Sudoku

							1	2
				3	5			
			6				7	
7						3		
			4			8		
1								
			1	2				
	8						4	
	5					6		

6	7	3	8	9	4	5	1	2
9	1	2	7	3	5	4	8	6
8	4	5	6	1	2	9	7	3
7	9	8	2	6	1	3	5	4
5	2	6	4	7	3	8	9	1
1	3	4	5	8	9	2	6	7
4	6	9	1	2	8	7	3	5
2	8	7	3	5	6	1	4	9
3	5	1	9	4	7	6	2	8

Solving Sudoku

Solving Sudoku puzzles involves a form of exhaustive search of possible configurations.

However, exploiting constraints to rule out certain possibilities for certain positions enables us to *prune* the search to the point people can solve Sudoku by hand.

Backtracking is the key to implementing exhaustive search programs correctly and efficiently.

Backtracking

Backtracking is a systematic method to iterate through all the possible configurations of a search space. It is a general algorithm/technique which must be customized for each individual application.

In the general case, we will model our solution as a vector $a = (a_1, a_2, ..., a_n)$, where each element a_i is selected from a finite ordered set S_i.

Such a vector might represent an arrangement where a_i contains the ith element of the permutation. Or the vector might represent a given subset S, where a_i is true if and only if the ith element of the universe is in S.

The Idea of Backtracking

At each step in the backtracking algorithm, we start from a given partial solution, say, $a = (a_1, a_2, ..., a_k)$, and try to extend it by adding another element at the end.

After extending it, we must test whether what we have so far is a solution.

If not, we must then check whether the partial solution is still potentially extendible to some complete solution.

If so, recur and continue. If not, we delete the last element from a and try another possibility for that position, if one exists.

Recursive Backtracking

```
Backtrack(a, k)
if a is a solution, print(a)
else {
        $k = k + 1$
        compute $S_k$
        while $S_k \neq \emptyset$ do
                $a_k$ = an element in $S_k$
                $S_k = S_k - a_k$
                Backtrack(a, k)
}
```

Backtracking and DFS

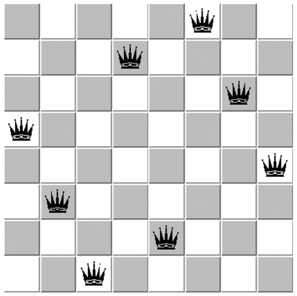
Backtracking is really just depth-first search on an implicit graph of configurations.

Backtracking can easily be used to iterate through all subsets or permutations of a set.

Backtracking ensures correctness by enumerating all possibilities.

For backtracking to be efficient, we must prune the search space.

The Eight-Queens Problem



The eight queens problem is a classical puzzle of positioning eight queens on an 8×8 chessboard such that no two queens threaten each other.

Eight Queens: Representation

What is concise, efficient representation for an n-queens solution, and how big must it be?

Since no two queens can occupy the same column, we know that the n columns of a complete solution must form a permutation of n. By avoiding repetitive elements, we reduce our search space to just $8! =$ 40,320 – clearly short work for any reasonably fast machine.

The critical routine is the candidate constructor. We repeatedly check whether the kth square on the given row is threatened by any previously positioned queen. If so, we move on, but if not we include it as a possible candidate:

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

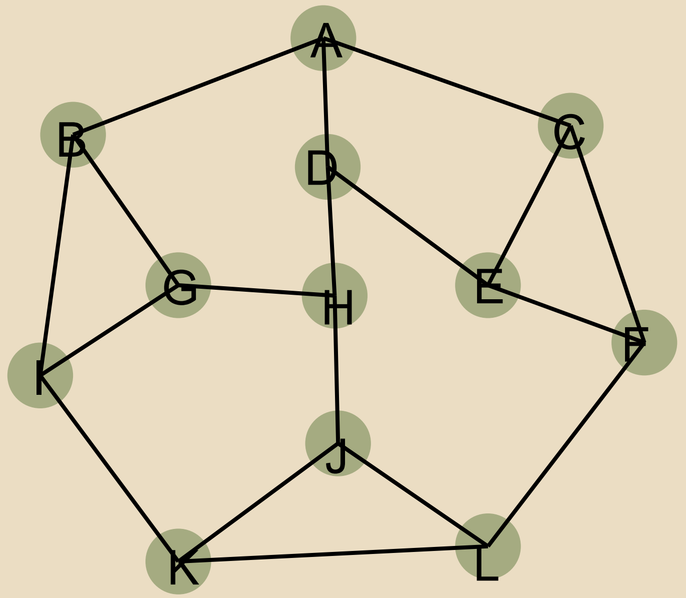
Given a graph with nodes representing people, with an edge between any two people who are enemies, find the largest set of people such that no two are enemies. In other words, given an undirected graph, find the largest set of vertices such that no two are joined by an edge.

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

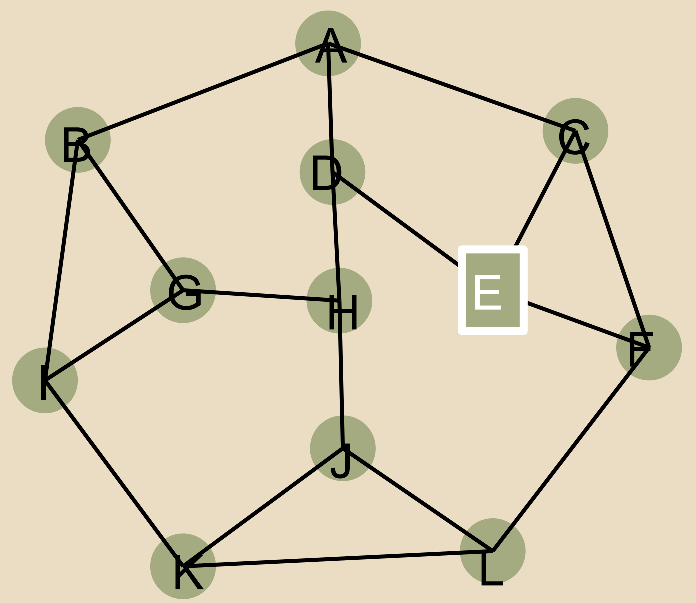
MAXIMAL INDEPENDENT SET

- 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.

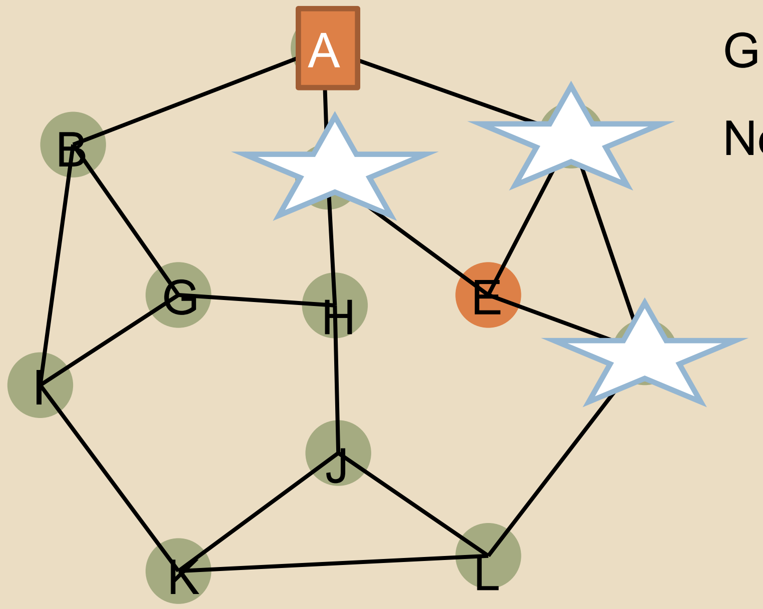
AN EXAMPLE



AN EXAMPLE



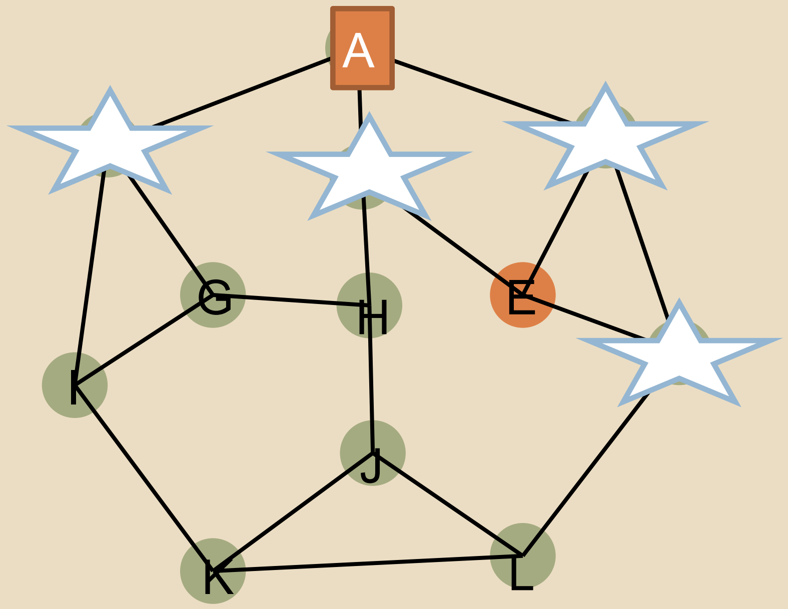
AN EXAMPLE



Greedy: all degree 3, pick any, say E

Neighbors (enemies) of E forced out of set

AN EXAMPLE

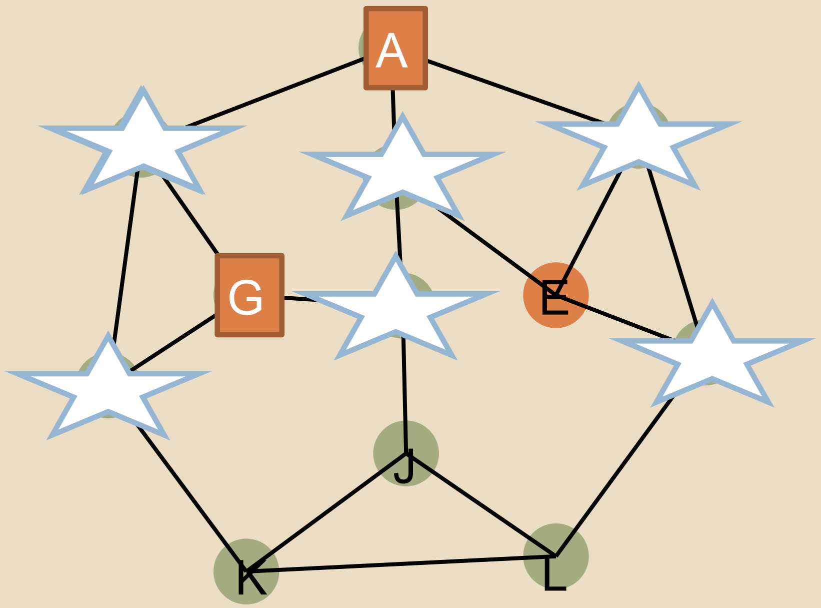


Greedy: all degree 3, pick any, say E

Neighbors (enemies) of E forced out of set

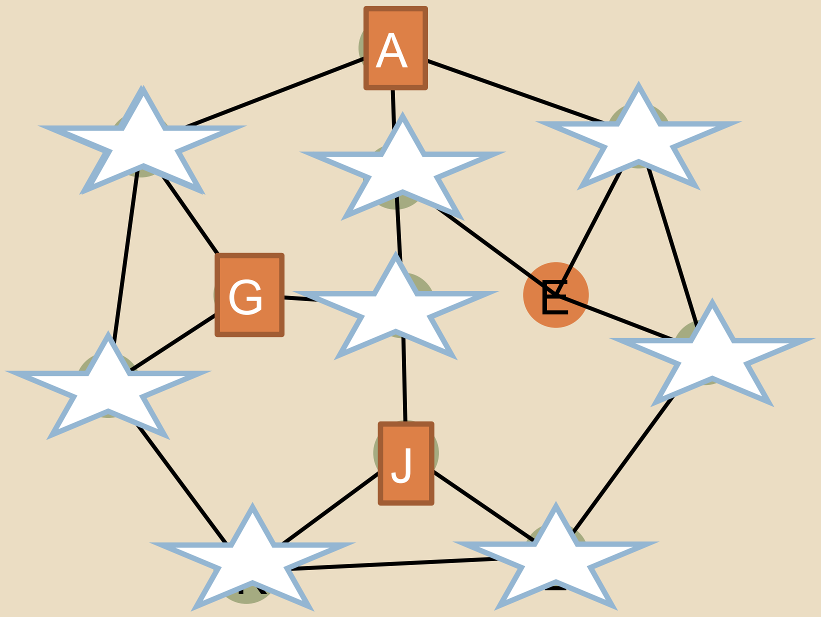
Lowest degree is now A

AN EXAMPLE



Many degree 2 vertices we could choose next, say G

AN EXAMPLE

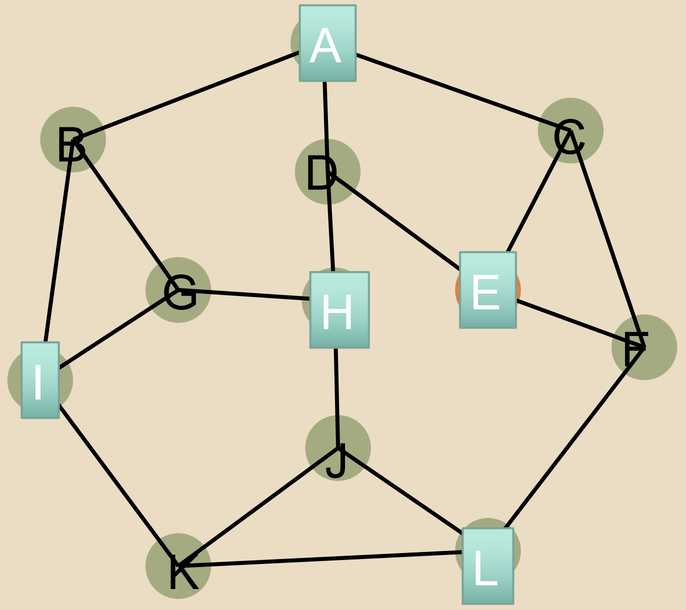


Many degree 2 vertices we could choose next, say G

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…..

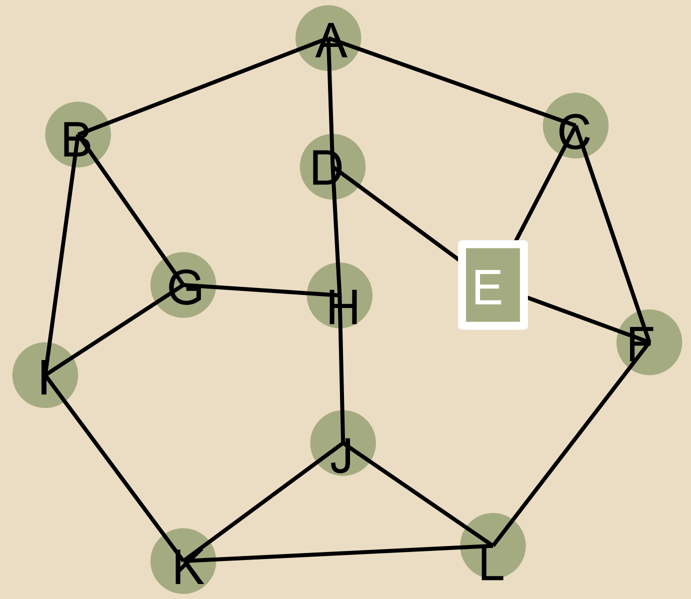
- What are the possible answers to this decision? Yes or No

- How do the answers affect the problem to be solved in the future?

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

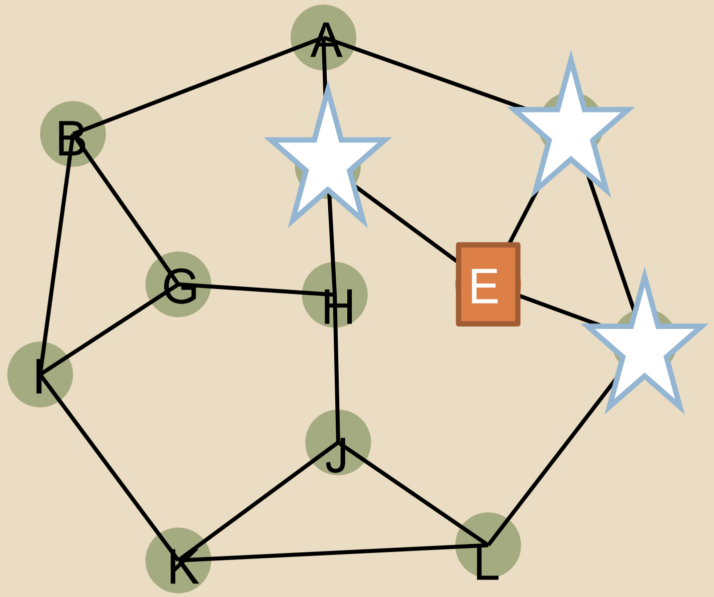
If we don’t pick E: Recurse on subgraph $G - \{E\}$.

AN EXAMPLE



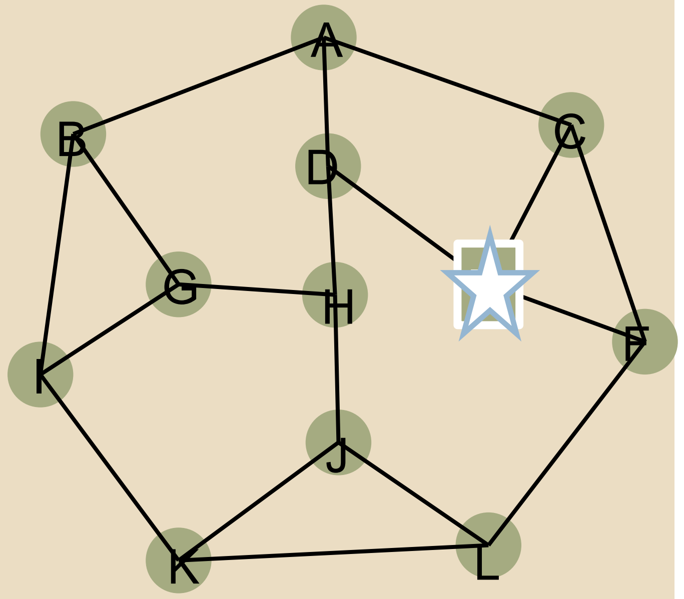
Local decision : Is E in S?
Possible answers: Yes, No

AN EXAMPLE

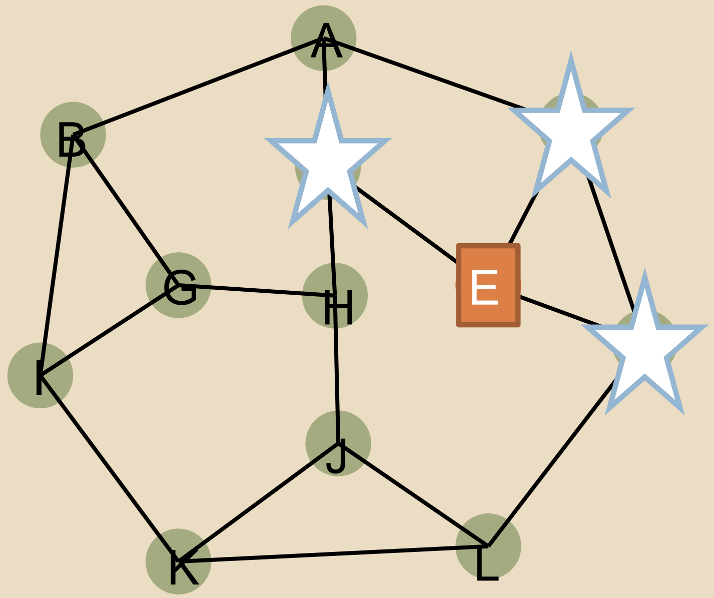


Local decision : Is E in S?
YES OR NO

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])

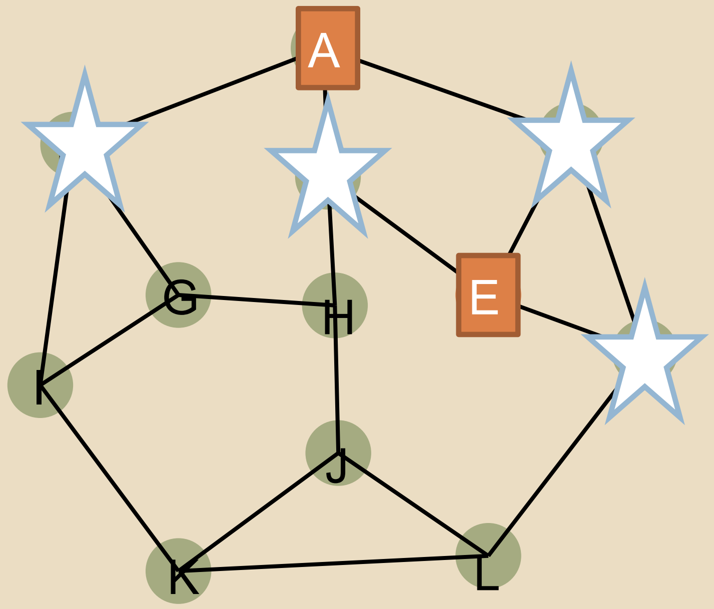


AN EXAMPLE



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

AN EXAMPLE



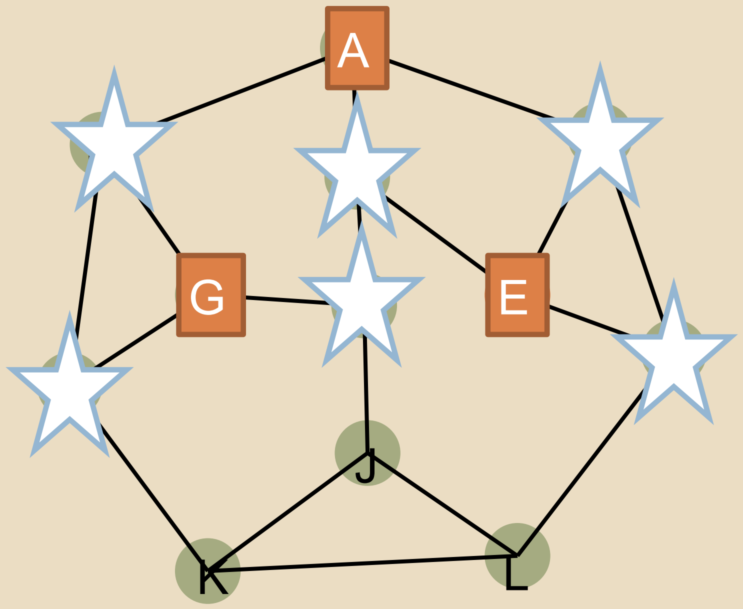
Local decision : Is E in S?
Case 1 : Yes
Consequences: Neighbors not in S

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

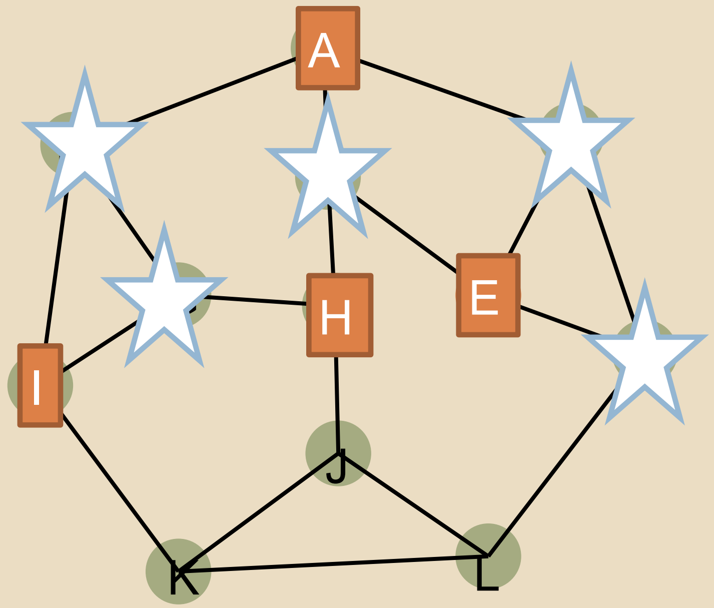
Is G in S?

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

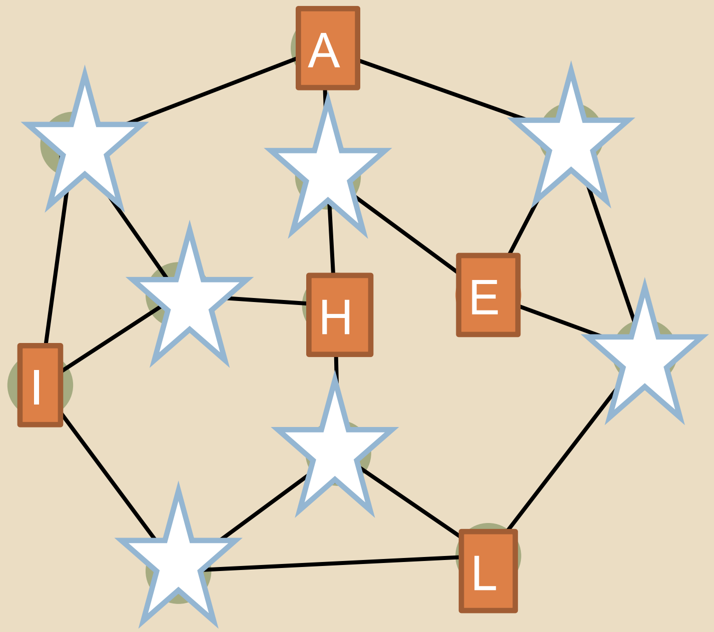
Go on to next local decision

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

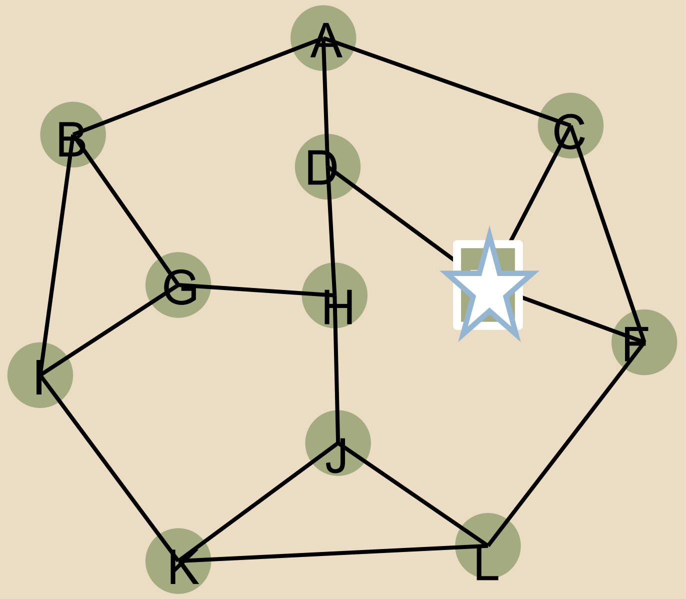
Is G in S?

Case 1b: No

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

Case 1 b: Get set of size 5

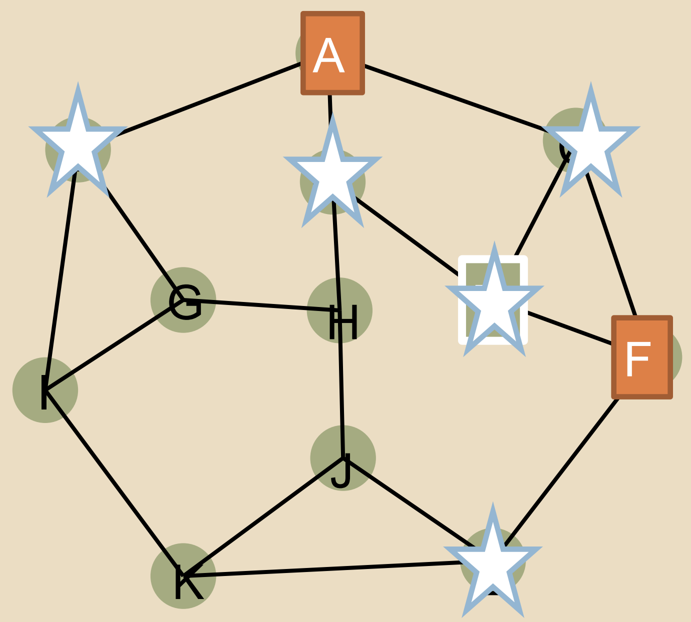
BACKTRACK AGAIN



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

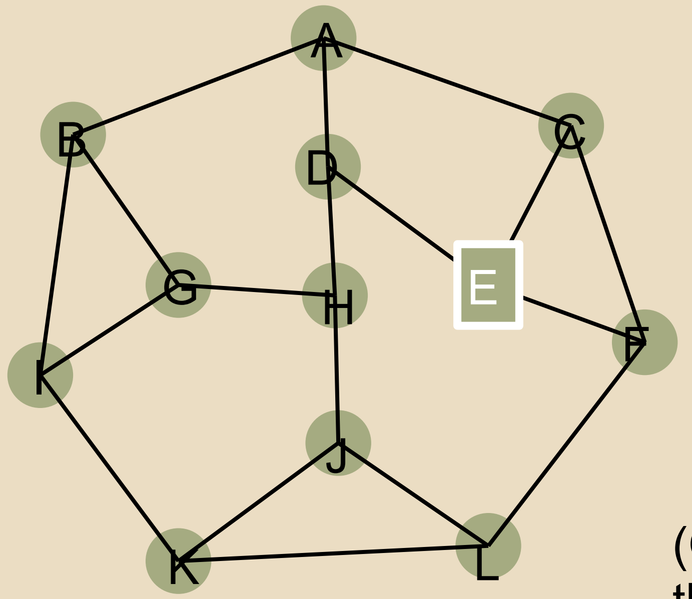
F is in S

Cycle of 5 : get 2

So this case eventually gets 4

Now we KNOW Case 1b is best

AN EXAMPLE

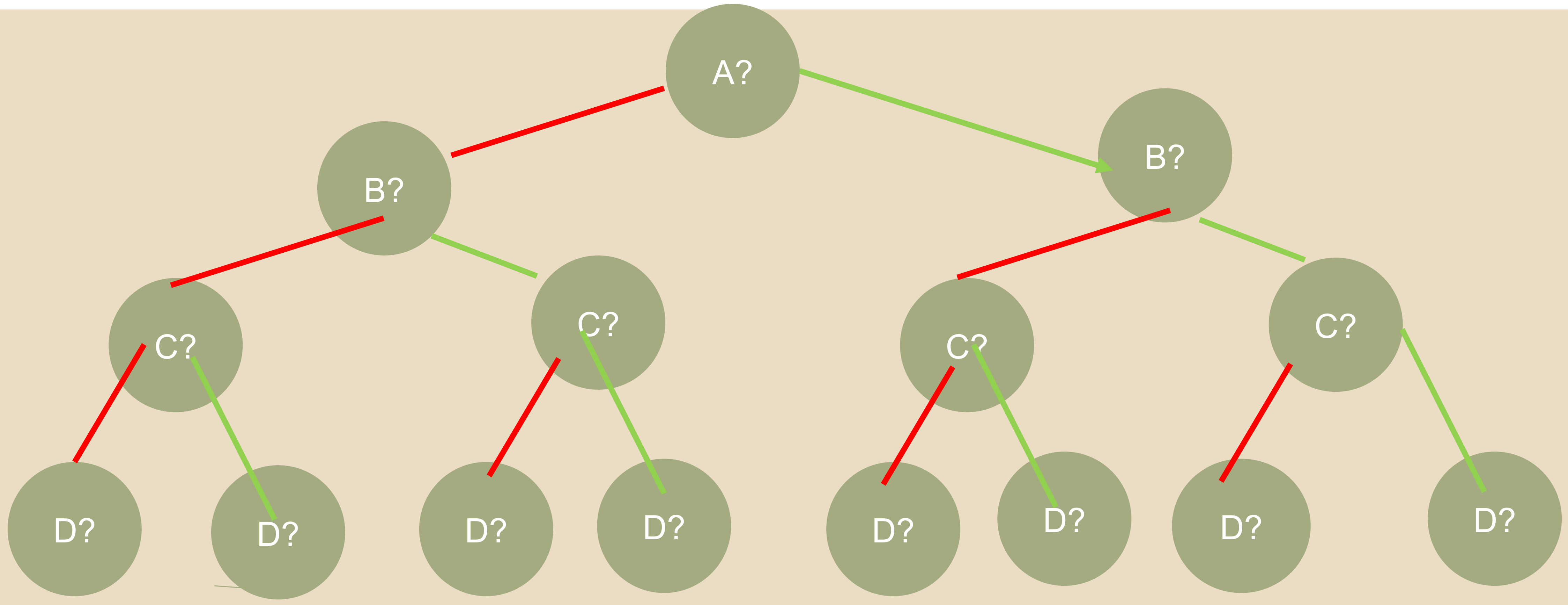


12 vertices means 4096 subsets

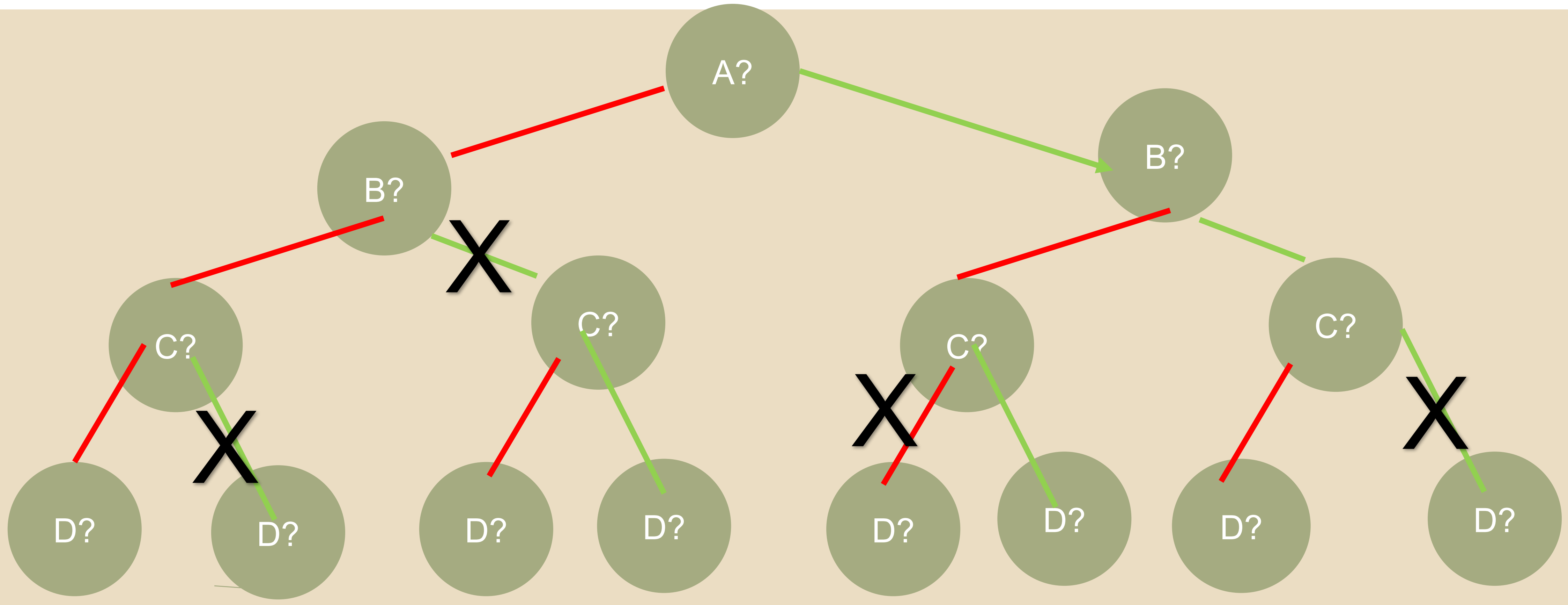
But in the end, we only needed 4 cases

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

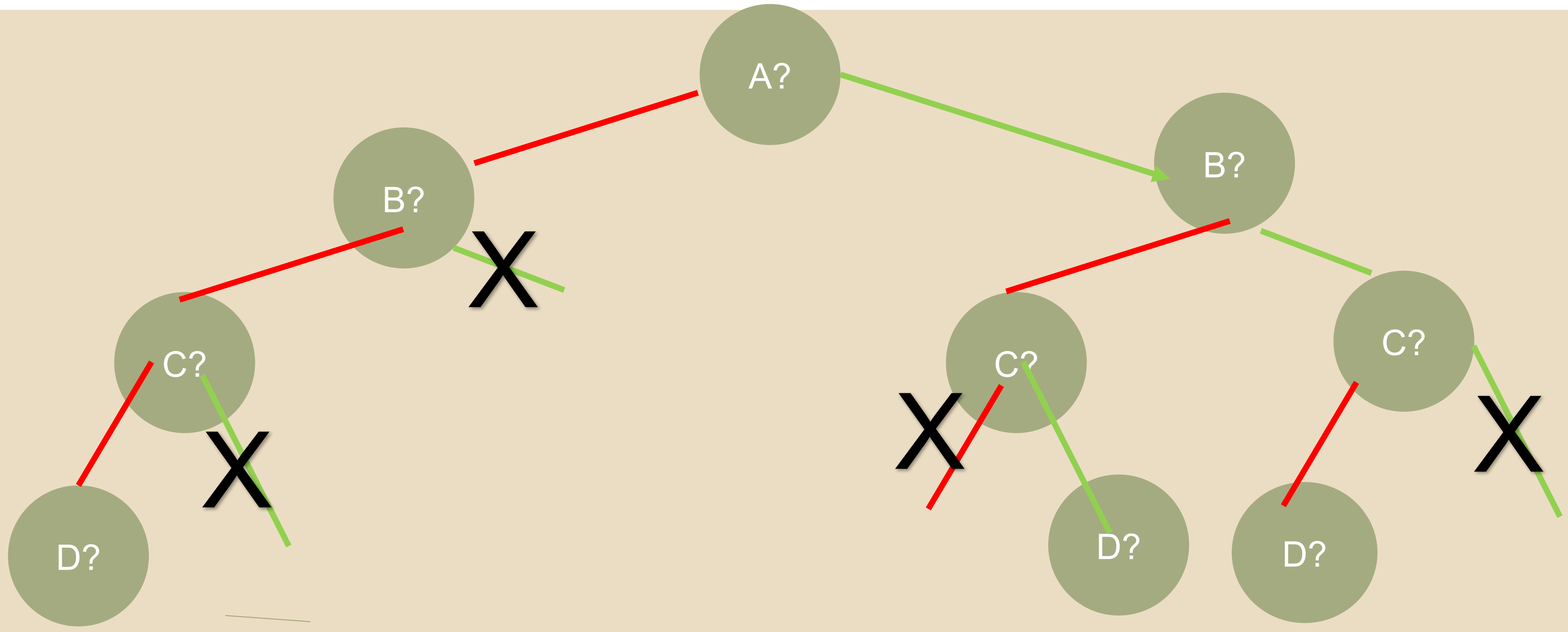
HOW BACKTRACKING HELPS



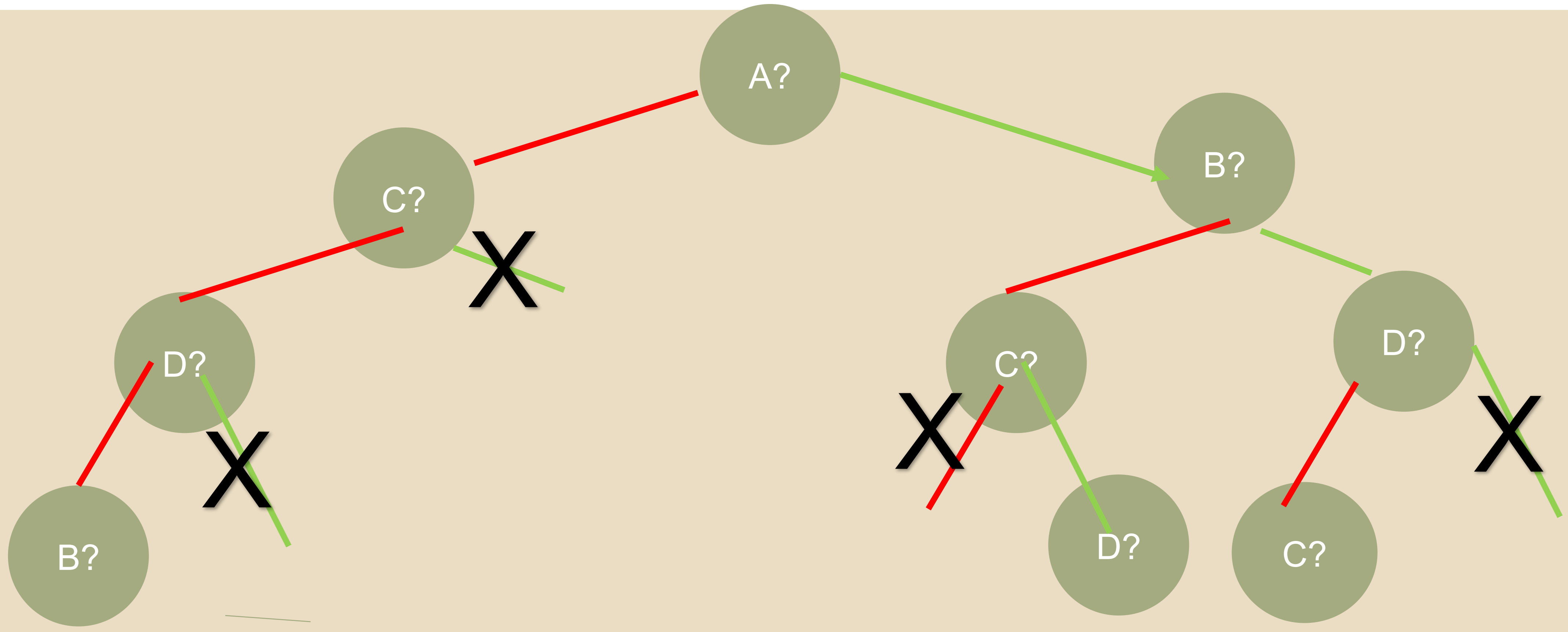
HOW BACKTRACKING HELPS



HOW BACKTRACKING HELPS



ORDER CAN BE ADAPTIVE


A?
C?
B?
X
D?
C?
D?
X
X
X
B?
D?
C?