Lecture 9: Search (I)

EMAT31530/Feb 2018/Raul Santos-Rodriguez

Have a look at ...

... Russell and Norvig (Ch. 3)

Question

A farmer wants to get his cabbage, goat, wolf across a river. He has a boat that only holds two. He cannot leave cabbage and goat alone or goat and wolf alone.

How many river crossings does he need?

- 4
- 5
- 6
- 7
- No solution

Question

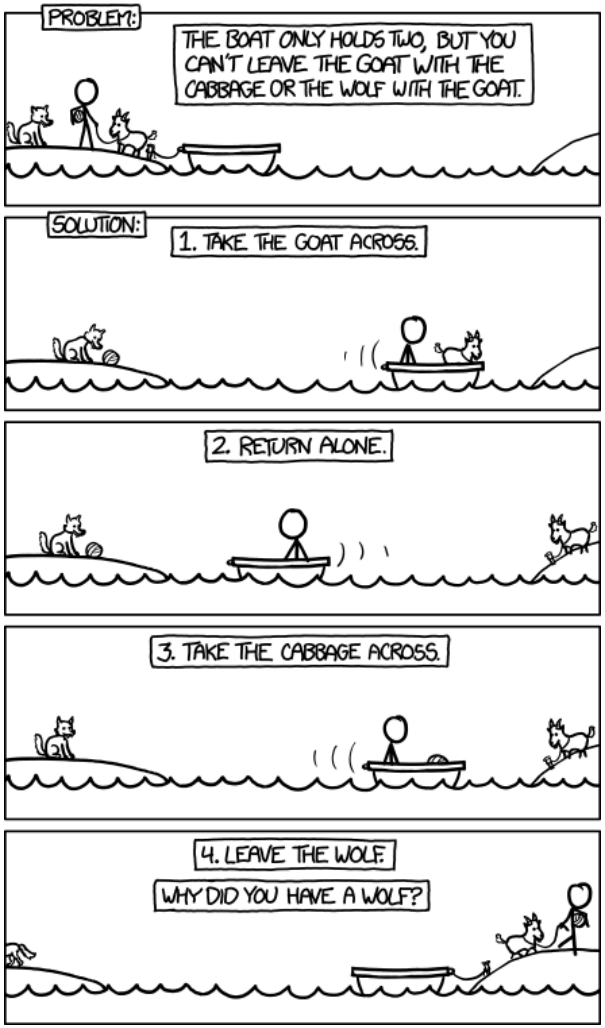
A farmer wants to get his cabbage, goat, wolf across a river. He has a boat that only holds two. He cannot leave cabbage and goat alone or goat and wolf alone.

How many river crossings does he need?

- 4
- 5
- 6
- 7
- No solution

When you solve this problem, try to think about how you did it. You probably simulated the scenario in your head, trying to send the farmer over with the goat, observing the consequences. If nothing got eaten, you might continue with the next action. Otherwise, you undo that move and try something else. How can we get a machine to do this automatically? One of the things we need is a systematic approach that considers all the possibilities. Search problems and their associated algorithms will allow us to do this.

A clever solution



http://xkcd.com/1134/

The road so far

- Machine learning

  Binary classification:

  $$x \to \boxed{?} \to y \in \{-1, +1\}, \text{ single action}$$

- Search
- Bayesian networks
- Markov decision process
- Game theory

The road so far

- Machine learning

  Binary classification:

  $x \rightarrow \boxed{?} \rightarrow y \in \{-1, +1\}$, single action

- Search

  Search problem:

  $x \rightarrow \boxed{?} \rightarrow$ action sequence $(a_1, a_2, a_3, a_4, \ldots)$

- Bayesian networks
- Markov decision process
- Game theory

The road so far

The road so far

- Machine learning
- Search
- Bayesian networks
- Markov decision process
- Game theory

Binary classification:

$x \to \boxed{?} \to y \in \{-1, +1\}$, single action

Search problem:

$x \to \boxed{?} \to$ action sequence $(a_1, a_2, a_3, a_4, \ldots)$

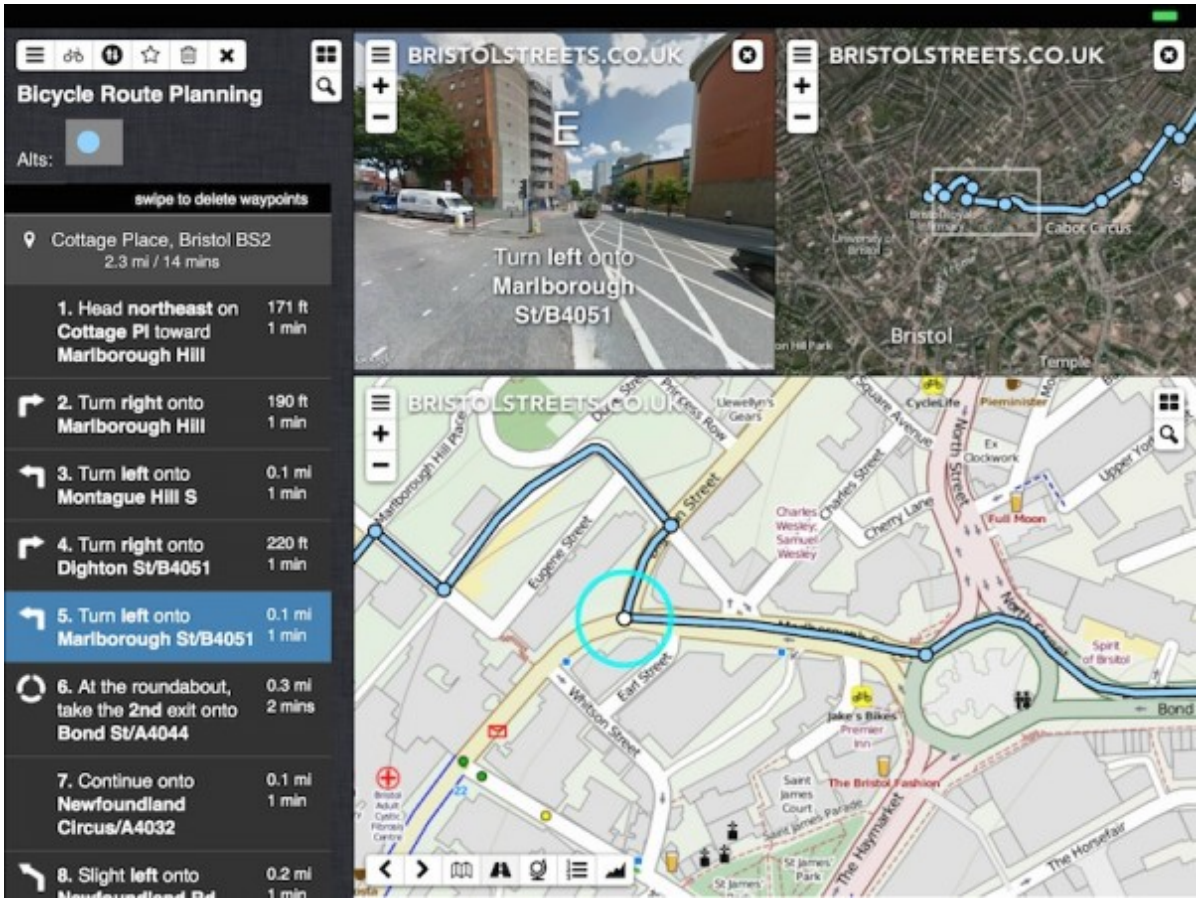
While simple machine learning models were appropriate for some applications such as sentiment classification or spam filtering, the applications we will look at today, such as solving puzzles, demand more. To tackle these new problems, we will introduce search problems, our first instance of a state-based model. In a search problem, in a sense, we are still building a predictor which takes an input x, but the predictor will now return an entire action sequence, not just a single action.

Outline

Search or how to find a sequence of actions that achieves a goal when no single action will do. Today we will cover

- Tree search
- Dynamic programming

Application: route finding

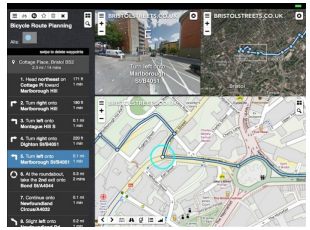


Actions: go straight, turn left, turn right

Objective: shortest? fastest? most scenic?

Application: route finding

Application: route finding

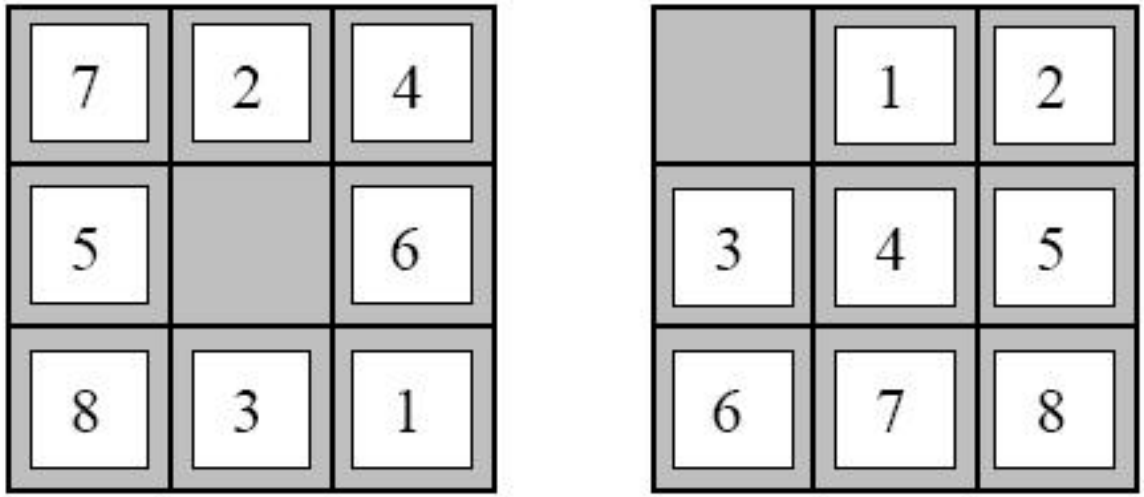


Actions: go straight, turn left, turn right

Objective: shortest? fastest? most scenic?

There are many real-world examples of this paradigm, which we will describe next. For each example, the key is to decompose the output solution into a sequence of primitive actions. In addition, we need to think about how to evaluate different possible outputs. Route finding is perhaps the most typical example of a search problem. We are given as the input x 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. In addition, we might evaluate action sequences based on distance, time, or pleasantness.

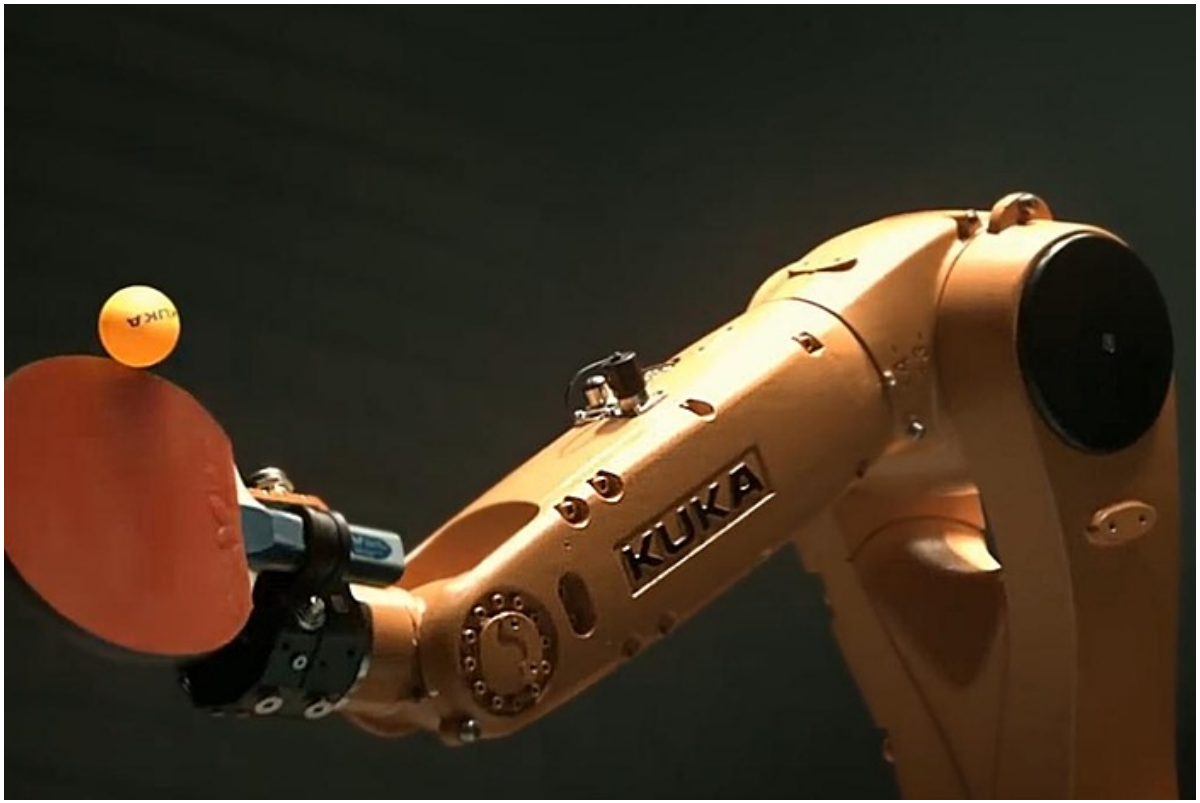
Application: solving puzzles



Actions: move pieces

Objective: reach a certain configuration

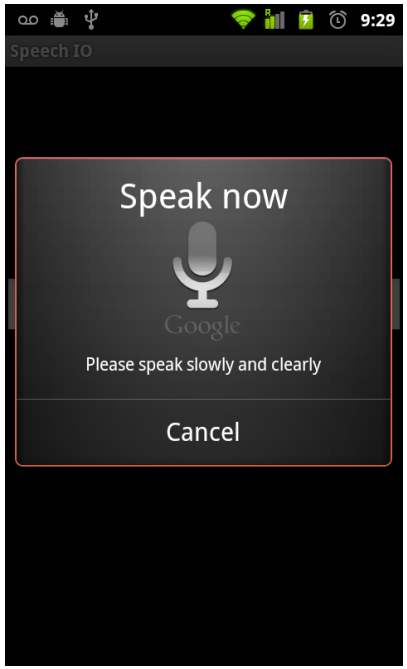
Application: robot planning



Actions: translate and rotate joints

Objective: fastest? most energy efficient? safest?

Application: speech recognition



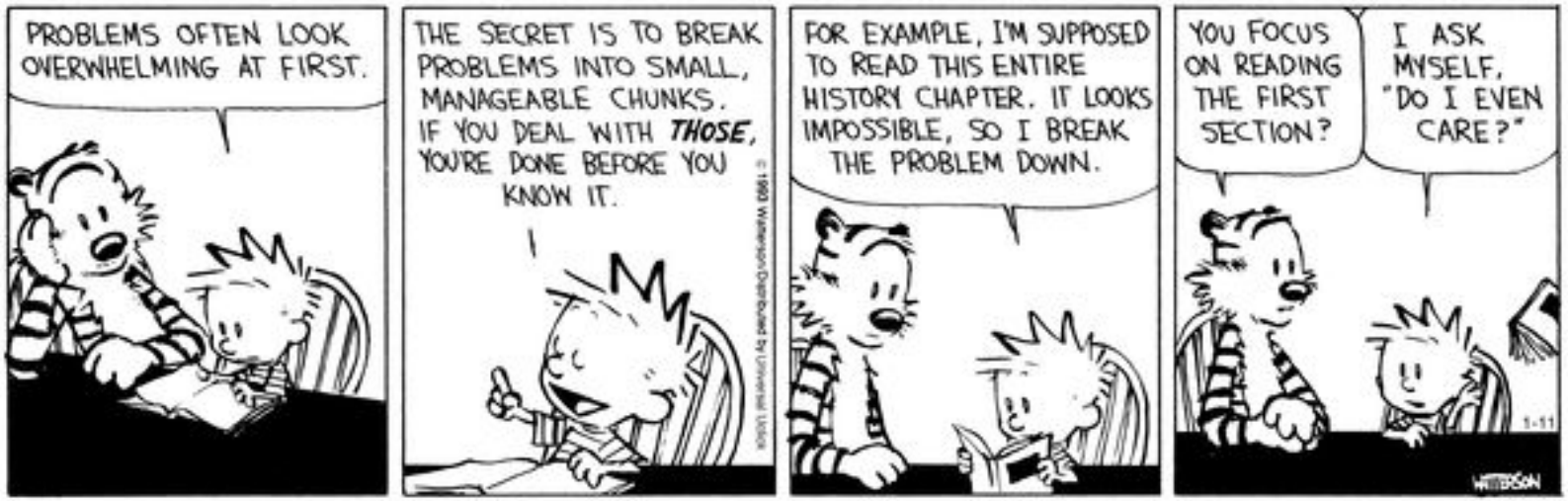
Actions: transcribe single sounds (phonemes)

Objective: fluent english? robust?

In all of these examples $\rightarrow$ sequence of actions

Idea

Decompose our very complex problem into small problems

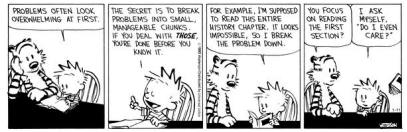


Search

In all of these examples → sequence of actions

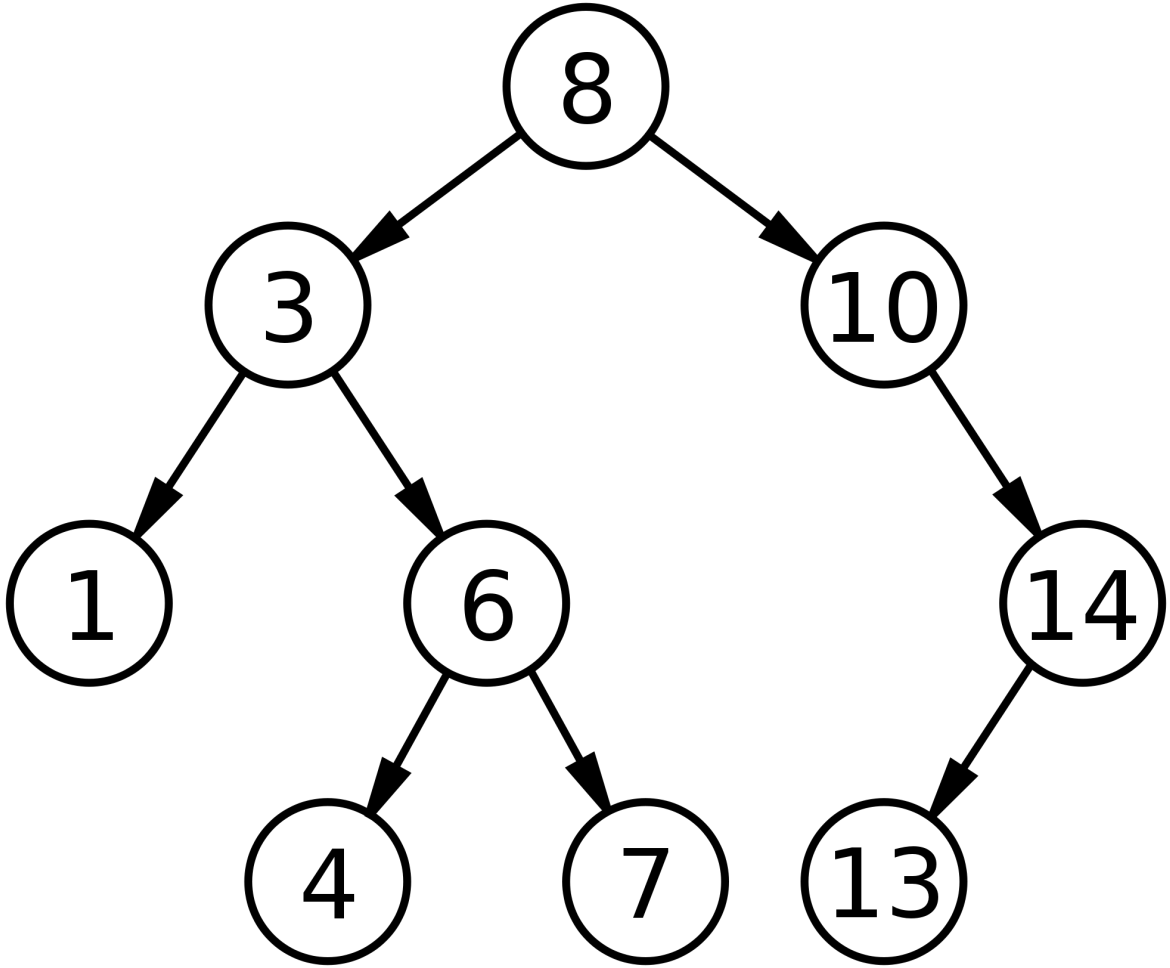
Idea

Decompose our very complex problem into small problems



Note that the number of possible action sequences of length n is exponential in n, which is too many to enumerate if we want to find the best one. So just decomposing the solution into actions isn't enough. Later, we will introduce the notion of a state. With an appropriate choice of state, we can construct efficient polynomial-time algorithms.

Tree search



Tree search: example

Farmer Cabbage Goat Wolf

Actions:

F▷	F◁
FC▷	FC◁
FG▷	FG◁
FW▷	FW◁

[Liang 2014]

Tree search: example

Farmer Cabbage Goat Wolf

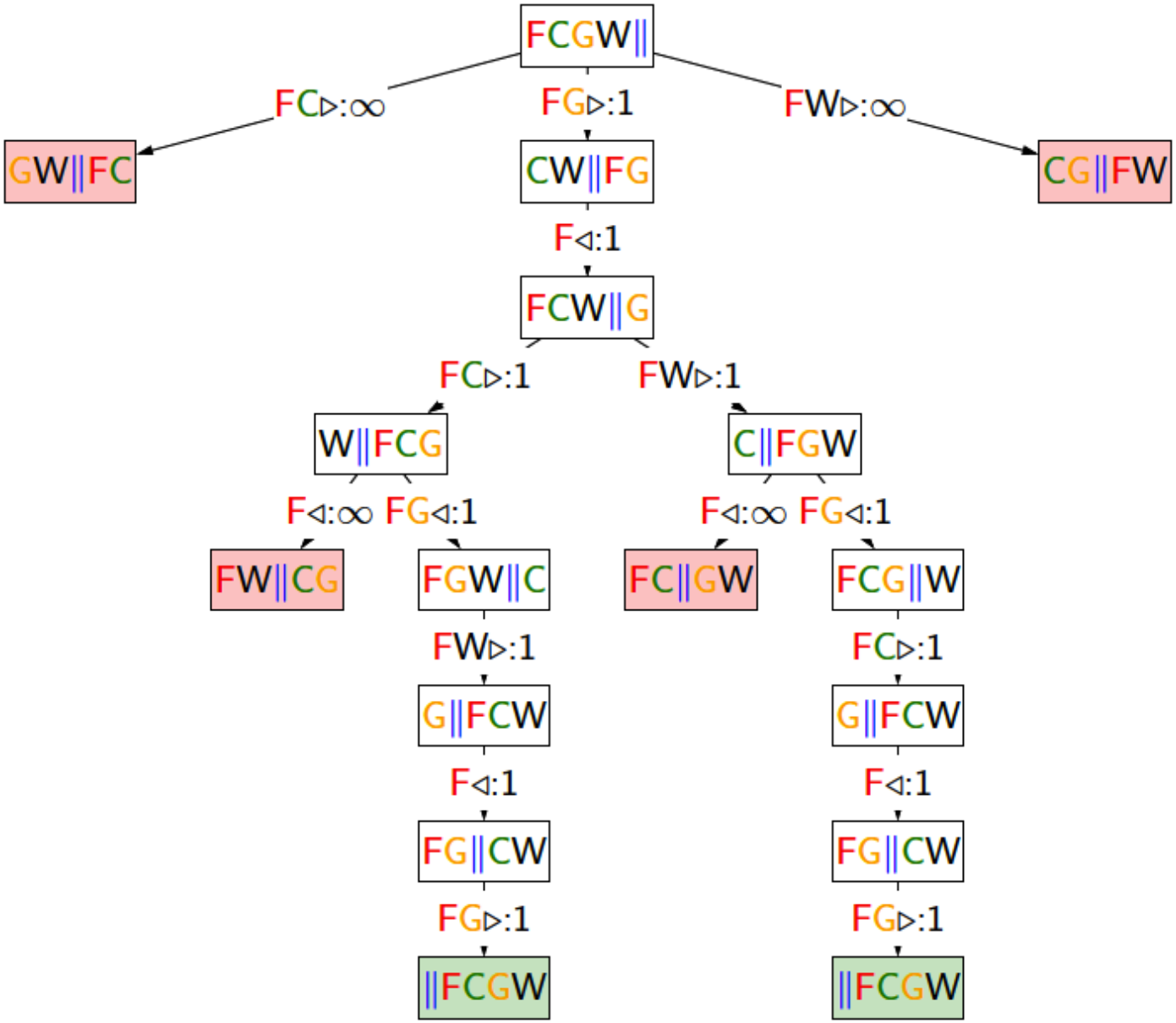
Actions:

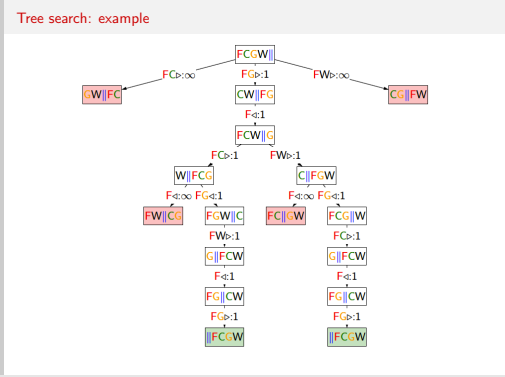
F▷ F◁
FC▷ FC◁
FG▷ FG◁
FW▷ FW◁

[Liang 2014]

For this problem, we have eight possible actions, which will be denoted by a concise set of symbols. For example, the action FG▷. means that the farmer will take the goat across to the right bank; F◁ means that the farmer is coming back to the left bank alone.

Tree search: example





We will build what we will call a search tree. The root of the tree is the start state, and the leaves are the goal states. Each edge leaving a node s corresponds to a possible action *a* 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, 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 has the minimum cost. For the boat crossing example, we have assumed each action (a safe river crossing) costs 1 unit of time. Invalid actions (ones that result in an eating event) cost ∞ and the successor is marked in red. We disallow actions that return us to an earlier configuration. The green nodes are the goal states. From this search tree, we see that there are exactly two solutions, each of which has a total cost of 7 steps.

Search problem

Search algorithms require a structure to keep track of the search tree that is being constructed:

Search

- s_{start}: starting state
- Actions(s): possible actions
- Cost(s,a): action cost
- Succ(s,a): successors
- Goal(s): found solution?

Search problem

Search algorithms require a structure to keep track of the search tree that is being constructed:

Search

- s_{start}: starting state
- Actions(s): possible actions
- Cost(s,a): action cost
- Succ(s,a): successors
- Goal(s): found solution?

The possible action sequences starting at the initial state form a search tree with the initial state at the root; the branches are actions and the nodes correspond to states in the state space of the problem. The first step is to test whether this is a goal state. Then we need to consider taking various actions. We do this by expanding the current state; that is, applying each legal action to the current state, thereby generating a new set of states. In this case, we add three branches from the parent node leading to the new child nodes. Now we must recursively choose which of these possibilities to consider further.

Measuring performance

Completeness: Is the algorithm guaranteed to find a solution when there is one?

Optimality: Does the strategy find the optimal solution?

Time complexity: How long does it take to find a solution?

Space complexity: How much memory is needed to perform the search?

Complexity

Complexity is expressed in terms of three quantities:

- b, the branching factor or maximum number of successors of any node;
- d, the depth of the shallowest goal node (i.e., the number of steps along the path from the root);
- D, the maximum length of any path in the state space.

Backtracking search

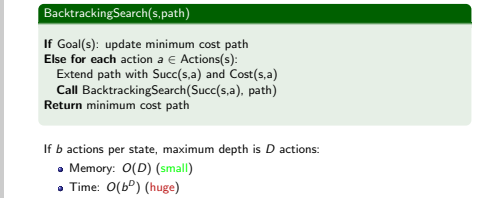
BacktrackingSearch(s,path)

If Goal(s): update minimum cost path
Else 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

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

- Memory: $O(D)$ (small)
- Time: $O(b^D)$ (huge)

Backtracking search



Backtracking search is the simplest algorithm which just tries all paths. The algorithm is called recursively on the current state s and the path *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. To analyze its memory and time complexities, let's 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.

Depth-first search

Idea: Backtracking search + stop when find the first goal state

Assumption: Action costs Cost(s,a) = 0

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

- Memory: $O(D)$ (small)
- Time: $O(b^D)$ (huge) but that is just for the worst case (could be much better if solutions are easy to find)

DFS is great when there are an abundance of solutions

Depth-first search

Idea: Backtracking search + stop when find the first goal state

Assumption: Action costs Cost(s,a) = 0

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

- Memory: $O(D)$ (small)
- Time: $O(b^D)$ (huge) but that is just for the worst case (could be much better if solutions are easy to find)

DFS is great when there are an abundance of solutions

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. 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). The worst time and space complexity are of the same order as backtracking search. In particular, if there is no path to the goal state, then we have to search the entire tree. However, if there are many ways to reach the goal state, then we can stop much earlier without exhausting the search tree. So DFS is great when there are an abundance of solutions.

Depth-first search

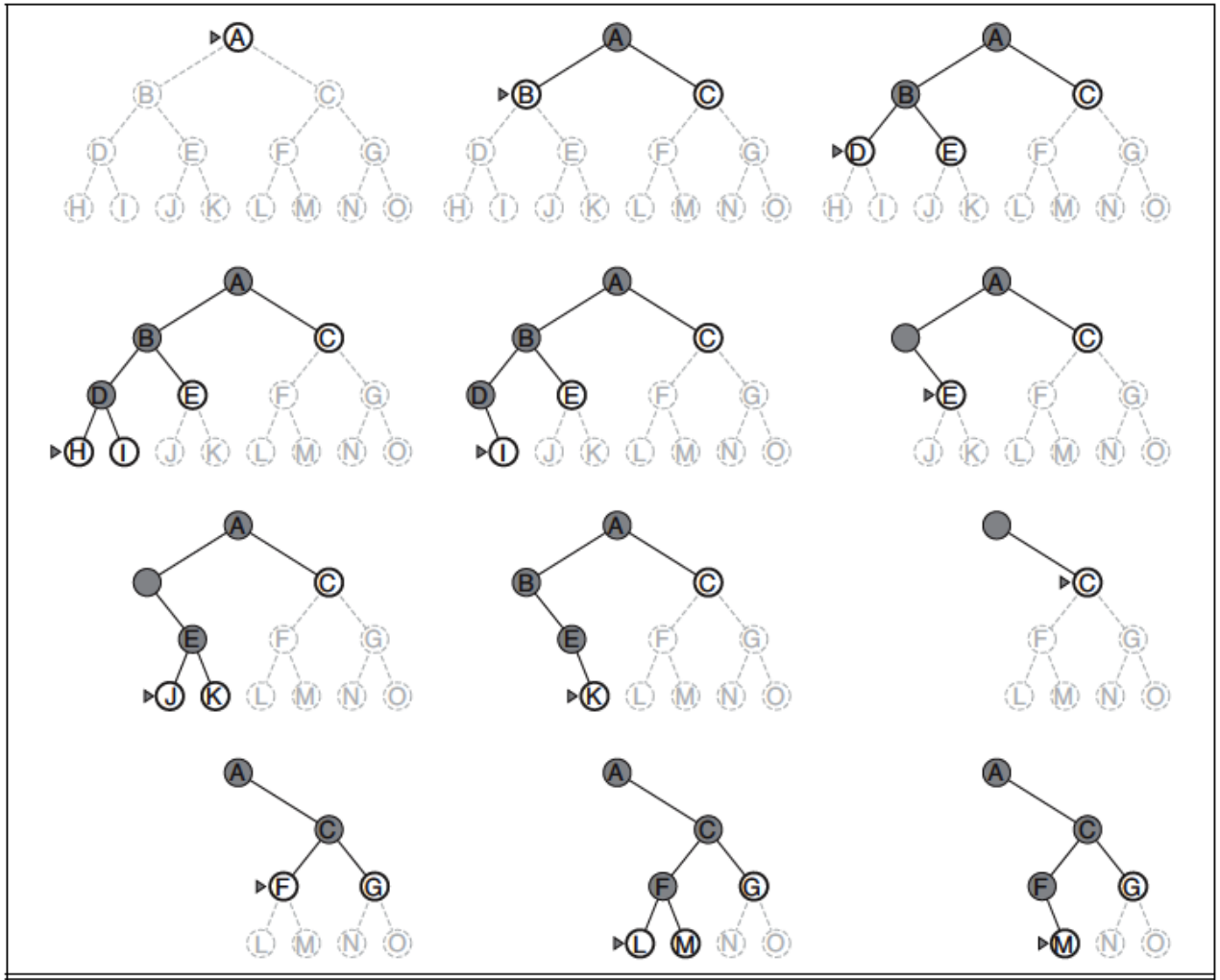


Figure 3.16 Depth-first search on a binary tree. The unexplored region is shown in light gray. Explored nodes with no descendants in the frontier are removed from memory. Nodes at depth 3 have no successors and M is the only goal node.

[Russell and Norvig]

Breadth-first search

Idea: explore all nodes in order of increasing depth

Assumption: Action costs Cost(s,a)=c for some $c \geq 0$

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

- Memory: $O(b^d)$ (worse)
- Time: $O(b^d)$ (depends on d instead of D)

Breadth-first search

Idea: explore all nodes in order of increasing depth

Assumption: Action costs Cost(s,a)=c for some $c \geq 0$

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

- Memory: $O(b^d)$ (worse)
- Time: $O(b^d)$ (depends on d instead of D)

Breadth-first search (BFS) 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 goal 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.

Breadth-first search

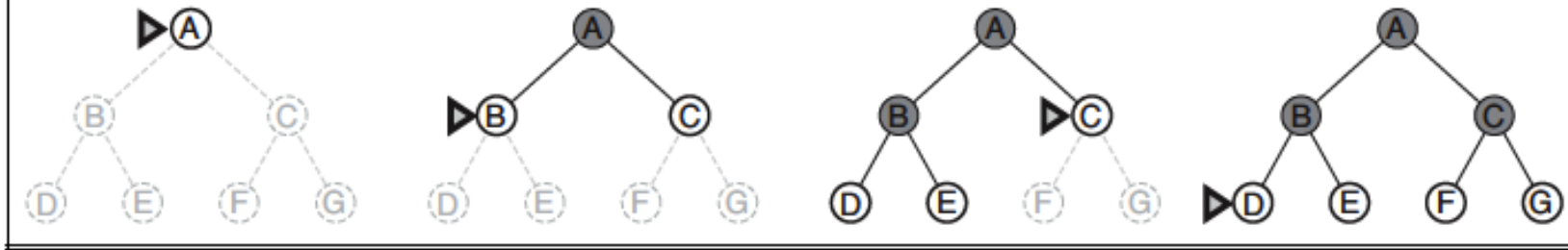


Figure 3.12 Breadth-first search on a simple binary tree. At each stage, the node to be expanded next is indicated by a marker.

[Russell and Norvig]

DFS with iterative deepening

Idea: Modify DFS to stop at a maximum depth

Idea: Call DFS for maximum depths $1, 2, \ldots$

Assumption: Action costs Cost(s, a) $= c$ for some $c \geq 0$

If b actions per state, solution size d:

- Memory: $O(d)$ (better!)
- Time: $O(b^d)$ (same as BFS)

DFS with iterative deepening

Idea: Modify DFS to stop at a maximum depth

Idea: Call DFS for maximum depths $1, 2, \ldots$

Assumption: Action costs Cost(s, a) $= c$ for some $c \geq 0$

If b actions per state, solution size d:

- Memory: $O(d)$ (better!)
- Time: $O(b^d)$ (same as BFS)

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.

DFS with iterative deepening

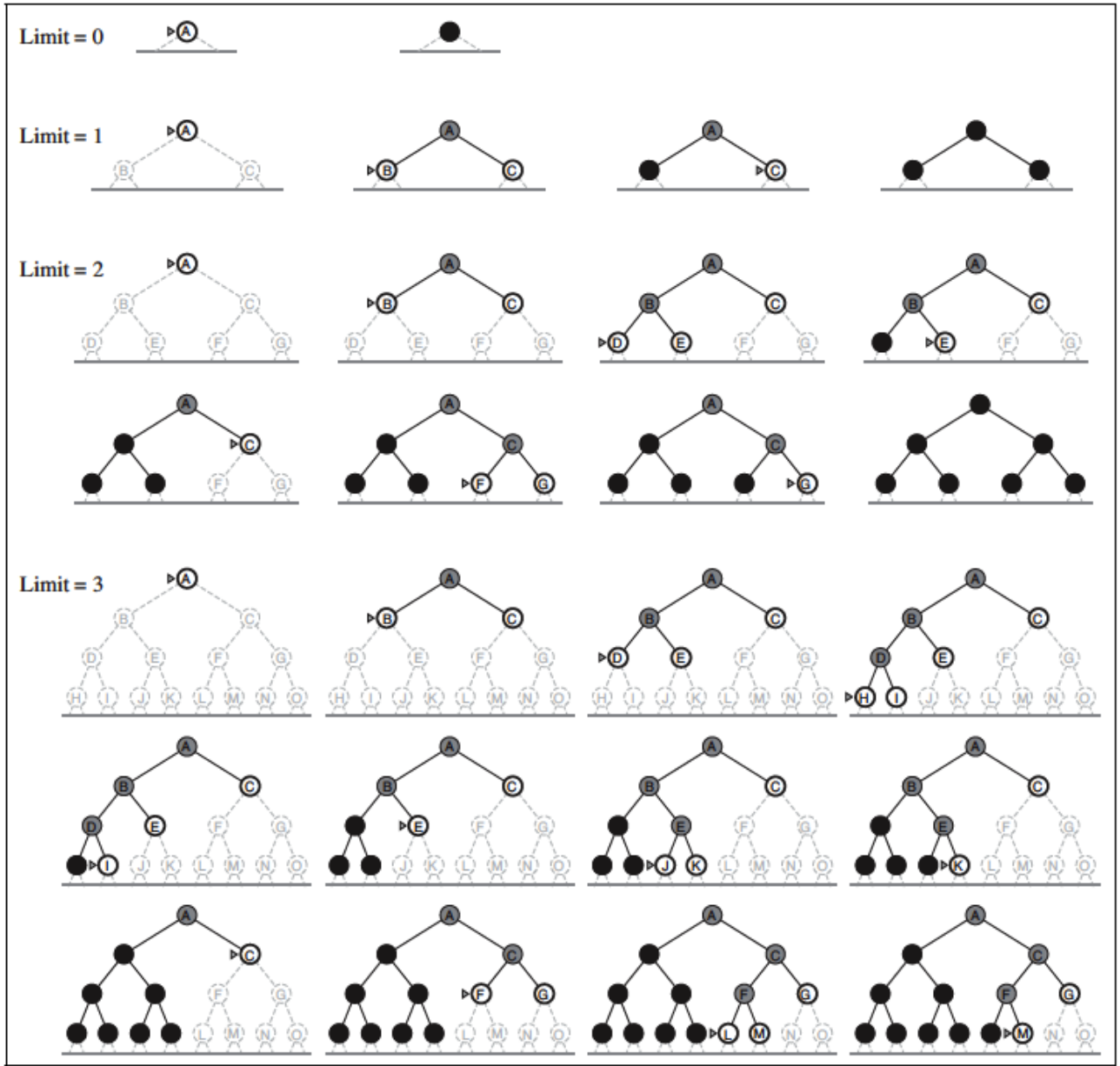


Figure 3.19 Four iterations of iterative deepening search on a binary tree.

Comparisson

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

b actions per state, solution depth d, maximum depth D

[Liang 2014]

Comparisson

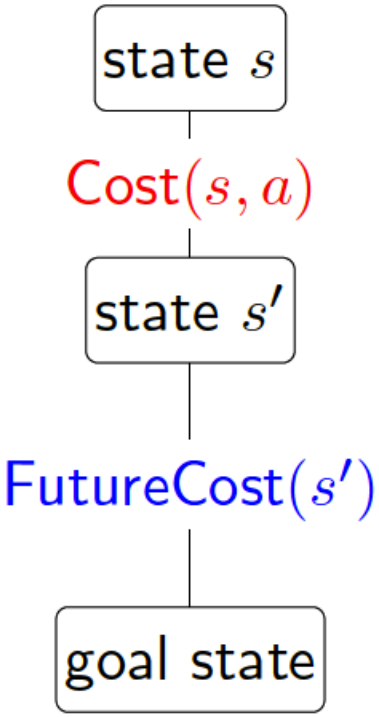
Algorithm	Action costs	Space	Time
DFS	zero	$O(D)$	$O(b^D)$
BFS	constant	$O(b^d)$	$O(b^d)$
DFS-ID	constant	$O(d)$	$O(b^d)$
Backtracking	any	$O(D)$	$O(b^D)$

b actions per state, solution depth d, maximum depth D

[Liang 2014]

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. Memory cannot magically grow, whereas time grows just by running things longer or parallelizing across multiple machines.

Dynamic programming



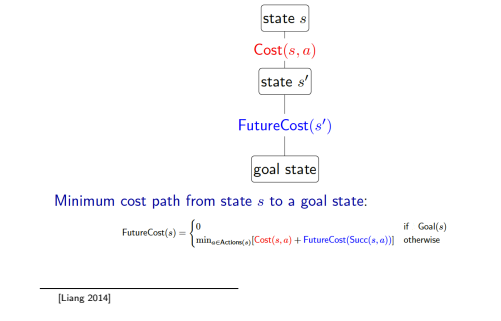
Minimum cost path from state s to a goal state:

$$\text{FutureCost}(s) = \begin{cases} 0 & \text{if } \text{Goal}(s) \\ \min_{a \in \text{Actions}(s)} [\text{Cost}(s, a) + \text{FutureCost}(\text{Succ}(s, a))] & \text{otherwise} \end{cases}$$

[Liang 2014]

Dynamic programming

Dynamic programming



[Liang 2014]

We will use dynamic programming to avoid the exponential running time of tree search. First, we will try to think about the minimum cost path in the search tree recursively. Define FutureCost(s) as the cost of the minimum cost path from s to a goal state. The minimum cost path starting with a state s to a goal state must take a first action a, which results in another state s', from which we should take a minimum cost path to a goal state. Written in symbols, we have a nice recurrence. The basic form is a base case (when s is a goal state) and an inductive case, which consists of taking the minimum over all possible actions a from s, taking an initial step resulting in an immediate action cost $Cost(s, a)$ and a future cost.

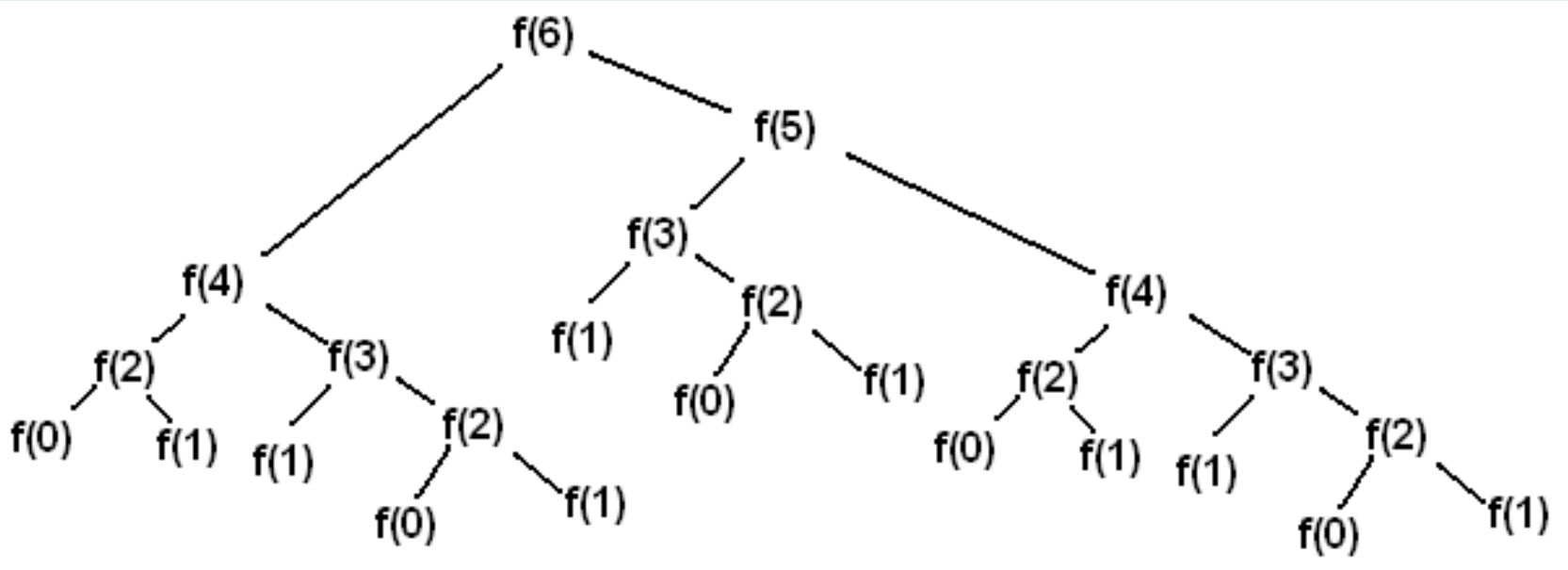
Dynamic programming: example

Fibonacci numbers

$$f(0) = 0$$

$$f(1) = 1$$

$$f(n) = f(n-1) + f(n-2)$$



[http://en.wikibooks.org/wiki/Algorithms/Dynamic_Programming]

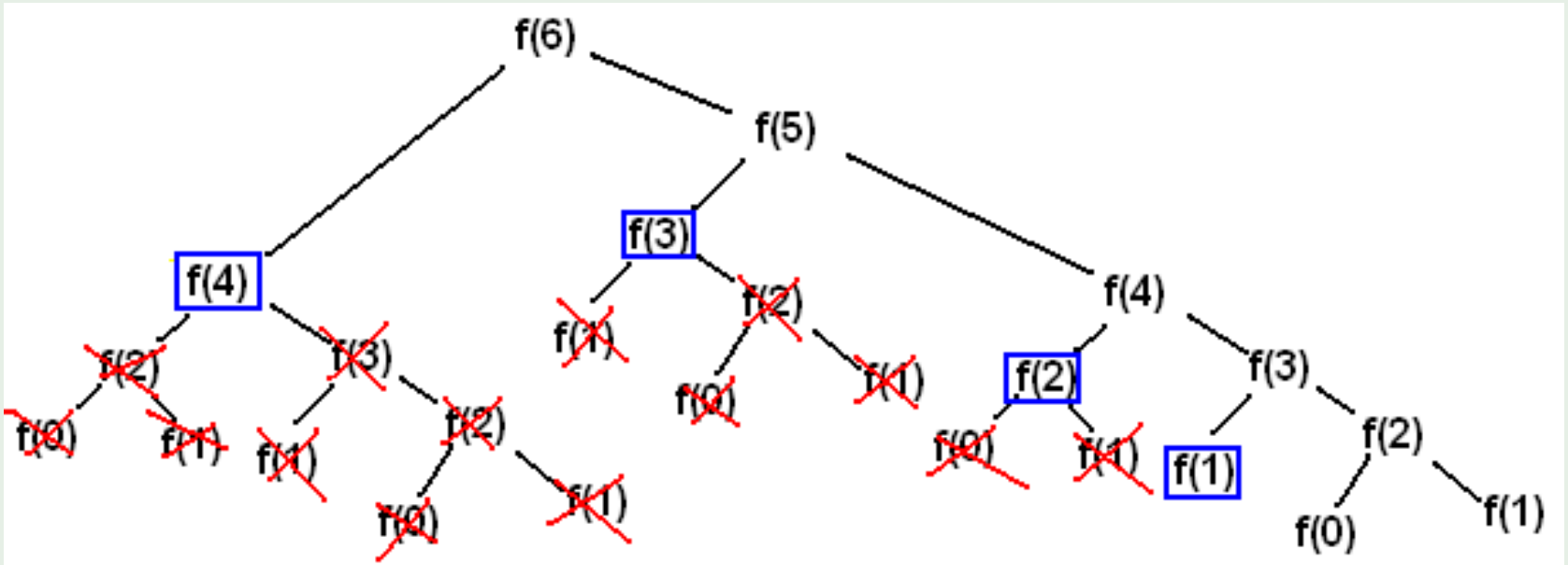
Dynamic programming: example

Fibonacci numbers

$$f(0) = 0$$

$$f(1) = 1$$

$$f(n) = f(n-1) + f(n-2)$$

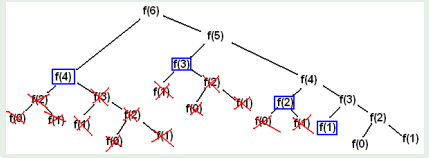


[http://en.wikibooks.org/wiki/Algorithms/Dynamic_Programming]

Dynamic programming: example

Fibonacci numbers

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n) = f(n-1) + f(n-2)$$

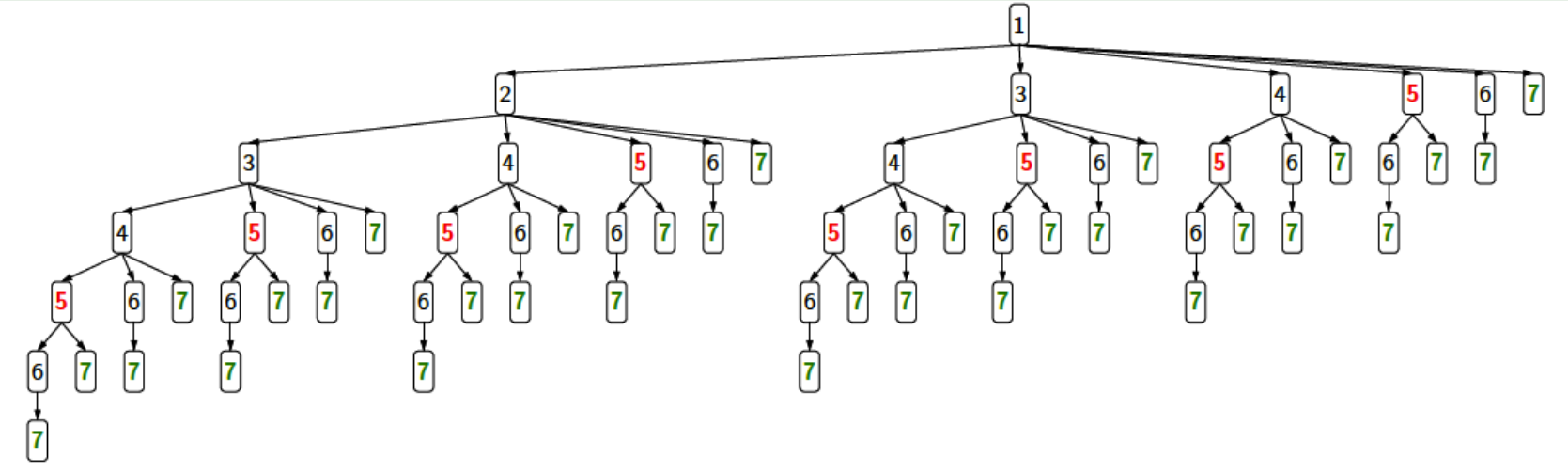


[http://en.wikibooks.org/wiki/Algorithms/Dynamic_Programming]

The values in the blue boxes are values that already have been calculated and the calls can thus be skipped. It is thus a lot faster than the straight-forward recursive algorithm.

Route planning

Find the minimum cost path from city 1 to city n, only moving forward. It costs c_{ij} to go from i to j.

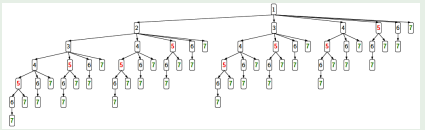


Idea: future costs only depend on current city

[Liang 2014]

Route planning

Find the minimum cost path from city 1 to city n, only moving forward. It costs c_{ij} to go from i to j.

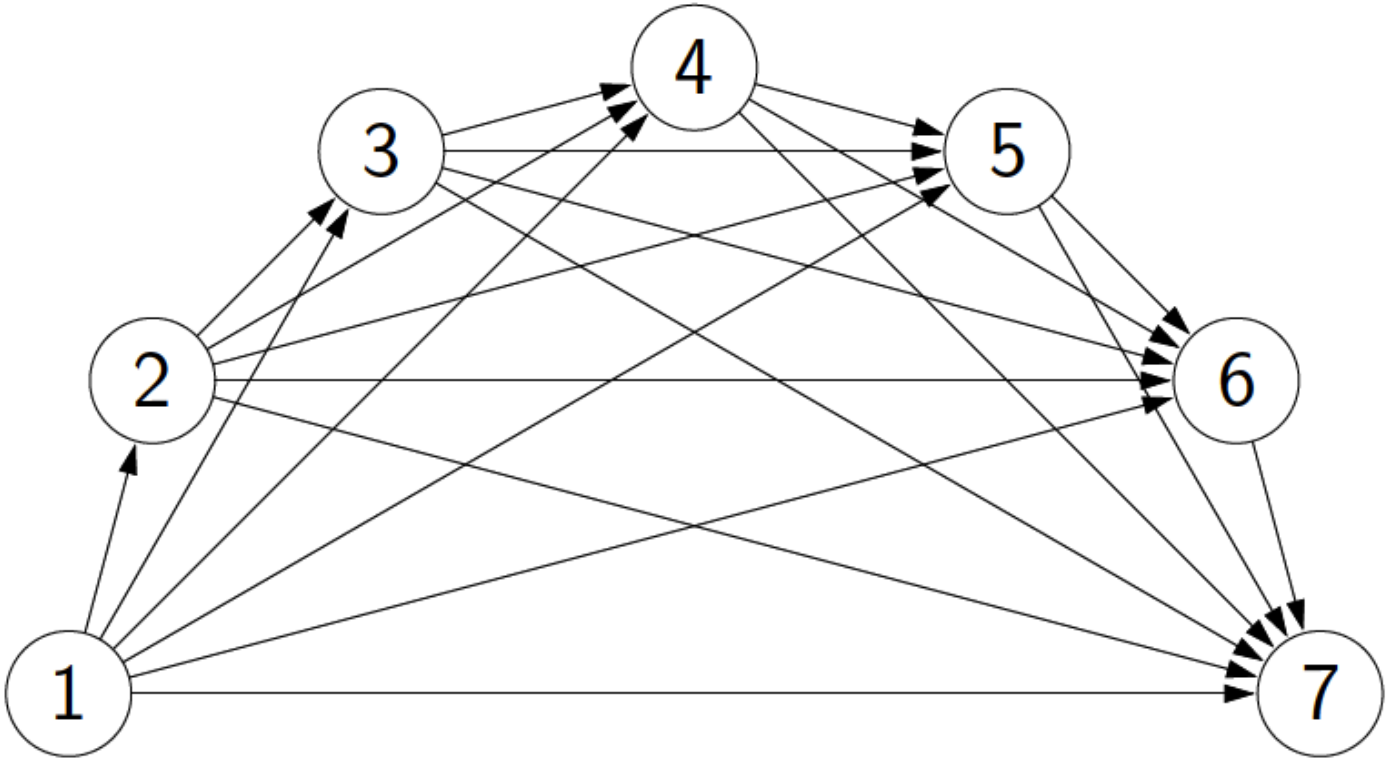


Idea: future costs only depend on current city

[Liang 2014]

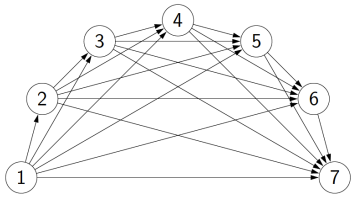
If we consider the simple route finding problem of traveling from city 1 to city n, the search tree grows exponentially with n. Again, this search tree has a lot of repeated structures. Moreover the future costs (the minimum cost of reaching a goal state) of a state only depends on the current city. So therefore, all the subtrees rooted at city 5, for example, have the same minimum cost. If we can just do that computation once, then we will have saved big time.

Dynamic programming: graph



Exponential saving in time and space!

Dynamic programming: graph



Exponential saving in time and space!

Let us collapse all the nodes that have the same city into one. This provides us with no longer a tree, but a directed acyclic graph with only n nodes rather than exponential in n nodes.

Dynamic programming: algorithm

Assumption: The state graph defined by Actions(s) and Succ(s,a) is acyclic

DynamicProgramming(s)

If already computed for s, **return** cached answer.
Else If Goal(s), **return** solution
Else for each action $a \in$ Actions(s): search algorithm!

State

A state is a summary of all the past actions sufficient to choose future actions optimally

Dynamic programming: algorithm

Assumption: The state graph defined by Actions(s) and Succ(s,a) is acyclic

DynamicProgramming(s)

If already computed for s, **return** cached answer.
Else If Goal(s), **return** solution
Else for each action $a \in$ Actions(s): search algorithm!

State

A state is a summary of all the past actions sufficient to choose future actions optimally

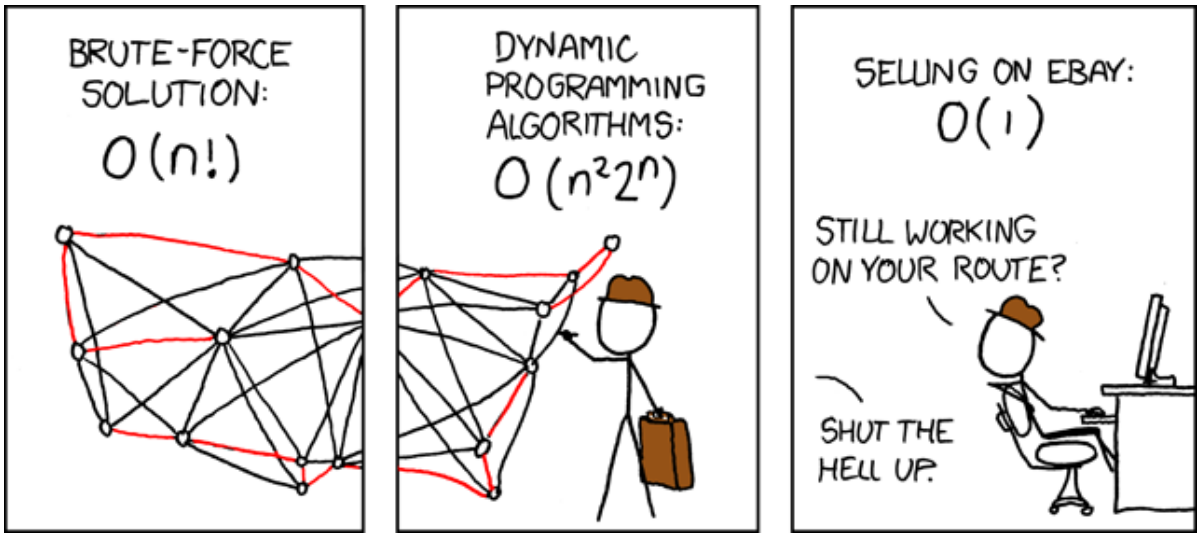
The dynamic programming algorithm is exactly backtracking search with one twist. At the beginning of the function, we check to see if we have already computed the future cost for s. If we have, then we simply return it (which takes constant time, using a hash map). Otherwise, we compute it and save it in the cache so we don't have to recompute it again. In this way, for every state, we are only computing its value once. For this particular example, the running time is $O(n^2)$, the number of edges. One important point is that the graph must be acyclic for dynamic programming to work. If there are cycles, the computation of a future cost for s might depend on s' which might depend on s. We will have infinite loop in this case. To deal with cycles, we need uniform cost search, which we will describe later.

A state is a summary of all the past actions sufficient to choose future actions optimally. What state is really about is forgetting the past. We can't forget everything because the action costs in the future might depend on what we did on the past. The more we forget, the fewer states we have, and the more efficient our algorithm. The idea is to find the minimal set of states that suffice.

Summary

State: summary of past actions sufficient to choose future actions optimally

Dynamic programming: backtracking search with memoization (exponential savings)



Dynamic programming only works for acyclic graphs...what if there are cycles?

Next lecture

We will discuss more search algorithms: uniform cost search