Winners also submit jobs late to get a better idea of the winning odds.

Next we investigate whether a bonus might have an impact on worker behavior. As

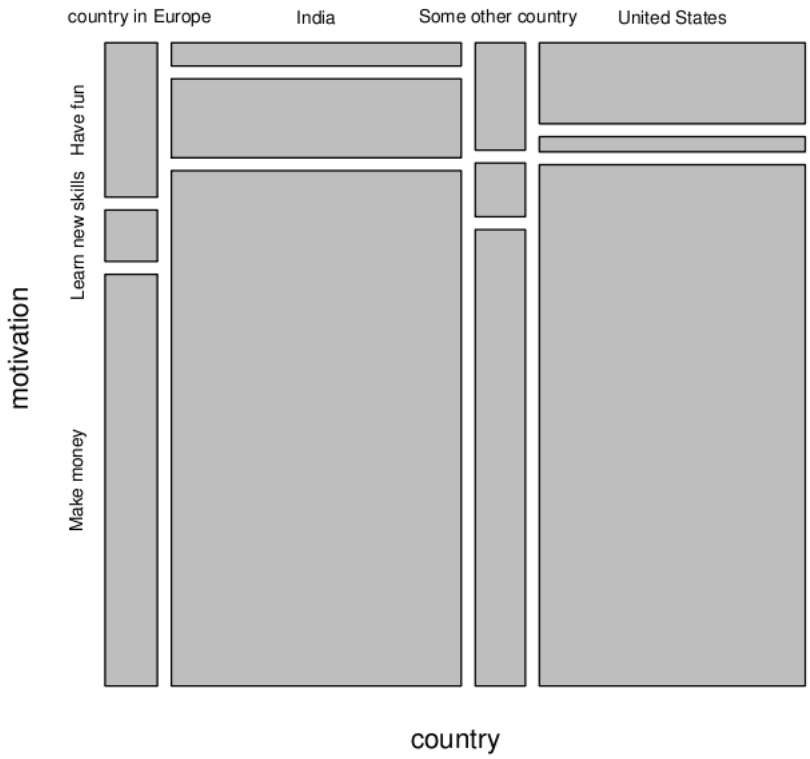


Figure 7.1: Motivation of workers on Mechanical Turk across countries. The primary motivation of all workers is to make money. Figure courtesy [31]

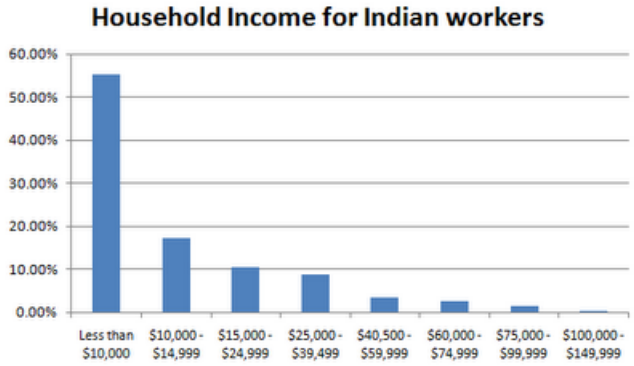


Figure 7.2: Household income of workers of Mechanical Turk from India. More than half of the workers have an annual income of under $10,000. Figure courtesy [69]

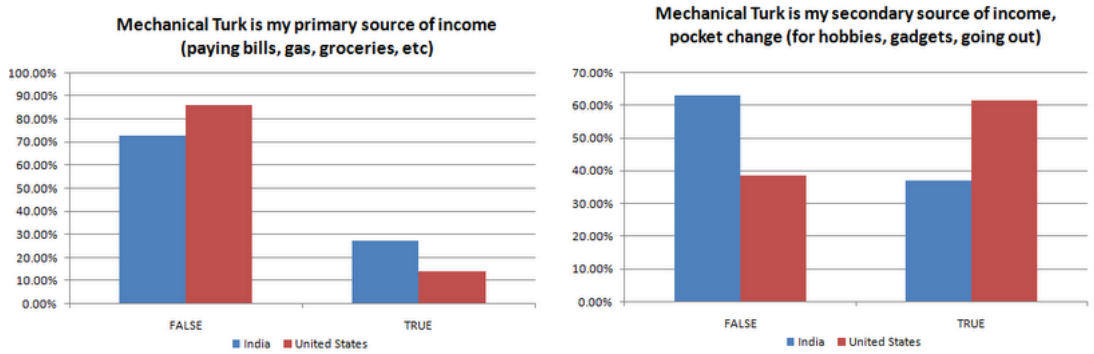


Figure 7.3: Workers whose primary and secondary household income is Amazon Mechanical Turk. The primary or secondary household income of around 70% of the workers are from Amazon Mechanical Turk. Figures courtesy [69]

Mechanical Turk workers are usually poorly paid compared to a classic job [76], it is natural to argue that a bonus might not stimulate workers' interests. This is true as study show that many workers are middle-classed and work mainly for recreational purposes [96, 88]. However, recent study reinforces that monetary reward is the primary motivation of the majority of workers on Mechanical Turk [31]. See Figure 7.1 for a distribution of the primary motivation of Mechanical Turk workers. Comparing to three years ago, when over 70% of the workers on Mechanical Turk are from the US, a recent survey on the demographics of workers [69] shows that the proportion of workers from the United States has dropped to below 50%. Instead, the percentage of workers from India has increased to 34%. Among the workers from India, 50% of them have a household income of less than $10,000 per year. Over 25% of the population Mechanical Turk is their primary source of income. If counting the secondary source of income the percentage is above 60%. Due to the discrepancy in labor values across geometry, payment rates on Mechanical Turk appear to remain competitive against full-time jobs in many countries.

As Mechanical Turk is not a mature labor market, some requesters have taken free rides by having workers do the jobs and later rejecting their submissions. This, somehow, discourages workers. To fight against these requesters, workers have built a website named Turker Nation [127], as an information exchange forum. Workers share their experiences in

interacting with various requesters. Some requesters also respond to workers' inquiries. On the one hand, information sharing helps better protect workers' rights. On the other hand, it is an excellent advertising venue for requesters with good reputations, such as fast pay, generous bonus, open to criticisms, *etc.*

Another question is whether workers are one-shot or long-term players. As HITs are contract-less, it is natural to think workers are unstable. However, if workers like a task they do many more of the same type. Nathan McFarland, co-founder of Castingwords, claimes that "They (workers) like to do one task they do very well and just keep doing it. If they find a HIT they like, they will come back day after day." [4]

All evidence tends to suggest that a bonus might be helpful in inspiring talented workers to build a long-term collaboration relationship with well-behaved requesters.

We propose one way of paying a bonus. Note one feature of iterative improvement is that the evaluation is strict. This is because the quality of an artifact is not only explicitly evaluated by ballots, but also implicitly evaluated by improvements in later iterations. Accordingly, the evaluation cost of our system is higher. For example, we may need to perform a few iterations of unsuccessful improvement before confirming that the current artifact is already very good. The evaluation cost of an artifact, in this sense, not only includes the the ballot costs in the same iteration, but also the costs for the unsuccessful improvements. As a result, we do not get much higher utility when receiving an exceptional artifact in an early iteration, as much evaluation costs will have to be spent later. Therefore, TurKontrol prefers a worker with higher competency than a exceptional work from an average worker. With this consideration, our bonus system should target on competent workers.[1]

The basic idea is that a bonus comes from our utility gains brought by an exceptional worker. From the worker-independent model (γ and $f_{Q'|q}$) one can calculate the expected gross utility u, the expected number of improvement HITs n_{imp} and ballot HITs n_b of one iterative process, when all participating workers are mediocre. For an exceptional worker x (with model γ_x, $f_{Q'|q,x}$), we compute $u(x)$, $n_{imp}(x)$ and $n_b(x)$ in the same way. The amount

[1]Although our bonus system focuses on good workers, the mathematics is general and can be easily extended to cover bonus on exceptional improvements as well.

of extra expected utility is:

$$\Delta U = [u(x) - c_{imp} \times n_{imp}(x) - c_b \times n_b(x)] - [u - c_{imp} \times n_{imp} - c_b \times n_b] \tag{7.1}$$

As a bonus, we pay back a proportion $\phi(< 1)$ of ΔU to worker x. We define

$$\theta = \frac{\Delta U \times \phi}{c_{imp} \times n_{imp}(x) + c_b \times n_b(x)}. \tag{7.2}$$

For worker x, we offer to pay a bonus of $\theta \times c_{imp}$ for each improvement HIT and $\theta \times c_b$ for each ballot HIT.

Other Incentive Types

Besides financial incentives such as a bonus, there are two other kinds of incentives worth investigation in the future.

The first one involves recognition. We design several leader boards, such as workers who have completed the greatest number of HITs, achieved the highest accuracies, and earned the most money, *etc.* We also plan to design personalized leader boards, *e.g.*, how much money a worker has earned relative to all her acquaintances. The goal is to stimulate workers to work hard and to compete against their colleagues and friends, in return for social satisfaction and accomplishments. Online games on social networking website such as Facebook [49] have designed many types of personalized leader boards for players, such as people who have received the largest volume of gifts during some activity. Users can publish their ranking information when they surpass a friend. From personal observation, many people like this feature and the leader boards encourage them competing against each other.

The second is coercive incentives, or a penalty system. We plan to blacklist workers who have made an unreasonable amount of mistakes, or been caught to have provided random answers or even malicious answers. Identifying those characteristics is challenging, since human may have personal strategies to game the system. We plan to try to model some representative worker types and categorize them using machine learning techniques.

Evaluation

Previous studies do not reach a consensus on whether higher reward helps improve the quality of a job [31, 69, 88]. A bonus can be regarded as a different form of (unguaranteed) reward. The two questions we would like to answer are: (1) Does any of the incentive methods work? (2) Which method works the best? We plan to answer these questions through empirical study.

Our first experiment is to try all mechanisms independently (or potentially jointly). We create multiple requesters, each of which presents the same or similar set of tasks while applies different (combination of) bonus mechanisms. Note that our baseline is a requester that does not offer any incentives. We evaluate the performance of all requesters after a period of time. Our performance metric should be built upon the following statistics: (1) the average quality per job, (2) the average accuracy/performance of participating workers, and (3) the proportion of exceptional participating workers.

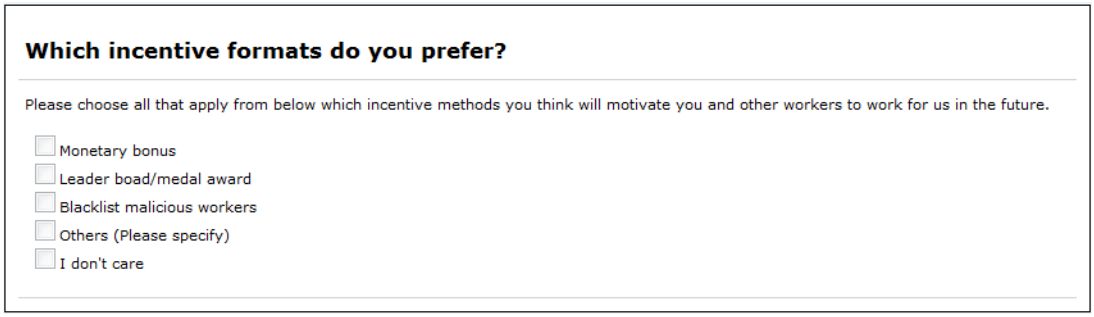


Figure 7.4: A simple survey example that asks workers which incentive methods they prefer.

Another evaluation method is online survey. We randomly pick workers who have taken our HITs for a survey question. The question presents all possible types of incentives, and asks them to pick the ones that will motivate them to work for more tasks in the future. Figure 7.4 shows an example of the survey. Notice that we care more about good workers, so in analyzing answers, we plan to give higher weights to workers with better performances.

Other Factors

To devise a successful incentive mechanism, a few other factors are also worth our attention. In a user study [70] on how workers react to a game of the Prisoners' Dilemma, Ipeirotis finds that workers do not act *optimally* as expected from applying the game theory. This could possibly due to the fact that a penalty is implemented in the form a reward, which disguises the essence of the game. But the implication is that there maybe other factors than the amount of reward in influencing workers' decision making process, such as risk-seeking, sub-optimality in reasoning, *etc.* Pointed out in studies [88, 96], recreational effects serve as good motivations for a worker to take a HIT. How to design a HIT and make it more interesting is another topic worth investigating. Horton and Chilton [64] discover an interesting *target effect* on the workers. They find that workers may set up some reward thresholds, achieving which makes them more likely to quit working. To overcome this effect, we can set the bonus point half-way between two thresholds, so as to encourage a worker achieving a higher threshold.

7.2.4 Model Transfer

Ultimately our quality control system should be applicable for various types of tasks. How to quickly and efficiently learn a worker model for a new task type is an imminent and challenging problem. Using a small amount of supervised data as a bootstrapping step is the solution we followed for the image description task, but it is potentially very expensive as it incurs a new set of expert labels for every task type. As an alternative, we plan to start the model for a new task from the model of an existing and similar task, *e.g.*, writing a movie review is a similar task of writing an image description. Later we plan to apply reinforcement learning techniques (discussed in Chapter 7.2.2) to update the model given new data.

7.3 New Applications

We are consistently looking forward to expanding our focus to new applications. One possible application is to find good playing strategies for multi-player games. It is very

difficult to solve multi-player games since the problem space grows exponentially in the number of players. We are interested in generating efficient representations and finding high-quality heuristics for such problems. Another potential area is in social networks. Recently there are an increasing number of companies trying to promote business using social networking websites (Facebook, Twitter, LinkedIn, *etc*). Due to the complex structure and the uncertainty (because of incomplete information) of the social networks, it is difficult for companies to come out with efficient strategies to target potential customers and handle various types of customers. Singla and Richardson [117] find that people who interact with each other more frequently on social networks often share similar interests. This information can be useful for building a more informed crowd-sourced worker model. We are interested in how decision making under uncertainty can be applied in these problems.

Chapter 8
CONCLUSIONS

Decision making under uncertainty is a very general and critical topic in Artificial Intelligence. Its representation ability covers almost all real-world decision making problems. Yet, its applicability has been largely limited by the scale of solvable problems. This thesis work tackles two major problems in the area of decision making under uncertainty. Firstly, it advances the state of the art of optimal planning by proposing new algorithms and improves its scalability. Secondly, it applies existing planning and machine learning techniques to the quality control problem of crowdsourcing, a new and rapidly-growing notion and platform, and demonstrates the usefulness of decision theory in a real-world setting, by controlling a dynamic, crowd-sourced workflow that successfully achieves statistically-significant, high-quality results than a traditional, non-adaptive workflow. We list the contributions of this work in more details.

8.1 Optimal Planning

1. We present a new optimal algorithm to solve MDPs, focused topological value iteration (FTVI). FTVI extends the topological value iteration (TVI) algorithm by focusing the construction of strongly connected components on transitions that likely belong to an optimal policy. FTVI does this by using a small amount of heuristic search to eliminate provably suboptimal actions. In contrast to TVI, which does not care about goal-state information, FTVI removes transitions that it determines to be irrelevant to an optimal policy for reaching the goal. In this sense, FTVI builds a much more informative topological structure than TVI.

2. We show empirically that FTVI outperforms TVI in a large number of domains, usually by an order of magnitude. This performance is due to the success of a more informed graphical structure, since the sizes of the connected components found by

FTVI are vastly smaller than those constructed by TVI's.

3. We find surprisingly that for many domains FTVI massively outperforms popular heuristic search algorithms, such as ILAO*, LRTDP, BRTDP and BaRTDP. After analyzing the performance of these algorithms over different problems, we find that a smaller number of goal states and long search depth to a goal are two key features of problems that are especially hard for heuristic search to handle. Our results show that FTVI outperforms heuristic search algorithms in such domains by an order of magnitude.

4. As a by-product we also compare ILAO*, LRTDP, BRTDP and BaRTDP (four popular, state-of-the-art heuristic search algorithms) and find that the strength of each algorithm is usually domain-specific. Generally, ILAO* is faster than other algorithms. BRTDP and BaRTDP are slow in some domains probably due to the fact that they are vulnerable to those problems' lack of informed upper bounds.

5. We present a novel optimal algorithm, partitioned external-memory value iteration (PEMVI). By partitioning a problem's state space and keeping them on disk, PEMVI may load the MDP model piecemeal into memory, and perform Bellman backups in an I/O-efficient manner. Our experiments demonstrate that PEMVI can solve problems much too large for internal-memory algorithms. Moreover, PEMVI converges an order of magnitude faster than the external-memory value iteration (EMVI) algorithm, a contemporary external-memory algorithm, since it has the ability to perform several backups in a single I/O scan and does not require continual sorting.

6. We demonstrate the practicality of automatically partitioning probabilistic planning problems by implementing a completely autonomous, domain-independent partitioning algorithm, which helps PEMVI optimally handle as yet unsolved problems in most IPC-06 domains. Our partitioning algorithm uses a novel approach for generating XOR groups, which generalizes those of [47, 136] and finds more, useful XOR formulae in practice.

7. In addition to the locality heuristic used by Zhou and Hansen [136], we propose two new heuristic, *coherence* and *balance*, for guiding the search for partitions and introduce a sampling method for its efficient computation. Our empirical study successfully shows that automatic partitioning can beat manual methods and demonstrates the utility of simple-negated groups.

8.2 Crowdsourcing

1. As complex workflows have become a commonplace in crowd-sourcing and are regularly employed for high-quality output, we introduce an exciting new application for artificial intelligence — control of crowd-sourced workflows.

2. We use decision theory, a partially-observable Markov decision process (POMDP) in particular, to model a popular class of iterative workflows and define equations that govern the various steps of the process. Our agent, TurKontrol, implements our mathematical framework and uses it to optimize and control the workflow. Extensive simulations show that TurKontrol is robust in a variety of scenarios and parameter settings, and results in higher utilities than previous, non-adaptive policies.

3. We present an efficient and cheap mechanism, as a bootstrapping step, to learn the parameters of the proposed POMDP model from limited, noisy training data. We validate the parameters independently and show that our learned model of worker accuracy significantly outperforms the popular majority-vote baseline when resources are constrained.

4. Our work conclusively demonstrates the benefits of AI, specifically decision-theoretic techniques, in controlling crowd-sourced workflows. We demonstrate the effectiveness of the decision-theoretic techniques, using an end-to-end system on a live crowdsourcing platform for the task of writing image descriptions. With our learned models guiding the controller, the dynamic workflows are vastly superior than non-adaptive workflows utilizing the same amount of money. Our results are statistically significant.

We investigate the qualitative behavior of our agent, illustrating interesting characteristics of iterative workflows on Mechanical Turk. Our agent is able to continuously improve its performance, with arrival of new data and off-line model reinforcement.

BIBLIOGRAPHY

[1] Pieter Abbeel, Adam Coates, Timothy Hunter, and Andrew Y. Ng. Autonomous autorotation of an rc helicopter. In *ISER*, pages 385–394, 2008.

[2] Pieter Abbeel, Dmitri Dolgov, Andrew Y. Ng, and Sebastian Thrun. Apprenticeship learning for motion planning with application to parking lot navigation. In *IROS*, pages 1083–1090, 2008.

[3] Douglas Aberdeen, Sylvie Thiébaux, and Lin Zhang. Decision-theoretic military operations planning. In *Proc. of the 14th International Conference on Automated Planning and Scheduling (ICAPS-04)*, pages 402–412, 2004.

[4] Amazon. Mechanical turk case study: Castingwords. `http://aws.amazon.com/solutions/ case-studies/castingwords-interview`.

[5] Askville. `http://askville.amazon.com`.

[6] Maria-Florina Balcan, Avrim Blum, and Yishay Mansour. The price of uncertainty. In *EC: Proceedings of the tenth ACM conference on Electronic commerce*, pages 285–294, 2009.

[7] Radha-Krishna Balla and Alan Fern. UCT for tactical assault planning in real-time strategy games. In *IJCAI*, pages 40–45, 2009.

[8] A.G. Barto, S.J. Bradtke, and S.P. Singh. Learning to act using real-time dynamic programming. *Artificial Intelligence J.*, 72:81–138, 1995.

[9] R. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, 1957.

[10] Daniel S. Bernstein, Robert Givan, Neil Immerman, and Shlomo Zilberstein. The complexity of decentralized control of markov decision processes. *Mathematics of Opererations Research*, 27(4):819–840, 2002.

[11] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control, Vol 1, 2nd ed.* Athena Scientific, 2000.

[12] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*, volume 2. Athena Scientific, 2000-2001.

[13] Dimitri P. Bertsekas. *Dynamic Programming and Optimal Control*, volume 2. Athena Scientific, 2 edition, 2001.

[14] Dimitri P. Bertsekas and John N. Tsitsiklis. *Neuro-Dynamic Programming*. Athena Scientific, Belmont, MA, 1996.

[15] Jeffrey P. Bigham, Chandrika Jayant, Hanjie Ji, Greg Little, Andrew Miller, Robert C. Miller, Robin Miller, Aubrey Tatarowicz, Brandyn White, Samuel White, and Tom Yeh. Vizwiz: nearly real-time answers to visual questions. In *UIST*, pages 333–342, 2010.

[16] B. Bonet and H. Geffner. Faster heuristic search algorithms for planning with uncertainty and full feedback. In *Proc. of 18th International Joint Conf. on Artificial Intelligence (IJCAI-03)*, pages 1233–1238. Morgan Kaufmann, 2003.

[17] B. Bonet and H. Geffner. Labeled RTDP: Improving the convergence of real-time dynamic programming. In *Proc. 13th International Conference on Automated Planning and Scheduling (ICAPS-03)*, pages 12–21, 2003.

[18] Blai Bonet. On the speed of convergence of value iteration on stochastic shortest-path problems. *Mathematics of Operations Research*, 32(2):365–373, 2007.

[19] Blai Bonet and Hector Geffner. mgpt: A probabilistic planner based on heuristic search. *J. Artif. Intell. Res. (JAIR)*, 24:933–944, 2005.

[20] Blai Bonet and Hector Geffner. Learning in depth-first search: A unified approach to heuristic search in deterministic non-deterministic settings, and its applications to MDPs. In *Proc. of the 16th International Conference on Automated Planning and Scheduling (ICAPS-06)*, pages 142–151, 2006.

[21] Ronen I. Brafman and Moshe Tennenholtz. R-max - a general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3:213–231, 2001.

[22] S.R.K. Branavan, Harr Chen, Luke S. Zettlemoyer, and Regina Barzilay. Reinforcement learning for mapping instructions to actions. In *ACL-IJCNLP*, 2009.

[23] John L. Bresina, Richard Dearden, Nicolas Meuleau, Sailesh Ramkrishnan, David E. Smith, and Richard Washington. Planning under continuous time and resource uncertainty: A challenge for AI. In *Proc. of 18th Conf. in Uncertainty in AI (UAI-02)*, pages 77–84, 2002.

[24] Emma Brunskill, Leslie Kaebling, Tomas Lozano-Perez, and Nicholas Roy. Continuous-state pomdps with hybrid dynamics. In *Symposium on Artificial Intelligence and Mathematics*, 2008.

[25] World's outsourcing market worth $373 billion, September 2009. http://www.businessweek.com/globalbiz/content/sep2009/gb20090925_476872.htm.

[26] Anthony R. Cassandra. A survey of pomdp applications. In *AAAI Fall Symposium: Planning with POMDPs*, 1998.

[27] Castingwords. http://castingwords.com.

[28] Yi-Jen Chiang, Michael T. Goodrich, Edward F. Grove, Roberto Tamassia, Darren Erik Vengroff, and Jeffrey Scott Vitter. External-memory graph algorithms. In *SODA*, pages 139–149, 1995.

[29] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, 2001.

[30] Dan Cosley, Shyong K. Lam, Istvan Albert, Joseph A. Konstan, and John Riedl. Is seeing believing?: how recommender system interfaces affect users' opinions. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 585–592, 2003.

[31] crowdflower.com. Crowdflower research board. http://blog.crowdflower.com/.

[32] Peng Dai and Judy Goldsmith. Topological value iteration algorithm for Markov decision processes. In *Proc. of IJCAI*, pages 1860–1865, 2007.

[33] Peng Dai and Judy Goldsmith. Finding best k policies. In *Proc. of ADT*, pages 144–155, 2009.

[34] Peng Dai and Judy Goldsmith. Finding best k policies. 59(1):107–123, 2010.

[35] Peng Dai and Eric A. Hansen. Prioritizing Bellman backups without a priority queue. In *Proc. of the 17th International Conference on Automated Planning and Scheduling (ICAPS-07)*, pages 113–119, 2007.

[36] Peng Dai, Mausam, and Daniel S. Weld. Partitioned external-memory value iteration. In *AAAI*, pages 898–904, 2008.

[37] Peng Dai, Mausam, and Daniel S. Weld. Domain-independent, automatic partitioning for probabilistic planning. In *IJCAI*, pages 1677–1683, 2009.

[38] Peng Dai, Mausam, and Daniel S. Weld. Focused topological value iteration. In *Proc. of ICAPS*, pages 82–89, 2009.

[39] Peng Dai, Mausam, and Daniel S. Weld. Focused topological value iteration. In *Proc. of ICAPS*, pages 82–89, 2009.

[40] Peng Dai, Mausam, and Daniel S. Weld. Decision-theoretic control of crowd-sourced workflows. In *AAAI*, 2010.

[41] Peng Dai, Mausam, and Daniel S. Weld. Artificial intelligence for artificial artificial intelligence. In *AAAI*, 2011.

[42] Thomas Dean, Robert Givan, and Sonia Leach. Model reduction techniques for computing approximately optimal solutions for Markov decision processes. In *UAI*, 1997.

[43] Jilles Steeve Dibangoye, Guy Shani, Brahim Chaib-draa, and Abdel-Illah Mouaddib. Topological order planner for pomdps. In *Proc. of IJCAI*, pages 1684–1689, 2009.

[44] Dominic DiPalantino and Milan Vojnovic. Crowdsourcing and all-pay auctions. In *EC: Proceedings of the tenth ACM conference on Electronic commerce*, pages 119–128, 2009.

[45] Pinar Donmez, Jaime G. Carbonell, and Jeff Schneider. A probabilistic framework to learn from multiple annotators with time-varying accuracy. In *SIAM International Conference on Data Mining (SDM)*, pages 826–837, 2010.

[46] Arnaud Doucet, Nando de Freitas, and Neil Gordon. *Sequential Monte Carlo Methods in Practice*. Springer, 2001.

[47] Stefan Edelkamp and Malte Helmert. Exhibiting knowledge in planning problems to minimize state encoding length. In *ECP*, pages 135–147, 1999.

[48] Stefan Edelkamp, Shahid Jabbar, and Blai Bonet. External memory value iteration. In *Proc. of the 17th International Conference on Automated Planning and Scheduling (ICAPS-07)*, pages 128–135, 2007.

[49] Facebook, June 2011. `http://www.facebook.com`.

[50] Zhengzhu Feng, Richard Dearden, Nicolas Meuleau, and Richard Washington. Dynamic programming for structured continuous markov decision problems. In *UAI*, pages 154–161, 2004.

[51] Zhengzhu Feng and Eric A. Hansen. Symbolic heuristic search for factored Markov decision processes. In *Proc. of the 17th National Conference on Artificial Intelligence (AAAI-05)*, 2002.

[52] Zhengzhu Feng, Eric A. Hansen, and Shlomo Zilberstein. Symbolic generalization for on-line planning. In *Proc. of the 19th Conference in Uncertainty in Artificial Intelligence (UAI-03)*, pages 209–216, 2003.

[53] Zhengzhu Feng and Shlomo Zilberstein. Region-based incremental pruning for pomdps. In *Proc. of UAI*, pages 146–153, 2004.

[54] Dave Ferguson and Anthony Stentz. Focussed dynamic programming: Extensive comparative results. Technical Report CMU-RI-TR-04-13, Carnegie Mellon University, Pittsburgh, PA, March 2004.

[55] Brent Frei. Paid crowdsourcing, current state & progress. Technical report, 2009.

[56] Sylvain Gelly and Yizao Wang. Exploration exploitation in go: UCT for monte-carlo go. In *NIPS On-line trading of Exploration and Exploitation Workshop*, 2006.

[57] Carlos Guestrin, Daphne Koller, Ronald Parr, and Shobha Venkataraman. Efficient solution algorithms for factored MDPs. *J. of Artificial Intelligence Research*, 19:399–468, 2003.

[58] Eric A. Hansen and Shlomo Zilberstein. LAO*: A heuristic search algorithm that finds solutions with loops. *Artificial Intelligence J.*, 129:35–62, 2001.

[59] Milos Hauskrecht, Nicolas Meuleau, Leslie Pack Kaelbling, Thomas Dean, and Craig Boutilier. Hierarchical solution of Markov decision processes using macro-actions. In *Proc. of UAI*, pages 220–229, 1998.

[60] Malte Helmert and Gabriele Röger. How good is almost perfect? In *Proc. of AAAI*, pages 944–949, 2008.

[61] J Heyman and D. Ariely. Effort for payment: A tale of two markets. *Psychological Science*, 15(11):787–793, 2004.

[62] Jesse Hoey, Robert St-Aubin, Alan Hu, and Craig Boutilier. SPUDD: Stochastic planning using decision diagrams. In *Proc. of the 15th Conference on Uncertainty in Artificial Intelligence (UAI-95)*, pages 279–288, 1999.

[63] Leah Hoffmann. Crowd control. *CACM*, 52(3):16–17, 2009.

[64] John Horton and Lydia Chilton. The labor economics of paid crowdsourcing. *CoRR*, abs/1001.0627, 2010.

[65] R.A. Howard. *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, Massachusetts, 1960.

[66] Eric Huang, Haoqi Zhang, David C. Parkes, Krzysztof Z. Gajos, and Yiling Chen. Toward automatic task design: A progress report. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, pages 77–85, 2010.

[67] 2006. `http://www.ldc.usb.ve/~bonet/ipc5/`.

[68] Panagiotis G. Ipeirotis, Foster Provost, and Jing Wang. Quality management on amazon mechanical turk. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, pages 64–67, 2010.

[69] Panos Ipeirotis. Demographics of mechanical turk. In *Working Paper*, 2010.

[70] Panos G. Ipeirotis. A computer scientist in a business school, 2010. http://behind-the-enemy-lines.blogspot.com/.

[71] Radu Jurca and Boi Faltings. Incentives for expressing opinions in online polls. In *EC: Proceedings of the 9th ACM conference on Electronic commerce*, pages 119–128, 2008.

[72] Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artif. Intell.*, 101(1-2):99–134, 1998.

[73] S. M. Kakade. *On the sample complexity of reinforcement learning.* PhD thesis, Gatsby Computational Neuroscience Unit, University College London, 2003.

[74] Robert Kern, Hannes Thies, and Gerhard Satzger. Statistical quality control for human-based electronic services. In *In Proc. of ICSOC*, pages 243–257, 2010.

[75] Aniket Kittur, Ed H. Chi, and Bongwon Suh. Crowdsourcing user studies with mechanical turk. In *CHI*, pages 453–456, 2008.

[76] Frank Kleemann, Gerd Gnter Vo, and Kerstin Rieder. Un(der)paid innovators: The commercial utilization of consumer work through crowdsourcing. *Science, Technology & Innovation Studies*, 4(1):5–26, 2008.

[77] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *ECML*, pages 282–293, 2006.

[78] Andrey Kolobov, Mausam, and Daniel S. Weld. ReTrASE: intergating paradigms for approximate probabilistic planning. In *Proc. of IJCAI*, pages 1746–1753, 2009.

[79] Andrey Kolobov, Mausam, and Daniel S. Weld. Classical planning in MDP heuristics: With a little help from generalization. In *Proc. of ICAPS*, pages 97–104, 2010.

[80] Andrey Kolobov, Mausam, and Daniel S. Weld. SixthSense: Fast and reliable recognition of dead ends in MDPs. In *Proc. of AAAI*, 2010.

[81] Ugur Kuter and Jiaqiao Hu. Computing and using lower and upper bounds for action elimination in mdp planning. In *SARA*, pages 243–257, 2007.

[82] Lihong Li and Michael L. Littman. Lazy approximation for solving continuous finite-horizon mdps. In *AAAI*, pages 1175–1180, 2005.

[83] Zhifei Li, Patrick Nguyen, and Geoffrey Zweig. Optimal dialog in consumer-rating systems using a pomdp framework. In *SIGdial: Proceedings of the 9th SIGdial Workshop on Discourse and Dialogue*, pages 104–111, 2008.

[84] Greg Little, Lydia B. Chilton, Max Goldman, and Robert C. Miller. Turkit: tools for iterative tasks on mechanical turk. In *KDD Workshop on Human Computation*, pages 29–30, 2009.

[85] Michael L. Littman, Thomas Dean, and Leslie Pack Kaelbling. On the complexity of solving Markov decision problems. In *Proc. of the 11th Annual Conference on Uncertainty in Artificial Intelligence (UAI-95)*, pages 394–402, Montreal, Quebec, Canada, 1995.

[86] Contact center in the cloud, December 2009. `http://liveops.com`.

[87] Omid Madani, Steve Hanks, and Anne Condon. On the undecidability of probabilistic planning and related stochastic optimization problems. *Artificial Intelligence*, 147(1-2):5–34, 2003.

[88] Winter Mason and Duncan J. Watts. Financial incentives and the "performance of crowds". In *HCOMP: Proceedings of the ACM SIGKDD Workshop on Human Computation*, pages 77–85, 2009.

[89] Mausam. *Stochastic Planning with Concurrent, Durative Actions*. PhD thesis, University of Washington, Seattle, 2007.

[90] Mausam, Emmanuel Benazera, Ronen I. Brafman, Nicolas Meuleau, and Eric A. Hansen. Planning with continuous resources in stochastic domains. In *Proc. of IJCAI*, pages 1244–1251, 2005.

[91] Mausam, Emmanuel Benazera, Ronen I. Brafman, Nicolas Meuleau, and Eric A. Hansen. Planning with continuous resources in stochastic domains. In *IJCAI*, pages 1244–1251, 2005.

[92] H. Brendan McMahan and Geoffrey J. Gordon. Fast exact planning in Markov decision processes. In *Proc. of the 15th International Conference on Automated Planning and Scheduling (ICAPS-05)*, 2005.

[93] H. Brendan McMahan, Maxim Likhachev, and Geoffrey J. Gordon. Bounded real-time dynamic programming: RTDP with monotone upper bounds and performance guarantees. In *Proceedings of the 22nd international conference on Machine learning (ICML-05)*, pages 569–576, 2005.

[94] Mechanical turk is a marketplace for work, December 2009. `http://www.mturk.com/mturk/welcome`.

[95] Nicolas Meuleau, Emmanuel Benazera, Ronen I. Brafman, Eric A. Hansen, and Mausam. A heuristic search approach to planning with continuous resources in stochastic domains. *J. of Artificial Intellegence Research (JAIR)*, 34:27–59, 2009.

[96] Katharine Mieszkowski. I make $1.45 a week and i love it, July 2006. `http://www.salon.com/technology/feature/2006/07/24/turks`.

[97] Andrew Moore and Chris Atkeson. Prioritized sweeping: Reinforcement learning with less data and less real time. *Machine Learning*, 13:103–130, 1993.

[98] Andrew W. Moore and Christopher G. Atkeson. The parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning*, 21:199–233, 1995.

[99] Rémi Munos and Andrew Moore. Rates of convergence for variable resolution schemes in optimal control. In *ICML*, pages 647–654, 2000.

[100] Rémi Munos and Andrew Moore. Variable resolution discretization in optimal control. *Machine Learning*, 49(2-3):291–323, 2002.

[101] D. J. Musliner, J. Carciofini, R. P. Goldman, J. Wu E. H. Durfee, and M. S. Boddy. Flexibly integrating deliberation and execution in decision-theoretic agents. In *ICAPS Workshop on Planning and Plan-Execution for Real-World Systems*, 2007.

[102] Nils J. Nilson. *Principles of Artificial Intelligence*. Tioga Publishing Company, Palo Alto, Ca., 1980.

[103] Ronald Parr. Flexible decomposition algorithms for weakly coupled Markov decision problems. In *Proc. of UAI*, pages 422–430, 1998.

[104] Relu Patrascu, Pascal Poupart, Dale Schuurmans, Craig Boutilier, and Carlos Guestrin. Greedy linear value-approximation for factored Markov decision processes. In *Proc. of the 17th National Conference on Artificial Intelligence (AAAI-02)*, pages 285–291, 2002.

[105] Joelle Pineau, Geoffrey Gordon, and Sebastian Thrun. Anytime point-based approximations for large pomdps. *J. Artificial Intelligence Research*, 27(1):335–380, 2006.

[106] Josep M. Porta, Matthijs T. J. Spaan, and Nikos A. Vlassis. Robot planning in partially observable continuous domains. In *Proc. Robotics: Science and Systems*, pages 217–224, 2005.

[107] Josep M. Porta, Nikos Vlassis, Matthijs T.J. Spaan, and Pascal Poupart. Point-based value iteration for continuous pomdps. *J. of Machine Learning Research*, 7:2329–2367, 2006.

[108] Pascal Poupart, Craig Boutilier, Relu Patrascu, and Dale Schuurmans. Piecewise linear value function approximation for factored MDPs. In *Proc. of the 18th National Conference on Artificial Intelligence (AAAI-02)*, pages 292–299, 2002.

[109] John A. Rice. *Mathematical Statistics and Data Analysis*. 2001.

[110] Stuart Russell and Eric Wefald. *Do the Right Thing*. MIT Press, Cambridge, MA, 1991.

[111] Scott Sanner, Robby Goetschalckx, Kurt Driessens, and Guy Shani. Bayesian real-time dynamic programming. In *Proc. of IJCAI*, pages 1784–1789, 2009.

[112] Dafna Shahaf and Eric Horvitz. Generlized markets for human and machine computation. In *AAAI*, 2010.

[113] Guy Shani, Ronen I. Brafman, and Solomon E. Shimony. Model-based online learning of pomdps. In *ECML*, pages 353–364, 2005.

[114] Guy Shani, Ronen I. Brafman, and Solomon Eyal Shimony. Prioritizing point-based pomdp solvers. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 38(6):1592–1605, 2008.

[115] Guy Shani, David Heckerman, and Ronen I. Brafman. An mdp-based recommender system. *Journal of Machine Learning Research*, 6:1265–1295, 2005.

[116] Reid Simmons and Sven Koenig. Probabilistic robot navigation in partially observable environments. In *Proceedings of IJCAI*, pages 1080–1087, 1995.

[117] Parag Singla and Matthew Richardson. Yes, there is a correlation: - from social networks to personal behavior on the web. In *In Proc. of WWW*, pages 655–664, 2008.

[118] Trey Smith and Reid G. Simmons. Focused real-time dynamic programming for mdps: Squeezing more out of a heuristic. In *Proc. of the 21th National Conference on Artificial Intelligence (AAAI-06)*, 2006.

[119] Rion Snow, Brendan O'Connor, Daniel Jurafsky, and Andrew Y. Ng. Cheap and fast - but is it good? evaluating non-expert annotations for natural language tasks. In *EMNLP*, pages 254–263, 2008.

[120] Edward J. Sondik. *The Optimal Control of Partially Observable markov Processes.* PhD thesis, Stanford, 1971.

[121] Alexander Sorokin and David Forsyth. Utility data annotation with amazon mechanical turk. In *Computer Vision and Pattern Recognition Workshop*, pages 1–8, 2008.

[122] Matthijs T. J. Spaan and Nikos Vlassis. Perseus: Randomized point-based value iteration for pomdps. *Journal of Artificial Intelligence Research*, 24:195–220, 2005.

[123] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning I: Introduction.* The MIT Press, 1998.

[124] Richard S. Sutton, David Mcallester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *NIPS*, pages 1057–1063, 1999.

[125] Taskcn, March 2010. `http://www.taskcn.com/`.

[126] Topcoder, December 2009. `http://topcoder.com`.

[127] Turker nation, 2010. `http://turkers.proboards.com`.

[128] Peter Welinder, Steve Branson, Serge Belongie, and Pietro Perona. The multidimensional wisdom of crowds. In *In Proc. of NIPS*, pages 2424–2432, 2010.

[129] Jacob Whitehill, Paul Ruvolo, Tingfan Wu, Jacob Bergsma, and Javier Movellan. Whose vote should count more: Optimal integration of labels from laberlers of unknown expertise. In *In Proc. of NIPS*, pages 2035–2043, 2009.

[130] Incentive. http://en.wikipedia.org/wiki/Incentive.

[131] Crowdsourcing, December 2009. `http://en.wikipedia.org/wiki/Crowdsourcing`.

[132] Jason D. Williams and Steve Young. Partially observable markov decision processes for spoken dialog systems. *Comput. Speech Lang.*, 21(2):393–422, 2007.

[133] David Wingate and Kevin D. Seppi. Prioritization methods for accelerating MDP solvers. *J. of Machine Learning Research*, 6:851–881, 2005.

[134] Jiang Yang, Lada A. Adamic, and Mark S. Ackerman. Crowdsourcing and knowledge sharing: strategic user behavior on taskcn. In *EC: Proceedings of the 9th ACM conference on Electronic commerce*, pages 246–255, 2008.

[135] Sungwook Yoon, Alan Fern, and Robert Givan. FF-Replan: A baseline for probabilistic planning. In *Proc. of the 17th International Conference on Automated Planning and Scheduling (ICAPS-07)*, pages 352–359, 2007.

[136] Rong Zhou and Eric A. Hansen. Domain-independent structured duplicate detection. In *Proc. of the 21th National Conference on Artificial Intelligence (AAAI-06)*, 2006.

[137] Rong Zhou and Eric A. Hansen. Edge partitioning in external-memory graph search. In *IJCAI*, pages 2410–2417, 2007.

[138] Rong Zhou and Eric A. Hansen. Parallel structured duplicate detection. In *Proc. of the 20th National Conference on Artificial Intelligence (AAAI-07)*, pages 1217–1223, 2007.

Appendix A

SNAPSHOTS OF TASKS OF TurKontrol

See below an image and a English description of it. Your task is to create a better description of it.

- Improve the current description by adding more details, fixing incorrect grammar and tightening the language.
- Do **not** try to tell a story, speculate or add your opinion.
- Your description will be used distinguish this picture from other similar photos; so accurate details and conciseness are crucial.

We'll check your descriptions and **reject spam results** .

Original text: A smiling woman is holding a box of strawberries at a farmers market. The space on the table in front of her is covered with more strawberries packed in green plastic boxes. The wall in the background is decorated with a mural of a farmer's market (including shoppers, assorted produce, a dog, and trash cans). To one side of the strawberries, peaches are piled up. On the other side, in the distance, you can see pears and other fruits displayed for sale.

A smiling woman is holding a box of strawberries at a farmers market. The space on the table in front of her is covered with more strawberries packed in green plastic boxes. The wall in the background is decorated with a mural of a farmer's market (including shoppers, assorted produce, a dog, and trash cans). To one side of the strawberries, peaches are piled up. On the other side, in the distance, you can see pears and other fruits displayed for sale.

Submit

Figure A.1: Snapshot of an improvement HIT.

Here is a photo followed by two textboxs. Click the arrow on the left of the textbox that contains the best description of the photo.

- Descriptions should be judged based on conciseness, level or detail, grammar and prose.
- Be sure to enter a vote for *every* task. You'll only get paid if you complete them all.

We'll **reject spam results** and compare your choices with those of other judges, so consider each score carefully.

Example

In this example, Description 2 is much better than Description 1, as it is more factual, comprehensive and concise.

Description 1:

This is a photograph of a tree and an empty grass field with more trees in the background. The field probably belongs to some farmer, who does not appear in the picture. The weather is quite nice – it is a bright summer day.

Description 2:

This is a photograph of a tree in an empty grass field. Behind the tree is a fence with more grass on the other side. There is a forest in the background. The sky is mostly clear and almost turquoise in color.

==========

Actual Task

> A smiling woman is holding a box of strawberries at a farmers market. The space on the table in front of her is covered with more strawberries packed in green plastic boxes. The wall in the background is decorated with a mural of a farmer's market (including shoppers, assorted produce, a dog, and trash cans). To one side of the strawberries, peaches are piled up. On the other side, in the distance, you can see pears and other fruits displayed for sale.

> A smiling lady is holding a green box of strawberries at a farmers market. The space on the table in front of her is covered with more strawberries packed in green boxes. The price is 3.95. The wall in the background is decorated with a mural of a farmer's market (including shoppers, assorted produce, a dog, and trash cans). On q one side of the strawberries, peaches are piled up. On the other side, in the distance, pears and other fruits displayed for sale.

Figure A.2: Snapshot of a complete ballot HIT.

Here is a photo followed by two textboxs. Click the arrow on the left of the textbox that contains the best description of the photo.

- Descriptions should be judged based on conciseness, level or detail, grammar and prose.
- Be sure to enter a vote for *every* task. You'll only get paid if you complete them all.

We'll **reject spam results** and compare your choices with those of other judges, so consider each score carefully.

Figure A.3: Snapshot of the instruction portion of a ballot HIT.

Example

In this example, Description 2 is much better than Description 1, as it is more factual, comprehensive and concise.

Description 1:

This is a photograph of a tree and an empty grass field with more trees in the background. The field probably belongs to some farmer, who does not appear in the picture. The weather is quite nice – it is a bright summer day.

Description 2:

This is a photograph of a tree in an empty grass field. Behind the tree is a fence with more grass on the other side. There is a forest in the background. The sky is mostly clear and almost turquoise in color.

Figure A.4: Snapshot of the example portion of a ballot HIT.

Actual Task

> A smiling woman is holding a box of strawberries at a farmers market. The space on the table in front of her is covered with more strawberries packed in green plastic boxes. The wall in the background is decorated with a mural of a farmer's market (including shoppers, assorted produce, a dog, and trash cans). To one side of the strawberries, peaches are piled up. On the other side, in the distance, you can see pears and other fruits displayed for sale.

> A smiling lady is holding a green box of strawberries at a farmers market. The space on the table in front of her is covered with more strawberries packed in green boxes. The price is 3.95. The wall in the background is decorated with a mural of a farmer's market (including shoppers, assorted produce, a dog, and trash cans). On q one side of the strawberries, peaches are piled up. On the other side, in the distance, pears and other fruits displayed for sale.

Figure A.5: Snapshot of the actual task portion of a ballot HIT.

Below is a picture followd by six descriptions. Your task is to give a score (between 0-10) to each description, using the button to the left of the possible scores. **We pay well, as we are looking for high-quality answers. Please take some time reading the instructions carefully** .

- A long description does **not** always means a high score
- A score of 0 means the description is so bad that anyone could improve it quickly and without much thinking.
- 10 means the description is perfect - no one is talented enough to improve the conciseness, level of detail, prose and grammar.
- Be sure to enter a score for *every* description. You'll only get paid if you complete them all.

We'll **reject spam results** and compare your choices with those of other judges, so consider each score carefully.

Figure A.6: Snapshot of the instruction portion of a scoring HIT.

Example

Here is an example picture with five descriptions, which we have scored. Do your best to score on roughly the same scale which we have u

Description 1

```
a picutre of
```

This a superficial description (with a typo) that anyone can improve. **Score: 0** .

Description 2

```
This is a photograph of a tall tree and empty grass field. Blue sky
can be seen in the background. Perhaps the field is used as a pasture
for a herd of horses. It looks like it would be fun to fly a kite
here if there were enough wind.
```

This starts out as a fair description, but it includes subjective and hypothetical details which are irrelevant and hurt the score. **Score: 3** .

Description 3

```
This is a photograph of a tree and an empty grass field with more
trees in the background. The field probably belongs to some farmer,
who does not appear in the picture. The weather is quite nice – it is
a bright summer day.
```

This is a fair description that describes the high-level idea of the picture, but it lacks detail. One can improve it fairly easily. **Score: 4** .

Description 4

```
This is a photograph of a large, deciduous tree in the middle of an
empty grass field. A fence divides the field right behind the tree
and a mixed forest occupies the background. The sky is blue with a
few wispy clouds.
```

This is a good and factual description. More detail can be put into it. **Score: 6** .

Description 5

```
This is a photograph of a tall deciduous tree, likely a White Oak
(Quercus alba), surrounded by an empty grass field which has been
recently mowed. A wire fence runs across the foreground (just behind
the oak), supported by short wooden poles.  In the lower right, an
antique hay rake lies in the field. In the background, behind the
field, lies a mixed forest. Blue sky with wispy clouds occupies the
top half of the photo.
```

This is an excellent description with considerable detail. It would be very hard to improve. **Score: 8** .

We can't create a description of score 10, as **almost nobody** can.

Figure A.7: Snapshot of the example portion of a scoring HIT.

Actual Task

Description 1

A smiling woman is holding a box of strawberries at a farmers market standing in front of a wall, which is decorated with a mural of a farmer's market (including shoppers, assorted produce, a dog, and trash cans).. The space on the table in front of her is covered with more plastic green baskets full of strawberries each of them costs 3.95$. To one side of the strawberries, peaches are piled up. On the other side, in the distance, you can see pears and other fruits kept for sale as well.

Poor ○0 ○1 ○2 ○3 ○4 ○5 ○6 ○7 ○8 ○9 ○10 Perfect

Description 2

A smiling woman is holding a box of strawberries at a farmers market. The space on the table in front of her is covered with more strawberries packed in green plastic boxes. The wall in the background is decorated with a mural of a farmer's market (including shoppers, assorted produce, a dog, and trash cans). To one side of the strawberries, peaches are piled up. On the other side, in the distance, you can see pears and other fruits displayed for sale.

Poor ○0 ○1 ○2 ○3 ○4 ○5 ○6 ○7 ○8 ○9 ○10 Perfect

Description 3

A smiling woman is holding a box of strawberries at a farmers market. The space on the table in front of her is covered with more strawberries packed in green plastic boxes. The wall in the background is decorated with a mural of a farmer's market (including shoppers, assorted produce, a dog, and trash cans). To one side of the strawberries, peaches are piled up. On the other side, in the distance, you can see pears and other fruits displayed for sale.

Poor ○0 ○1 ○2 ○3 ○4 ○5 ○6 ○7 ○8 ○9 ○10 Perfect

Description 4

A smiling woman is holding a box of strawberries at a farmers market standing in front of a wall, which is decorated with a mural of a farmer's market (including shoppers, assorted produce, a dog, and trash cans).. In front of her there are so many green colored plastic baskets with full of strawberries.cost has displayed (3.95$). To one side of the strawberries, peaches are piled up. On the other side, in the distance, you can see pears and other fruits kept for sale as well.

Poor ○0 ○1 ○2 ○3 ○4 ○5 ○6 ○7 ○8 ○9 ○10 Perfect

Description 5

A smiling woman is holding a box of strawberries at a farmers market. The space on the table in front of her is covered with more strawberries packed in green plastic boxes. The wall in the background is decorated with a mural of a farmer's market (including shoppers, assorted produce, a dog, and trash cans). To one side of the strawberries, peaches are piled up. On the other side, in the distance, you can see pears and other fruits displayed for sale.

Poor ○0 ○1 ○2 ○3 ○4 ○5 ○6 ○7 ○8 ○9 ○10 Perfect

Figure A.8: Snapshot of the actual task portion of a scoring HIT.

VITA

Peng Dai was born in Nanjing, China. He received a B.S. degree in Computer Science from Nanjing University, and M.S. degrees in Computer Science from National University of Singapore and University of Kentucky. His research interests in computer science are in the areas of Artificial Intelligence, with emphasis on decision making under uncertainty, scalable, automated planning, and decision-theoretic planning applications for human computation.

Mr. Dai is the author of over 15 papers and technical reports. He is a member of AAAI since 2005. Mr. Dai has served the program committee member for three top AI conferences and has reviewed for over 10 top AI conferences and journals.