MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Fall Semester, 2007

Lecture 11 Notes

Long-term decision-making and search

Long-term decision-making

So far in this course, we have taken a couple of different approaches to action-selection. In the utility-functions section, we considered a set of possible next actions, evaluated the utility of the results, and chose the one with the best outcome. When we were studying control loops, we didn't have the program explicitly consider alternative courses of action, but we, as designers, made models of the controller's behavior in the world and tried to prove whether it would behave in the way we wanted it to, taking into account a longer-term pattern of behavior.

Often, we will want a system to generate complex long-term patterns of behavior, but we will not be able to write a simple control rule to generate those behavior patterns. In that case, we'd like the system to evaluate alternatives for itself, but instead of evaluating single actions, it will have to evaluate whole sequences of actions, deciding whether they're a good thing to do given the current state of the world.

Let's think of the problem of navigating through a city, given a road map, and knowing where we are. We can't usually decide whether to turn left at the next intersection without deciding on a whole path.

As always, the first step in the process will be to come up with a formal model of a real-world problem that abstracts away the irrelevant detail. So, what, exactly, is a path? The car we're driving will actually follow a trajectory through continuous space(time), but if we tried to plan at that level of detail we would fail miserably. Why? First, because the space of possible trajectories through two-dimensional space is just too enormous. Second, because when we're trying to decide which roads to take, we don't have the information about where the other cars will be on those roads, which will end up having a huge effect on the detailed trajectory we'll end up taking.

So, we can divide the problem into two levels: planning in advance which turns we'll make at which intersections, but deciding dynamically exactly how to control the steering wheel and the gas to best move from intersection to intersection, given the current circumstances (other cars, stop-lights, etc.).

We can make an abstraction of the driving problem to include road intersections and the way they're connected by roads. Then, given a start and a goal intersection, we could consider all possible paths between them, and choose the one that is best.

What criteria might we use to evaluate a path? There are all sorts of reasonable ones: distance, time, gas mileage, traffic-related frustration, scenery, etc. Generally speaking, the approach we'll outline below can be extended to any criterion that is additive: that is, your happiness with the whole path is the sum of your happiness with each of the segments. For now, though, we'll adopt the simple criterion of wanting to find a path with the fewest "steps" from intersection to intersection.

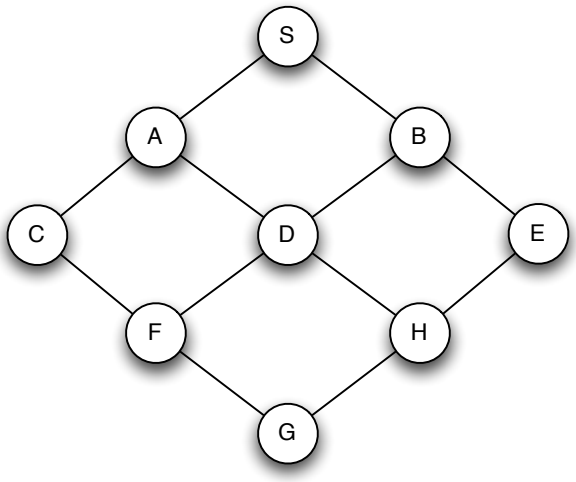

Figure 1: Map of small city 1

Now, one algorithm for deciding on the best path through a map of the road intersections in our (very small) world (shown in figure 1), would be to enumerate all the paths, evaluate each one according to our criterion, and then return the best one. The problem is that there are *lots* of paths. Even in our little domain, with 9 intersections, there are 210 paths from the intersection labeled S to the one labeled G.

We can get a much better handle on this problem, by formulating it as an instance of a graph search (or a "state-space search") problem, for which there are simple algorithms that perform well.

State-space search

We'll model a state-space search problem formally as consisting of

- a set (possibly infinite) of *states* the system can be in
- a *starting state* which is an element of the set of states
- a *goal test*, which is a function that can be applied to any state, and returns `True` if that state can serve as a goal[1]
- a *successor function*, which is a function that can be applied to a state, yielding a set of states that can be reached from the original state with one primitive step

The decision about what constitutes a *primitive step* is a modeling decision. It could be to drive to the next intersection, or to drive a meter, or a variety of other things, depending on the domain. The only requirement is that it terminate in a well-defined next state (and that, when it is time to execute the plan, we will know how to to execute the primitive step.)

We can think of this model as specifying a *graph* (in the computer scientist's sense), in which the states are the nodes and the successor function specifies what arcs exist between the nodes.

So, for the little world in figure 1, we might make a model in which

[1] Although in many cases we have a particular goal state (such as the intersection in front of my house), in other cases, we may have the goal of going to any gas station, which can be satisfied by many different intersections.

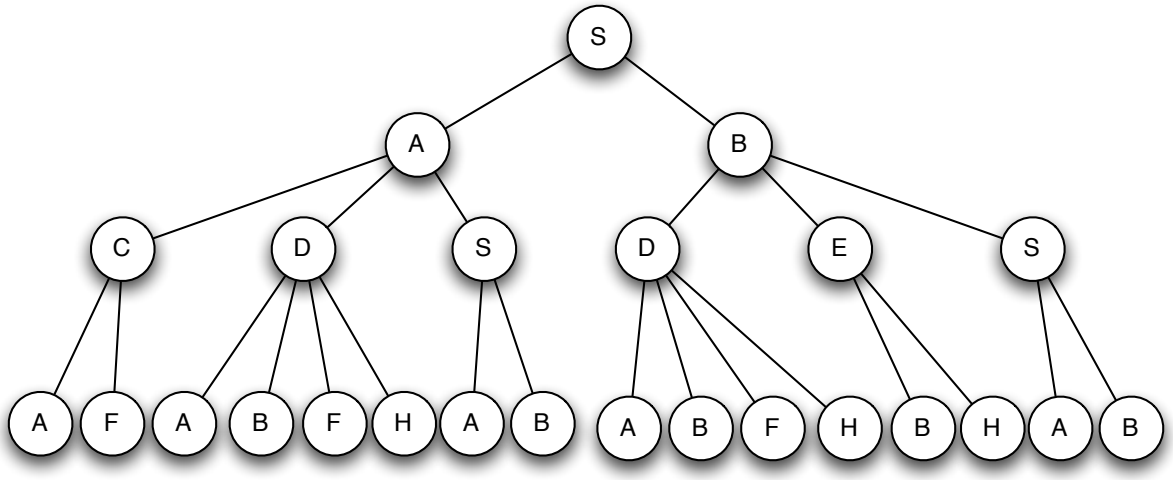

Figure 2: Partially expanded search tree for city 1

- The set of states is the intersections $\{S, A, B, C, D, E, F, G, H\}$.
- The starting state is S.
- The goal test is

```
def g(x):
    return x == 'E'
```

- The successor function is defined using a dictionary:

```
succ = {'S' : ['A','B'], 'A': ['S', 'C', 'D'], 'B': ['S', 'D', 'E'], \
        'C' : ['A', 'F'], 'D': ['A', 'B', 'F', 'H'], 'E' : ['B', 'H'], \
        'F' : ['C', 'D', 'G'], 'H' : ['D', 'E', 'G'], 'G' : ['F', 'H']}
def successors(x):
    return succ[x]
```

We can think of this structure as defining a *search tree*, as shown in figure 2. It has the starting state, S, at the root node, and then each node has its successor states as children. Layer k of this tree contains all possible paths through the graph of length k.

Some of these paths are completely ridiculous. It can never be reasonable, if we're trying to find the shortest path between two states, to go back to a state we have previously visited on that same path. So we can adopt the following rule:

Pruning rule 1: Don't consider any path that visits the same state twice.

This rule will let us remove a number of branches from the tree, as shown in figure 3.

This seems like a much more reasonable set of choices. Now, how can we systematically search for a path to the goal? There are two plausible strategies:

- Start down a path, keep trying to extend it until you get stuck, in which case, go back to the last choice you had, and go a different way. This is how kids often solve mazes. We'll call it *depth-first* search.
- Go layer by layer through the tree, first considering all paths of length 1, then all of length 2, etc. We'll call this *breadth-first* search.

In either case, while we're doing the search, we'll have a set of paths that we have partially explored and need to remember for further consideration. We'll store them in a list, called the *agenda*.

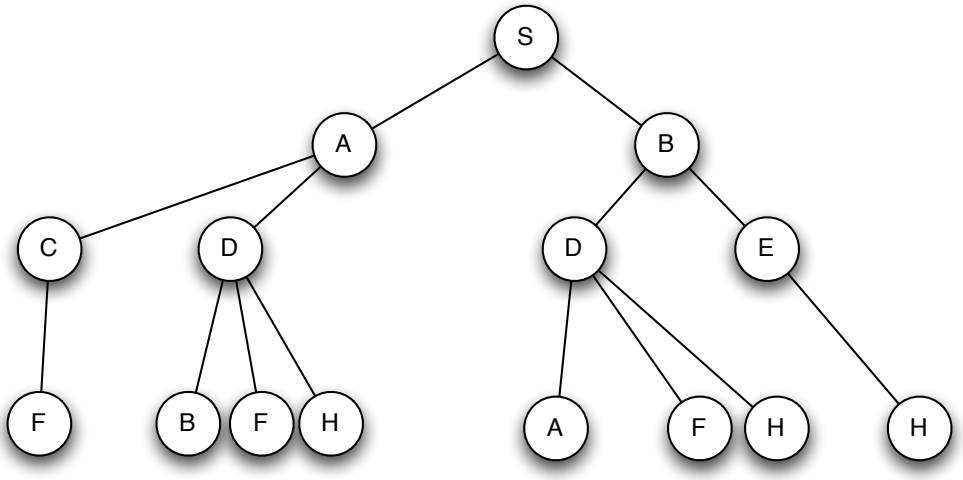

Figure 3: Partially expanded search tree for city 1, after pruning rule 1

Generic Search algorithm

The basic algorithms for searching graphs have the following form:

```
def search(initialState, goalTest, successors):
    agenda = [[initialState]]
    while agenda != []:
        path = selectNextPathToExpand(agenda)
        newPaths = []
        for newState in successors(path[-1]):
            if goalTest(newState):
                return path + [newState]
            elif newState in path:
                pass
            else:
                newPaths.append(path + [newState])
        add(newPaths, agenda)
    return None
```

We start by initializing the agenda to contain a single path, which itself contains a single initial state. This is the state from which the search will begin. Now, we enter a loop that will run until the agenda is empty (we have no more paths to consider), but could stop sooner.

Inside the loop, we select a path from the agenda (more on how we decide which one to take out in a bit) and *expand it*. To expand a path, we get ahold of the the last state in that path (`path[-1]`), get all possible successors of that state, and consider extending our current path with each of those successor states. We'll call this process *visiting* each of the successor states.

First, we look to see of one of those new states satisfies the goal test. If it does, we return the path with this good state added to it, and we're done! Next, we check to see if the state we're thinking about adding is already in the path. If it is, we can apply pruning rule 1, and just skip this path. Finally, if neither of those conditions holds, we add the new state to the path and put this extended path on our list of new paths. Once we have a list of new paths derived from the path we're expanding, we add them to the agenda.

We continue this process until we find a goal state or the agenda becomes empty. This is not quite yet an algorithm, though, because we haven't said anything about the order in which to expand

the paths that are currently in the agenda. First, we need to engage in a brief digression...

Stacks and Queues

In designing algorithms we frequently make use of two simple data structures: stacks and queues. You can think of them both as abstract data types that support two operations: `push` and `pop`. The `push` operation adds an element to the stack or queue, and the `pop` operation removes an element. The difference between a stack and a queue is what element you get back when you do a `pop`.

stack : When you `pop` a stack, you get back the element that you most recently put in. A stack is also called a LIFO, for *last in, first out*.

queue : When you `pop` a queue, you get back the element that you put in earliest. A queue is also called a FIFO, for *first in, first out*.

Depth-First Search

So, now we can easily describe depth-first search by saying that it's an instance of the generic search procedure described above, but in which the agenda is a *stack*: that is, that we always expand the path we most recently put into the agenda.

The code listing below shows our implementation of depth-first search. We are using `agenda = newPaths + agenda` to add our list of new paths to the beginning of the agenda. And `agenda.pop(0)` will return the first element of a list and remove it from the list. So, the agenda is behaving like a stack for us.

```
def search(initialState, goalTest, successors):
    agenda = [[initialState]]
    while agenda != []:
        path = agenda.pop(0)
        newPaths = []
        for newState in successors(path[-1]):
            if goalTest(newState):
                return path + [newState]
            elif newState in path:
                pass
            else:
                newPaths.append(path + [newState])
        agenda = newPaths + agenda
    return None
```

So, let's see how this search method behaves on City 1, with start state S and goal state H. Here is a trace of the algorithm (you can get this in the code we distribute by setting `verbose = True` before you run it.)

```
>>> depthFirst('S', lambda x: x == 'H', successorsCity1)
agenda:  []
   expanding:  ['S']
agenda:  [['S', 'B']]
   expanding:  ['S', 'A']
agenda:  [['S', 'A', 'D'], ['S', 'B']]
   expanding:  ['S', 'A', 'C']
agenda:  [['S', 'A', 'D'], ['S', 'B']]
   expanding:  ['S', 'A', 'C', 'F']
agenda:  [['S', 'A', 'C', 'F', 'G'], ['S', 'A', 'D'], ['S', 'B']]
   expanding:  ['S', 'A', 'C', 'F', 'D']
10  states visited
['S', 'A', 'C', 'F', 'D', 'H']
```

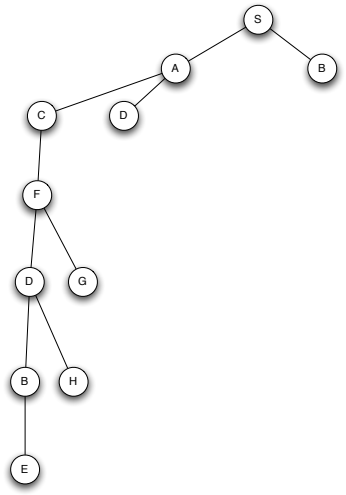

Figure 4: Search tree generated by depth-first search for city 1

You can see that in this world, the search never needs to "backtrack", that is, to go back and try expanding an older path on its agenda. It is always able to push the current path forward until it reaches the goal. Figure 4 shows the search tree generated during the depth-first search process.

Figure 5 shows another city map (it's a weird city, we know, but maybe a bit like trying to drive in Boston). In this city, depth-first search behaves a bit differently (trying to go from S to G this time):

```
>>> depthFirst('S', lambda x: x == 'G', successorsCity2)
agenda:  []
   expanding:  ['S']
agenda:  [['S', 'B']]
   expanding:  ['S', 'A']
agenda:  [['S', 'A', 'D'], ['S', 'B']]
   expanding:  ['S', 'A', 'C']
agenda:  [['S', 'A', 'D'], ['S', 'B']]
   expanding:  ['S', 'A', 'C', 'D']
agenda:  [['S', 'B']]
   expanding:  ['S', 'A', 'D']
agenda:  [['S', 'B']]
   expanding:  ['S', 'A', 'D', 'C']
agenda:  []
   expanding:  ['S', 'B']
agenda:  [['S', 'B', 'F']]
   expanding:  ['S', 'B', 'E']
agenda:  []
   expanding:  ['S', 'B', 'F']
10  states visited
['S', 'B', 'F', 'G']
```

In this case, it explores all possible paths down in the left branch of the world, and then has to backtrack up and over to the right branch.

Here are some important properties of depth-first search:

- It may run forever in an infinite domain (as long as the path it's on has a new successor that hasn't been previously visited, it can go down that path forever).
- It will run forever if we don't apply pruning rule 1, potentially going back and forth from one state to another, forever.

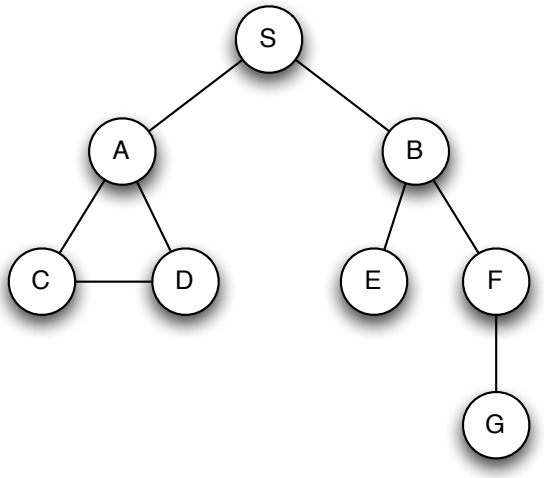

Figure 5: Map of small city 2

- It doesn't necessarily find the shortest path (as we can see from the very first example).
- It is generally efficient in the amount of space it requires to store the agenda, which will be a constant factor times the depth of the path it is currently considering (we'll explore this in more detail later.)

Breadth-First Search

To change to breadth-first search, we need to choose the oldest, rather than the newest paths from the agenda to expand. There are two easy ways to change the depth-first code to make this happen: either continue adding new paths to the beginning of the agenda and select paths to be expanded off of the end of the agenda; or continue to select paths to be expanded from the beginning of the agenda and add new paths to the end of the agenda. We'll take the second approach, for the cosmetic reason that it will make both searches have a "left-to-right" bias in going through the tree, so it will be easier for us to discuss and predict the results in detail (but there's no reason to expect it to work any better or worse).

Here is how breadth-first search works, looking for a path from S to H in city 1:

```
>>> breadthFirst('S', lambda x: x == 'H', successorsCity1)
agenda:  [['S']]
   expanding:  ['S']
agenda:  [['S', 'A'], ['S', 'B']]
   expanding:  ['S', 'A']
agenda:  [['S', 'B'], ['S', 'A', 'C'], ['S', 'A', 'D']]
   expanding:  ['S', 'B']
agenda:  [['S', 'A', 'C'], ['S', 'A', 'D'], ['S', 'B', 'D'], ['S', 'B', 'E']]
   expanding:  ['S', 'A', 'C']
agenda:  [['S', 'A', 'D'], ['S', 'B', 'D'], ['S', 'B', 'E'], ['S', 'A', 'C', 'F']]
   expanding:  ['S', 'A', 'D']
11  states visited
['S', 'A', 'D', 'H']
```

We can see it proceeding systematically through paths of length two, then length three, then length four, finding the goal among the length-four paths.

This systematic search through paths of increasing length, coupled with the termination as soon as a path to a goal state is found, means that breadth-first search will always return the shortest path to the goal (or one of the shortest paths, if there are multiple paths of the same length). We'll see later that this can come at the price of requiring more space to store the agenda than in depth-first search.

Dynamic programming

Let's look at breadth-first search in the city 1 example, but this time with goal state G:

```
>>> breadthFirst('S', lambda x: x == 'G', successorsCity1)
agenda:  [['S']]
   expanding:  ['S']
agenda:  [['S', 'A'], ['S', 'B']]
   expanding:  ['S', 'A']
agenda:  [['S', 'B'], ['S', 'A', 'C'], ['S', 'A', 'D']]
   expanding:  ['S', 'B']
agenda:  [['S', 'A', 'C'], ['S', 'A', 'D'], ['S', 'B', 'D'], ['S', 'B', 'E']]
   expanding:  ['S', 'A', 'C']
agenda:  [['S', 'A', 'D'], ['S', 'B', 'D'], ['S', 'B', 'E'],
          ['S', 'A', 'C', 'F']]
   expanding:  ['S', 'A', 'D']
agenda:  [['S', 'B', 'D'], ['S', 'B', 'E'], ['S', 'A', 'C', 'F'],
          ['S', 'A', 'D', 'B'], ['S', 'A', 'D', 'F'], ['S', 'A', 'D', 'H']]
   expanding:  ['S', 'B', 'D']
agenda:  [['S', 'B', 'E'], ['S', 'A', 'C', 'F'], ['S', 'A', 'D', 'B'],
          ['S', 'A', 'D', 'F'], ['S', 'A', 'D', 'H'], ['S', 'B', 'D', 'A'],
          ['S', 'B', 'D', 'F'], ['S', 'B', 'D', 'H']]
   expanding:  ['S', 'B', 'E']
agenda:  [['S', 'A', 'C', 'F'], ['S', 'A', 'D', 'B'], ['S', 'A', 'D', 'F'],
          ['S', 'A', 'D', 'H'], ['S', 'B', 'D', 'A'], ['S', 'B', 'D', 'F'],
          ['S', 'B', 'D', 'H'], ['S', 'B', 'E', 'H']]
   expanding:  ['S', 'A', 'C', 'F']
17  states visited
['S', 'A', 'C', 'F', 'G']
```

The first thing that is notable about this trace is that it ends up visiting 23 states in a domain with 9 different states. The issue is that it is exploring multiple paths to the same state. For instance, it has both SAD and SBD in the agenda. Even worse, it has both SA and SBDA in there! We really don't need to consider all of these paths. We can make use of the following example of the dynamic programming principle:

The shortest path from X *to* Z *that goes through* Y *is made up of the shortest path from* X *to* Y *and the shortest path from* Y *to* Z*.*

So, as long as we find the shortest path from the start state to some intermediate state, we don't need to consider any other paths between those two states; there is no way that they can be part of the shortest path between the start and the goal. This insight is the basis of a new pruning principle:

Pruning rule 2: Don't consider any path that visits a state that you have already visited via some other path.

In breadth-first search, because of the orderliness of the expansion of the layers of the search tree, we can guarantee that the first time we visit a state, we do so along the shortest path.[2] So, whenever

[2]Note that for more sophisticated search techniques, such as uniform cost, in which different arcs have different costs, or A*, in which we use an estimate of cost-to-go to make the search more efficient, we can do a version of this trick, but it has to be a bit more complicated. You'll learn this someday in 6.034.

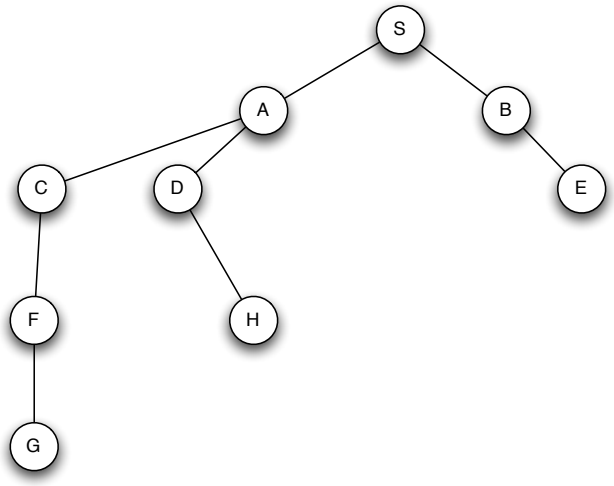

Figure 6: Search tree generated by breadth-first search with dynamic programming, for city 1

we visit a state for the first time, we'll remember that path to the state. And because we'll never consider multiple paths to a state, our agenda can just be a list of states we need to think about, rather than paths. This will let us develop a breadth-first search algorithm that is both somewhat simpler and a good deal more efficient than the one we had before.

```
def breadthFirstDP(initialState, goalTest, successors):
    agenda = [initialState]
    pathTo = {initialState: [initialState]}
    while agenda != []:
        state = agenda.pop(0)
        newStates = []
        for newState in successors(state):
            if not pathTo.has_key(newState):
                pathTo[newState] = pathTo[state] + [newState]
                if goalTest(newState):
                    return pathTo[newState]
                else:
                    newStates.append(newState)
        agenda = agenda + newStates
    return None
```

We are actually keeping two data structures: the `agenda`, which is a list of states that have been visited but not yet expanded, and `pathTo`, which is a dictionary that maps a state `s` to a path from the initial state to `s`, if `s` has been visited, and otherwise has no entry for `s`. We initialize `pathTo` by saying that we know a path to the initial state, consisting of the list containing the initial state.

Now, we enter into a loop, repeatedly popping the first state off of the agenda. For each successor, `newState`, of that state, we check to see whether we already know a path to it. If so, there's nothing more to be done (we know that its successors have already been visited). If not, we add the path to that state into `pathTo`. The path to `newState` is just the path to the state we are expanding, with `newState` added onto the end. We check to see if `newState` satisfies the goal test, and, if so, we terminate and return the path to it. If not, we put `newState` on the list of successor states to be added to the agenda.

As before, if the agenda becomes exhausted, we fail and return `None`.

So, let's see how this performs on the task of going from `S` to `G` in city 1:

```
>> breadthFirstDP('S', lambda x: x == 'G', successorsCity1)
agenda:  ['S']
   expanding:  S
agenda:  ['A', 'B']
   expanding:  A
agenda:  ['B', 'C', 'D']
   expanding:  B
agenda:  ['C', 'D', 'E']
   expanding:  C
agenda:  ['D', 'E', 'F']
   expanding:  D
agenda:  ['E', 'F', 'H']
   expanding:  E
agenda:  ['F', 'H']
   expanding:  F
9  states visited
['S', 'A', 'C', 'F', 'G']
```

As you can see, this results in visiting significantly fewer states. You can also see the search tree generated by this process in figure 6. In bigger problems, this effect will be amplified hugely, and will make the difference between whether the algorithm can run in a reasonable amount of time, or not.

We can make the same improvement to depth-first search. It still will not guarantee that the shortest path will be found, but will guarantee that we never expand more paths than the actual number of states. The only change is that the new states are added to the beginning of the agenda.

```
def breadthFirstDP(initialState, goalTest, successors):
    agenda = [initialState]
    pathTo = {initialState: [initialState]}
    while agenda != []:
        state = agenda.pop(0)
        newStates = []
        for newState in successors(state):
            if not pathTo.has_key(newState):
                pathTo[newState] = pathTo[state] + [newState]
                if goalTest(newState):
                    return pathTo[newState]
                else:
                    newStates.append(newState)
        agenda = newStates + agenda
    return None
```

Numeric example

Many different kinds of problems can be formulated in terms of finding the shortest path through a space of states. A famous one, which is very appealing to beginning calculus students, is to take a derivative of a complex equation by finding a sequence of operations that takes you from the starting expression to one that doesn't contain any derivative operations. We'll explore a different simple one here:

- The states are the integers.
- The initial state is some integer; let's say 1
- The successors of a state n are: $\{2n, n+1, n-1, n^2, -n\}$
- The goal test is `lambda x:  x == 10`

So, the idea would be to find a short sequence of operations to move from 1 to 10.

First of all, this is a bad domain for applying depth-first search. Why? Because it will go off on a gigantic chain of doubling the starting state, and never find the goal. We can run breadth-first search, though. Without dynamic programming, here is what happens:

```
>>> numberTest(1, 10, breadthFirst)
   expanding:  [1]
   expanding:  [1, 2]
   expanding:  [1, 0]
   expanding:  [1, -1]
   expanding:  [1, 2, 3]
   expanding:  [1, 2, 4]
   expanding:  [1, 2, -2]
   expanding:  [1, 0, -1]
   expanding:  [1, -1, 0]
   expanding:  [1, -1, -2]
   expanding:  [1, 2, 3, 6]
   expanding:  [1, 2, 3, 4]
   expanding:  [1, 2, 3, 9]
39  states visited
[1, 2, 3, 9, 10]
```

We find a nice short path, but visit 52 states. Let's try it with DP:

```
>>> numberTest(1, 10, breadthFirstDP)
   expanding:  1
   expanding:  2
   expanding:  0
   expanding:  -1
   expanding:  3
   expanding:  4
   expanding:  -2
   expanding:  6
   expanding:  9
20  states visited
[1, 2, 3, 9, 10]
```

We find the same path, but with many fewer states.

To experiment with depth-first search, we can make a version of the problem where the state space is limited to the integers in $[-10, 10]$. Here's what happens:

```
>>> numberTestFinite(1, 10, depthFirst, 11)
   expanding:  [1]
   expanding:  [1, 2]
   expanding:  [1, 2, 3]
   expanding:  [1, 2, 3, 6]
   expanding:  [1, 2, 3, 6, 7]
   expanding:  [1, 2, 3, 6, 7, 8]
   expanding:  [1, 2, 3, 6, 7, 8, 9]
19  states visited
[1, 2, 3, 6, 7, 8, 9, 10]
```

We visit fewer states than regular breadth-first search, but find a path with 7 steps, instead of 4. If we use DP with depth-first search, we get:

```
>>> numberTestFinite(1, 10, depthFirstDP, 11)
   expanding:  1
   expanding:  2
   expanding:  3
   expanding:  6
   expanding:  7
   expanding:  8
   expanding:  -8
   expanding:  -9
   expanding:  -10
19  states visited
[1, 2, 3, 6, 7, 8, -8, -9, -10, 10]
```

Ugh. We visit even fewer states, but get an even longer path. Note that the relationship between `depthFirst` and `depthFirstDP` won't always be this way; it can be highly problem dependent.

We can see from trying lots of different searches in this space that (a) the DP makes the search much more efficient and (b) that the difficulty of these search problems varies incredibly widely.

Actions

We have been modeling our state spaces so far by successor functions that return a list of states that are connected to the current state. This is an excellent model for problems like navigating in a map. In other problems, such as the numeric example, above we think of having a discrete set of *actions* that produce resulting states. In that case, we can model the successor functions as producing a list of *(action, state)* pairs. A solution to a problem is then a list of these pairs, taking the first element of each pair gives us a *plan* made up of a list of actions to perform.

For example, for the numeric problem described above, here's a possible successor function.

```
def successors (state):  # state is an integer
    return [('x*2', state*2),
            ('x+1', state+1),
            ('x-1', state-1),
            ('x**2', state**2),
            ('-x', -state)])
```

Note that each element of the resulting list is a tuple, whose first element is the name of the action and whose second element is the new state, which in this case is an integer computed as a function of the input state. In practice, we would want to remove duplicates states from this list of resulting states, but that is an optimization.

In cases such as the city example, we don't think of there being a fixed set of actions. However, we can simply make up arbitrary names for the actions, for example integers. Actions can produce `None` in some states, indicating that they are not defined for that state. In this way, we can model any domain, even those with variable number of successors, as having some fixed number of actions.

The code we will be working with in the lab this week uses this representation. This requires some detailed changes to the search code presented above but the basic algorithm remains the same.

Computational complexity

To finish up this segment, let's consider the computational complexity of these algorithms. As we've already seen, there can be a huge variation in the difficulty of a problem that depends on the exact structure of the graph, and is very hard to quantify in advance. It can sometimes be possible to analyze the average case running time of an algorithm, if you know some kind of distribution over the problems you're likely to encounter. We'll just stick with the traditional *worst-case analysis*, which tries to characterize the Ω running time of the worst possible input for the algorithm.

First, we need to establish a bit of notation. Let

- `b` be be the *branching factor* of the graph; that is, the number of successors a node can have. If we want to be truly worst-case in our analysis, this needs to be the maximum branching factor of the graph.

- d be the *maximum depth* of the graph; that is, the length of the longest path in the graph. In an infinite space, this could be infinite.
- l be the *solution depth* of the problem; that is, the length of the shortest path from the start state to the shallowest goal state.
- n be the *state space size* of the graph; that is the total number of states in the domain.

Depth-first search, in the worst case, will search the entire search tree. It has d levels, each of which has b times as many paths as the previous one. So, there are b^d paths on the d^{th} level. The algorithm might have to visit all of the paths at all of the levels, which is about b^{d+1} states. But the amount of storage it needs for the agenda is only $b \cdot d$.

Breadth-first search, on the other hand, only needs to search as deep as the depth of the best solution. So, it might have to visit as many as b^{l+1} nodes. The amount of storage required for the agenda can be as bad as b^l, too.

So, to be clear, consider the numeric search problem. The branching factor $b = 5$, in the worst case. So, if we have to find a sequence of 10 steps, breadth-first search could potentially require visiting as many as $5^{11} = 48828125$ nodes!

This is all pretty grim. What happens when we consider the DP version of breadth-first search? We can promise that every state in the state space is visited at most once. So, it will visit at most n states. Sometimes n is *much* smaller than b^l (for instance, in a road network). In other cases, it can be much larger (for instance, when you are solving an easy (short solution path) problem embedded in a very large space). Even so, the DP version of the search will visit fewer states, except in the very rare case in which there are never multiple paths to the same state (the graph is actually a tree). For example, in the numeric search problem, the shortest path from 1 to 91 is 9 steps long, but using DP it only requires visiting 1973 states, rather than $5^{10} = 9765625$.