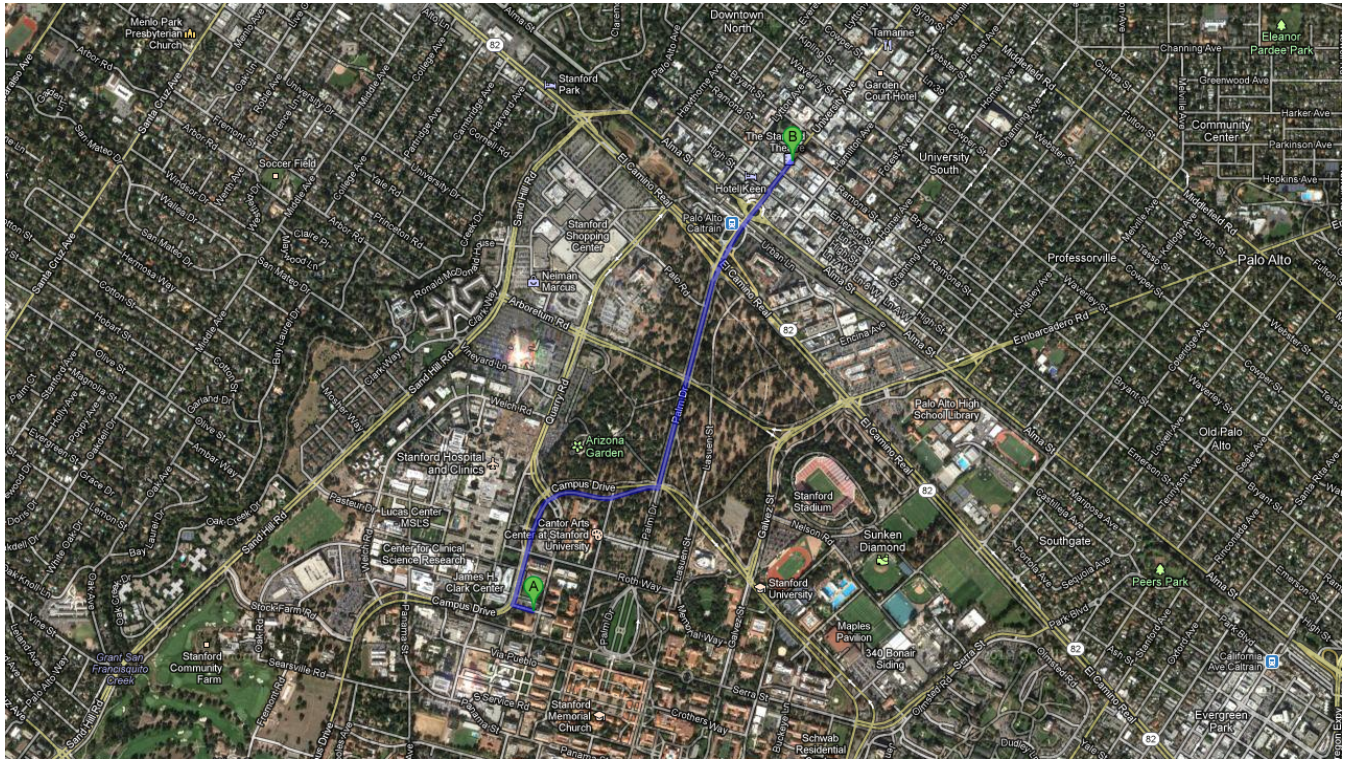
Application: route finding



Objective: shortest? fastest? most scenic?

Actions: go straight, turn left, turn right

- Route finding is perhaps the most canonical example of a search problem. We are given as the input a map, a source point and a destination point. The goal is to output a sequence of actions (e.g., go straight, turn left, or turn right) that will take us from the source to the destination.
- We might evaluate action sequences based on an objective (distance, time, or pleasantness).

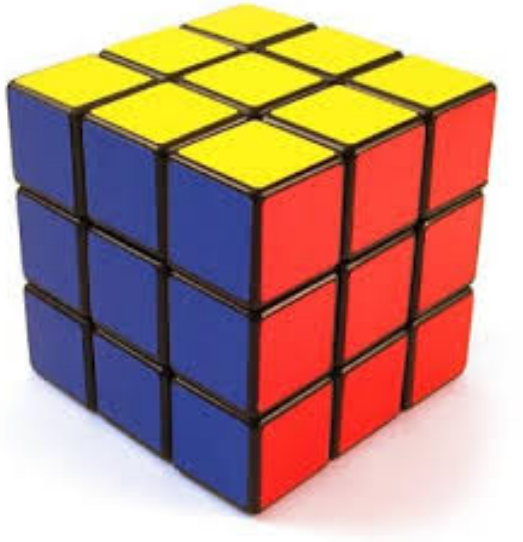
Application: robot motion planning

Objective: fastest? most energy efficient? safest?

Actions: translate and rotate joints

- In robot motion planning, the goal is get a robot to move from one position/pose to another. The desired output trajectory consists of individual actions, each action corresponding to moving or rotating the joints by a small amount.
- Again, we might evaluate action sequences based on various resources like time or energy.

Application: solving puzzles



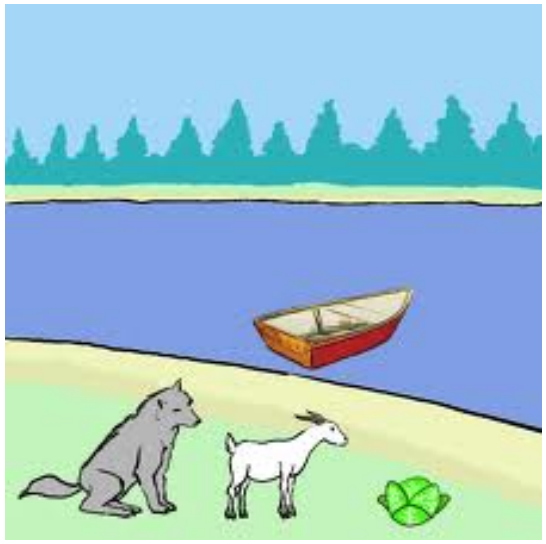
Objective: reach a certain configuration

Actions: move pieces (e.g., `Move12Down`)

- In solving various puzzles, the output solution can be represented by a sequence of individual actions. In the Rubik's cube, an action is rotating one slice of the cube. In the 15-puzzle, an action is moving one square to an adjacent free square.
- In puzzles, even finding one solution might be an accomplishment. The more ambitious might want to find the best solution (say, minimize the number of moves).

Roadmap

Tree search



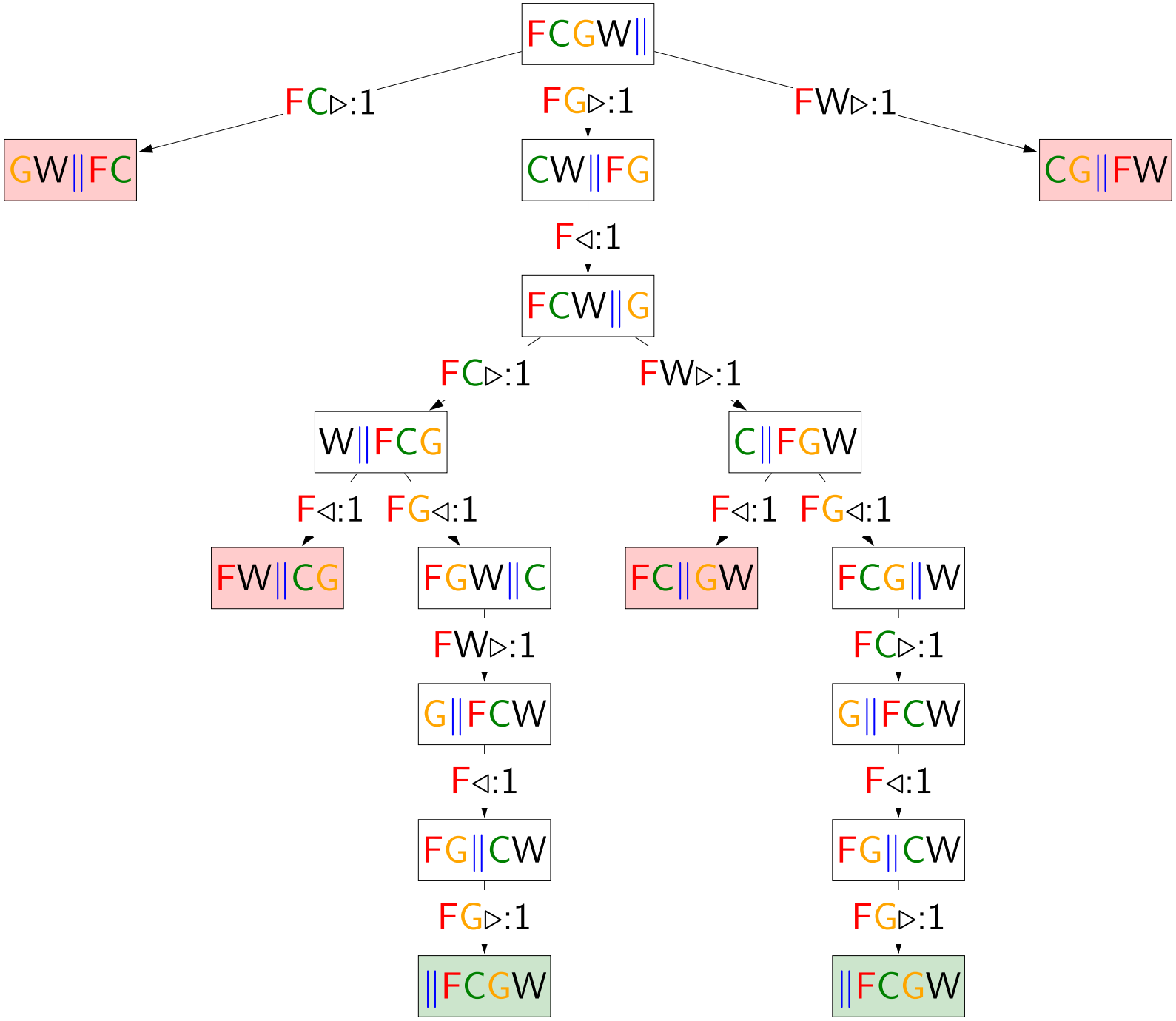
Farmer Cabbage Goat Wolf

Actions:

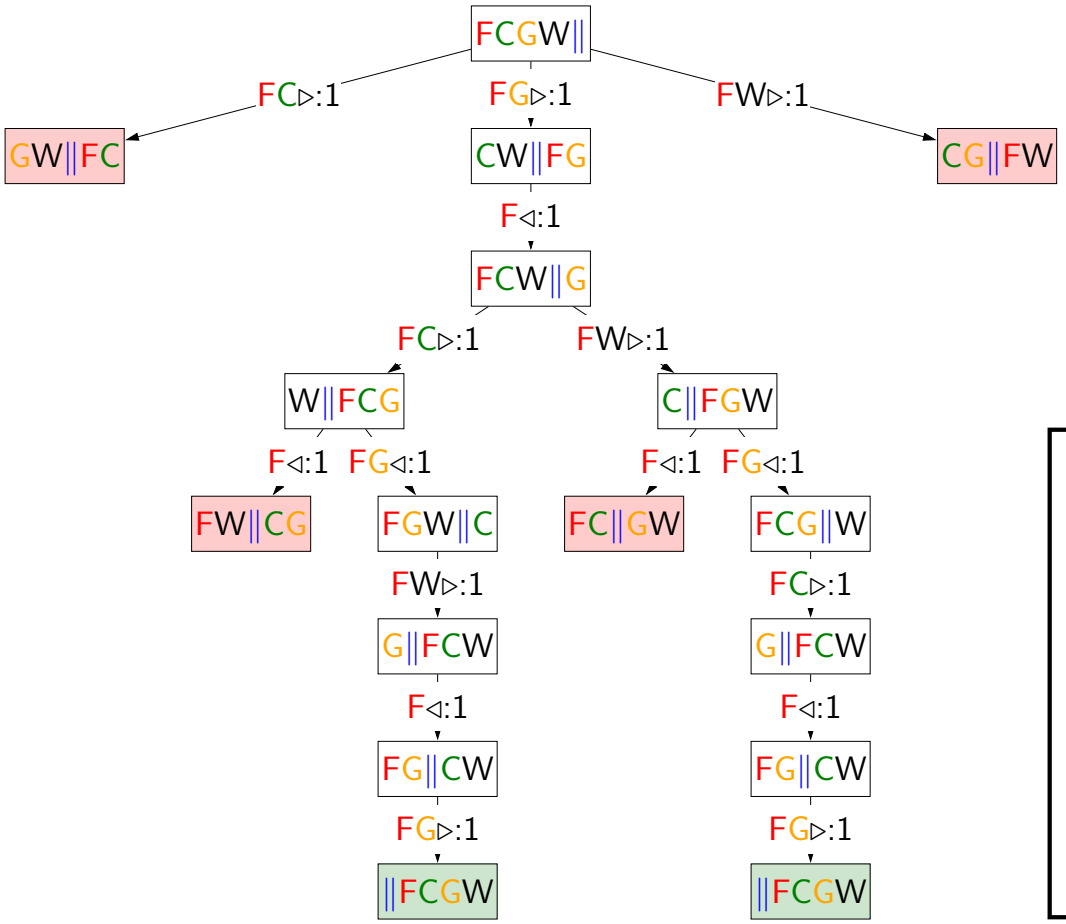
F▷	F◁
FC▷	FC◁
FG▷	FG◁
FW▷	FW◁

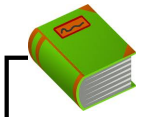
Approach: build a **search tree** ("what if?")

- We first start with our boat crossing puzzle. While you can possibly solve it in more clever ways, let us approach it in a very brain-dead, simple way, which allows us to introduce the notation for search problems.
- For this problem, we have eight possible actions, which will be denoted by a concise set of symbols. For example, the action FG$\triangleright$ means that the farmer will take the goat across to the right bank; F$\triangleleft$ means that the farmer is coming back to the left bank alone.


FCGW||
FC▷:1
FG▷:1
FW▷:1
GW||FC
CW||FG
CG||FW
F◁:1
FCW||G
FC▷:1
FW▷:1
W||FCG
C||FGW
F◁:1
FG◁:1
F◁:1
FG◁:1
FW||CG
FGW||C
FC||GW
FCG||W
FW▷:1
FC▷:1
G||FCW
G||FCW
F◁:1
F◁:1
FG||CW
FG||CW
FG▷:1
FG▷:1
||FCGW
||FCGW

Search problem



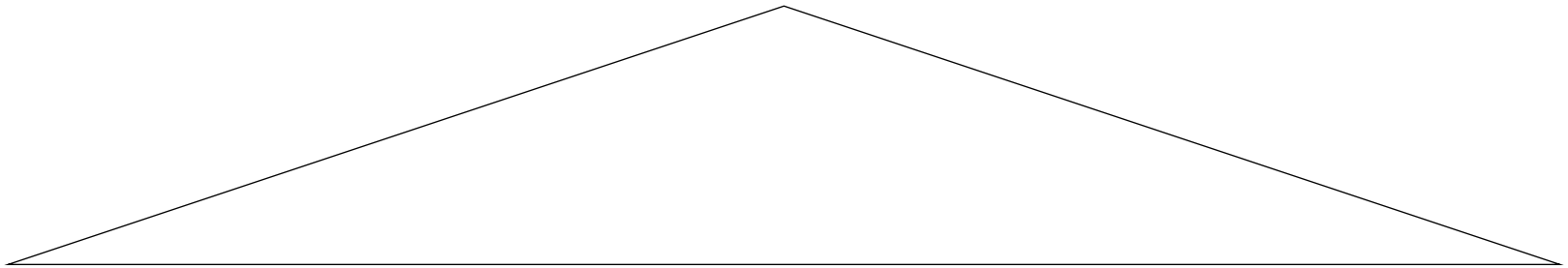


Definition: search problem

- s_{start}: starting state
- Actions(s): possible actions
- Cost(s, a): action cost
- Succ(s, a): successor
- IsEnd(s): reached end state?

- We will build what we will call a **search tree**. The root of the tree is the start state s_{start}, and the leaves are the end states (IsEnd(s) is true). Each edge leaving a node s corresponds to a possible action $a \in$ Actions(s) that could be performed in state s. The edge is labeled with the action and its cost, written a : Cost(s, a). The action leads deterministically to the successor state Succ(s, a), represented by the child node.
- In summary, each root-to-leaf path represents a possible action sequence, and the sum of the costs of the edges is the cost of that path. The goal is to find the root-to-leaf path that ends in a valid end state with minimum cost.
- Note that in code, we usually do not build the search tree as a concrete data structure. The search tree is used merely to visualize the computation of the search algorithms and study the structure of the search problem.
- For the boat crossing example, we have assumed each action (a safe river crossing) costs 1 unit of time. We disallow actions that return us to an earlier configuration. The green nodes are the end states. The red nodes are not end states but have no successors (they result in the demise of some animal or vegetable). From this search tree, we see that there are exactly two solutions, each of which has a total cost of 7 steps.

Backtracking search



[whiteboard: search tree]

If b actions per state, maximum depth is D actions:

- Memory: $O(D)$ (small)
- Time: $O(b^D)$ (huge) [$2^{50} = 1125899906842624$]

Backtracking search

Algorithm: backtracking search

def backtrackingSearch(s, path):
    If IsEnd(s): update minimum cost path
    For each action $a \in$ Actions(s):
        Extend path with Succ(s, a) and Cost(s, a)
        Call backtrackingSearch(Succ(s, a), path)
    Return minimum cost path

[semi-live solution: `backtrackingSearch`]

- Now let's put modeling aside and suppose we are handed a search problem. How do we construct an algorithm for finding a **minimum cost path** (not necessarily unique)?
- We will start with **backtracking search**, the simplest algorithm which just tries all paths. The algorithm is called recursively on the current state s and the path leading up to that state. If we have reached a goal, then we can update the minimum cost path with the current path. Otherwise, we consider all possible actions a from state s, and recursively search each of the possibilities.
- Graphically, backtracking search performs a depth-first traversal of the search tree. What is the time and memory complexity of this algorithm?
- To get a simple characterization, assume that the search tree has maximum depth D (each path consists of D actions/edges) and that there are b available actions per state (the **branching factor** is b).
- It is easy to see that backtracking search only requires $O(D)$ memory (to maintain the stack for the recurrence), which is as good as it gets.
- However, the running time is proportional to the number of nodes in the tree, since the algorithm needs to check each of them. The number of nodes is $1 + b + b^2 + \cdots + b^D = \frac{b^{D+1}-1}{b-1} = O(b^D)$. Note that the total number of nodes in the search tree is on the same order as the number of leaves, so the cost is always dominated by the last level.
- In general, there might not be a finite upper bound on the depth of a search tree. In this case, there are two options: (i) we can simply cap the maximum depth and give up after a certain point or (ii) we can disallow visits to the same state.
- It is worth mentioning that the greedy algorithm that repeatedly chooses the lowest action myopically won't work. Can you come up with an example?

Depth-first search

Assumption: zero action costs

Assume action costs $\text{Cost}(s, a) = 0$.

Idea: Backtracking search + stop when find the first end state.

If b actions per state, maximum depth is D actions:

- Space: still $O(D)$
- Time: still $O(b^D)$ worst case, but could be much better if solutions are easy to find

- Backtracking search will always work (i.e., find a minimum cost path), but there are cases where we can do it faster. But in order to do that, we need some additional assumptions — there is no free lunch.
- Suppose we make the assumption that all the action costs are zero. In other words, all we care about is finding a valid action sequence that reaches the goal. Any such sequence will have the minimum cost: zero.
- In this case, we can just modify backtracking search to not keep track of costs and then stop searching as soon as we reach a goal. The resulting algorithm is **depth-first search** (DFS), which should be familiar to you. The worst time and space complexity are of the same order as backtracking search. In particular, if there is no path to an end state, then we have to search the entire tree.
- However, if there are many ways to reach the end state, then we can stop much earlier without exhausting the search tree. So DFS is great when there are an abundance of solutions.

Ver Algoritmo DFS

Breadth-first search

Assumption: constant action costs

Assume action costs $\text{Cost}(s, a) = c$ for some $c \geq 0$.

Idea: explore all nodes in order of increasing depth.

Legend: b actions per state, solution has d actions

- Space: now $O(b^d)$ (a lot worse!)
- Time: $O(b^d)$ (better, depends on d, not D)

- **Breadth-first search** (BFS), which should also be familiar, makes a less stringent assumption, that all the action costs are the same non-negative number. This effectively means that all the paths of a given length have the same cost.
- BFS maintains a queue of states to be explored. It pops a state off the queue, then pushes its successors back on the queue.
- BFS will search all the paths consisting of one edge, two edges, three edges, etc., until it finds a path that reaches a end state. So if the solution has d actions, then we only need to explore $O(b^d)$ nodes, thus taking that much time.
- However, a potential show-stopper is that BFS also requires $O(b^d)$ space since the queue must contain all the nodes of a given level of the search tree. Can we do better?

Ver Algoritmo BFS

DFS with iterative deepening

Assumption: constant action costs

Assume action costs $\text{Cost}(s, a) = c$ for some $c \geq 0$.

Idea:

- Modify DFS to stop at a maximum depth.
- Call DFS for maximum depths $1, 2, \ldots$.

  DFS on d asks: is there a solution with d actions?

Legend: b actions per state, solution size d

- Space: $O(d)$ (saved!)
- Time: $O(b^d)$ (same as BFS)

- Yes, we can do better with a trick called **iterative deepening**. The idea is to modify DFS to make it stop after reaching a certain depth. Therefore, we can invoke this modified DFS to find whether a valid path exists with at most d edges, which as discussed earlier takes $O(d)$ space and $O(b^d)$ time.
- Now the trick is simply to invoke this modified DFS with cutoff depths of $1, 2, 3, \ldots$ until we find a solution or give up. This algorithm is called DFS with iterative deepening (DFS-ID). In this manner, we are guaranteed optimality when all action costs are equal (like BFS), but we enjoy the parsimonious space requirements of DFS.
- One might worry that we are doing a lot of work, searching some nodes many times. However, keep in mind that both the number of leaves and the number of nodes in a search tree is $O(b^d)$ so asymptotically DFS with iterative deepening is the same time complexity as BFS.

Tree search algorithms

Legend: b actions/state, solution depth d, maximum depth D

Algorithm	**Action costs**	**Space**	**Time**
DFS	zero	$O(D)$	$O(b^D)$
BFS	constant ≥ 0	$O(b^d)$	$O(b^d)$
DFS-ID	constant ≥ 0	$O(d)$	$O(b^d)$
Backtracking	any	$O(D)$	$O(b^D)$

- Always exponential time
- Avoid exponential space with DFS-ID

- Here is a summary of all the tree search algorithms, the assumptions on the action costs, and the space and time complexities.
- The take-away is that we can't avoid the exponential time complexity, but we can certainly have linear space complexity. Space is in some sense the more critical dimension in search problems. Memory cannot magically grow, whereas time "grows" just by running an algorithm for a longer period of time, or even by parallelizing it across multiple machines (e.g., where each processor gets its own subtree to search).