COL702: Backtracking and Dynamic Programming

Thanks to Miles Jones, Russell Impagliazzo, and Sanjoy Dasgupta at UCSD for these slides.

SEARCH AND OPTIMIZATION PROBLEMS

Many problems involve finding the best solution from among a large space of possibilities.

- **Instance**: What does the input look like?
- **Solution format**: What does an output look like?
- **Constraints**: What properties must a solution have?
- **Objective function**: What makes a solution better or worse?

GLOBAL SEARCH VS LOCAL SEARCHES

- Like greedy algorithms,
  backtracking algorithms break the massive global search for a solution, into a series of simpler local searches.
  "Which edge do we take first? Then second? …"

- Unlike greedy algorithms, which guess the best local choice and only consider this possibility,
  backtracking uses exhaustive search to try out all combinations of local decisions.

GLOBAL SEARCH VS LOCAL SEARCHES

- However, we can often use the **constraints** of the problem to **prune** cases that are dead ends. Applying this recursively, we get a substantial savings over exhaustive search.

- This might take a long time to do. What are some other ideas in general?

BACKTRACKING: PROS AND CONS

The good:

Very general, applies to almost any search problem

Can lead to exponential improvement over exhaustive search

Often better as heuristic than worst-case analysis

FIRST STEP TO DYNAMIC PROGRAMMING

The bad:

Since it works for very hard problems, usually only improved exponential time, not poly time

Hard to give exact time analysis

MAXIMAL INDEPENDENT SET

Given a graph with nodes representing people, with an edge between any two people who are enemies, find the largest set of people such that no two are enemies. In other words, given an undirected graph, find the largest set of vertices such that no two are joined by an edge.

MAXIMAL INDEPENDENT SET

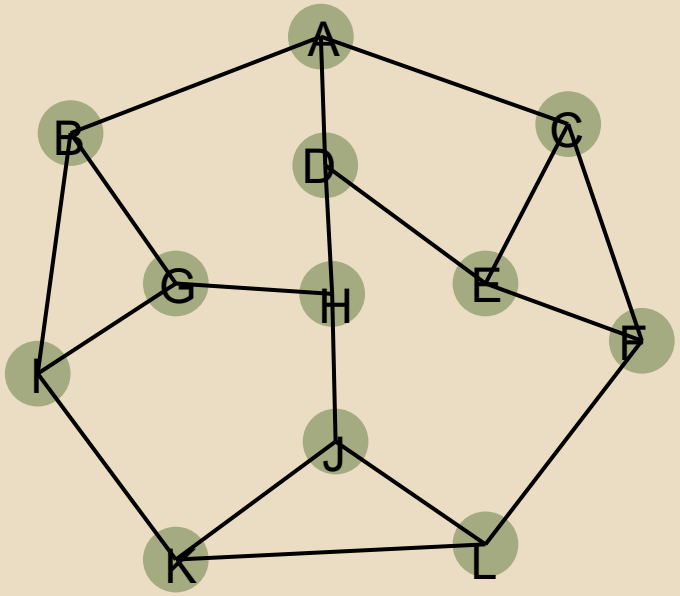
Given a graph with nodes representing people, with an edge between any two people who are enemies, find the largest set of people such that no two are enemies. In other words, given an undirected graph, find the largest set of vertices such that no two are joined by an edge.

- **Instance:**
- **Solution format:**
- **Constraint:**
- **Objective:**

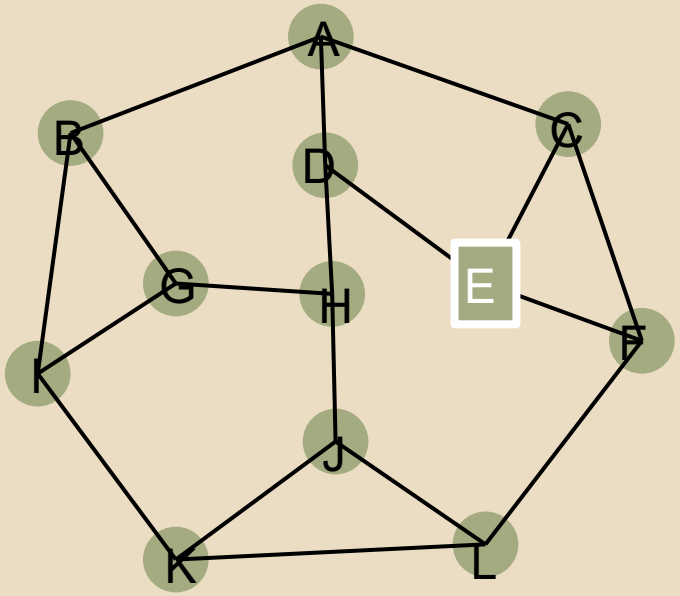
MAXIMAL INDEPENDENT SET

- Greedy approaches?
- One may be tempted to choose the person with the fewest enemies, remove all of his enemies and recurse on the remaining graph.
- This is fast, but does not always find the best solution.

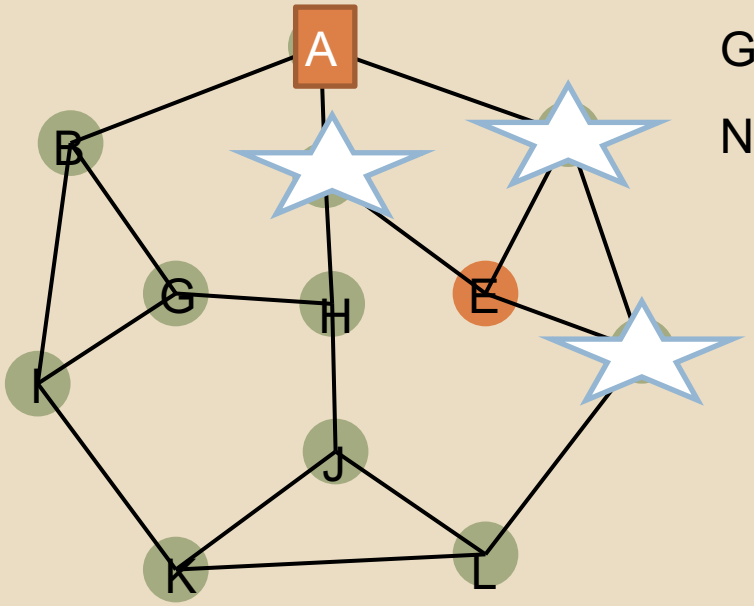
AN EXAMPLE



AN EXAMPLE



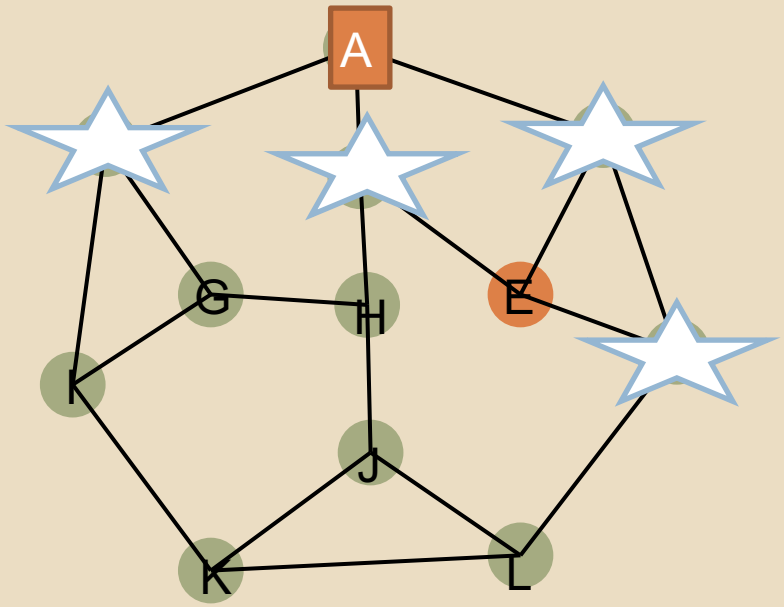
AN EXAMPLE



Greedy: all degree 3, pick any, say E

Neighbors (enemies) of E forced out of set

AN EXAMPLE

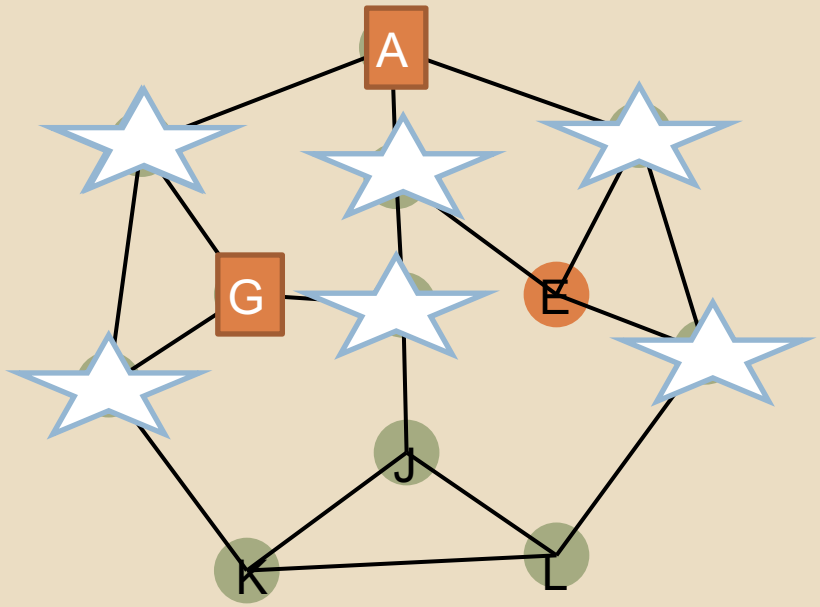


Greedy: all degree 3, pick any, say E

Neighbors (enemies) of E forced out of set

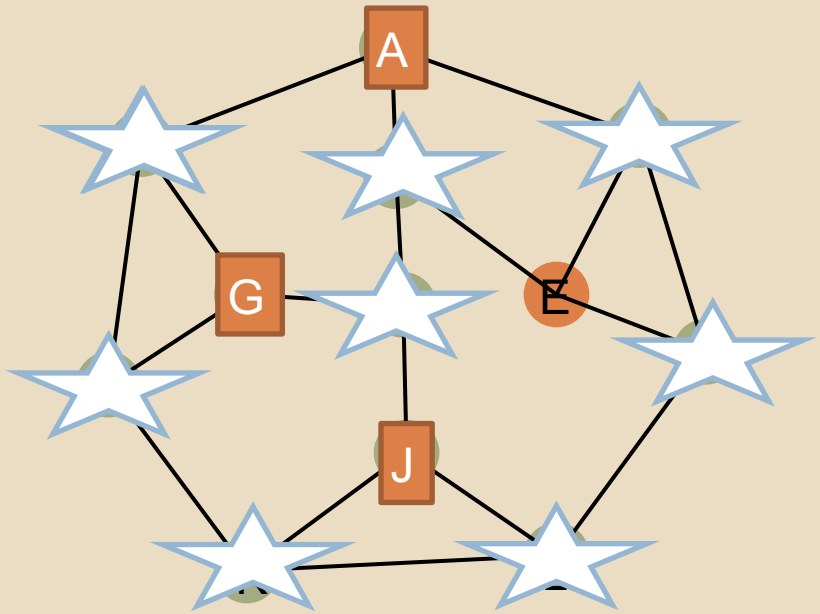
Lowest degree is now A

AN EXAMPLE



Many degree 2 vertices we could choose next, say G

AN EXAMPLE

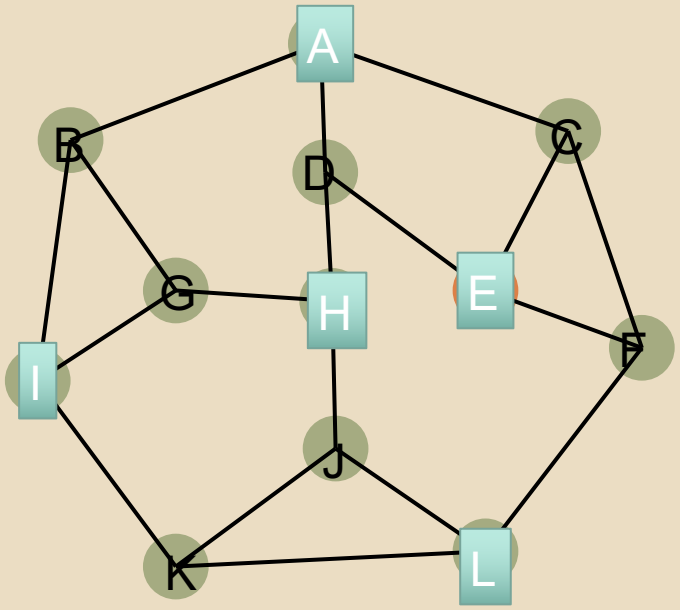


Many degree 2 vertices we could choose next, say G

Can pick any remaining one

Solution found by greedy is size 4

BETTER SOLUTION



MAXIMAL INDEPENDENT SET

- What is the solution space?
- How much is exhaustive search?
- What are the constraints?
- What is the objective?

MAXIMAL INDEPENDENT SET

- What is the solution space?

All subsets S of V

- How much is exhaustive search?

2^{|V|}

- What are the constraints?

For each edge e={u,v}, cannot have both u and v in S

- What is the objective?

|S|

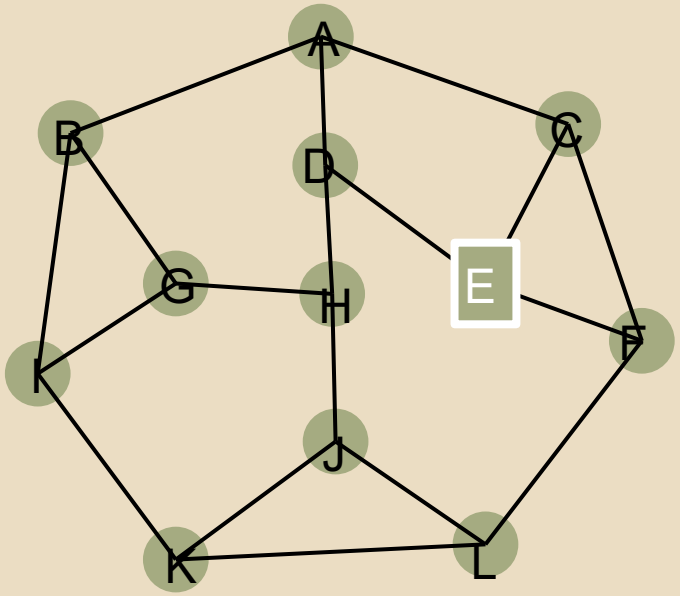
MAXIMAL INDEPENDENT SET

- Backtracking: Do exhaustive search locally. Use constraints to simplify problem along the way.
- What is a local decision? Do we pick vertex E or not…..
- What are the possible answers to this decision? Yes or No
- How do the answers affect the problem to be solved in the future?

If we pick E: Recurse on subgraph $G - \{E\} - \{E\text{'s neighbors}\}$ (and add 1)

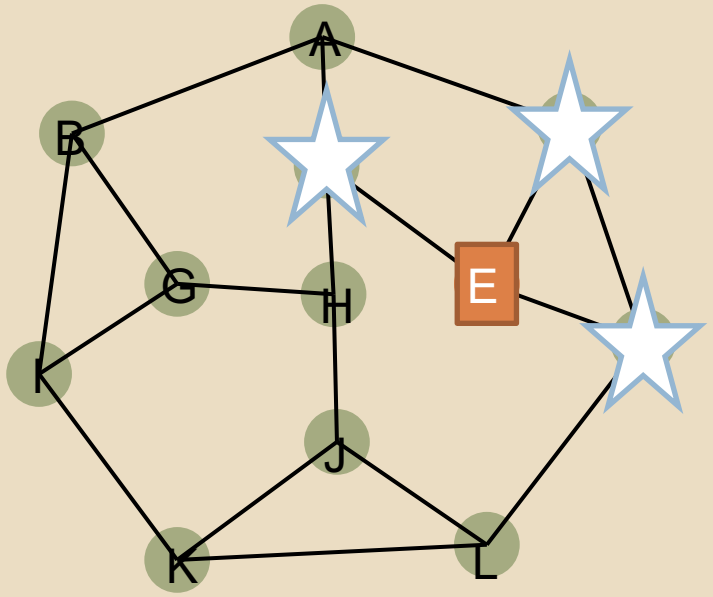
If we don't pick E: Recurse on subgraph $G - \{E\}$.

AN EXAMPLE



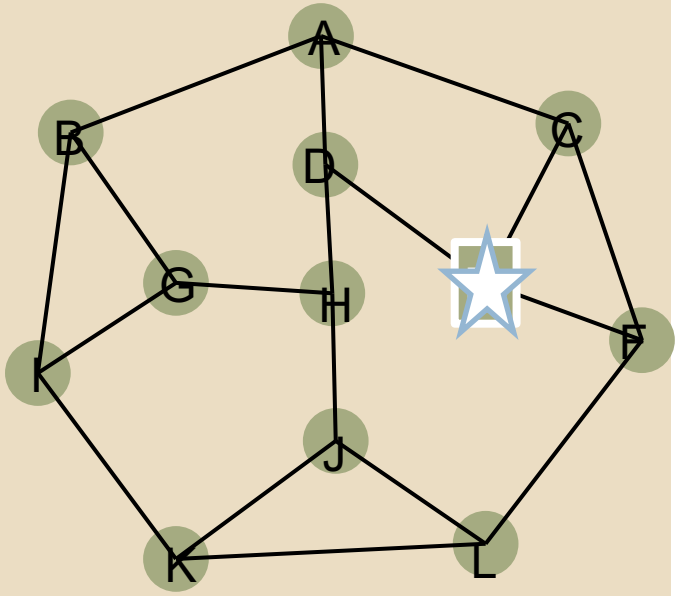
Local decision : Is E in S?
Possible answers: Yes, No

AN EXAMPLE

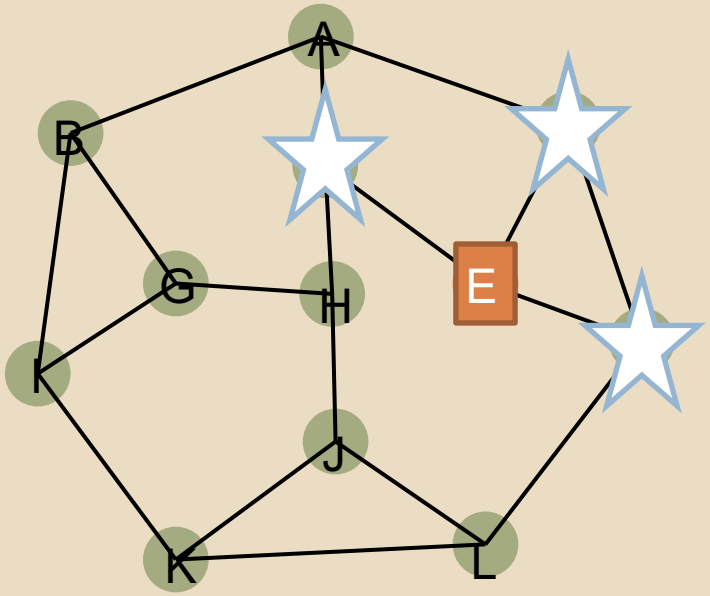


Local decision : Is E in S?
YES OR NO

MIS([A,B,C,D,E,F,G,H,I,J,K,L])=
(YES) 1 + MIS([A,B, G,H,I,J,K,L])
(NO) MIS([A,B,C,D, ,F,G,H,I,J,K,L])

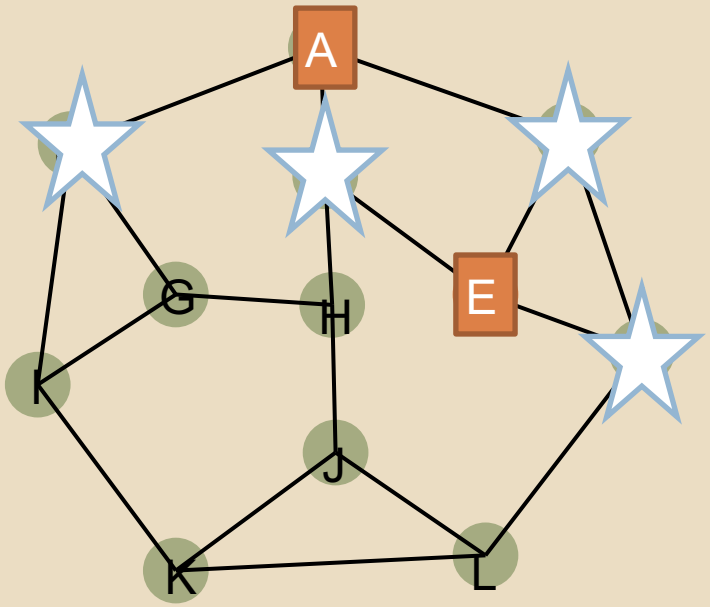


AN EXAMPLE



Local decision : Is E in S?
Case 1 : Yes
Consequences: Neighbors not in S

AN EXAMPLE



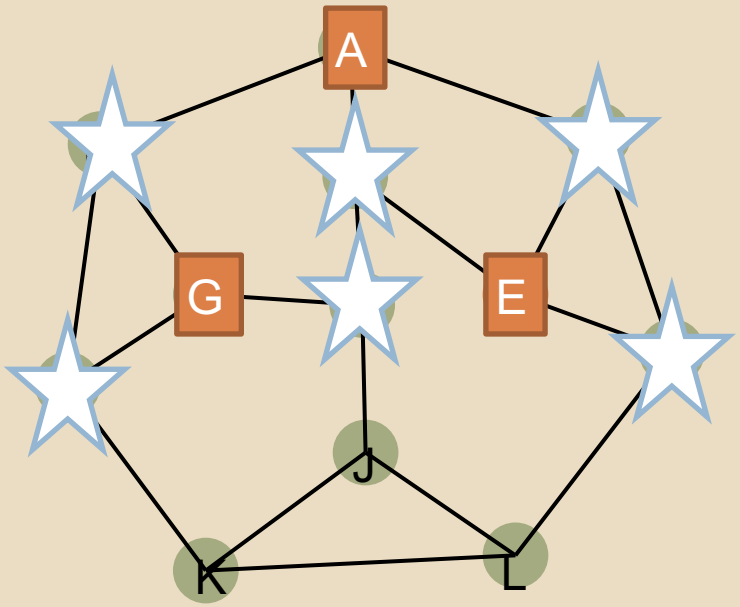
Local decision : Is E in S?
Case 1 : Yes
Consequences: Neighbors not in S

Claim: A is now in some largest IS

Go on to next local decision

Is G in S?

AN EXAMPLE



Local decision : Is E in S?
Case 1 : Yes
Consequences: Neighbors not in S

Claim: A is now in some largest IS

Go on to next local decision

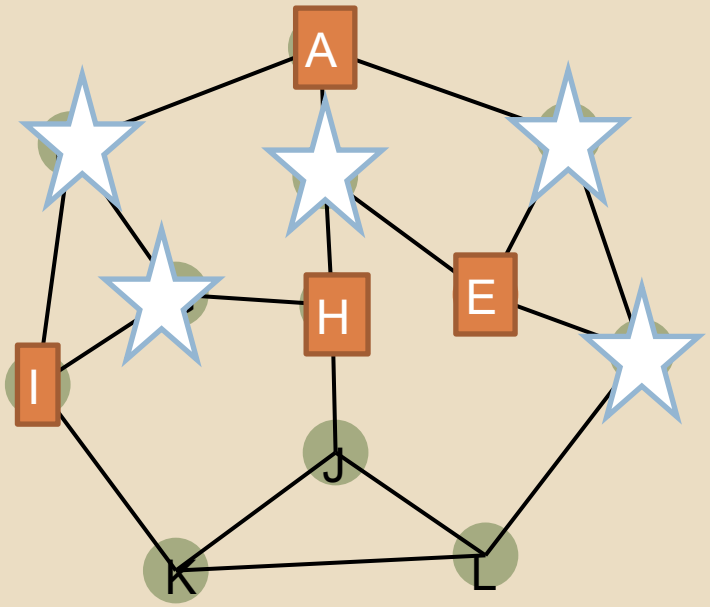
Is G in S?

Case 1a: Yes

Other three symmetrical: Get one more

Best set for Case 1a: 4, e.g, A,G,E,J

BUT NOW WE BACKTRACK



Local decision : Is E in S?
Case 1 : Yes
Consequences: Neighbors not in S

Claim: A is now in some largest IS

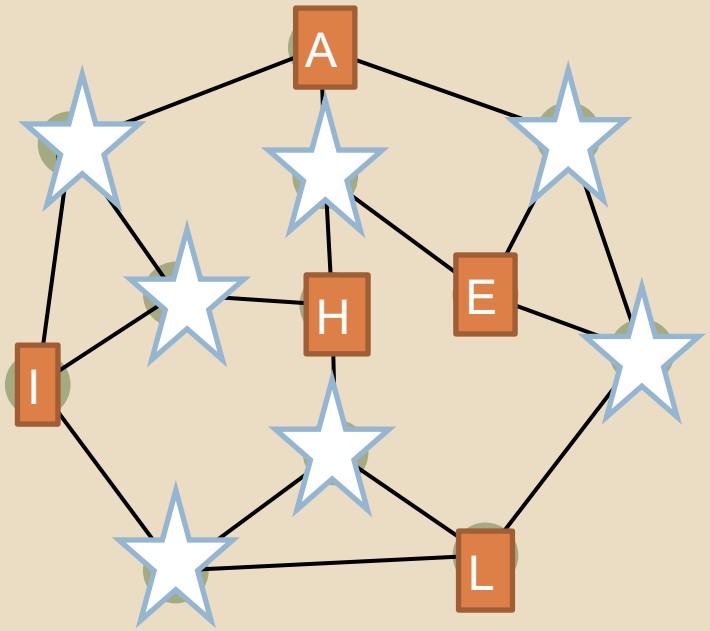
Go on to next local decision

Is G in S?

Case 1b: No

Claim: I, H in some smallest MIS in Case 1b

BUT NOW WE BACKTRACK



Local decision : Is E in S?
Case 1 : Yes
Consequences: Neighbors not in S

Claim: A is now in some largest IS

Go on to next local decision

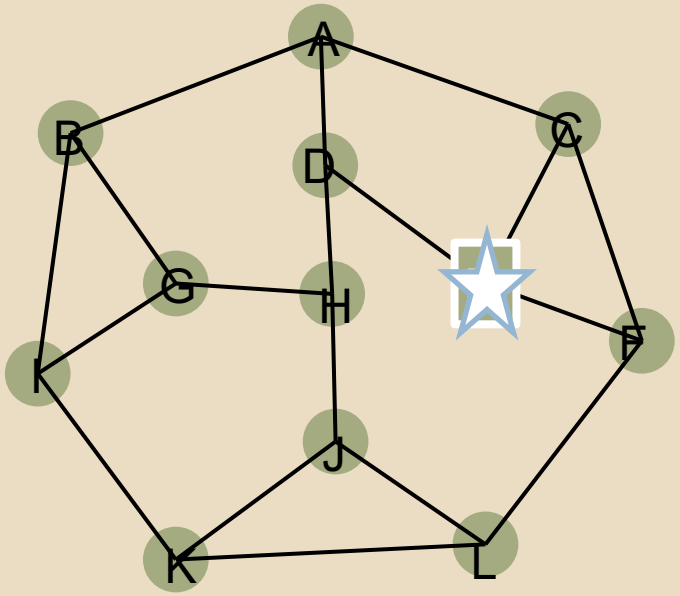
Is G in S?

Case 1b: No

Claim: I, H in some smallest MIS in Case 1b

Case 1 b: Get set of size 5

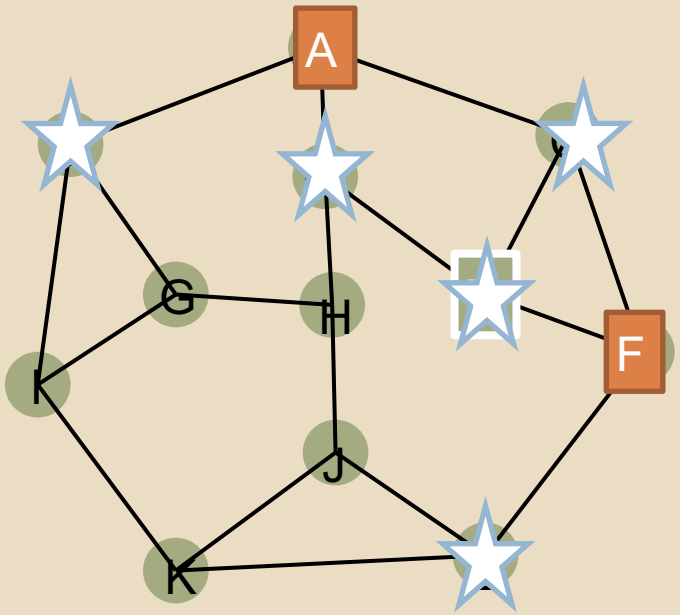
BACKTRACK AGAIN



Case 1b is better than Case 1a, but we still don't know its optimal

Need to consider Case 2: E is not in S

BACKTRACK AGAIN



Case 1b is better than Case 1a, but we still don't know its optimal

Need to consider Case 2: E is not in S

Case 2a: A is in S

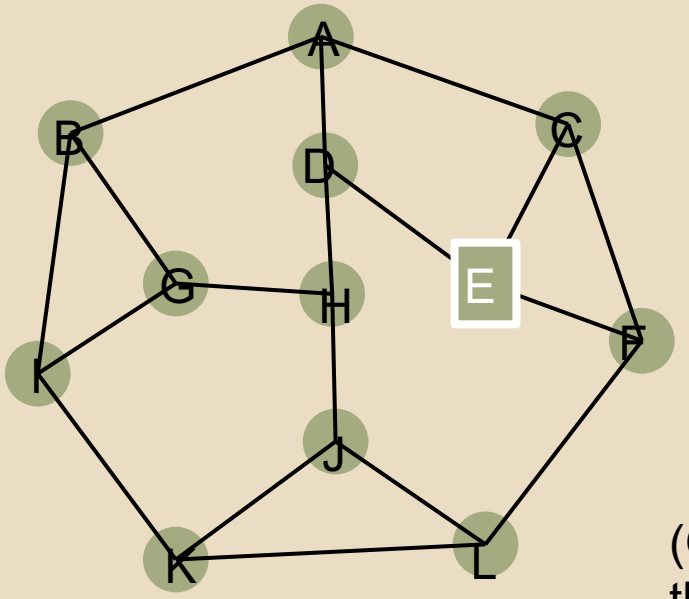
F is in S

Cycle of 5 : get 2

So this case eventually gets 4

Now we KNOW Case 1b is best

AN EXAMPLE



12 vertices means 4096 subsets

But in the end, we only needed 4 cases

(OK, I used some higher principles, e.g. symmetry that our BT algorithm might not have)

CASE ANALYSIS AS RECURSION

MIS1(G= (V,E))

- IF |V|=0 return the empty set
- Pick vertex v
- S_1:= v + MIS1(G-v-N(v))
- S_2: = MIS1(G-v)
- IF |S_2| > |S_1| return S_2, else return S_1

CORRECTNESS

MIS1(G= (V,E))

- IF |V|=0 return the empty set
- Pick vertex v
- S_1:= v + MIS1(G-v-N(v))
- S_2: = MIS1(G-v)
- IF |S_2| > |S_1| return S_2, else return S_1

Induction on n. Base case n=0: MIS1 correctly returns empty set.

Otherwise, use strong induction: S_1 is max ind set containing v,

S_2 max ind. set not containing v. Better of two is MIS in G.

TIME ANALYSIS

MIS1(G= (V,E))

- IF |V|=0 return the empty set
- Pick vertex v
- S_1:= v + MIS1(G-v-N(v))
- S_2: = MIS1(G-v)
- IF |S_2| > |S_1| return S_2, else return S_1

TIME ANALYSIS

MIS1(G= (V,E))

- IF |V|=0 return the empty set
- Pick vertex v:
- S_1:= v + MIS1(G-v-N(v)) — Worst-case: T(n-1)
- S_2: = MIS1(G-v) — T(n-1)
- IF |S_2| > |S_1| return S_2, else return S_1 — poly(n)

T(n)= 2 T(n-1) + poly(n)

- Idea: bottom-heavy, so exact poly(n) doesn't affect asymptotic time
- T(n)=2^n

WHAT IS THE WORST CASE FOR MIS1?

WHAT IS THE WORST CASE FOR MIS1?

- An empty graph with no edges, i.e., the whole graph is an independent set
- But then we should just return all vertices without trying cases
- More generally, if a vertex has no neighbors, the case when we include it v+MIS(G-v) is always better than the case when we don't include it, MIS(G-v)

GETTING RID OF THAT STUPID WORST CASE

MIS2($G = (V, E)$)

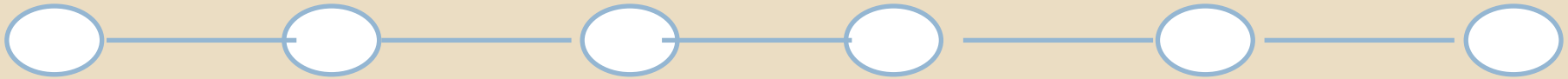
- IF $|V| = 0$ return the empty set
- Pick vertex v
- $S_1 := v + MIS2(G - v - N(v))$
- IF deg(v)=0 return S_1
- $S_2 := MIS2(G - v)$
- IF $|S_2| > |S_1|$ return S_2, else return S_1

- Correctness: If deg(v) = 0, $|S_2| < |S_1|$ so we'd return S_1 anyway
- So does same thing as MIS1

WHAT IS THE WORST CASE FOR MIS2?

WHAT IS THE WORST CASE FOR MIS2?

If the graph is a line and we always pick the end, we recurse on one line of size $n-1$ and one of size $n-2$



$$T(n) = T(n-1) + T(n-2) + poly(n)$$

$$T(n) = O(Fib(n)) = O(2^{0.7n})$$

Still exponential but for medium sized n, makes huge difference

$n = 80$: 2^{56} = minute of computer time, 2^{80} = 16 million minutes

CAN WE DO BETTER?

In the example, we actually argued that we should add vertices of degree 1 as well.

Modify-the-solution proof:

- Say v has one neighbor u.
- Let S_1 be the largest independent set with v and let S_2 be the largest ind. set without v.
- Let $S' = S_2 - \{u\} + \{v\}$. S' is an independent set, and is at least as big as S_2, and contains v. Thus, S_1 is at least as big as S', which is at least as big as S_2. So don't bother computing S_2 in this case.

IMPROVED ALGORITHM

MIS3($G = (V, E)$)

- IF $|V| = 0$ return the empty set
- Pick vertex v
- $S_1 := v + MIS3(G - v - N(v))$
- IF deg(v)=0 or 1 return S_1
- $S_2 := MIS3(G - v)$
- IF $|S_2| > |S_1|$ return S_2, else return S_1

Correctness: If deg(v) =0 or 1 $|S_2|$ is at most $|S_1|$, so we'd return S_1 anyway, so does same thing as MIS1

TIME ANALYSIS

- $T(n)$ is at most $T(n-1) + T(n-3)$ + small amount

- Similar to Fibonacci numbers, but a bit better, about
- $2^{0.6n}$ rather than $2^{0.7n}$.

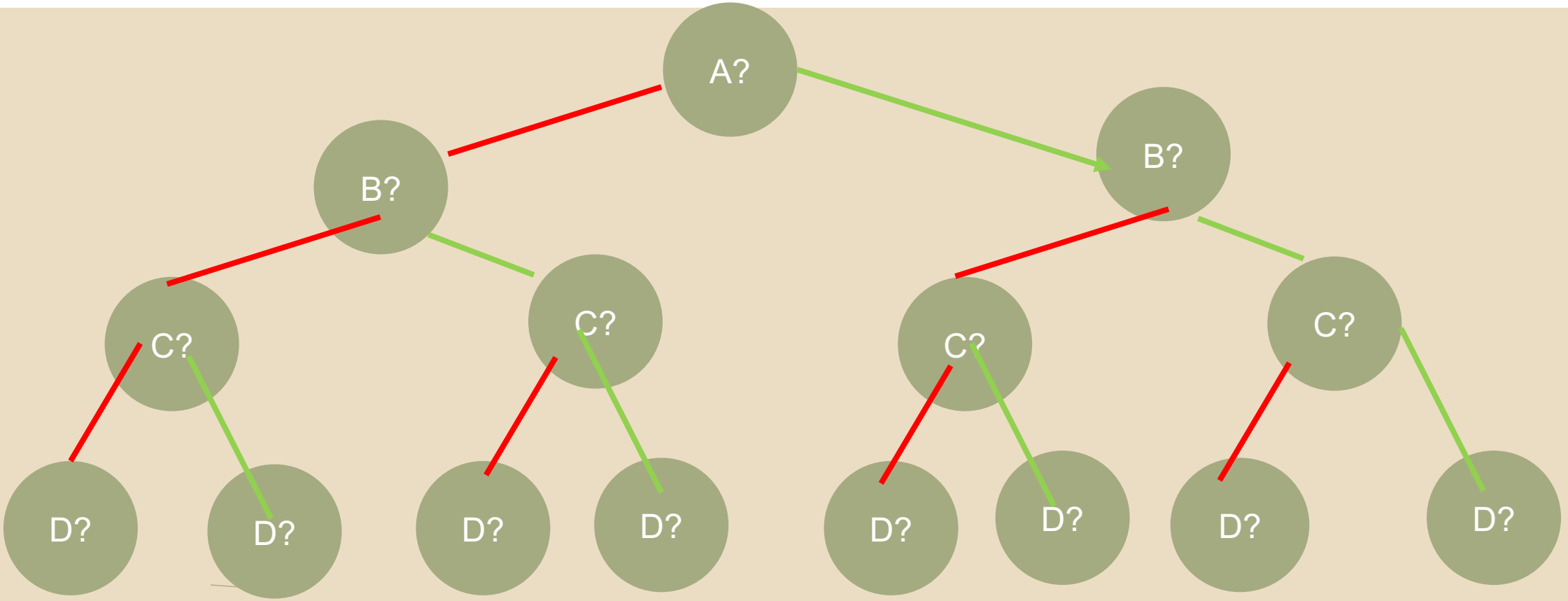
- $n = 80$: $2^{0.6n}$=2^{48}, less than a second.
- $n = 100$: 2^{60} = 16 minutes, 2^{70}=16,000 minutes
- So while still exponential, big win for moderate n

IS THIS TIGHT?

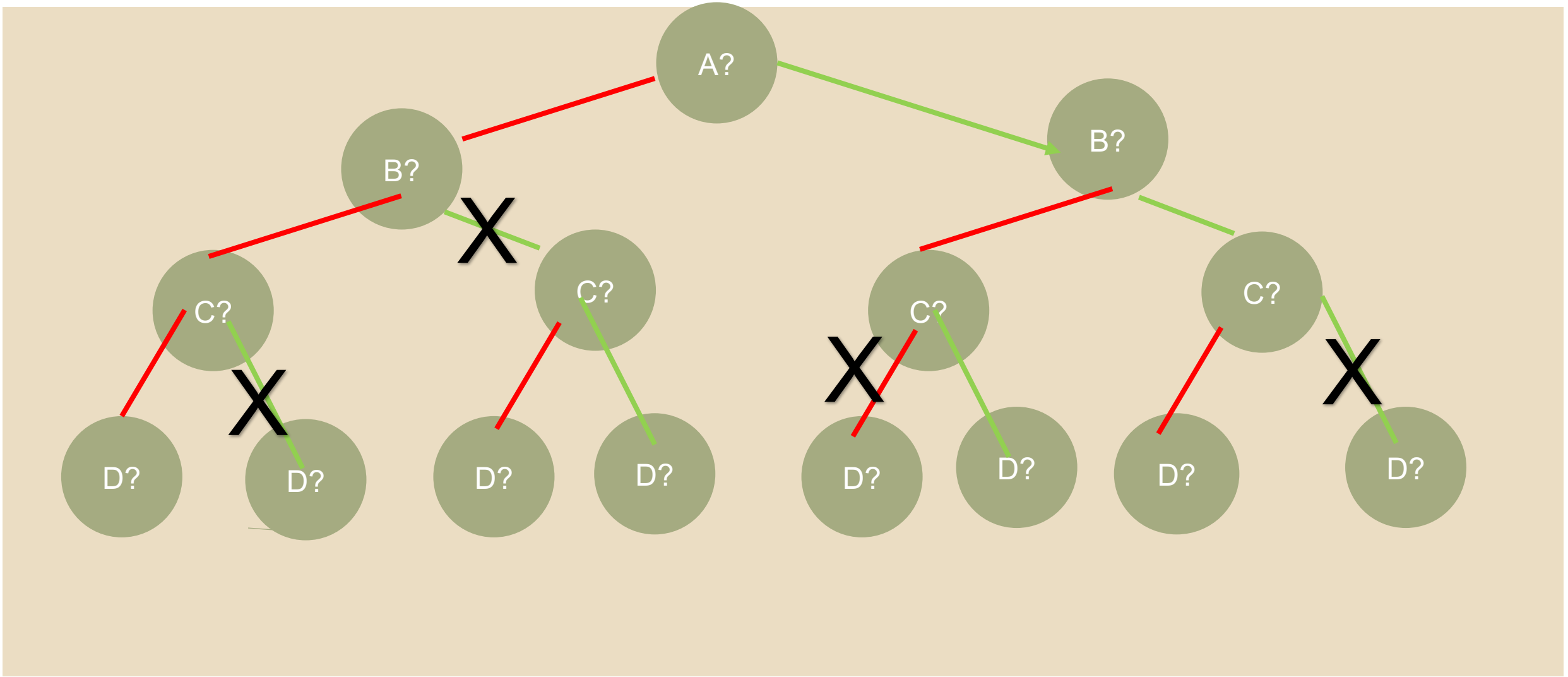
IS THIS TIGHT?

- I don't know whether there is any graph where MIS3 is that bad.
- Best known MIS algorithm around $2^{n/4}$ by Robson, building on Tarjan and Trojanowski. Does much more elaborate case analysis for small degree vertices
- Interesting research question: is there a limit to improvements?
- This question= Exponential Time Hypothesis, has interesting ramifications whether true or false

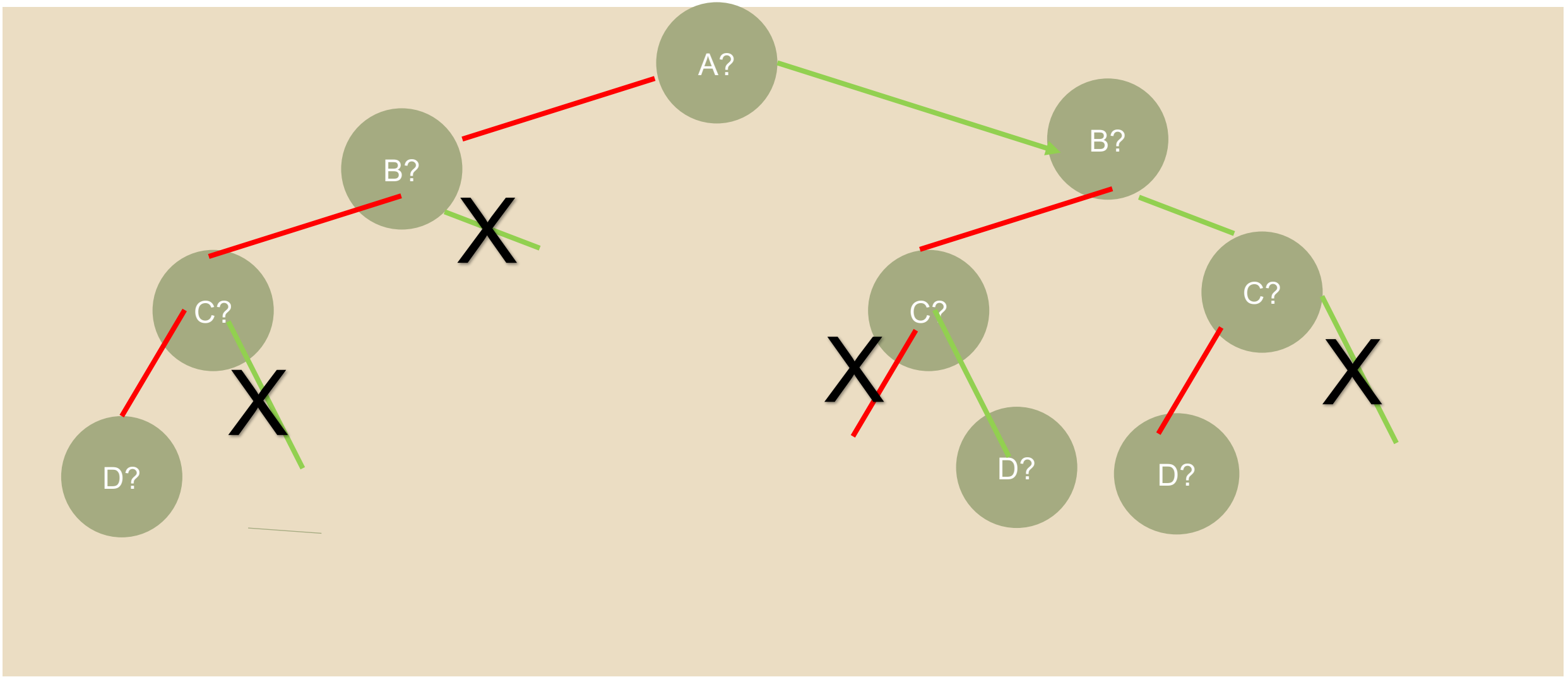
HOW BACKTRACKING HELPS



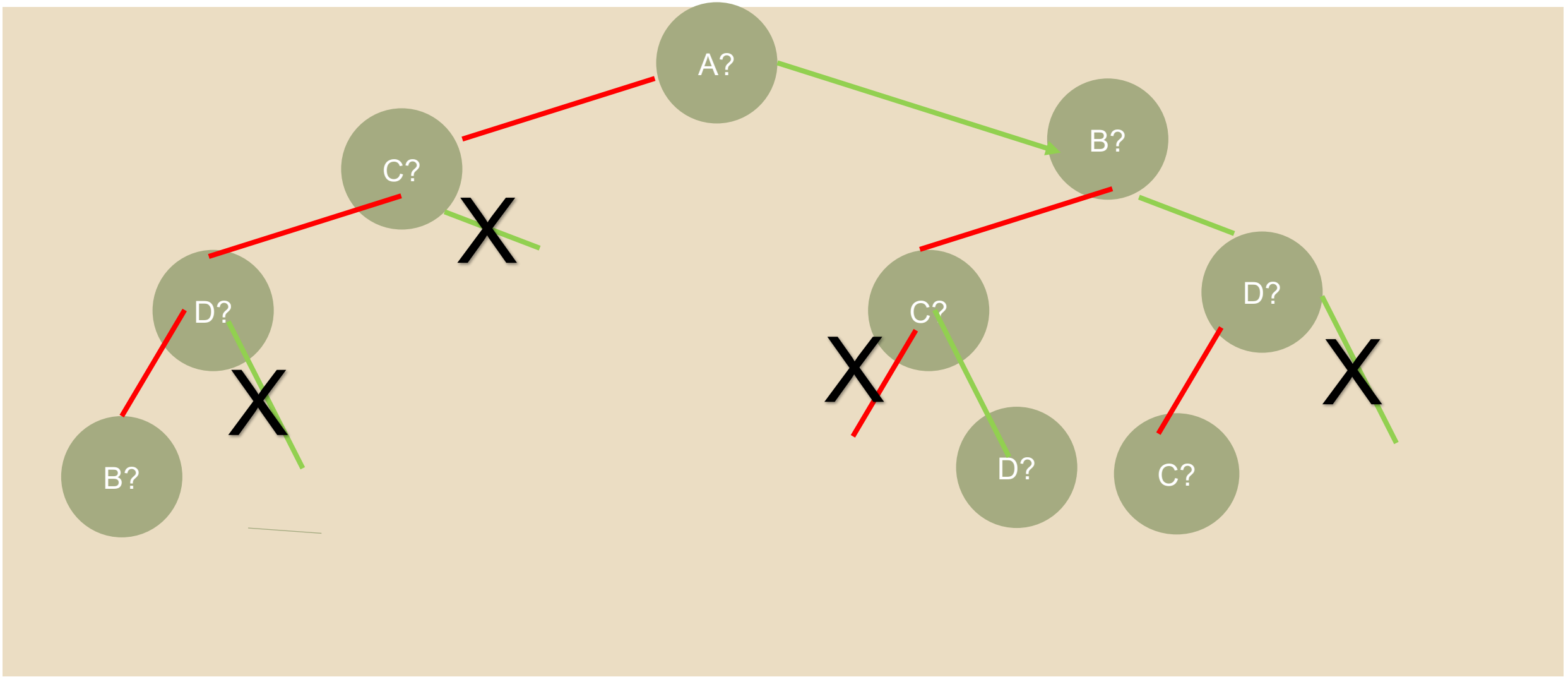
HOW BACKTRACKING HELPS



HOW BACKTRACKING HELPS



ORDER CAN BE ADAPTIVE



WHEN CAN WE PRUNE?

- Basic: when constraints would be violated
- Subtler: when that choice is dominated by another; the other choice is at least as likely to lead to a (good) solution (Need “modify-the-solution” argument)
- Branch-and-bound: dynamically track “best-so-far” solution. If current path cannot do better (using some function that bounds the achievable best), then we can prune our path.

“SELF-SIMILARITY”?

- Self-similarity: Problem+ choice = smaller problem of same type
- If we have self-similarity, it makes recursion in BT (and hence, DP) very clean.
- But if we don’t have it, we can still use BT (and hence DP)
- Generalize the problem to keep partial solution
- Generalized problem will have self-similarity, original becomes special case

3-COLORING

- Instance: undirected graph G
- Solution format: Give each vertex v a color C(v) $\in \{R, G, B\}$
- Constraint: If {u,v} is an edge, C(v) ≠ C(u)
- Problem: Existence: is there any 3-coloring of G? (True, False)
- Originally came up in making maps: vertices=countries, colors must be distinct to show borders. Famous 4-color theorem said all planar graphs (including possible maps) can be colored with 4 colors

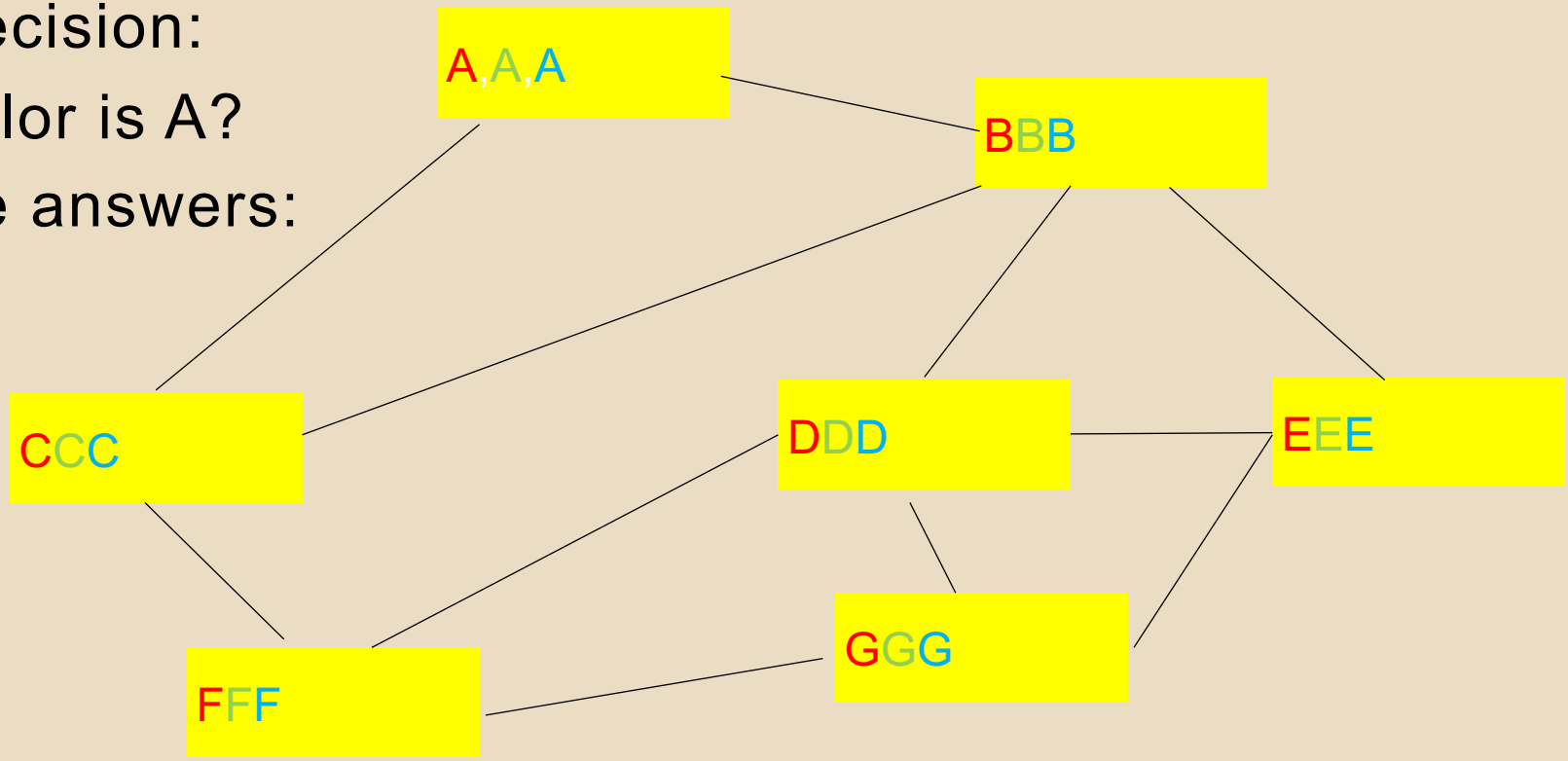
EXAMPLE

Local decision:

What color is A?

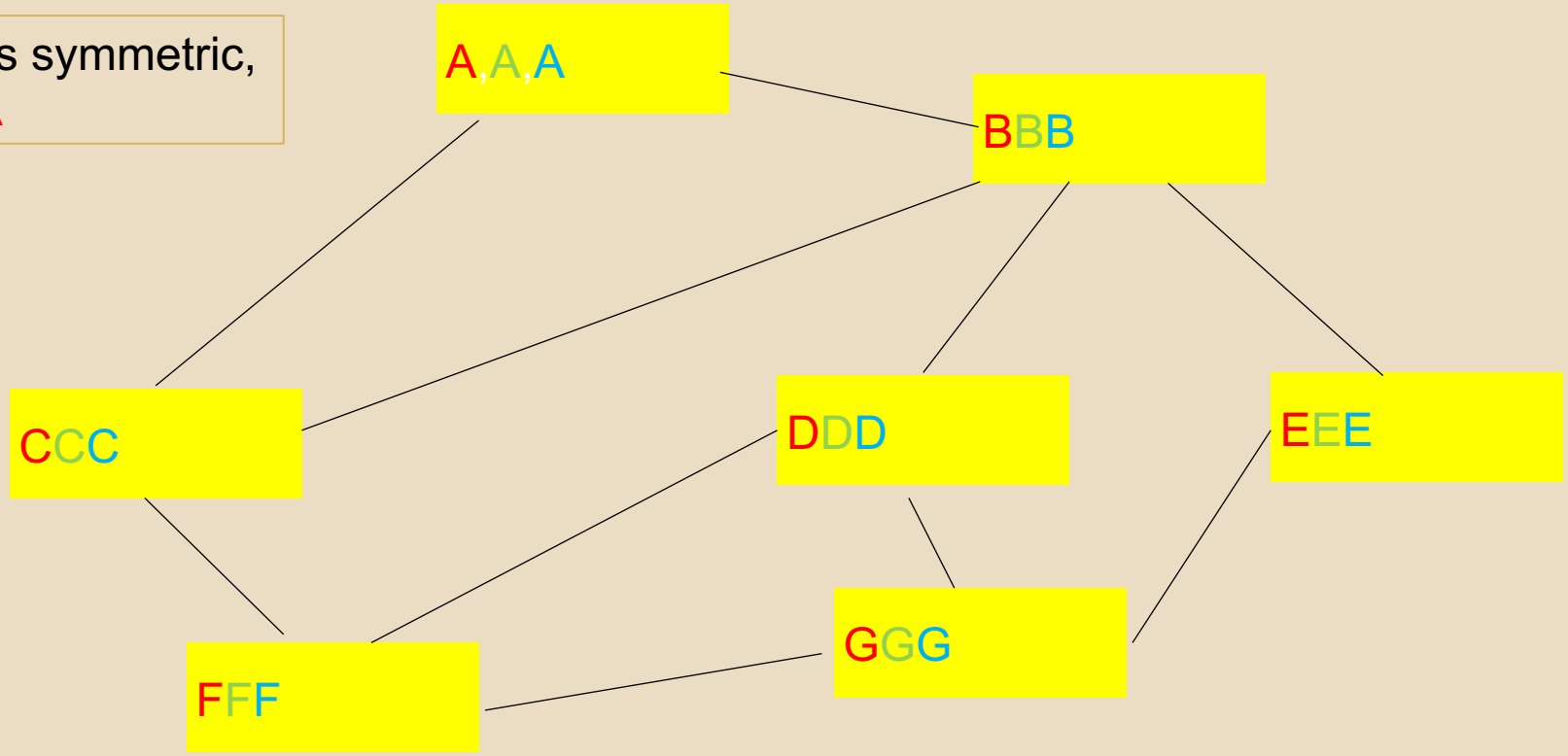
Possible answers:

AAA



EXAMPLE

All answers symmetric,
Just pick A

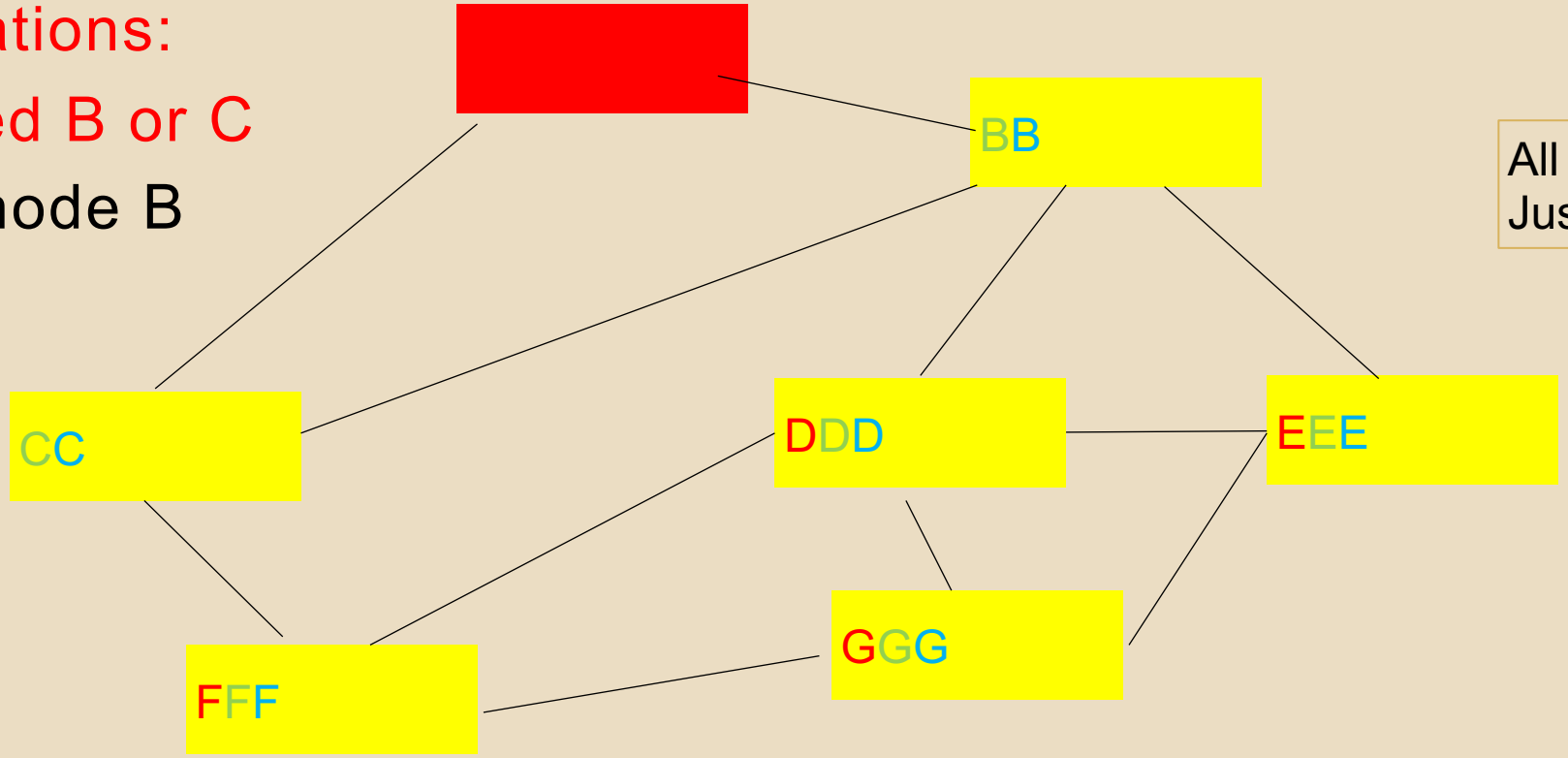


EXAMPLE

Implications:

- No red B or C

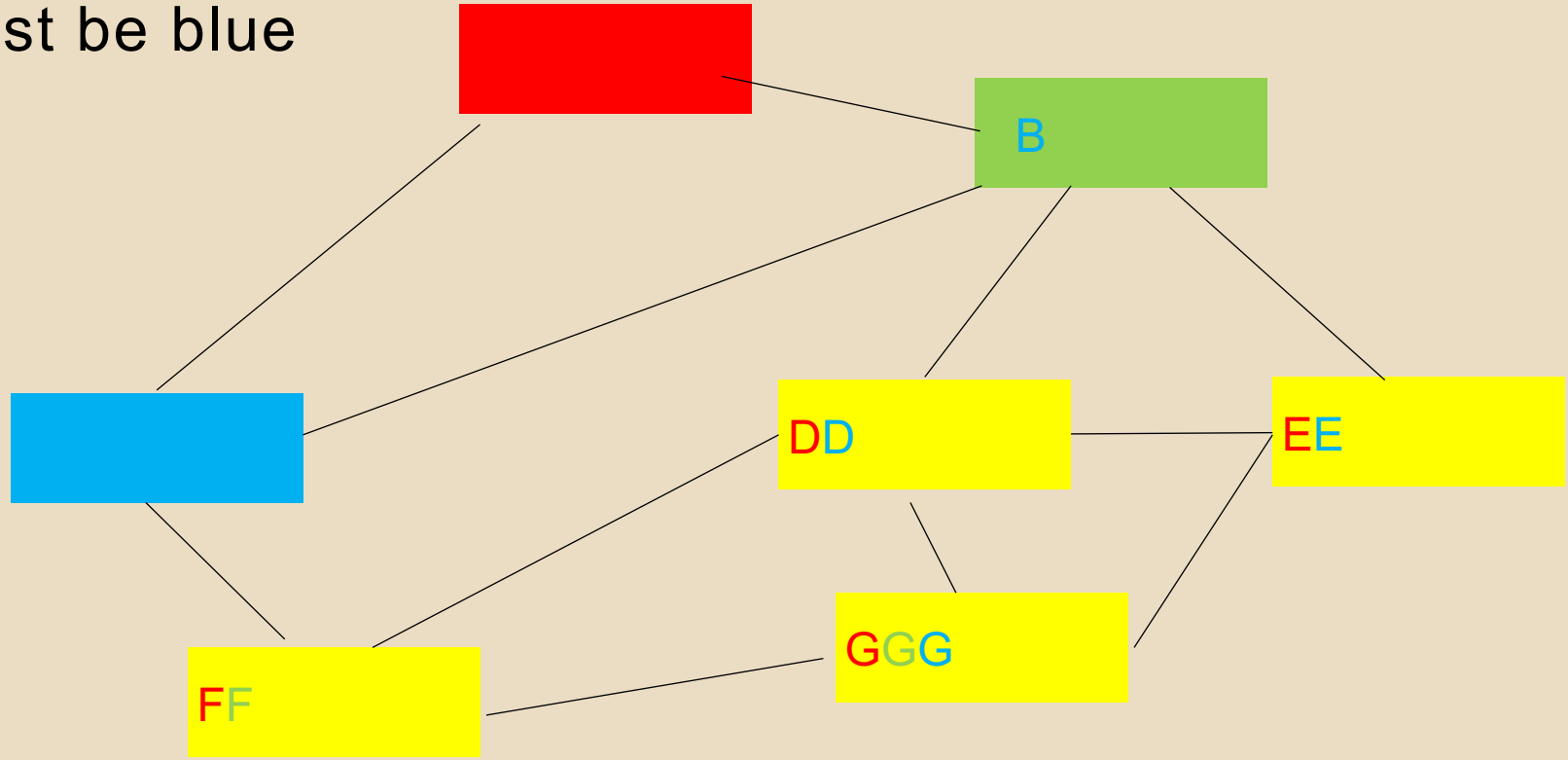
Next: node B



All answers symmetric,
Just pick B

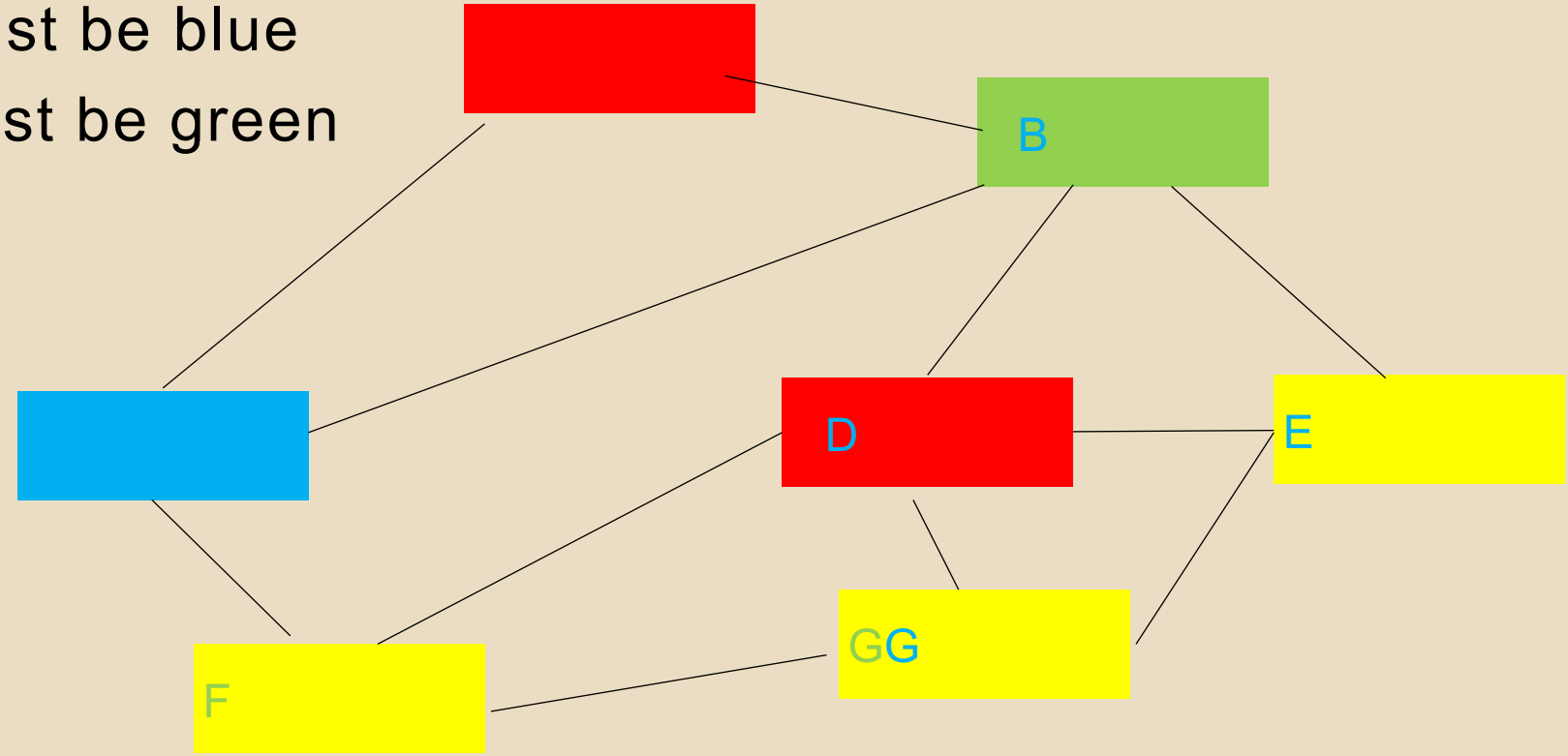
EXAMPLE

- C must be blue



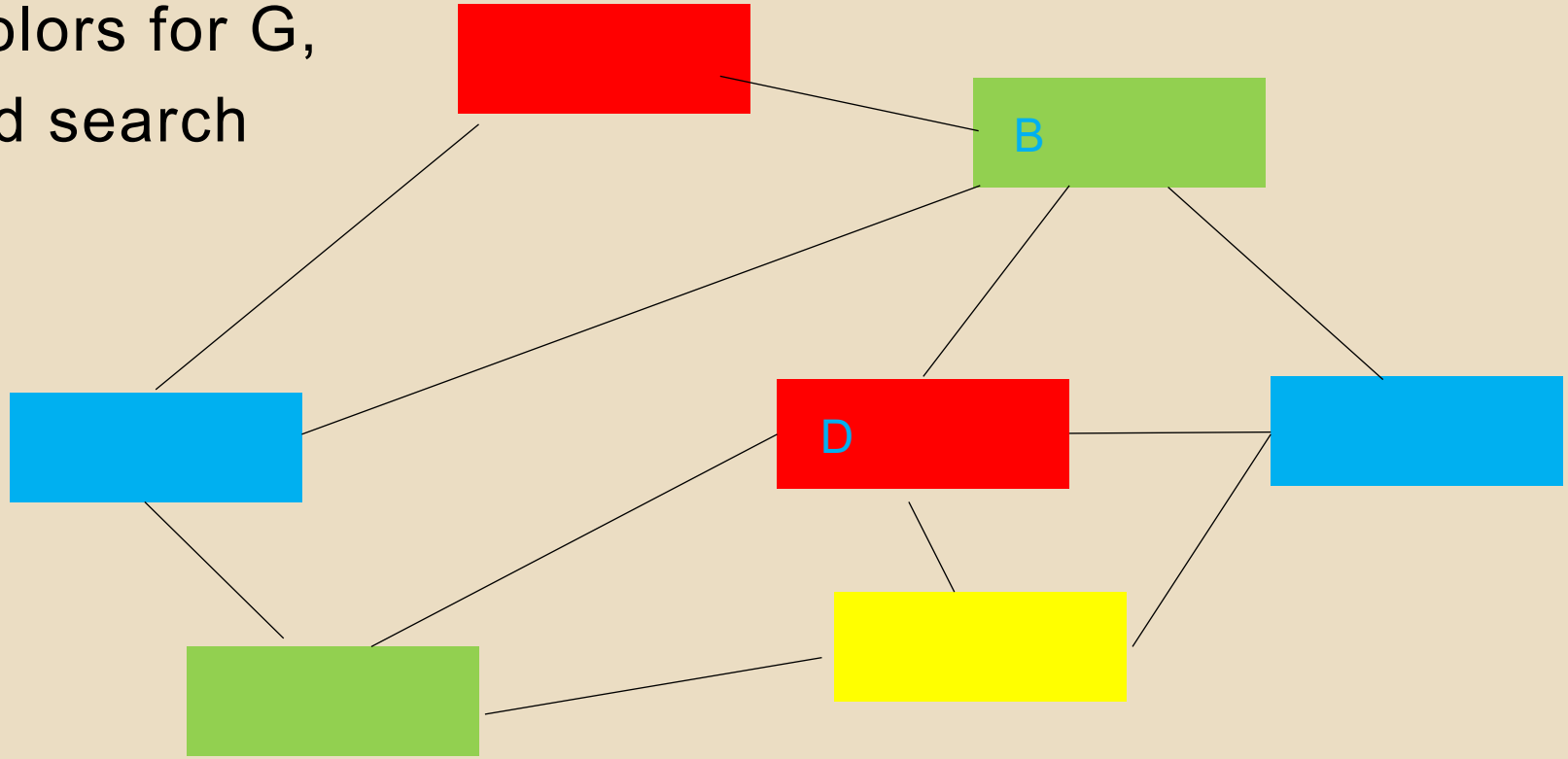
CASE 1: D (CASE 2: D)

- E must be blue
- F must be green



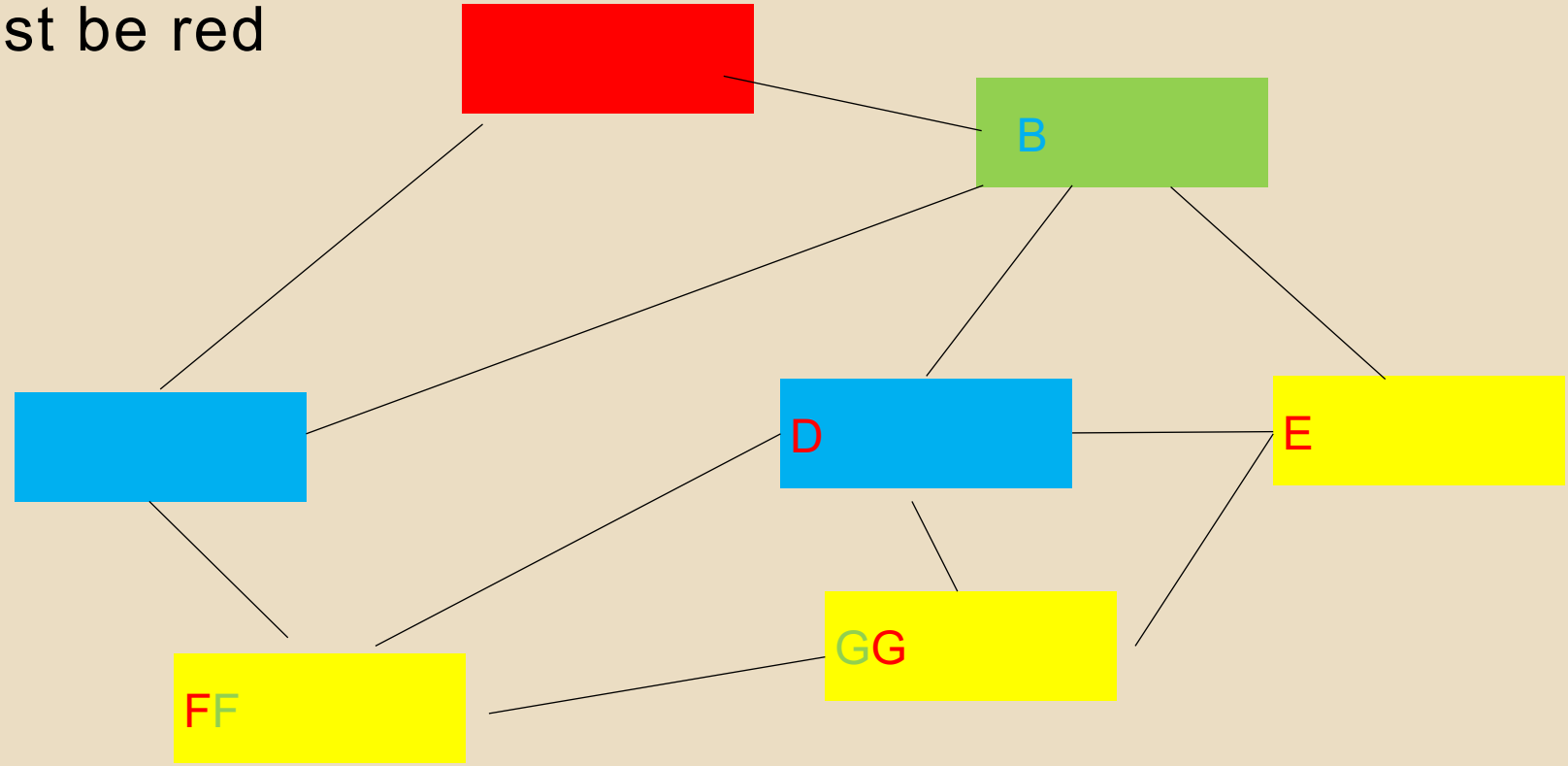
CASE 1: D (CASE 2: D)

- No colors for G,
- Failed search



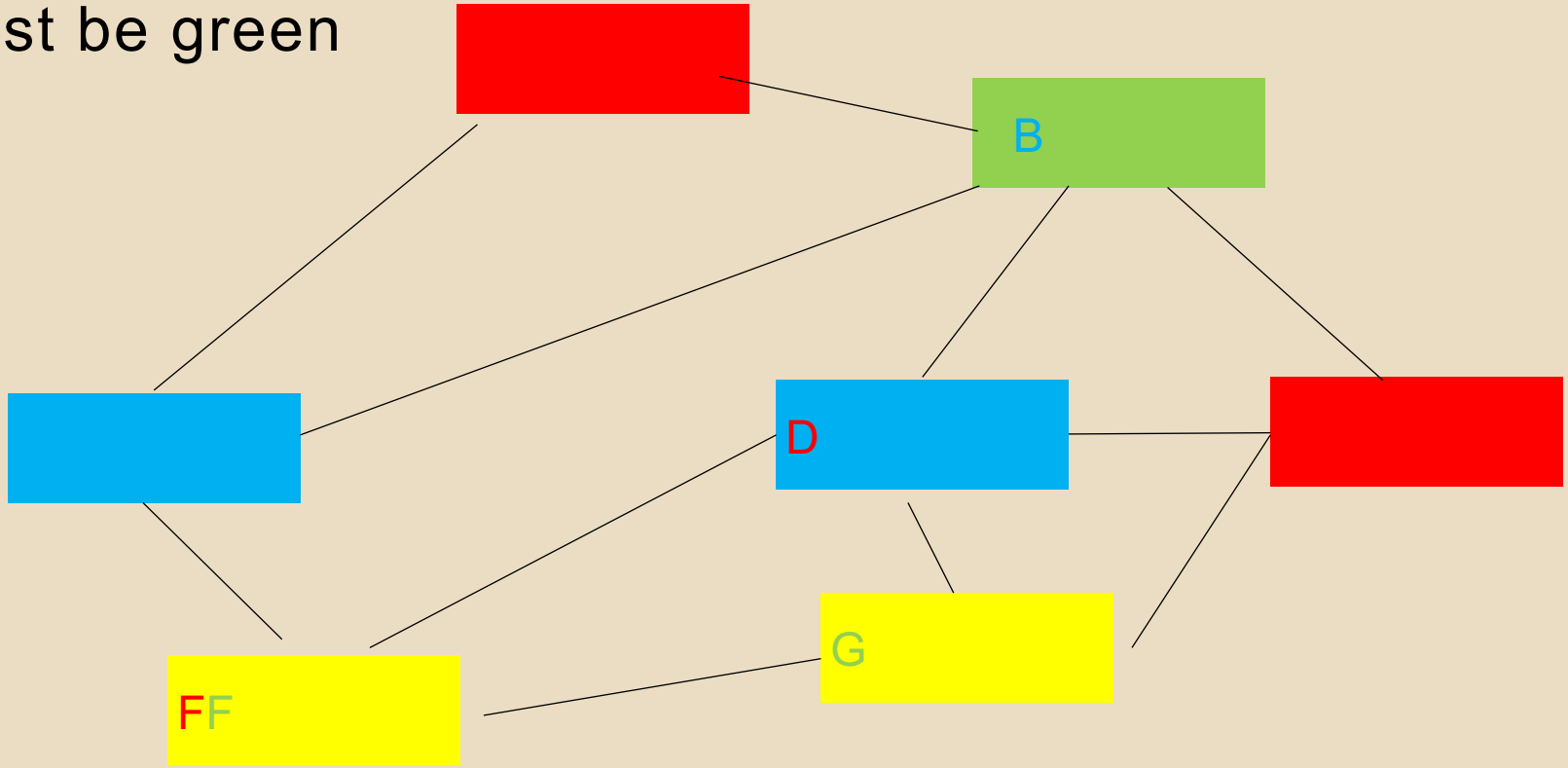
CASE 2: D

- E must be red



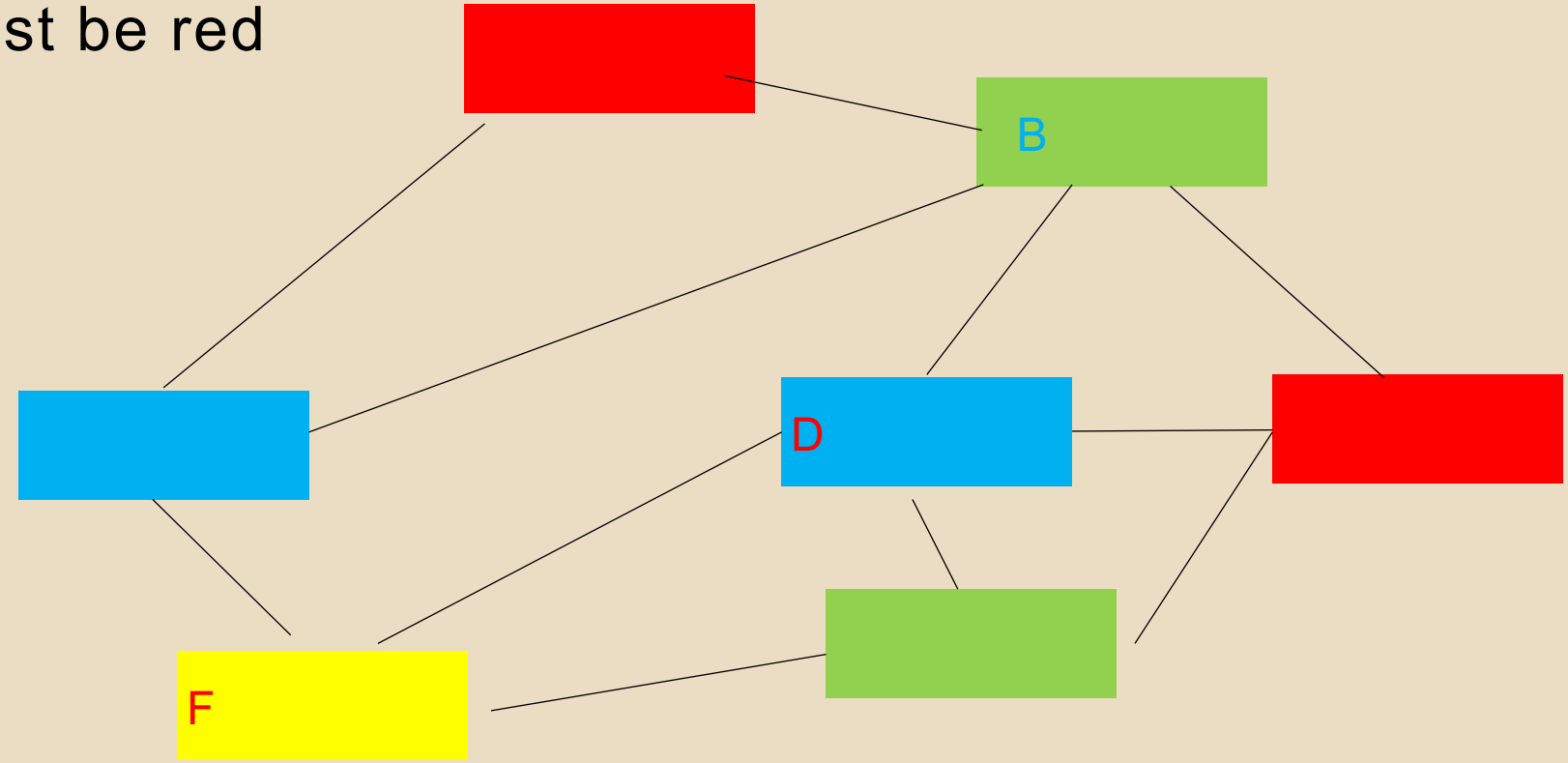
CASE 2: D

- G must be green



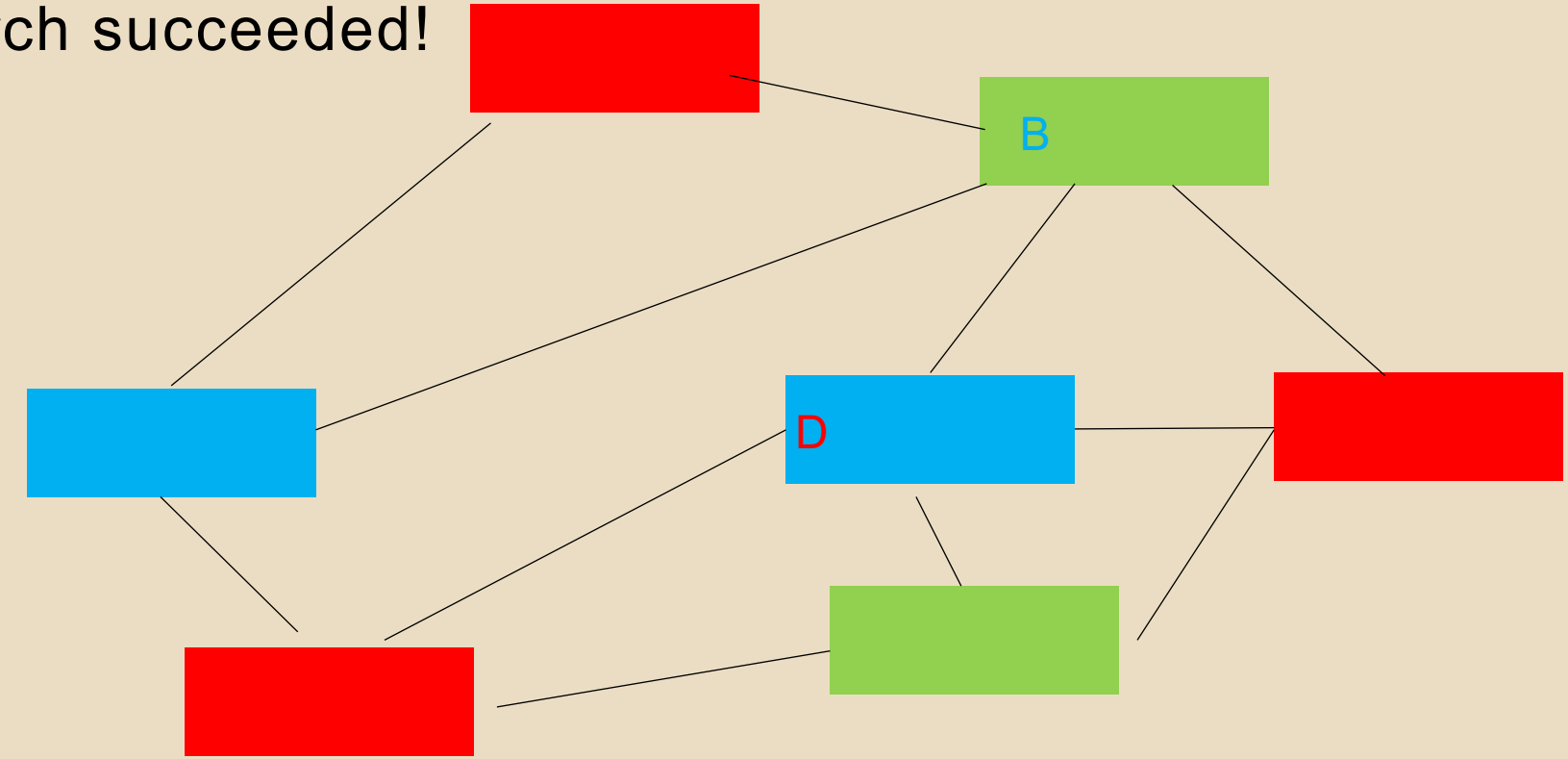
CASE 2: D

- F must be red



CASE 2: D

- Search succeeded!



PARTIAL INFORMATION

- This approach used partial information about the previous solution to generalize the problem so we could solve it recursively

- CL(u)= list of possible colors for vertex u. Initially, CL(u) is

all three colors, but we'll delete colors as we make recursive calls

The list 3-coloring problem, L3C(G,CL), adds the constraints that

C(u) must be in CL(u)

BACKTRACKING ALGORITHM

L3C(G,CL)

- If |V|=0 return True
- If there is any v with |CL(v)|=0 return False
- If there is a v with CL(v)={c}, then :
- Delete c from CL(u) for each neighbor u of v
- Return L3C(G-{v},CL)
- If all vertices v have |CL(v)|=3, then pick some v and
- Delete R from CL(u) for each neighbor u of v
- Return L3C(G-{v},CL)

BACKTRACKING ALGORITHM CONTINUED

- Remaining case: the smallest size of CL(u) is 2
- Pick v with CL(v)={c_1,c_2}
- Let CL_1 be CL(u),

  except that we delete c_1 from CL(u) for neighbors u of v
- Let CL_2 be CL(u),

  except that we delete c_2 from CL(u) for neighbors u of v
- IF L3C(G-{v}, CL_1)= True: return True
- Return L3C(G-{v}, CL_2)

BACKTRACKING TIME ANALYSIS

- Here, exhaustive search is O(3^n) time, because there are three possible colors per vertex

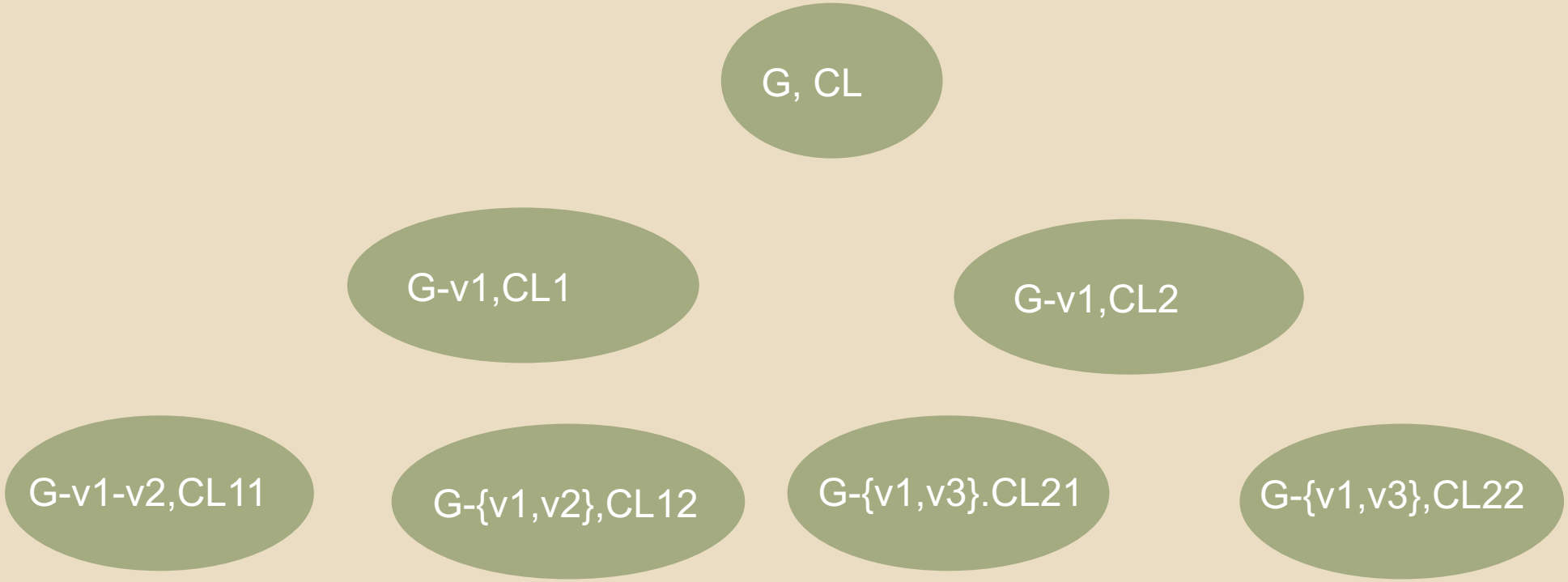
- General way to bound BT algorithms: View recursions as forming tree based on sub-calls

Time = number of leaves

If every node in tree makes at most f= fan-out recursive calls, then

The number of leaves = O(f^{$depth$})

THIS ALGORITHM



TIME ANALYSIS CONT

- Depth ≤ n-1, because graph decreases every rec. call
- Fan-out =2 , because at most two rec. calls

- So time is $O(2^n)$
- Still exponential, but much better than 3^n for moderate sized
- inputs

TOWARDS DYNAMIC PROGRAMMING

- Dynamic Programming = Backtracking + Memoization
- Memoization = store and re-use, like the Fibonacci algorithm from first class

Two simple ideas, but easy to get confused if you rush:
1. Where is the recursion?
(It disappears into the memoization, like the Fib. example did.)
2. Have I made a decision?
(Only temporarily, like BT)

If you don't rush, a surprisingly powerful and simple algorithm technique.

One of the most useful ideas around