6.825 Techniques in Artificial Intelligence

Problem Solving and Search

Problem Solving

Last time we talked about different ways of constructing agents and why it is that you might want to do some sort of on-line thinking. It seems like, if you knew enough about the domain, that off-line you could do all this compilation and figure out what program should go in the agent and put it in the agent. And that's right. But, sometimes when the agent has a very rich and complicated environment, it's easier to leave some of that not worked out, to let the agent work some of it out on-line. We ended up talking about planning, about considering sequences of actions and deciding what to do based on the projected value of doing different sequences of actions.

Now, we're going to spend time doing something fairly basic. We're going to talk about search as an entree to a whole bunch of more complicated applications of search. I'm assuming that all of you have seen basic search algorithms sometime before, so we'll go through the beginning part fairly quickly but then we'll do some more complicated ones in more detail.

We're going to talk about what I would call a simple form of planning, but what the book calls "problem solving". A problem-solving problem has these properties:

6.825 Techniques in Artificial Intelligence

Problem Solving and Search

Problem Solving

- Agent knows world dynamics

The agent knows the dynamics of the world, that is, it knows that if it takes a particular action in a particular situation, here's what's going to happen.

6.825 Techniques in Artificial Intelligence

Problem Solving and Search

Problem Solving

- Agent knows world dynamics
- World state is finite, small enough to enumerate

The world state is finite and not too big (there's a good technical term! Say, smaller than 1000 or 10000; small enough to enumerate in your computer).

6.825 Techniques in Artificial Intelligence

Problem Solving and Search

Problem Solving

- Agent knows world dynamics
- World state is finite, small enough to enumerate

- World is deterministic

The world dynamics are deterministic: when the world is in some state and the agent does some action, there's only one thing that could happen in the world.

6.825 Techniques in Artificial Intelligence

Problem Solving and Search

Problem Solving

- Agent knows world dynamics
- World state is finite, small enough to enumerate

- World is deterministic
- Utility for a sequence of states is a sum over path

The utility for sequences of states is a sum over the path of the utilities of the individual states. What I mean by that is that if I travel some trajectory, how good it is to have traveled that trajectory is going to be a sum of some function of the states that I'm in at each step and the actions that I took along the way. That's a structure that makes a lot of things easy. And it's one that's pretty hard to back out of.

6.825 Techniques in Artificial Intelligence

Problem Solving and Search

Problem Solving

- Agent knows world dynamics
- World state is finite, small enough to enumerate

- World is deterministic
- Utility for a sequence of states is a sum over path
- Agent knows current state

The agent knows exactly what state it is in.

6.825 Techniques in Artificial Intelligence

Problem Solving and Search

Problem Solving

- Agent knows world dynamics [learning]
- World state is finite, small enough to enumerate

- World is deterministic
- Utility for a sequence of states is a sum over path
- Agent knows current state

Relaxation of assumptions later in the course

To look ahead: We are going to relax the assumption that the agent knows the world dynamics at the very end of this course, when we talk about learning.

6.825 Techniques in Artificial Intelligence

Problem Solving and Search

Problem Solving

- Agent knows world dynamics [learning]
- World state is finite, small enough to enumerate [logic]
- World is deterministic
- Utility for a sequence of states is a sum over path
- Agent knows current state

Relaxation of assumptions later in the course

We're going to relax the assumption that the world state is small when we talk about logic, because logic gives us a way to do abstraction that lets us deal with very large or infinite state spaces.

6.825 Techniques in Artificial Intelligence

Problem Solving and Search

Problem Solving

- Agent knows world dynamics [learning]
- World state is finite, small enough to enumerate [logic]
- World is deterministic [uncertainty]
- Utility for a sequence of states is a sum over path
- Agent knows current state

Relaxation of assumptions later in the course

We're going to relax the assumption that the world is deterministic when we talk about uncertainty or probability.

6.825 Techniques in Artificial Intelligence

Problem Solving and Search

Problem Solving

- Agent knows world dynamics [learning]
- World state is finite, small enough to enumerate [logic]
- World is deterministic [uncertainty]
- Utility for a sequence of states is a sum over path
- Agent knows current state [logic]

Relaxation of assumptions later in the course

And we're going to relax the assumption that the agent knows the current state when we talk about logic and about probability.

6.825 Techniques in Artificial Intelligence

Problem Solving and Search

Problem Solving

- Agent knows world dynamics [learning]
- World state is finite, small enough to enumerate [logic]
- World is deterministic [uncertainty]
- Utility for a sequence of states is a sum over path
- Agent knows current state [logic, uncertainty]

Relaxation of assumptions later in the course

And this one (utility is a sum) we're not going to relax at all because it breaks so much of what we do.

6.825 Techniques in Artificial Intelligence

Problem Solving and Search

Problem Solving

- Agent knows world dynamics [learning]
- World state is finite, small enough to enumerate [logic]
- World is deterministic [uncertainty]
- Utility for a sequence of states is a sum over path
- Agent knows current state [logic, uncertainty]

Few real problems are like this, but this may be a useful abstraction of a real problem

Relaxation of assumptions later in the course

Let's say that you want to formulate some problems in the world as problem-solving problems of this kind. In fact, the hardest part is figuring out how you could take a real problem and abstract it like this. Except for totally synthetic problems, nothing is really like this. On the other hand, it can often be useful to think of a problem this way. But, there's usually a lot of hard work that goes into going from the real, honest-to-goodness, messy problem into an abstraction like this.

Example: Route Planning in a Map

A map is a graph where nodes are cities and links are roads. This is an abstraction of the real world.

The example problem that we'll use in looking at problem solving methods is route planning in a map.

Example: Route Planning in a Map

A map is a graph where nodes are cities and links are roads. This is an abstraction of the real world.

- Map gives world dynamics: starting at city X on the map and taking some road gets to you to city Y.

If I give you a map, you know the world dynamics, because you know that if you are in this place and you travel down that road, then you're going to end up at this other place.

Example: Route Planning in a Map

A map is a graph where nodes are cities and links are roads. This is an abstraction of the real world.

- Map gives world dynamics: starting at city X on the map and taking some road gets to you to city Y.
- World (set of cities) is finite and enumerable.

The world state is finite, again as an abstraction. If I give you a map that has dots on it, standing for the towns that somebody thought were big enough to merit a dot, somebody decided that was a good level of abstraction to think about driving around this place.

Example: Route Planning in a Map

A map is a graph where nodes are cities and links are roads. This is an abstraction of the real world.

- Map gives world dynamics: starting at city X on the map and taking some road gets to you to city Y.
- World (set of cities) is finite and enumerable.
- World is deterministic: taking a given road from a given city leads to only one possible destination.

The world is deterministic. Again, in the view of a map, there aren't probabilities that tell you how likely it is that if you're trying to go here, you'll end up over there. Now, we all know that there's an error model overlaid over this and if we're trying to go here, there is some probability that we'll end up over there. But, for now, we're going to work at the level of abstraction that does not take that into consideration. Later on in this course we're actually going to think about those other kinds of models.

Example: Route Planning in a Map

A map is a graph where nodes are cities and links are roads. This is an abstraction of the real world.

- Map gives world dynamics: starting at city X on the map and taking some road gets to you to city Y.
- World (set of cities) is finite and enumerable.
- World is deterministic: taking a given road from a given city leads to only one possible destination.
- Utility for a sequence of states is usually either total distance traveled on the path or total time for the path.

Usually the cost that you pay is something like how long it took, or how much gas you used, or how pretty the scenery was, integrated over time, or something like that. In those cases, your utility can be expressed as a sum. But that's not always the case. It depends on how you formulate the problem. For example, assume you have enough gas to go 20 miles. Then, you're going to have a situation where any path that's longer than 20 miles has really bad utility and any shorter path is ok. And, that can be hard to express as a sum. So, there are things that you could want to express that you can't write down as a sum. Which makes this a real restriction on the kinds of problems we can address.

Example: Route Planning in a Map

A map is a graph where nodes are cities and links are roads. This is an abstraction of the real world.

- Map gives world dynamics: starting at city X on the map and taking some road gets to you to city Y.
- World (set of cities) is finite and enumerable.
- World is deterministic: taking a given road from a given city leads to only one possible destination.
- Utility for a sequence of states is usually either total distance traveled on the path or total time for the path.
- We assume current state is known

Usually, if you're trying to work with a map the assumption is that you know where you are, although we all know it is not always true.

Formal Definition

Problem:

So now, let's talk about the formal definition of a problem-solving problem.

Formal Definition

Problem:

- Set of states: S

We have a set of states, which is finite, and not too big,

Formal Definition

Problem:

- Set of states: S
- Initial state

We have an initial state (known to us),

Formal Definition

Problem:

- Set of states: S
- Initial state
- Operators (actions): $S \rightarrow S$

We have a set of what are called operators, which is another name for actions. An operator is basically a mapping from states to states; we've said it's deterministic and we know what it is. There's a set of operators, and each operator moves forward or fills up the gas tank or colors in the square in front of me. Each operator says, if I'm in one state, what's the next one.

Formal Definition

Problem:

- Set of states: S
- Initial state
- Operators (actions): $S \rightarrow S$
- Goal test: $S \rightarrow \{ t, f \}$

We have a goal test. The assumption in a problem solving problem is that you're trying to finish a job. We're going to assume that this is an episodic task; that we do some work and it gets over. You can extend some of these techniques to living very long or potentially infinite lives but we're not going to think about this until the very end of the course when we discuss Markov processes.

Let's assume that in the driving analogy you have a utility function that we add up as we take steps along the path, but that there's a particular place that we're trying to get to. So, there's a goal test that is a mapping from states into Booleans. Some kind of a function that says "is this state the goal?" "is this the place I'm trying to get to?" "is this considered to be a stopping point for this problem?"

Formal Definition

Problem:

- Set of states: S
- Initial state
- Operators (actions): $S \rightarrow S$
- Goal test: $S \rightarrow \{ t, f \}$
- Path cost: $(S, O)^* \rightarrow \mathbb{R}$
  - Sum of costs: $\sum c(S,O)$

And, we have path cost. It's going to be a mapping from paths to real numbers. A step on the path is a state followed by an operator and we can consider strings of those and we could ask how much do they cost. A state and operator star, that means some sequence of them. I could have written just a sequence of states into real numbers; why didn't I? Assume that there are three ways of getting home, one involves walking, the other involves driving, and another involves taking the train. They may all get me home but it may matter which operator I use since they may each cost different amounts. Typically, this is really going to be the sum of some cost of each particular state and operator combination.

So that's the formal definition of a problem-solving problem.

Formal Definition

Problem:

- Set of states: S
- Initial state
- Operators (actions): $S \rightarrow S$
- Goal test: $S \rightarrow \{ t, f \}$
- Path cost: $(S, O)^* \rightarrow \mathbb{R}$
  - Sum of costs: $\sum c(S,O)$

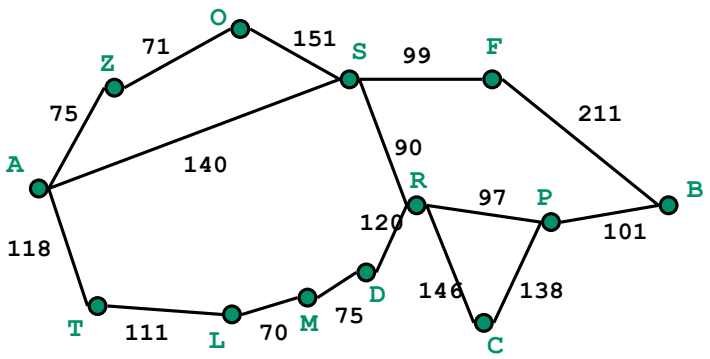
Criteria for algorithms:

- Computation time/space
- Solution quality

When you think about solving such a problem you want to think about a couple of different things. We're going to look for algorithms for solving this kind of problem, so you want to think about how well the algorithm does. How can we think about that?

We're going to worry about computation time and space and we're also going to worry about solution quality and this is going to prefigure something that's going to come up over and over again, which is, that if you want good answers, you're going to have to think harder. So, there's pretty much always a tradeoff between the quality of the answers and how much we have to think. We always have to keep that in mind, so we can look at different algorithms, some of which guarantee the optimal solution but take forever, some of which run pretty quickly but you never know how good the answer is going to be. There is no right way to make the tradeoff between these two and an agent operating on-line might want to make this tradeoff differently at different times or in different circumstances. So, computing which way to dodge the oncoming truck, you want to have one kind of tradeoff; computing where to go to graduate school, there are other decisions that merit a lot more thought. It has to do with how much time pressure there is and the magnitude of the consequences of the decision.

Route Finding



We'll think about route-finding as an example for understanding how different search algorithms work.

Here's a map; we'll use it as our example (it's in the book). We can really think of driving around in this map as a formal problem as defined in the last slide. We're in some start state, labeled Arad, and here's the goal state, labeled Bucharest (this is Romania, in case you didn't recognize it). This is part of the graph that's in the book and it has the property that it is a set of 26 cities beginning with unique letters. We're going to try to find our way through Romania.

Obviously, this is an abstraction. We've left out all kind of things. In fact, we don't know the world dynamics, some of these bridges may be out or roads closed for construction. The state of the world is really enormous; when you are driving around, it's not just what city you're in that matters, it's how much gas you have and whether you're hungry and what the traffic is like, but we're not going to worry about that right now. The world is not really deterministic (as we said before), maybe you don't even know what city you're in, etc. But, it still seems to be useful to solve this little problem of "if I knew all this stuff, what would I do?" And, if nothing else, we can use this abstract route finding as a sub-procedure in a more complicated system.

Romania Map

Just for fun, here's a real map of Romania. It has a lot more detail, but still leaves out everything but the raw geography.

Search

So, we're going to do searching; we'll cover the basic methods really fast. But first, let's write the general structure of these algorithms. We have something called an "agenda", this is AI people's term for it - a list of states that are waiting for us to expand them.

Search

- Put start state in the agenda

We’re going to begin by putting the start state on the agenda.

Search

- Put start state in the agenda
- Loop

Now we loop.

Search

- Put start state in the agenda
- Loop
  - Get a state from the agenda

We're going to get a state from the agenda,

Search

- Put start state in the agenda
- Loop
  - Get a state from the agenda
    - If goal, then return

If it's the goal, then return it.

Search

- Put start state in the agenda
- Loop
  - Get a state from the agenda
    - If goal, then return
    - Expand state (put children in agenda)

Otherwise, we expand the state and put the children on the agenda. So, what does it mean to expand the state? In this case, it means to take all the operators and apply them to the state to get a set of possible successor states. From any state, there's a set of things that you could do, and each of those operators yields a possible next state.

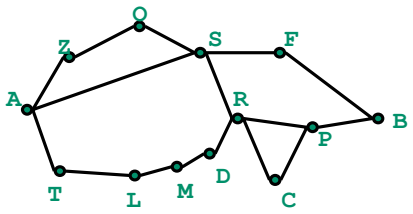
Search

- Put start state in the agenda
- Loop
  - Get a state from the agenda
    - If goal, then return
    - Expand state (put children in agenda)

Which state is chosen from the agenda defines the type of search and may have huge impact on effectiveness.

Which state is chosen from the agenda defines the type of search and may have a huge impact on its efficiency and/or the quality of the solutions it produces.

Depth-First Search

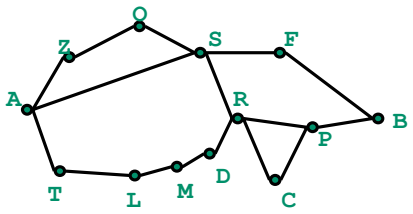


Let's start by looking at Depth-First Search (DFS). The search strategy is entirely defined by how we choose a node from the agenda to expand.

So, in depth-first search what's the strategy? Whenever possible, you choose one of the children that you just added to the agenda. Otherwise, you go back up to the most recent child that you have not yet expanded.

Depth-First Search

- Treat agenda as a stack (get most recently added node)
  - Expansion: put children at top of stack
  - Get new nodes from top of stack

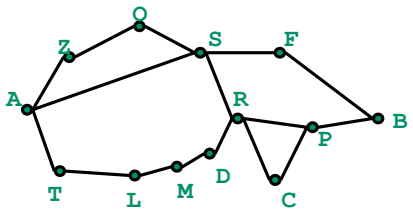


We can make this happen by treating the agenda as a stack. You always put children at the top of the stack and pop when you need a new node. This search method has the character of making a commitment and seeing what might happen after that, and seeing what might happen after that, and so on until it's not going to work out.

Depth-First Search

- Treat agenda as a stack (get most recently added node)
  - Expansion: put children at top of stack
  - Get new nodes from top of stack

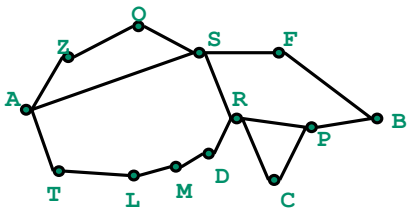
A



Let's just see what would happen if we did DFS on this graph. We would start with A, as our start state. What are the children of A? Z,S, and T.

Depth-First Search

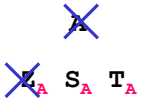
- Treat agenda as a stack (get most recently added node)
  - Expansion: put children at top of stack
  - Get new nodes from top of stack

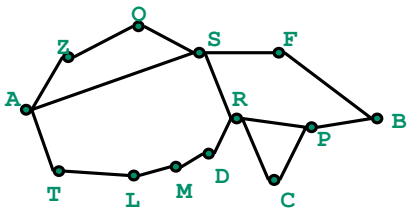


So, we pop A off the stack, expand it, and then push Z,S,T.

Depth-First Search

- Treat agenda as a stack (get most recently added node)
  - Expansion: put children at top of stack
  - Get new nodes from top of stack





Now, we are going to pick Z to expand. What are the children of Z? A and O.

We have a potential problem here. If we put A onto the agenda again, we'll pick it, then put on Z, S, and T; pick Z, add A and O to the agenda, etc. So, we'll have to do something about loops.

Avoiding Loops

This discussion of how to deal with loops actually applies to all of the search methods, not just depth-first.

Avoiding Loops

- Method 1:
  - Don’t add a node to the agenda if it’s already in the agenda
  - Causes problems when there are multiple paths to a node and we want to be sure to get the shortest

The first thought about how to fix this problem might be not to add a node to the agenda if it’s already in the agenda. But there’s a potential problem with this strategy. It might be that there are two paths (to node S, for instance), with different lengths. It might also be that the second path we discover is shorter than the first. Thus, it could be important (especially in later searches) to consider both alternatives.

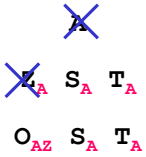
Avoiding Loops

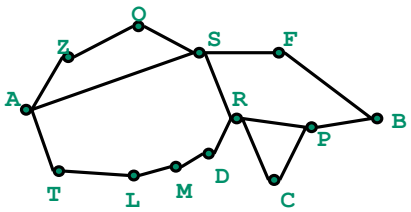
- Method 1:
  - Don't add a node to the agenda if it's already in the agenda
  - Causes problems when there are multiple paths to a node and we want to be sure to get the shortest
- Method 2:
  - Don't expand a node (or add it to the agenda) if it has already been expanded.
  - We'll adopt this one for all of our searches

Another way to fix the problem is to say that we're not going to expand a node or add a node to the agenda if it has already been expanded. This is a pretty generally useful way to deal with loops. So, when we expand a node, we'll set a bit (either in the node, or in some other data structure we use to keep track of which nodes we've visited). Then, when it's time to add a node to the agenda, we'll check to see if it has already been expanded and, if so, we won't bother adding it.

Depth-First Search

- Treat agenda as a stack (get most recently added node)
  - Expansion: put children at top of stack
  - Get new nodes from top of stack

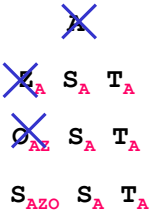


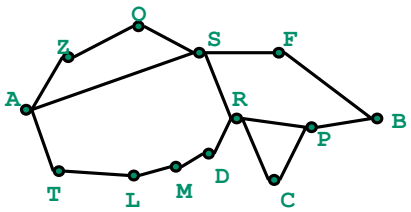


Okay. Let's go back to our depth-first search. We just expanded Z, which leads to putting A and O onto the stack. But before we put them on we notice that A has already been expanded, so we just add O.

Depth-First Search

- Treat agenda as a stack (get most recently added node)
  - Expansion: put children at top of stack
  - Get new nodes from top of stack

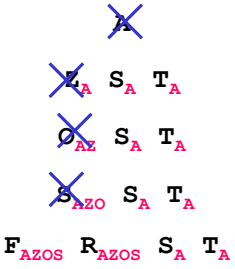


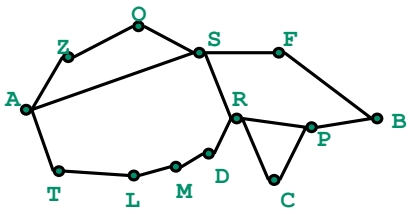


Now, we pop the next node off the stack. It's O. The children of O are Z and S. Z has already been expanded, so we add S to the agenda (note that although S has been added to the agenda, it has not yet been expanded. The red subscript letters indicate the path that was followed to this node).

Depth-First Search

- Treat agenda as a stack (get most recently added node)
  - Expansion: put children at top of stack
  - Get new nodes from top of stack

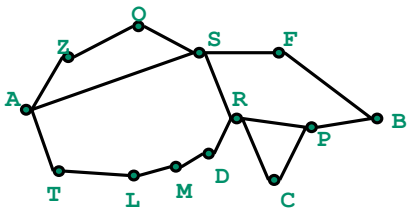




Now, we pop S, and add children F and R (because children O and A have already been expanded).

Depth-First Search

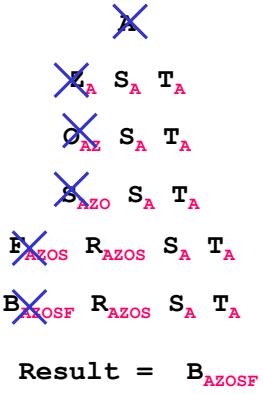
- Treat agenda as a stack (get most recently added node)
  - Expansion: put children at top of stack
  - Get new nodes from top of stack

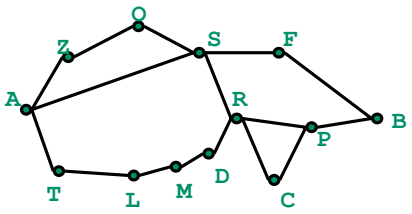


We pop F, and add B.

Depth-First Search

- Treat agenda as a stack (get most recently added node)
  - Expansion: put children at top of stack
  - Get new nodes from top of stack

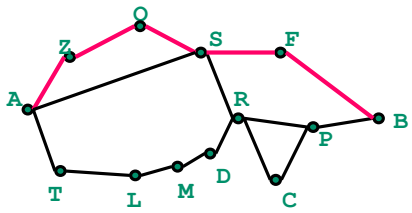




Then, finally, we pop B, see that it is the goal state, and terminate. Yay!

Properties of DFS

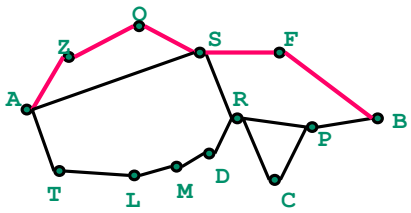
Sub-optimal answer:



This graph doesn't have any dead ends, so we don't have to backtrack, we'll eventually find our way to the goal. But, will we find our way there the shortest way? Not necessarily. In our example, we ended up with the path AZOSFB, which has a higher path cost than the one that starts by going straight to S. So, depth-first search won't necessarily find the shortest path.

Properties of DFS

Sub-optimal answer:



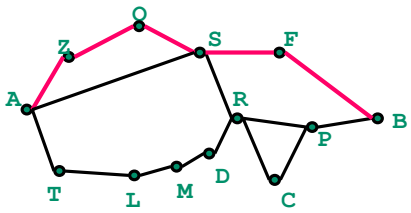
Let
- b = branching factor

So, how efficient is DFS?

When people talk about the complexity of search methods, they usually think of them in terms of trees, and, in particular, about the branching factor (**b**), which is the maximum number of children a node can have.

Properties of DFS

Sub-optimal answer:

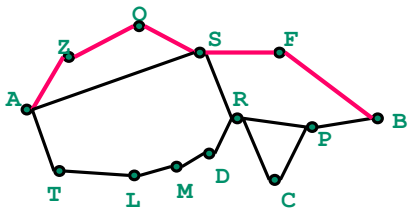


Let
- b = branching factor
- m = maximum depth

M stands for the maximum depth of the tree. In a graph, what would the maximum depth be (if you're pruning repeated nodes)? How deep could your search tree be? As many nodes as the length of the longest cycle, which has to be less than the number of nodes.

Properties of DFS

Sub-optimal answer:

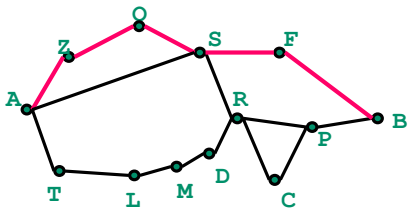


Let
- b = branching factor
- m = maximum depth
- d = goal depth

We'll let **d** stand for the goal depth, that is, the number of arcs on the shortest path (measured in the number of arcs) between the start and goal nodes.

Properties of DFS

Sub-optimal answer:



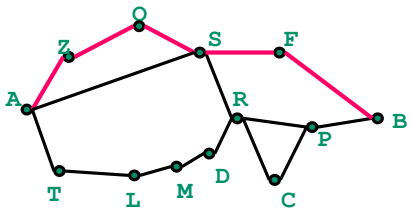
Let
- b = branching factor
- m = maximum depth
- d = goal depth

- $O(b^m)$ time

So, how much time, in big O terms, does DFS take? Does it depend on how close the goal is to you? In the worst case, no. You could go all the way through the tree before you find the one path where the goal is next door, depending on the order that you add the children to the agenda (which is unspecified by the algorithm). So, if it's a tree, we're talking order b^m time (really bad).

Properties of DFS

Sub-optimal answer:



Let
- b = branching factor
- m = maximum depth
- d = goal depth

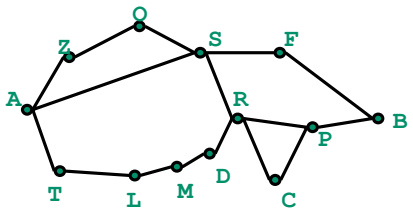
- $O(b^m)$ time
- O(mb) space

But, DFS does have a redeeming feature; what is it? why do people use it all the time?

Space! How much space do you need? O(m), or O(mb): you have to remember up to b different alternatives at up to m different depths. So, this is good. People usually implement DFS recursively and the stack gets stored in the stack of the program.

Breadth-First Search

- Treat agenda as a queue (get least recently added node)
  - Expansion: put children at end of queue
  - Get new nodes from the front of queue

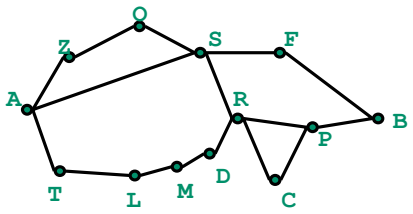


Now let's talk about Breadth-First Search (BFS). How does it differ from DFS at the algorithmic level? You expand all the nodes at one level of the tree, before you go to the next level. To make this happen, you put children at the end of the agenda and you pop them from the front. So, this has the effect of expanding cities going out by depth.

Breadth-First Search

- Treat agenda as a queue (get least recently added node)
  - Expansion: put children at end of queue
  - Get new nodes from the front of queue

A



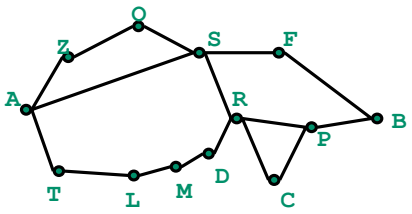
So, in our example, we start with A.

Breadth-First Search

- Treat agenda as a queue (get least recently added node)
  - Expansion: put children at end of queue
  - Get new nodes from the front of queue

~~A~~

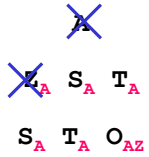
Z_A S_A T_A

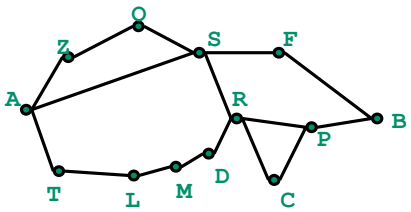


Then, we pop A and add children Z, S, and T.

Breadth-First Search

- Treat agenda as a queue (get least recently added node)
  - Expansion: put children at end of queue
  - Get new nodes from the front of queue

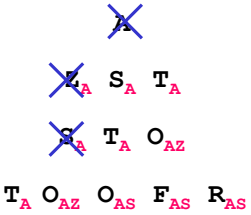


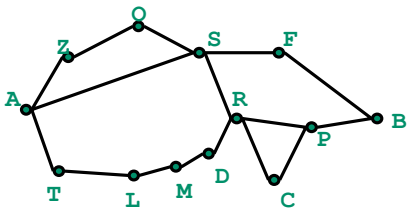


Now, we pop Z, and add O (but not A, because it's already been expanded).

Breadth-First Search

- Treat agenda as a queue (get least recently added node)
  - Expansion: put children at end of queue
  - Get new nodes from the front of queue

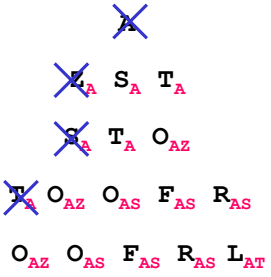


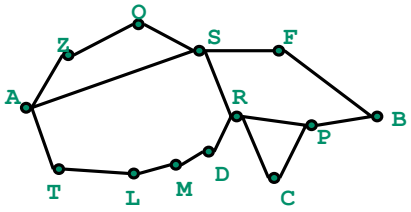


Now, we pop S, and add O, F, and R.

Breadth-First Search

- Treat agenda as a queue (get least recently added node)
  - Expansion: put children at end of queue
  - Get new nodes from the front of queue

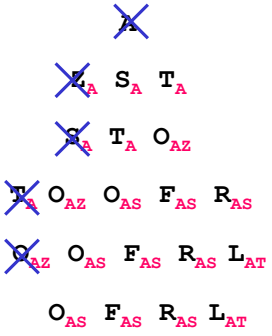


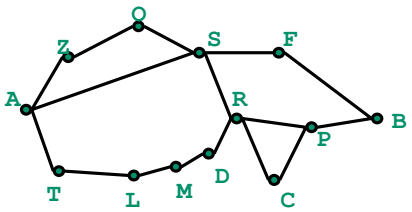


Next, we pop T and add L.

Breadth-First Search

- Treat agenda as a queue (get least recently added node)
  - Expansion: put children at end of queue
  - Get new nodes from the front of queue

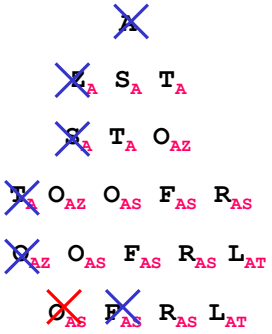


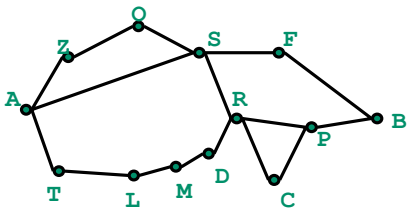


Now, we pop O. It has children Z and S, but both have been expanded, so we don't add anything.

Breadth-First Search

- Treat agenda as a queue (get least recently added node)
  - Expansion: put children at end of queue
  - Get new nodes from the front of queue

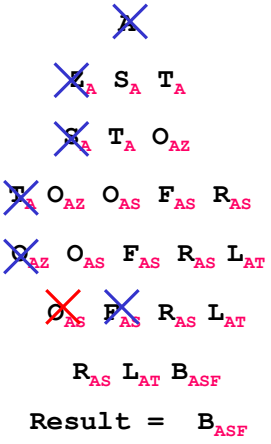


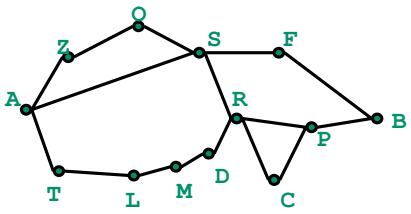


Now, we get O again (via a different path). We ignore it because it's already been expanded and pop F.

Breadth-First Search

- Treat agenda as a queue (get least recently added node)
  - Expansion: put children at end of queue
  - Get new nodes from the front of queue

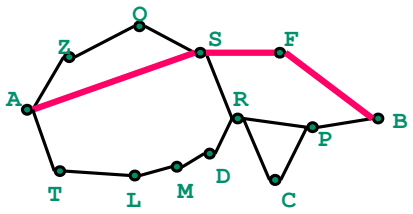




Now we add B to the agenda. In some later searches, it will be important to keep going, because we might find a shorter path to B. But in BFS, the minute we hit the goal state, we can stop and declare victory. Yay!

Properties of BFS

Guaranteed to return shortest path (measured in number of arcs) from start to goal.



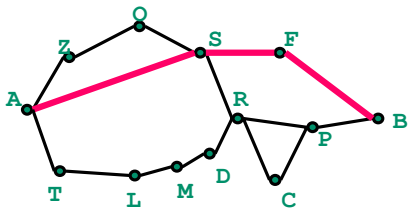
What about BFS, what can we say about the answer we get? It's optimal in that it returns the path with the fewest steps. Not necessarily the shortest route, but fewest steps. If there's a one-step answer it will give you that, if not it will look for a two step answer, and so on.

Properties of BFS

Guaranteed to return shortest path (measured in number of arcs) from start to goal.

Let

- b = branching factor
- m = maximum depth
- d = goal depth



- $O(b^d)$ time

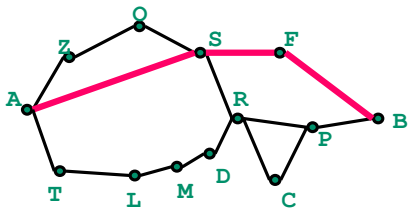
How long does this take to run? O(b^d). We having a branching factor of b, and even if the graph goes on forever, if we know that the goal is at depth d and we're expanding it out depth by depth, we know that we will only expand the tree to depth d.

Properties of BFS

Guaranteed to return shortest path (measured in number of arcs) from start to goal.

Let

- b = branching factor
- m = maximum depth
- d = goal depth



- $O(b^d)$ time
- $O(b^d)$ space

How much space? What do we have to remember in order to go from one level to the next level? O(b^d). In order to make the list of nodes at level 4, you need to know all the level 3 nodes. So, the drawback of BFS is that we need O(b^d) space. Do you need to save all the nodes? No, you only have to save the one layer, but exponential growth is such that there are as many nodes in the last layer as in the sum of all previous layers.

Iterative Deepening

- DFS is efficient in space, but has no path-length guarantee
- BFS finds min-step path but requires exponential space
- Iterative deepening: Perform a sequence of DFS searches with increasing depth-cutoff until goal is found.

Can you get the best of both worlds? That's always nice if you can and in this case, you kind of can. The way to get the best of both worlds is a method called iterative deepening. The idea here is that DFS is a real win in space but it might go forever down a really bad path. BFS gets you a short answer if there is one, but it is expensive in space. So, in iterative deepening, you do a DFS with a cutoff of depth 1; you just do DFS but if you ever get to a node at depth greater than 1, then you backtrack. Depth 1 is really boring, that just means examine the root node. Maybe the root node is the goal, otherwise, you do a DFS with the cutoff at depth 2 and if there's a solution at depth 2, then you will find it; otherwise, you do DFS with cutoff at depth three and so on.

Iterative Deepening

- DFS is efficient in space, but has no path-length guarantee
- BFS finds min-step path but requires exponential space
- Iterative deepening: Perform a sequence of DFS searches with increasing depth-cutoff until goal is found.

DFS cutoff depth	Space	Time
1	O(b)	O(b)
2	O(2b)	$O(b^2)$
3	O(3b)	$O(b^3)$
4	O(4b)	$O(b^4)$
…	…	…
d	O(db)	$O(b^d)$
Total	Max = O(db)	Sum = $O(b^{d+1})$

It seems a little bit wasteful. The crucial thing is that these steps are going to take time on the order of: b, b^2, b^3, b^4, etc., because a DFS that cuts off at depth d is going to take time b^d. The total time is going to be the sum of the times at each depth. The sum of b through b^4 is about b^5. It's on the order of b^(d+1) which is sort of like b^d.

The space required for DFS is m*b, and the total space is the max required at any level, which is the max at the last level, or d*b.

So, we have a method with about the same time requirement as BFS (probably double (if b=2)) but the space requirement of DFS. So, this really is the best of both worlds. It is standard practice in many search applications. Chess playing computers usually do iterative deepening. Is there a mate in one move? Is there a mate in two? And so on.

Uniform Cost Search

- Breadth-first and Iterative-Deepening find path with fewest steps (hops).
- If steps have unequal cost, this is not interesting.

Both BFS and iterative deepening find the path with the shortest number of steps, but if the steps have unequal costs then that's not interesting. You can go from small town to small town, and it takes a lot of small towns to get from here to Chicago, but it may actually shorter than taking the freeway (probably not in time, but in distance).

Uniform Cost Search

- Breadth-first and Iterative-Deepening find path with fewest steps (hops).
- If steps have unequal cost, this is not interesting.
- How can we find the shortest path (measured by sum of distances along path)?

If you wanted to find the shortest path from Araj to Bucharest and you thought about the options and you made an agenda that said, I could start by going to Z for a cost of 75 or I could go to S for 140 or I could go to T for 118, what would your natural inclination be to think about next. Z, clearly! And, then you say, alright, let's think about Z. From Z, I can go to O for total path length so far of 146. Or I could go to A for 150. So what do we want to think about next? T! From T, I could go to L for a total of 229.

Uniform Cost Search

- Breadth-first and Iterative-Deepening find path with fewest steps (hops).
- If steps have unequal cost, this is not interesting.
- How can we find the shortest path (measured by sum of distances along path)?
- Uniform Cost Search:
  - Nodes in agenda keep track of total path length from start to that node
  - Agenda kept in priority queue ordered by path length
  - Get shortest path in queue

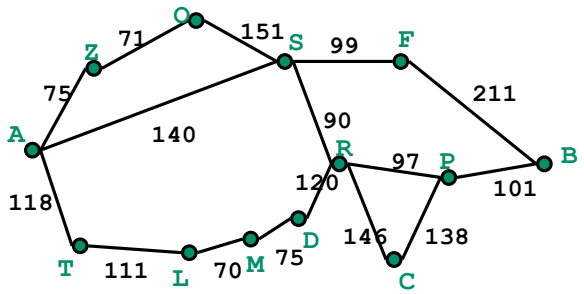
So, uniform cost search says, we're going to store the agenda in a priority queue and when we get out a new node, we get out the one with the shortest path so far.

Uniform Cost Search

- Breadth-first and Iterative-Deepening find path with fewest steps (hops).
- If steps have unequal cost, this is not interesting.
- How can we find the shortest path (measured by sum of distances along path)?
- Uniform Cost Search:
  - Nodes in agenda keep track of total path length from start to that node
  - Agenda kept in priority queue ordered by path length
  - Get shortest path in queue
- Explores paths in contours of total path length; finds optimal path.

What can we say about this search method? Is it going to give us the shortest path to a goal? Yes, because we're exploring cities in contours of real cost (as opposed to BFS where we explored them in order of number of steps) of how long it takes to get there from here.

Uniform Cost Search

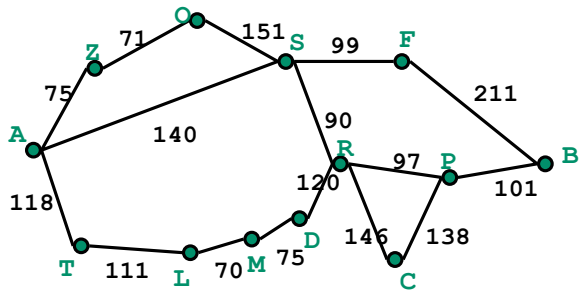


- We examine a node to see if it is the goal only when we take it off the agenda, not when we put it in.

Note that we examine a node to see if it's the goal only when we take it off the agenda, not when we put it in. So, that means if I find a ridiculously long path to the goal, I add it in, but I keep checking all the shorter paths until I'm sure that there's no better way to go. It seems like an optimization to check right now to see if it's the goal, but if you're worrying about finding the shortest path, you should not do that.

Uniform Cost Search

A

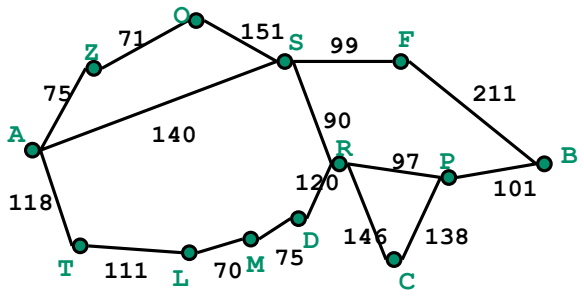


So, let's see how it works on our standard example. We start with A in the agenda.

Uniform Cost Search

~~A~~

Z_{75} T_{118} S_{140}



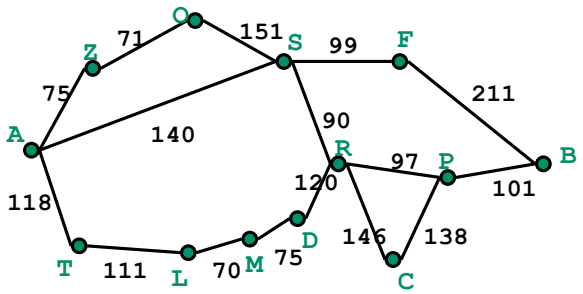
Then we remove it and add Z with cost 75, T with cost 118, and S with cost 140.

Uniform Cost Search

~~A~~

~~Z~~$_{75}$ T$_{118}$ S$_{140}$

T$_{118}$ S$_{140}$ O$_{146}$



Now, we remove the node with the shortest path, which is Z. And we add its children. We don't need to add A, because we've already expanded it. So we just add O, with cost 146.

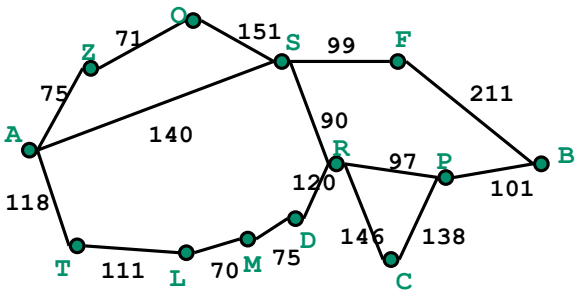
Uniform Cost Search

~~A~~

~~Z~~$_{75}$ T$_{118}$ S$_{140}$

~~T~~$_{118}$ S$_{140}$ O$_{146}$

S$_{140}$ O$_{146}$ L$_{229}$



Now the shortest path goes to T, so we remove it and add child L with cost 229.

Uniform Cost Search

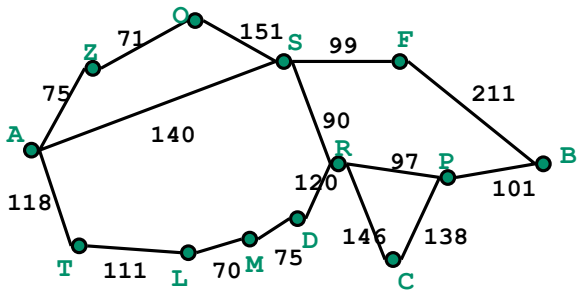
~~A~~

~~Z~~$_{75}$ T$_{118}$ S$_{140}$

~~T~~$_{118}$ S$_{140}$ O$_{146}$

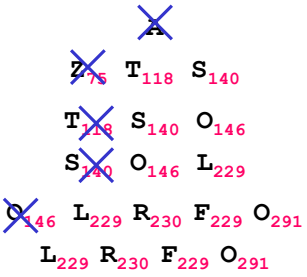
~~S~~$_{140}$ O$_{146}$ L$_{229}$

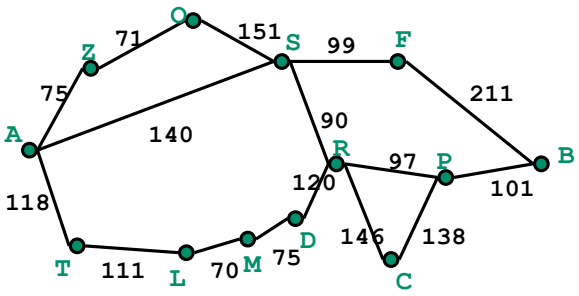
O$_{146}$ L$_{229}$ R$_{230}$ F$_{229}$ O$_{291}$



The shortest path now goes to S, so we remove it and add O with path length 291 (because, although we've "seen" O, it hasn't been expanded yet), F with path length 239, and R with path length 230.

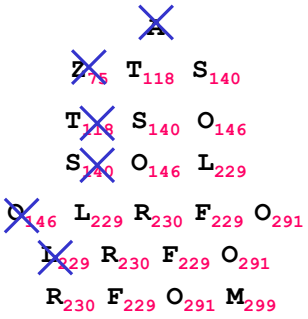
Uniform Cost Search

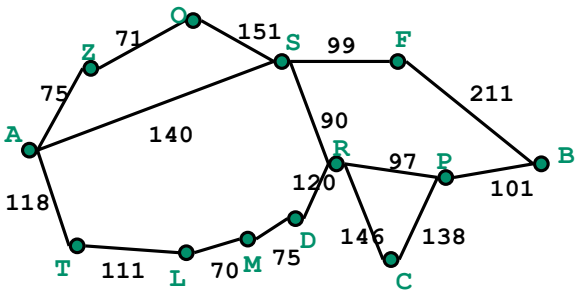




Now we remove O, and since it doesn't have any unexpanded neighbors, we don't add anything.

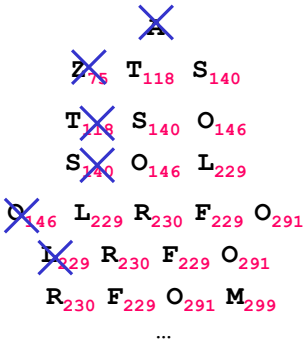
Uniform Cost Search

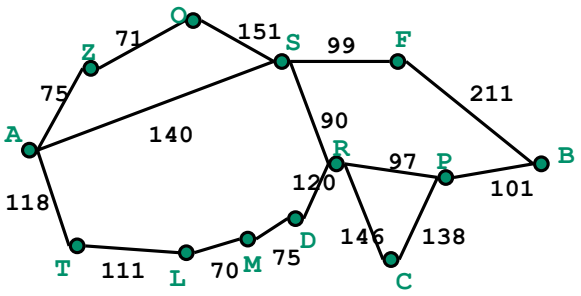




So next we remove L, and add M with length 299.

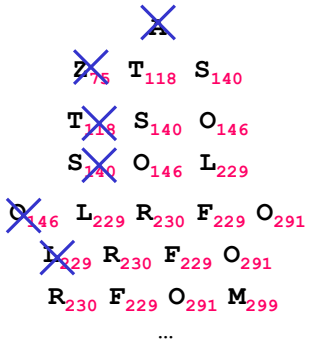
Uniform Cost Search

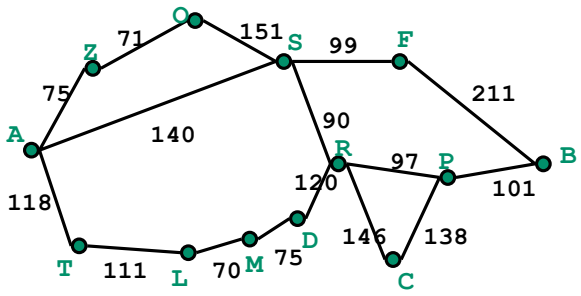




Whew. I'm tired of this. You can see how it's going to come out. We'll expand R, add P, D, and C. Then expand F and add B with a path cost of 440. But we're not willing to stop yet, because there might be a shorter path. Eventually, we'll add B with a length of 428 (via S, R, and P). And when that's the shortest path left in the agenda, we'll know it's the shortest path to the goal.

Uniform Cost Search

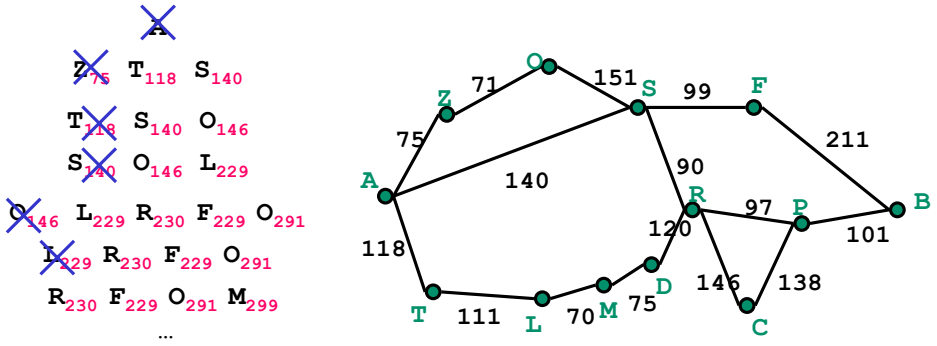




- We examine a node to see if it is the goal only when we take it off the agenda, not when we put it in.

Although we won't go through a formal proof, it seems pretty clear that in the process of removing nodes from the agenda, we're enumerating all the paths in the graph in order of their length from the start state. So, the first path that we remove from the agenda that reaches the goal state is in fact the shortest path to that state.

Uniform Cost Search



- We examine a node to see if it is the goal only when we take it off the agenda, not when we put it in.
- The algorithm is optimal only when the costs are non-negative.

There is one cautionary note. The algorithm is optimal only in some circumstances. What? We have to disallow something; it doesn't come up in road networks. Let's say that your cost is money, not distance. Distance has the property that it keeps getting bigger, there's no way that you could contrive to have traveled less distance than you did yesterday. But, with money, you can earn money. So, the condition is: no negative costs. Because, otherwise you may have something over here that costs 400, which is so bad that you would never want to look at it but then it turns out that you have to drive 400 miles but you win 600 dollars. If this is true, then uniform cost search is not going to work. If you know the range of costs, you can always shift the costs to make them all positive, and since we're just adding them up as you go, you get the same optimum.

Uniform Cost Search

- Time cost is $O(b^m)$ [not $O(b^d)$ as in Russell&Norvig]

What can we say about how long the search takes? The book says, it takes time on the order of b^d. It seems to me that I can construct a map where I have a step of length 200 and then a step of length 1 and there's my goal. Alternatively, I have 200 steps of length 1 and then another step of length 10. So, I'm going to walk down all these little guys first until I take my 200th step and then I'm going to find that nice shallow goal. So, it should really be O(b^m).

Uniform Cost Search

- Time cost is $O(b^m)$ [not $O(b^d)$ as in Russell&Norvig]
- Space cost could also be $O(b^m)$, but is probably more like $O(b^d)$ in most cases.

By a similar argument, I can make a very big tree that has most of the b^m possible fringe elements in the agenda, so the space cost could be as bad as b^m, but it's probably going to be more like b^d in typical cases.

Uninformed vs. Informed Search

- Depth-first, breadth-first and uniform-cost searches are uninformed.

If you don't know anything else about your problem, uniform cost search is about the cleverest thing that you can do. We call this search because we have to look around because we don't know where to find the right answer. But, sometimes we know some information that might be helpful. A lot of times people characterize the methods we have been talking about as "uninformed" or "blind" search methods because there is no extra information that's brought to bear.

Uninformed vs. Informed Search

- Depth-first, breadth-first and uniform-cost searches are uninformed.
- In informed search there is an estimate available of the cost (distance) from each state (city) to the goal.

We'll talk now about "informed" search. What if you knew the shortest line distance to Bucharest from everywhere? So, the roads might be all contorted and twisty and there might be mountains in the way, but you could take your ruler out and measure the straight-line distance.

Uninformed vs. Informed Search

- Depth-first, breadth-first and uniform-cost searches are uninformed.
- In informed search there is an estimate available of the cost (distance) from each state (city) to the goal.
- This estimate (heuristic) can help you head in the right direction.

What does that give you? It gives you some intuition about which places are closer to others. It can head you kind of in the right direction. There will always be some cases when you really have to go around the long way. But, in general, it's at least good to know whether you're headed in the right direction. We can use this information as a "heuristic". Twenty years ago, AI was sort of synonymous with heuristics (it usually defined as a "rule of thumb"; it's related to the Greek word "eureka," which means "I found it!"). It's usually some information that helps you but does not directly contain the answer).

Uninformed vs. Informed Search

- Depth-first, breadth-first and uniform-cost searches are uninformed.
- In informed search there is an estimate available of the cost (distance) from each state (city) to the goal.
- This estimate (heuristic) can help you head in the right direction.
- Heuristic embodied in function h(n), estimate of remaining cost from search node n to the least cost goal.

For us, a heuristic is going to be a function that gives us an estimate of how much it's going to cost to go from here to the goal. Concretely, h(n) is an estimate of the remaining cost from n to goal on the least-cost path..

Uninformed vs. Informed Search

- Depth-first, breadth-first and uniform-cost searches are uninformed.
- In informed search there is an estimate available of the cost (distance) from each state (city) to the goal.
- This estimate (heuristic) can help you head in the right direction.
- Heuristic embodied in function h(n), estimate of remaining cost from search node n to the least cost goal.
- Graph being searched is a graph of states. Search algorithm defines a tree of search nodes. Two paths to the same state generate two different search nodes.

The map we've been looking at is a graph of states, we're talking about states of the world that we can move between. Any particular algorithm that we apply to that graph is going to result in a search tree. And, the search tree has nodes and that search tree might visit the same state twice along different paths, so two ways to get to the same state will be considered as two different nodes in our search tree, but there's really only one underlying state in the world.

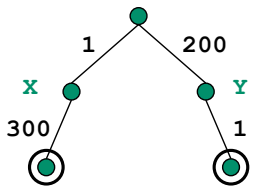
Uninformed vs. Informed Search

- Depth-first, breadth-first and uniform-cost searches are uninformed.
- In informed search there is an estimate available of the cost (distance) from each state (city) to the goal.
- This estimate (heuristic) can help you head in the right direction.
- Heuristic embodied in function h(n), estimate of remaining cost from search node n to the least cost goal.
- Graph being searched is a graph of states. Search algorithm defines a tree of search nodes. Two paths to the same state generate two different search nodes.
- Heuristic could be defined on underlying state; the path to a state does not affect estimate of distance to the goal.

So, in fact, our heuristic function h can apply to states just as well as nodes since the cost of the cheapest path to the goal from the current state s will be independent of the path that we took from the start state to state s. Recall that h is only an estimate; if we knew the actual remaining cost to the goal, we could just use that to find the optimal path.

Using Heuristic Information

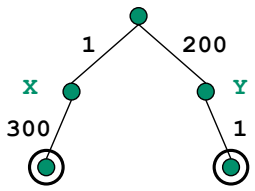
- Should we go to X or Y?



Given a heuristic function, how can we use it? Let's look at a made-up case, where it costs 200 to go to state Y and then 1 to get to the goal. And it costs 1 to go to state X and then 300 to get to the goal.

Using Heuristic Information

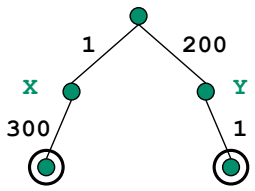
- Should we go to X or Y?
- Uniform cost says go to X



If we were doing uniform cost search, we'd look at these two nodes and we would say, it's only cost us 1 so far to get to X, so let's expand X out some more. But, if we have an estimate of how far it is from X to the goal state, we could use that to help decide where we should go next.

Using Heuristic Information

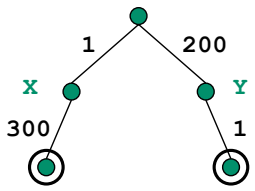
- Should we go to X or Y?
- Uniform cost says go to X
- If h(X) ≫ h(Y), this should affect our choice



How would you like to use that estimate? Add it to the path length so far! How good do we think it would be to keep going this way? Well, I know it's cost me this much to get here and I can guess how much it will cost me to go to the goal, so the sum of these two is an estimate of how far it's likely to be on the best path that goes this way.

Using Heuristic Information

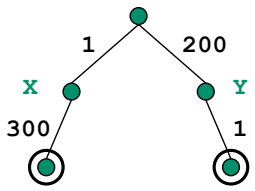
- Should we go to X or Y?
- Uniform cost says go to X
- If h(X) $\gg$ h(Y), this should affect our choice
- If g(n) is path-length of node n, we can use g(n) + h(n) to prioritize the agenda



So, let's define g(n) to be the cost so far (n is a node in the search tree) and we have an estimate of the remaining cost for the path, h(n). And, so the suggestion is to use an algorithm like uniform cost search, but to prioritize the agenda according to g + h.

Using Heuristic Information

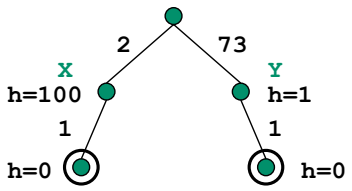
- Should we go to X or Y?
- Uniform cost says go to X
- If h(X) $\gg$ h(Y), this should affect our choice
- If g(n) is path-length of node n, we can use g(n) + h(n) to prioritize the agenda
- This method is called A* [pronounced "A star"]



In fact, this is an algorithm called A*.

Admissibility

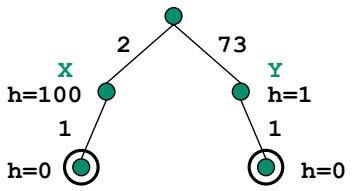
- What must be true about h for A* to find optimal path?



Let's think about when A* is going to be really good and when A* is going to be not so good. What has to be true about h for A* to find the optimal path?

Admissibility

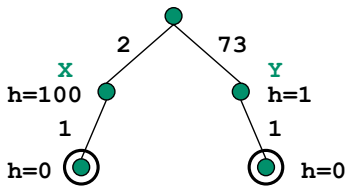
- What must be true about h for A* to find optimal path?
- A* finds optimal path if h is admissible; h is admissible when it never overestimates.



A* is guaranteed to find the optimal path when h is an *admissible* heuristic. h is admissible when it never overestimates the shortest distance to the goal from a node. The book has a proof of the optimality of A*; we're not going to go over it in class, but it's worth reading.

Admissibility

- What must be true about h for A* to find optimal path?
- A* finds optimal path if h is admissible; h is admissible when it never overestimates.
- In this example, h is not admissible.



g(X)+h(X) = 102

g(Y)+h(Y) = 74

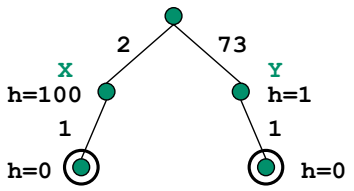
Optimal path is not found!

Otherwise, here's the problem that you could get into. Look at the node for state X in the example it cost 2 to get here and it only costs one more to get to the goal, but you have an h of 100. It costs 2 to get this far but you have an h of 100. On the other branch, it costs 73 to get to Y, and you have an h of 1 and in fact it costs 1 to get to the goal. So, the path via X has total cost 3 and the path via Y has total cost 74.

What would happen if we did A* search in this example? We'd put node X in the agenda with priority (g + h) of 102 and node Y in with priority 74. So, we'd expand Y and insert the goal node with priority 74 (h is 0 at the goal). Then, we'd take out the goal node and return as our result the path that goes through Y. We didn't even expand node X because our guess was that it was terrible. So, for h to be admissible it has to not overestimate.

Admissibility

- What must be true about h for A* to find optimal path?
- A* finds optimal path if h is admissible; h is admissible when it never overestimates.
- In this example, h is not admissible.
- In route finding problems, straight-line distance to goal is admissible heuristic.

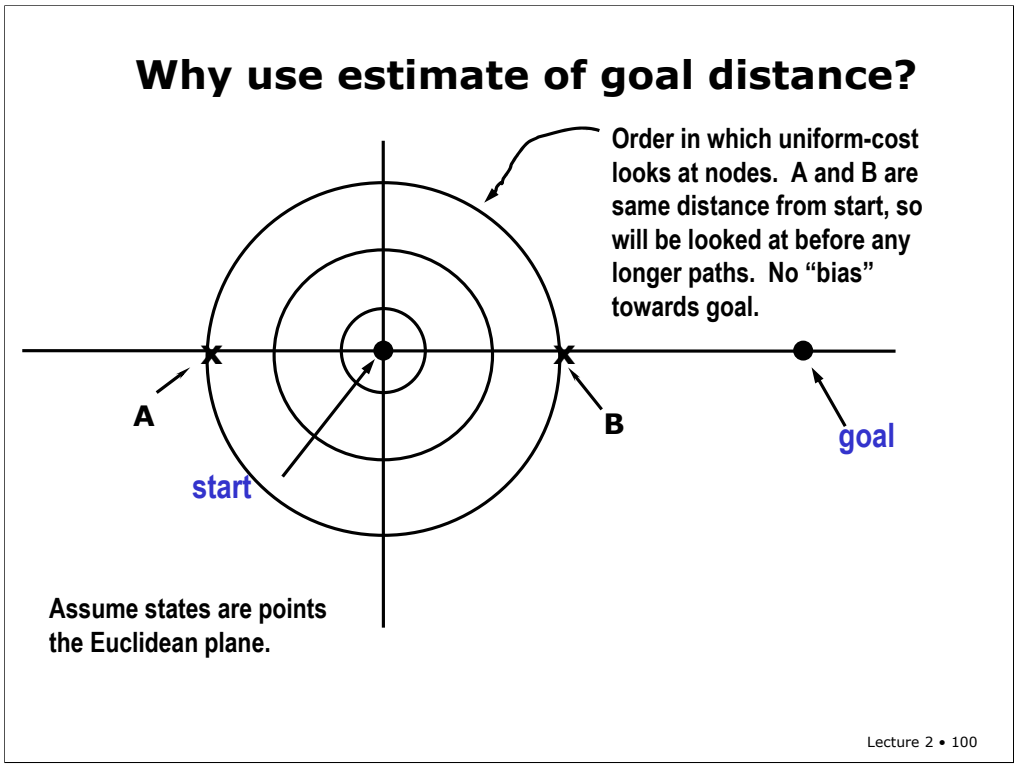


g(X)+h(X) = 102

g(Y)+h(Y) = 74

Optimal path is not found!

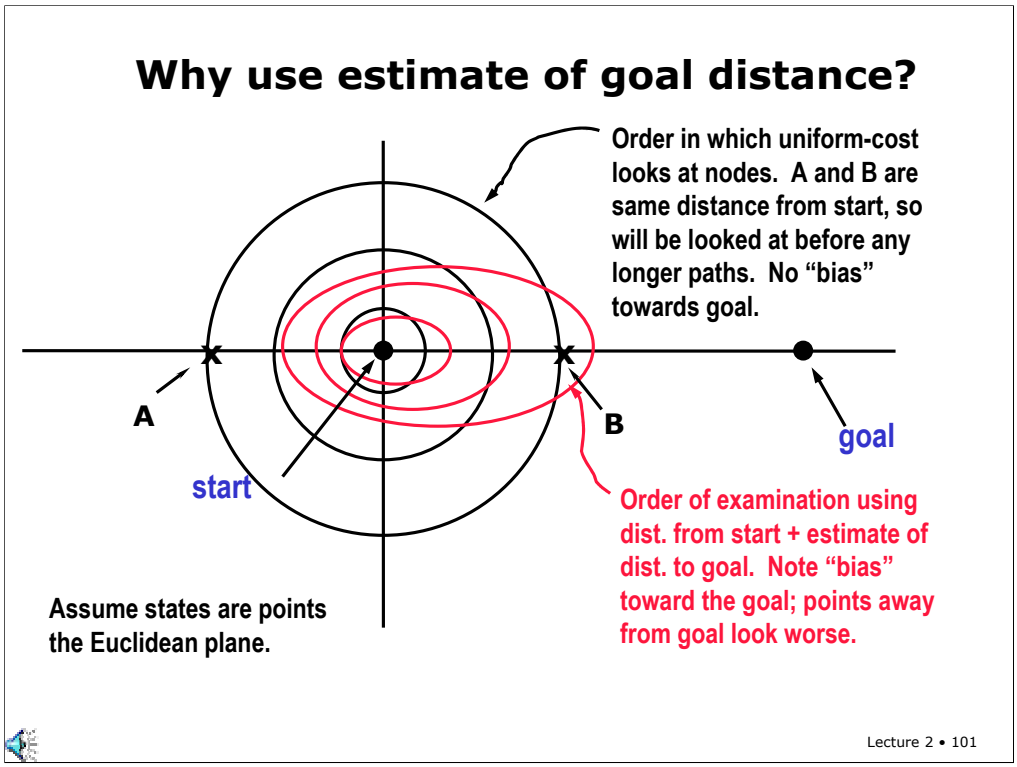
In our path planning example, if in fact the cost is distance, is shortest line distance admissible? Yes! That makes it an extra good heuristic. No matter how hard you try you can't find a path shorter than 150 between A and C (we assume that the triangle inequality holds). So, if we have an admissible heuristic, A* finds the optimal path.



In this slide and the next, we just want to gain some intuition about why A* is a better search method than uniform cost.

Imagine that there are a lot of other roads radiating from the start state. We can think of breadth-first search searching out from the start state in contours of numbers of steps from the start state (all states that are one step away are expanded first, then those that are two steps away, etc). In uniform-cost search, we search in contours of distance from the start state, expanding states in order of the length of the shortest path from the start.

In this figure, we see that states A and B will be visited at about the same time, even though A is quite far away from the goal, because they're the same distance from the start state.



You can think of A* as searching contours of distance from the start state + estimated distance to the goal. So, the estimated distance term should skew the search in the direction of the goal; so that you are searching contours that are warped out in the direction of the goal. As long as you do that in the right order and the heuristic doesn't mislead you, you're ok. And you will expand fewer nodes than you would have if you were using uniform cost search.

Heuristics

- If we set h=0, then A* is uniform-cost search; h=0 is admissible heuristic (when all costs are non-negative).

Uniform cost search is an instance of A* then. What's the heuristic? h=0. It's admissible. If you say that everywhere I go, that's it, I'm going to be at the goal (being very optimistic) that's going to give you uniform cost search.

Heuristics

- If we set h=0, then A* is uniform-cost search; h=0 is admissible heuristic (when all costs are non-negative).
- Very difficult to find heuristics that guarantee sub-exponential worst-case cost.

There's been some theoretical work on the efficiency of heuristics. I want to have a heuristic that cuts down the search space. It would be great to prove a theorem that says that if your heuristic has some property, then you'll only have to search some much smaller fraction of the space. There are such theorems, we won't go into them here, there are some pointers in the book, but your heuristic has to be really, really good in order for it give you sub-exponential search time; it nearly has to know the answer with a little bit of uncertainty.

Heuristics

- If we set h=0, then A* is uniform-cost search; h=0 is admissible heuristic (when all costs are non-negative).
- Very difficult to find heuristics that guarantee sub-exponential worst-case cost.
- Heuristic functions can be solutions to "relaxed" version of original problem, e.g. straight line distance is solution to route-finding when we relax constraint to follow the roads.

How do you find a heuristic? In the path-planning problem, it wasn't too hard to think of the shortest-line distance. In general, you can often find heuristic functions by relaxing your problem. Sure, we have the problem of finding our way from A to B on the road map, which is the more constrained version of the problem. You can relax the problem and say, let's find the way from A to B given that we can go any way we want to. And, sometimes you can take a hard problem and find a relaxed problem that's really easy to solve such that the cost of any path in the relaxed problem is less than the corresponding path in the original problem and then you can use the solutions to the relaxed problem as your heuristic.

Search Problems

- In problem-solving problems, we want a path as the answer, that is, a sequence of actions to get from one state to another.

When we wrote the problem-solving problem definition, it really captured the notion of planning problems where you need to find some trajectory of actions that you can do to get from a start to a goal state, and what you're interested in is the trajectory. But there are a lot of other situations where search comes up where you are not really interested in the trajectory; all you're really interested in is finding an answer. So, for example, the traveling salesman problem (TSP) is a paradigmatic example. The question is not how do I get from A to B most efficiently but "is there a trajectory through this graph that satisfies certain properties, like that it visits all the cities." So, another real class of search problems are ones where we don't care about the path.

Search Problems

- In problem-solving problems, we want a path as the answer, that is, a sequence of actions to get from one state to another.
- In search problems, all we want is the best state (that satisfies some constraints).
  - Set of states: S
  - Initial state
  - Operators: $S \rightarrow S$
  - Cost (Utility): $S \rightarrow \mathbb{R}$

Let's call this a search problem rather than a problem-solving problem. In a search problem, we have a set of states; we typically still have a set of operators that lets us go from state to state and we now have a utility function that just maps states into utility. and now our goal is to find the best state in the space. One view that you could take of a problem like this is to say "who needs the operators?" It's not like you are trying to think about how to really walk around in this space, but now you can think about these operators not as things that you do in the world to move you from city to city, but as things that you do in the information space to move you from potential solution to potential solution to the problem.

Example: Traveling Salesman

- In traveling salesman problem (TSP) we want a least-cost path that visits all the cities in a graph once.

Okay. So, in TSP we want to find the least-cost path that visits all the cities in a graph exactly once.

Example: Traveling Salesman

- In traveling salesman problem (TSP) we want a least-cost path that visits all the cities in a graph once.
- Note that this is not a route-finding problem, since we must visit every city, only the order of visit changes.

Note that this is not a route finding problem, since we must visit every city. The question is only the order in which we visit them.

Example: Traveling Salesman

- In traveling salesman problem (TSP) we want a least-cost path that visits all the cities in a graph once.
- Note that this is not a route-finding problem, since we must visit every city, only the order of visit changes.
- A state in the search for TSP solution is a complete tour of the cities.

So, in TSP, we're not going to think about the space being the cities in the graph, instead we're going to think of the states as being possible tours (solutions).

Example: Traveling Salesman

- In traveling salesman problem (TSP) we want a least-cost path that visits all the cities in a graph once.
- Note that this is not a route-finding problem, since we must visit every city, only the order of visit changes.
- A state in the search for TSP solution is a complete tour of the cities.
- An operator is not an action in the world that moves from city to city, it is an action in the information space that moves from one potential solution (tour) to another.

Now, we can think about solving TSP by taking a tour and changing it around in various ways and looking to see how good they are. So, what are some operators that you may use on tours in TSP?

Example: Traveling Salesman

- In traveling salesman problem (TSP) we want a least-cost path that visits all the cities in a graph once.
- Note that this is not a route-finding problem, since we must visit every city, only the order of visit changes.
- A state in the search for TSP solution is a complete tour of the cities.
- An operator is not an action in the world that moves from city to city, it is an action in the information space that moves from one potential solution (tour) to another.
- Possible TSP operators: Swap two cities in a tour

You could swap two cities. A TSP solution is the order in which you visit the cities and you say, “I’m going to swap these two”. That gives you another tour and it has some cost.

You can think about finding the best thing in the space as a search, but not one that’s related to acting in the world, it’s a search that’s informational.

Example: Square Root

- Given $y=x^2 \in \mathbb{R}$ find $x \in \mathbb{R}$
- Utility of x is a measure of error, e.g. $U = ½ (x^2 - y)^2$
- Operator: $x \rightarrow x - r \nabla_x U$ (for small stepsize r)
  - take a step down gradient (wrt x) of the error
  - For example, $x = x - r (x^2 - y) 2x$
  - Assume y = 7, start with guess x = 3, let r = 0.01
  - Next guesses are: 2.880, 2.805, 2.756, …, 2.646

Another example of search is any kind of gradient descent method. Let's say that you'd like to find the square root of a number. You can actually apply this in continuous spaces. What are the states? The states are real numbers. The utility is going to be the distance between x squared and the answer that we want. What's a typical operator in this space? I'm trying to find the square root of 7, I think it's 3, I look at the difference between 3 squared and 7 and I can do what? You could just guess higher or lower, if I'm too high, go down a little if I'm too low, go up a little. What's a more direct thing to do? Step to the median. Or, if we can compute the derivative, we can take a step along the gradient.

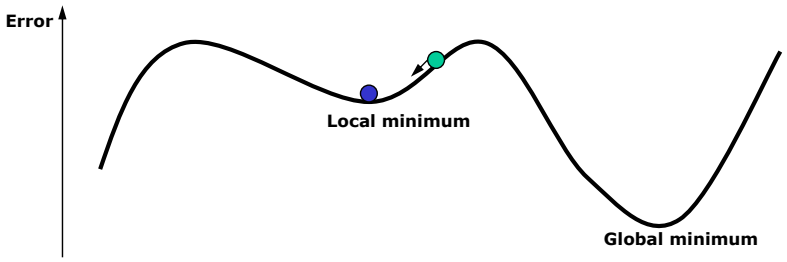
Example: Square Root

- Given $y=x^2 \in \mathbb{R}$ find $x \in \mathbb{R}$
- Utility of x is a measure of error, e.g. $U = ½ (x^2 - y)^2$
- Operator: $x \rightarrow x - r \nabla_x U$ (for small stepsize r)
  - take a step down gradient (wrt x) of the error
  - For example, $x = x - r (x^2 - y) 2x$
  - Assume y = 7, start with guess x = 3, let r = 0.01
  - Next guesses are: 2.880, 2.805, 2.756, …, 2.646
- We can prove that there is a unique x whose error value is minimal and that applying this operator repeatedly (for some value of r) will find this minimum x (to some specified accuracy).

It turns out that in the square root case you can prove something about this search. You can run a deterministic algorithm and prove that you're going to an optimal point in the space if you start out right.

Multiple Minima

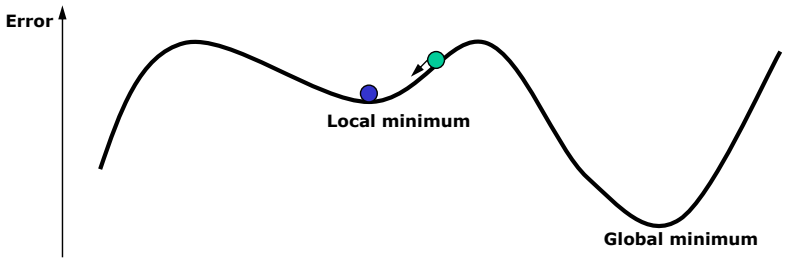
- Most problems of interest do not have unique global minima that can be found by gradient descent from an arbitrary starting point.
- Typically, local search methods (such as gradient descent) will find local minima and get stuck there.



A lot of times you can't guarantee that you'll find the optimal solution. I'll illustrate it in a continuous space, but the ideas apply equally well in a discrete space. Imagine that I'm trying to find the minimum of a space and I find myself at the green point and I decide that I'm going to take a step in the direction of the gradient, which seems well justified. Well, I'm going to find myself at the blue point, and I'll stay there happily. In more complicated spaces, sometimes deterministic algorithms don't work out very well.

Multiple Minima

- Most problems of interest do not have unique global minima that can be found by gradient descent from an arbitrary starting point.
- Typically, local search methods (such as gradient descent) will find local minima and get stuck there.
- How can we escape from local minima?
  - Take some random steps!
  - Re-start from randomly chosen starting points



So, what do people do? Randomize! You can try multiple runs from different initial conditions. You can say, going down the gradient seems to be a good thing most of the time but sometimes I just want to be wild and crazy because being wild and crazy might get me out of this bowl and put me in that bowl over there that contains the global minimum.

Simulated Annealing

- T = initial temperature
- x = initial guess
- v = Energy(x)
- Repeat while T > final temperature
  - Repeat n times
    - $x' \leftarrow$ Move(x)
    - v' = Energy(x')
    - If $v' < v$ then accept new x [$x \leftarrow x'$]
    - Else accept new x with probability exp(-($v' - v$)/kT)
  - T = 0.95T /* for example */
- At high temperature, most moves accepted (and can move between "basins")
- At low temperature, only moves that improve energy are accepted

Probably many of you have been exposed to the idea of simulated annealing, which, by analogy to cooling of a metal, says: I want to reach some minimum energy state, so I tend to take steps that decrease the energy. But, depending on a parameter that is called temperature (when the temperature is high) I am going to every now and then take really wild and crazy steps and as the temperature decreases, I am less and less likely to take such wild and crazy steps. This has the characteristic that at the beginning I am going to be hopping around almost entirely at random but as I bring the temperature down, I hop less and less randomly and I do something more like gradient descent. The intuition is that these big, long, random hops, hop you into the nice big basins, and as the temperature gets lower and lower you do something like gradient descent which helps get you to the bottom of the big bowl. You can also do this in discrete spaces. What we're going to do in the first homework assignment is explore the relationship between some systematic search methods and some randomized search methods for solving problems of this kind that are more like TSP. We'll look at planning problems later on.

Just for reference, we're including pseudocode on the slide. But we won't study this technique in any further detail.

Search Demonstration

- There's a cool applet at UBC for playing around with search algorithms: www.cs.ubc.ca/labs/lci/CIspace

Recitation Problems

Problems 3.17 a, b, f (Russell & Norvig)

What would be a good heuristic function in that domain? (from 3.17)

What would be a good heuristic function for the Towers of Hanoi problem? (look this up on the web, if you don't know about it)

Other practice problems that we might talk about in recitation: 4.4, 4.11a,b