Chapter 8
Long-term decision-making and search

In the lab exercises of this course, we have implemented several brains for our robots. We used wall-following to navigate through the world and we used various linear controllers to drive down the hall and to control the robot head. In the first case, we just wrote a program that we hoped would do a good job. When we were studying linear controllers, 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.

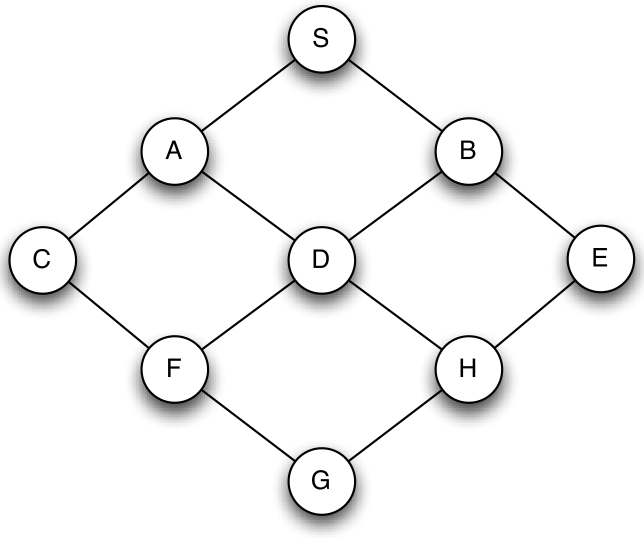
As always, the first step in the process will be to come up with a formal model of a real-world problem that abstracts away the irrelevant detail. So, what, exactly, is a path? The car we're driving will actually follow a trajectory through continuous space(time), but if we tried to plan at that level 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 'on-line', while we're driving, 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. We'll start with the simple criterion of wanting to find a path with the fewest "steps"; in this case, it will be the path that traverses the fewest intersections. Then in **section 8.5** we will generalize our methods to handle problems where different actions have different costs.

One possible algorithm for deciding on the best path through a map of the road intersections in this (very small) world



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.

8.1 State-space search

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

- a (possibly infinite) set 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 procedure that can be applied to any state, and returns `True` if that state can serve as a goal;[47]
- a *successor function*, which takes a state and an action as input, and returns the new state that will result from taking the action in the state; and
- a *legal action list*, which is just a list of actions that can be legally executed in this domain.

The decision about what constitutes an *action* 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

[47] 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.

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 execute the action.)

We can think of this model as specifying a *labeled graph* (in the computer scientist's sense), in which the states are the nodes, action specifies which of the arcs leading out of a node is to be selected, and the successor function specifies the node at the end of each of the arcs.

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

- 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 something like:

```
    lambda x: x == 'H'
```

- The legal actions in this domain are the numbers `0`, `1`, ..., `n-1`, where `n` is the maximum number of successors in any of the states.
- The map can be defined using a dictionary:

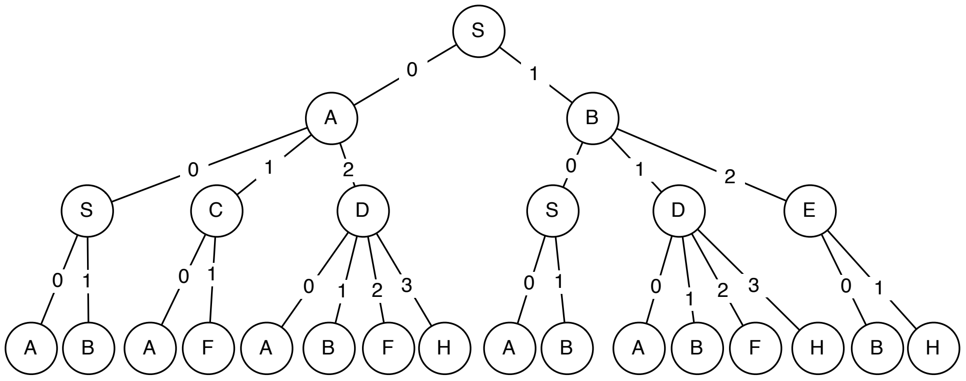
```
map1 = {'S' : ['A', 'B'],
        'A' : ['S', 'C', 'D'],
        'B' : ['S', 'D', 'E'],
        'C' : ['A', 'F'],
        'D' : ['A', 'B', 'F', 'H'],
        'E' : ['B', 'H'],
        'F' : ['C', 'D', 'G'],
        'H' : ['D', 'E', 'G'],
        'G' : ['F', 'H']}
```

  where each key is a state, and its value is a list of states that can be reached from it in one step. Now we can define the successor function as

```
def map1successors(s, a):
    return map1[s][a]
```

  but with an additional test to be sure that if we attempt to take an action that doesn't exist in `s`, it just results in state `s`. So, for example, the successor reached from state `'A'` by taking action `1` is state `'C'`.

We can think of this structure as defining a *search tree*, like this:



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 of length k through the graph.

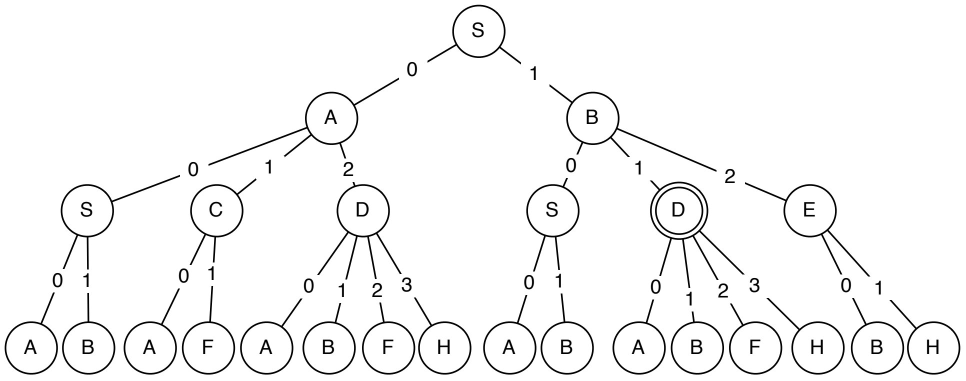
8.1.1 Representing search trees

We will need a way to represent the tree as a Python data structure as we construct it during the search process. We will start by defining a class to represent a *search node*, which is one of the circles in the tree.

Each search node represents:

- The state of the node;
- the action that was taken to arrive at the node; and
- the search node *from which* this node can be reached.

We will call the node from which a node can be reached its *parent* node. So, for example, in the figure below



we will represent the node with double circles around it with its state, `'D'`, the action that reached it, `1`, and its parent node, which is the node labeled `'B'` above it.

Note that *states and nodes are not the same thing!* In this tree, there are many nodes labeled by the same state; they represent different paths to and through the state.

Here is a Python class representing a search node. It's pretty straightforward.

```
class SearchNode:
    def __init__(self, action, state, parent):
        self.state = state
        self.action = action
        self.parent = parent
```

There are a couple of other useful methods for this class. First, the `path` method, returns a list of pairs `(a, s)` corresponding to the path starting at the top (root) of the tree, going down to this node. It works its way up the tree, until it reaches a node whose parent is `None`.

```
    def path(self):
        if self.parent == None:
            return [(self.action, self.state)]
        else:
            return self.parent.path() + [(self.action, self.state)]
```

The path corresponding to our double-circled node is `((None, 'S'), (1, 'B'), (1, 'D'))`.

Another helper method that we will find useful is the `inPath` method, which takes a state, and returns `True` if the state occurs anywhere in the path from the root to the node.

```
    def inPath(self, s):
        if s == self.state:
            return True
        elif self.parent == None:
            return False
        else:
            return self.parent.inPath(s)
```

8.1.2 Basic search algorithm

We'll describe a sequence of search algorithms of increasing sophistication and efficiency. An ideal algorithm will take a problem description as input and return a path from the start to a goal state, if one exists, and return None, if it does not. Some algorithms will not be capable of finding a path in all cases.

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.

Both of the search strategies described above can be implemented using a procedure with this basic structure:

```
def search(initialState, goalTest, actions, successor):
    if goalTest(initialState):
        return [(None, initialState)]
    agenda = EmptyAgenda()
    add(SearchNode(None, initialState, None), agenda)
    while not empty(agenda):
        parent = getElement(agenda)
        for a in actions(parent.state):
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            else:
                add(newN, agenda)
    return None
```

We start by checking to see if the initial state is a goal state. If so, we just return a path consisting of the initial state.

Otherwise, we have real work to do. We make the *root* node of the tree. It has no parent, and there was no action leading to it, so all it needs to specify is its state, which is `initialState`, so it is created with

```
SearchNode(None, initialState, None)
```

During the process of constructing the search tree, we will use a data structure, called an *agenda*, to keep track of which nodes in the partially-constructed tree are on the fringe, ready to be expanded, by adding their children to the tree. We initialize the agenda to contain the root note. 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 node from the agenda (more on how we decide which one to take out in a bit) and *expand it*. To expand a node, we determine which actions can be taken from the state that is stored in the node, and *visit* the *successor* states that can be reached via the actions.

When we visit a state, we make a new search node (`newN`, in the code) that has the node we are in the process of expanding as the parent, and that remembers the state being visited and the action that brought us here from the parent.

Next, we check to see if the new state satisfies the goal test. If it does, we're done! We return the path associated with the new node.

If this state it doesn't satisfy the goal test, then we add the new node 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 what it means to add and extract nodes from the agenda. And, we'll find, that it will do some very stupid things, in its current form.

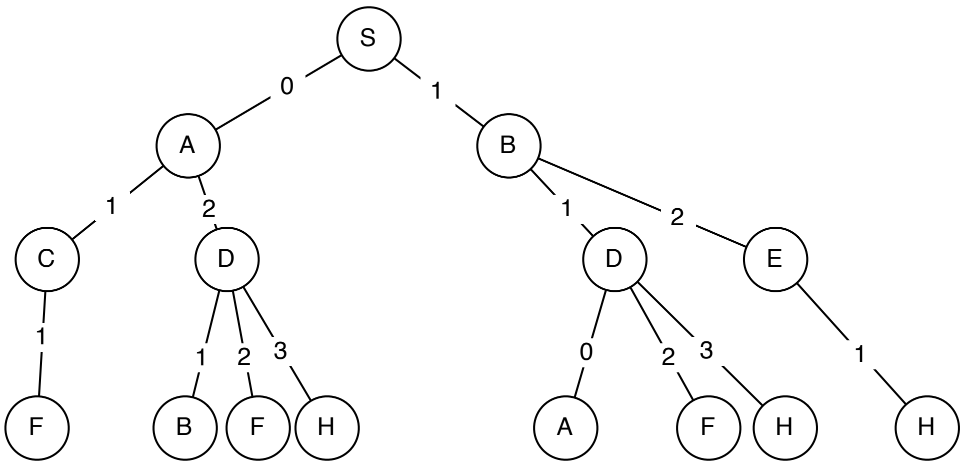
We'll start by curing the stupidities, and then return to the question of how best to select nodes from the agenda.

8.1.3 Basic pruning, or How not to be completely stupid

If you examine the full search tree, you can see that some of the paths it contains 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, to avoid trivial infinite loops, we can adopt the following rule:

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

If we can apply this rule, then we will be able to remove a number of branches from the tree, as shown here:



It is relatively straightforward to modify our code to implement this rule:

```
def search(initialState, goalTest, actions, successor):
    if goalTest(initialState):
        return [(None, initialState)]
    agenda = [SearchNode(None, initialState, None)]
    while agenda != []:
        parent = getElement(agenda)
        for a in actions(parent.state):
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            elif parent.inPath(newS):
                pass
            else:
                add(newN, agenda)
    return None
```

The red text above shows the code we've added to our basic algorithm. It just checks to see whether the current state already exists on the path to the node we're expanding and, if so, it doesn't do anything with it.

The next pruning rule doesn't make a difference in the current domain, but can have a big effect in other domains:

Pruning Rule 2. If there are multiple actions that lead from a state r to a state s, consider only one of them.

To handle this in the code, we have to keep track of which new states we have reached in expanding this node, and if we find another way to reach one of those states, we just ignore it. The changes to the code for implementing this rule are shown in red:

```
def search(initialState, goalTest, actions, successor):
    if goalTest(initialState):
        return [(None, initialState)]
    agenda = [SearchNode(None, initialState, None)]
    while agenda != []:
        parent = getElement(agenda)
        newChildStates = []
        for a in actions(parent.state):
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            elif newS in newChildStates:
                pass
            elif parent.inPath(newS):
                pass
            else:
                newChildStates.append(newS)
                add(newN, agenda)
    return None
```

Each time we pick a new node to expand, we make a new empty list, `newChildStates`, and keep track of all of the new states we have reached from this node.

Now, we have to think about how to extract nodes from the agenda.

8.1.4 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*.

In Python, we can use lists to represent both stacks and queues. If `data` is a list, then `data.pop(0)` removes the first element from the list and returns it, and `data.pop()` removes the last element and returns it.

Here is a class representing stacks as lists. It always adds new elements to the end of the list, and pops items off of the same end, ensuring that the most recent items get popped off first.

```
class Stack:
    def __init__(self):
        self.data = []
    def push(self, item):
        self.data.append(item)
    def pop(self):
        return self.data.pop()
    def isEmpty(self):
        return self.data is []
```

Here is a class representing stacks as lists. It always adds new elements to the end of the list, and pops items off of the front, ensuring that the oldest items get popped off first.

```
class Queue:
    def __init__(self):
        self.data = []
    def push(self, item):
        self.data.append(item)
    def pop(self):
        return self.data.pop(0)
    def isEmpty(self):
        return self.data is []
```

We will use stacks and queues to implement our search algorithms.

8.1.5 Depth-First Search

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, we always expand the node we most recently put into the agenda.

The code listing below shows our implementation of depth-first search.

```
def depthFirstSearch(initialState, goalTest, actions, successor):
    agenda = Stack()
    if goalTest(initialState):
        return [(None, initialState)]
    agenda.push(SearchNode(None, initialState, None))
    while not agenda.isEmpty()
        parent = agenda.pop()
        newChildStates = []
        for a in actions(parent.state):
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            elif newS in newChildStates:
                pass
            elif parent.inPath(newS):
                pass
            else:
                newChildStates.append(newS)
                agenda.push(newN)
    return None
```

You can see several operations on the agenda. We:

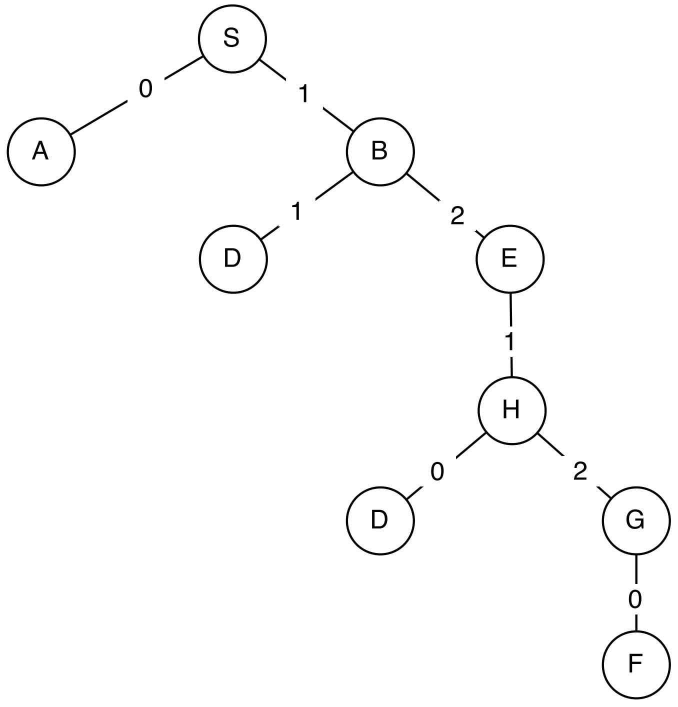
- Create an empty `Stack` instance, and let that be the agenda.
- Push the initial node onto the agenda.
- Test to see if the agenda is empty.
- Pop the node to be expanded off of the agenda.
- Push newly visited nodes onto the agenda.

Because the agenda is an instance of the `Stack` class, subsequent operations on the agenda ensure that it will act like a stack, and guarantee that children of the most recently expanded node will be chosen for expansion next.

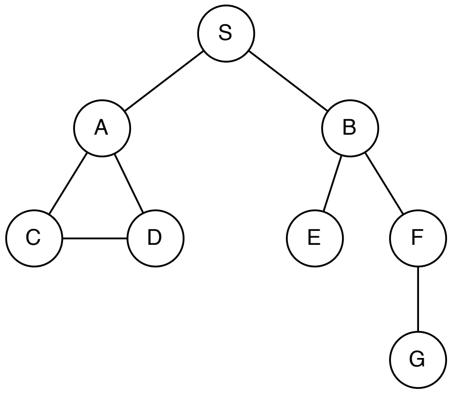
So, let's see how this search method behaves on our city map, with start state S and goal state F. Here is a trace of the algorithm (you can get this in the code we distribute by setting `verbose = True` before you run it.)

```
depthFirst('S', lambda x: x == 'F', map1LegalActions, map1successors)
agenda:  Stack([S])
   expanding:  S
agenda:  Stack([S-0->A, S-1->B])
   expanding:  S-1->B
agenda:  Stack([S-0->A, S-1->B-1->D, S-1->B-2->E])
   expanding:  S-1->B-2->E
agenda:  Stack([S-0->A, S-1->B-1->D, S-1->B-2->E-1->H])
   expanding:  S-1->B-2->E-1->H
agenda:  Stack([S-0->A, S-1->B-1->D, S-1->B-2->E-1->H-0->D, S-1->B-2->E-1->H-2->G])
   expanding:  S-1->B-2->E-1->H-2->G
8  states visited
[(None, 'S'), (1, 'B'), (2, 'E'), (1, 'H'), (2, 'G'), (0, 'F')]
```

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. Here is the search tree generated during the depth-first search process.



Here is 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 D this time):

```
depthFirst('S', lambda x: x == 'D', map2LegalActions, map2successors)
agenda:  Stack([S])
   expanding:  S
agenda:  Stack([S-0->A, S-1->B])
   expanding:  S-1->B
agenda:  Stack([S-0->A, S-1->B-1->E, S-1->B-2->F])
   expanding:  S-1->B-2->F
agenda:  Stack([S-0->A, S-1->B-1->E, S-1->B-2->F-1->G])
   expanding:  S-1->B-2->F-1->G
agenda:  Stack([S-0->A, S-1->B-1->E])
   expanding:  S-1->B-1->E
agenda:  Stack([S-0->A])
   expanding:  S-0->A
7  states visited
[(None, 'S'), (0, 'A'), (2, 'D')]
```

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 will run forever if we don't apply pruning rule 1, potentially going back and forth from one state to another, forever.
- 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; we'll see an example of this in the last section).
- 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 in section 8.4.)

8.1.6 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. All we have to do is change the agenda to be a queue instead of a stack, and everything else stays the same, in the code.

```
def breadthFirstSearch(initialState, goalTest, actions, successor):
    agenda = Queue()
    if goalTest(initialState):
        return [(None, initialState)]
    agenda.push(SearchNode(None, initialState, None))
    while not agenda.isEmpty():
        parent = agenda.pop()
        newChildStates = []
        for a in actions(parent.state):
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            elif newS in newChildStates:
                pass
            elif parent.inPath(newS):
                pass
            else:
                newChildStates.append(newS)
                agenda.push(newN)
    return None
```

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

```
>>> breadthFirst('S', lambda x: x == 'F', map1LegalActions, map1successors)
agenda:  Queue([S])
   expanding:  S
agenda:  Queue([S-0->A, S-1->B])
   expanding:  S-0->A
agenda:  Queue([S-1->B, S-0->A-1->C, S-0->A-2->D])
   expanding:  S-1->B
agenda:  Queue([S-0->A-1->C, S-0->A-2->D, S-1->B-1->D, S-1->B-2->E])
   expanding:  S-0->A-1->C
7  states visited
[(None, 'S'), (0, 'A'), (1, 'C'), (1, 'F')]
```

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

Here are some important properties of breadth-first search:

- Always returns a shortest (least number of steps) path to a goal state, if a goal state exists in the set of states reachable from the start state.
- It may run forever if there is no solution and the domain is infinite.
- It requires more space than depth-first search.

8.1.7 Dynamic programming

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

```
>>> breadthFirst('S', lambda x: x == 'G', map1LegalActions, map1successors)
agenda:  Queue([S])
   expanding:  S
agenda:  Queue([S-0->A, S-1->B])
```

```
   expanding:  S-0->A
agenda:  Queue([S-1->B, S-0->A-1->C, S-0->A-2->D])
   expanding:  S-1->B
agenda:  Queue([S-0->A-1->C, S-0->A-2->D, S-1->B-1->D, S-1->B-2->E])
   expanding:  S-0->A-1->C
agenda:  Queue([S-0->A-2->D, S-1->B-1->D, S-1->B-2->E, S-0->A-1->C-1->F])
   expanding:  S-0->A-2->D
agenda:  Queue([S-1->B-1->D, S-1->B-2->E, S-0->A-1->C-1->F, S-0->A-2->D-1->B, S-0->A-2->D-2-
>F, S-0->A-2->D-3->H])
   expanding:  S-1->B-1->D
agenda:  Queue([S-1->B-2->E, S-0->A-1->C-1->F, S-0->A-2->D-1->B, S-0->A-2->D-2->F, S-0->A-2-
>D-3->H, S-1->B-1->D-0->A, S-1->B-1->D-2->F, S-1->B-1->D-3->H])
   expanding:  S-1->B-2->E
agenda:  Queue([S-0->A-1->C-1->F, S-0->A-2->D-1->B, S-0->A-2->D-2->F, S-0->A-2->D-3->H,
S-1->B-1->D-0->A, S-1->B-1->D-2->F, S-1->B-1->D-3->H, S-1->B-2->E-1->H])
   expanding:  S-0->A-1->C-1->F
16  states visited
[(None, 'S'), (0, 'A'), (1, 'C'), (1, 'F'), (2, 'G')]
```

The first thing that is notable about this trace is that it ends up visiting 16 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 `S-0->A-2->D` and `S-1->B-1->D` in the agenda. Even worse, it has both `S-0->A` and `S-1->B-1->D-0->A` 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 3. 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. So, we'll keep track of the states that we have visited so far, by using a dictionary, called `visited` that has an entry for every state we have visited.[48] Then, if we are considering adding a new node to the tree that goes to a state we have already visited, we just ignore it. This test can take the place of the test we used to have for pruning rule 1; it's clear that if the path we are considering already contains this state, then the state has been visited before. Finally, we have to remember, whenever we add a node to the agenda, to add the corresponding state to the visited list.

Here is our breadth-first search code, modified to take advantage of dynamic programming.

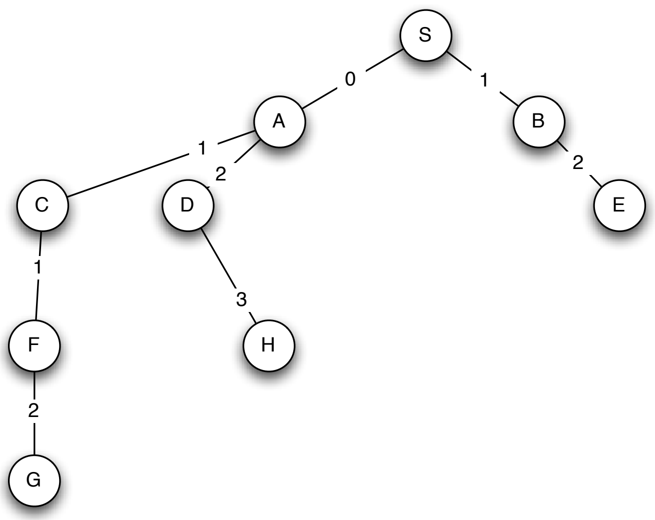
[48] An alternative representation would be just to keep a Python `set` of visited nodes.

```
def breadthFirstDP(initialState, goalTest, actions, successor):
    agenda = Queue()
    if goalTest(initialState):
        return [(None, initialState)]
    agenda.push(SearchNode(None, initialState, None))
    visited = {initialState: True}
    while not agenda.isEmpty():
        parent = agenda.pop()
        for a in actions(parent.state):
            newS = successor(parent.state, a)
            newN = SearchNode(a, newS, parent)
            if goalTest(newS):
                return newN.path()
            elif visited.has_key(newS):
                pass
            else:
                visited[newS] = True:
                agenda.push(newN)
    return None
```

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

```
>>> breadthFirstDP('S', lambda x: x == 'G', map1LegalActions, map1successors)
agenda:  Queue([S])
   expanding:  S
agenda:  Queue([S-0->A, S-1->B])
   expanding:  S-0->A
agenda:  Queue([S-1->B, S-0->A-1->C, S-0->A-2->D])
   expanding:  S-1->B
agenda:  Queue([S-0->A-1->C, S-0->A-2->D, S-1->B-2->E])
   expanding:  S-0->A-1->C
agenda:  Queue([S-0->A-2->D, S-1->B-2->E, S-0->A-1->C-1->F])
   expanding:  S-0->A-2->D
agenda:  Queue([S-1->B-2->E, S-0->A-1->C-1->F, S-0->A-2->D-3->H])
   expanding:  S-1->B-2->E
agenda:  Queue([S-0->A-1->C-1->F, S-0->A-2->D-3->H])
   expanding:  S-0->A-1->C-1->F
8  states visited
[(None, 'S'), (0, 'A'), (1, 'C'), (1, 'F'), (2, 'G')]
```

As you can see, this results in visiting significantly fewer states. Here is the tree generated by this process:



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, and not.

We can make the same improvement to depth-first search; we just need to use a stack instead of a queue in the algorithm above. It still will not guarantee that the shortest path will be found, but will guarantee that we never visit more paths than the actual number of states. The only change to breadth-first search with dynamic programming is that the new states are added to the beginning of the agenda.

8.1.8 Configurable search code

Because all of our search algorithms (breadth-first and depth-first, with and without dynamic programming) are all so similar, and we don't like to repeat code, we provide (in file `search.py`) a single, configurable search procedure. It also prints out some information as it goes, if you have the `verbose` or `somewhatVerbose` variables set to `True`, and has a limit on the maximum number of nodes it will expand (to keep from going into an infinite loop).

```
def search(initialState, goalTest, actions, successor,
           depthFirst = False, DP = True, maxNodes = 10000):
    if depthFirst:
        agenda = Stack()
    else:
        agenda = Queue()

    startNode = SearchNode(None, initialState, None)
    if goalTest(initialState):
        return startNode.path()
    agenda.push(startNode)
    if DP: visited = {initialState: True}
    count = 1
    while not agenda.isEmpty() and maxNodes > count:
        n = agenda.pop()
        newStates = []
        for a in actions:
            newS = successor(n.state, a)
            newN = SearchNode(a, newS, n)
            if goalTest(newS):
                return newN.path()
            elif newS in newStates:
                pass
            elif ((not DP) and n.inPath(newS)) or \
                  (DP and visited.has_key(newS)):
                pass
            else:
                count += 1
                if DP: visited[newS] = True
                newStates.append(newS)
                agenda.push(newN)
    return None
```

8.2 Connection to state machines

We can use state machines as a convenient representation of state-space search problems. Given a state machine, in its initial state, what sequence of inputs can we feed to it to get it to enter a done state? This is a search problem, analogous to determining the sequence of actions that can be taken to reach a goal state.

The `getNextValues` method of a state machine can serve as the `successor` function in a search (the inputs to the machine are the actions). Our standard machines do not have a notion of legal actions; but we will add an attribute called `legalInputs`, which is a list of values that are legal inputs to the machine (these are the actions, from the planning perspective) to machines that we want to use with a search.

The `startState` attribute can serve as the initial state in the search and the `done` method of the machine can serve as the goal test function.

Then, we can plan a sequence of actions to go from the start state to one of the done states using this function, where `smToSearch` is an instance of `sm.SM`.

```
def smSearch(smToSearch, initialState = None, goalTest = None, maxNodes = 10000,
             depthFirst = False, DP = True):
   if initialState == None:
       initialState = smToSearch.startState
   if goalTest == None:
       goalTest = smToSearch.done
   return search(initialState, goalTest, smToSearch.legalInputs,
                 # This returns the next state
                 lambda s, a: smToSearch.getNextValues(s, a)[0],
                 maxNodes = maxNodes,
                 depthFirst=depthFirst, DP=DP)
```

It is mostly clerical: it allows us to specify a different initial state or goal test if we want to, and it extracts the appropriate functions out of the state machine and passes them into the search procedure. Also, because `getNextValues` returns both a state and an output, we have to wrap it inside a function that just selects out the next state and returns it.

8.3 Numeric search domain

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 legal actions are to apply the following operations: $\{2n, n+1, n-1, n^2, -n\}$.
- 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.

Here it is, formalized as state machine in Python:

```
class NumberTestSM(sm.SM):
    startState = 1
    legalInputs = ['x*2', 'x+1', 'x-1', 'x**2', '-x']
    def __init__(self, goal):
        self.goal = goal
    def nextState(self, state, action):
        if action == 'x*2':
            return state*2
        elif action == 'x+1':
            return state+1
        elif action == 'x-1':
            return state-1
        elif action == 'x**2':
            return state**2
        elif action == '-x':
            return -state
    def getNextValues(self, state, action):
        nextState = self.nextState(state, action)
        return (nextState, nextState)
    def done(self, state):
        return state == self.goal
```

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 (we have set `verbose = False` and `somewhatVerbose = True` in the search file):

```
>>> smSearch(NumberTestSM(10), initialState = 1, depthFirst = False, DP = False)
   expanding:  1
   expanding:  1-x*2->2
   expanding:  1-x-1->0
   expanding:  1--x->-1
   expanding:  1-x*2->2-x*2->4
   expanding:  1-x*2->2-x+1->3
   expanding:  1-x*2->2--x->-2
   expanding:  1-x-1->0-x-1->-1
   expanding:  1--x->-1-x*2->-2
   expanding:  1--x->-1-x+1->0
   expanding:  1-x*2->2-x*2->4-x*2->8
   expanding:  1-x*2->2-x*2->4-x+1->5
33  states visited
[(None, 1), ('x*2', 2), ('x*2', 4), ('x+1', 5), ('x*2', 10)]
```

We find a nice short path, but visit 33 states. Let's try it with DP:

```
>>> smSearch(NumberTestSM(10), initialState = 1, depthFirst = False, DP = True)
   expanding:  1
   expanding:  1-x*2->2
   expanding:  1-x-1->0
   expanding:  1--x->-1
   expanding:  1-x*2->2-x*2->4
   expanding:  1-x*2->2-x+1->3
   expanding:  1-x*2->2--x->-2
   expanding:  1-x*2->2-x*2->4-x*2->8
   expanding:  1-x*2->2-x*2->4-x+1->5
17  states visited
[(None, 1), ('x*2', 2), ('x*2', 4), ('x+1', 5), ('x*2', 10)]
```

We find the same path, but visit noticeably fewer states. If we change the goal to 27, we find that we visit 564 states without DP and 119, with. If the goal is 1027, then we visit 12710 states without DP and 1150 with DP, which is getting to be a very big difference.

To experiment with depth-first search, we can make a version of the problem where the state space is limited to the integers in some range. We do this by making a subclass of the `NumberTestSM`, which remembers the maximum legal value, and uses it to restrict the set of legal inputs for a state (any input that would cause the successor state to go out of bounds just results in staying at the same state, and it will be pruned.)

```
class NumberTestFiniteSM(NumberTestSM):
    def __init__(self, goal, maxVal):
        self.goal = goal
        self.maxVal = maxVal
    def getNextValues(self, state, action):
        nextState = self.nextState(state, action)
        if abs(nextState) < self.maxVal:
            return (nextState, nextState)
        else:
            return (state, state)
```

Here's what happens if we give it a range of -20 to +20 to work in:

```
>>> smSearch(NumberTestFiniteSM(10, 20), initialState = 1, depthFirst = True,
             DP = False)
   expanding:  1
   expanding:  1--x->-1
   expanding:  1--x->-1-x+1->0
   expanding:  1--x->-1-x*2->-2
   expanding:  1--x->-1-x*2->-2--x->2
   expanding:  1--x->-1-x*2->-2--x->2-x+1->3
   expanding:  1--x->-1-x*2->-2--x->2-x+1->3--x->-3
   expanding:  1--x->-1-x*2->-2--x->2-x+1->3--x->-3-x**2->9
20  states visited
[(None, 1), ('-x', -1), ('x*2', -2), ('-x', 2), ('x+1', 3), ('-x', -3), ('x**2', 9), ('x+1',
10)]
```

We generate a much longer path!

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.

8.4 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 approximate running time of the worst possible input for the algorithm.

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

- b 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!

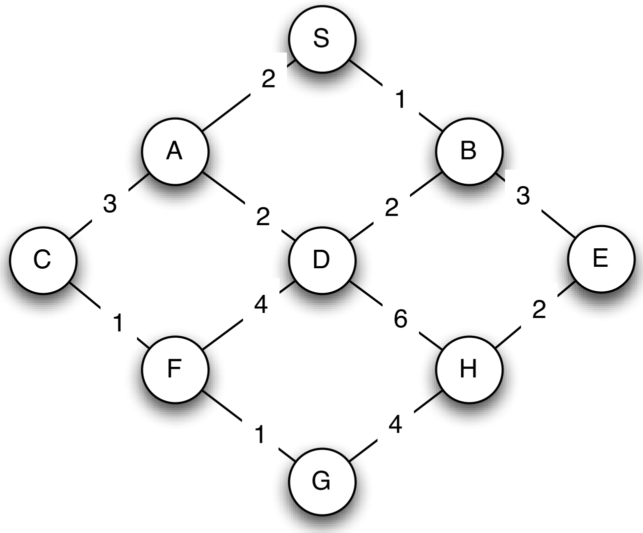
This is all pretty grim. What happens when we consider the DP version of breadth-first search? We can promise that every state in the state space is visited at most once. So, it will visit at most n states. Sometimes n is *much* smaller than b^l (for instance, in a road network). In other cases, it can 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 $5^{10} = 9765625$.

8.5 Uniform cost search

In many cases, the arcs in our graph will actually have different costs. In a road network, we would really like to find the shortest path in miles (or in time to traverse), and different road segments have different lengths and times. To handle a problem like this, we need to extend our representation of search problems, and add a new algorithm to our repertoire.

We will extend our notion of a successor function, so that it takes a state and an action, as before, but now it returns a pair (newS, cost), which represents the resulting state, as well as the cost that is incurred in traversing that arc. To guarantee that all of our algorithms are well behaved, we will require that all costs be positive (not zero or negative).

Here is our original city map, now with distances associated with the roads between the cities.



We can describe it in a dictionary, this time associating a cost with each resulting state, as follows:

```
map1dist = {'S' : [('A', 2), ('B', 1)],
            'A' : [('S', 2), ('C', 3), ('D', 2)],
            'B' : [('S', 1), ('D', 2), ('E', 3)],
            'C' : [('A', 3), ('F', 1)],
            'D' : [('A', 2), ('B', 2), ('F', 4), ('H', 6)],
            'E' : [('B', 3), ('H', 2)],
            'F' : [('C', 1), ('D', 4), ('G', 1)],
            'H' : [('D', 6), ('E', 2), ('G', 4)],
            'G' : [('F', 1), ('H', 4)]}
```

When we studied breadth-first search, we argued that it found the shortest path, in the sense of having the fewest nodes, by seeing that it investigate all of the length 1 paths, then all of the length 2 paths, etc. This orderly enumeration of the paths guaranteed that when we first encountered a goal state, it would be via a shortest path. The idea behind *uniform cost search* is basically the same: we are going to investigate paths through the graph, in the order of the sum of the costs on their arcs. If we do this, we guarantee that the first time we extract a node with a given state from the agenda, it will be via a shortest path, and so the first time we extract a node with a goal state from the agenda, it will be an optimal solution to our problem.

Priority Queue

Just as we used a stack to manage the agenda for depth-first search and a queue to manage the agenda for bread-first search, we will need to introduce a new data structure, called a *priority queue* to manage the agenda for uniform-cost search. A priority queue is a data structure with the same basic operations as stacks and queues, with two differences:

- Items are pushed into a priority queue with a numeric score, called a *cost*.
- When it is time to pop an item, the item in the priority queue with the least *cost* is returned and removed from the priority queue.

There are many interesting ways to implement priority queues so that they are very computationally efficient. Here, we show a very simple implementation that simply walks down the entire contents of the priority queue to find the least-cost item for a pop operation. Its `data` attribute consists of a list of `(cost, item)` pairs. It calls the `argmaxIndex` procedure from our utility package, which takes a list of items and a scoring function, and returns a pair consisting of the index of the list with the highest scoring item, and the score of that item. Note that, because `argmaxIndex` finds the item with the *highest score*, and we want to extract the item with the *least cost*, our scoring function is the *negative* of the cost.

```
class PQ:
    def __init__(self):
        self.data = []
    def push(self, item, cost):
        self.data.append((cost, item))
    def pop(self):
        (index, cost) = util.argmaxIndex(self.data, lambda (c, x): -c)
        return self.data.pop(index)[1] # just return the data item
    def isEmpty(self):
        return self.data is []
```

UC Search

Now, we're ready to study the uniform-cost search algorithm itself. We will start with a simple version that doesn't do any pruning or dynamic programming, and then add those features back in later. First, we have to extend our definition of a `SearchNode`, to incorporate costs. So, when we create a new search node, we pass in an additional parameter `actionCost`, which represents the cost just for the action that moves from the parent node to the state. Then, we create an attribute `self.cost`, which encodes the cost of this entire path, from the starting state to the last state in the path. We compute it by adding the path cost of the parent to the cost of this last action, as shown by the red text below.

```
class SearchNode:
    def __init__(self, action, state, parent, actionCost):
        self.state = state
        self.action = action
        self.parent = parent
        if self.parent:
            self.cost = self.parent.cost + actionCost
        else:
            self.cost = actionCost
```

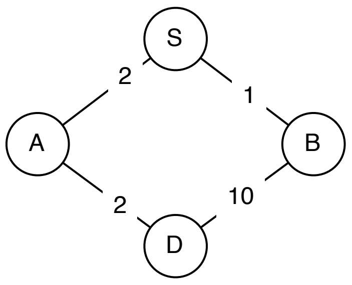
Now, here is the search algorithm. It looks a lot like our standard search algorithm, but there are two important differences:

- The agenda is a priority queue.
- Instead of testing for a goal state when we put an element *into* the agenda, as we did in breadth-first search, we test for a goal state when we take an element *out of* the agenda. This is crucial, to ensure that we actually find the shortest path to a goal state.

```
def ucSearch(initialState, goalTest, actions, successor):
    startNode = SearchNode(None, initialState, None, 0)
    if goalTest(initialState):
        return startNode.path()
    agenda = PQ()
    agenda.push(startNode, 0)
    while not agenda.isEmpty():
        n = agenda.pop()
        if goalTest(n.state):
            return n.path()
        for a in actions:
            (newS, cost) = successor(n.state, a)
            if not n.inPath(newS):
                newN = SearchNode(a, newS, n, cost)
                agenda.push(newN, newN.cost)
    return None
```

Example

Consider the following simple graph:



Let’s simulate the uniform-cost search algorithm, and see what happens when we try to start from S and go to D:

- The agenda is initialized to contain the starting node. The agenda is shown as a list of cost, node pairs.

```
agenda:  PQ([(0, S)])
```

- The least-cost node, S, is extracted and expanded, adding two new nodes to the agenda. The notation S-0->A means that the path starts in state S, takes action 0, and goes to state A.

```
    0 :   expanding:  S
agenda:  PQ([(2, S-0->A), (1, S-1->B)])
```

- The least-cost node, S-1->B, is extracted, and expanded, adding one new node to the agenda. Note, that, at this point, we have discovered a path to the goal: S-1->B-1->D is a path to the goal, with cost 11. But we cannot be sure that it is the shortest path to the goal, so we simply put it into the agenda, and wait to see if it gets extracted before any other path to a goal state.

```
    1 :   expanding:  S-1->B
agenda:  PQ([(2, S-0->A), (11, S-1->B-1->D)])
```

- The least-cost node, S-0->A is extracted, and expanded, adding one new node to the agenda. At this point, we have two different paths to the goal in the agenda.

```
    2 :   expanding:  S-0->A
agenda:  PQ([(11, S-1->B-1->D), (4, S-0->A-1->D)])
```

- Finally, the least-cost node, S-0->A-1->D is extracted. It is a path to the goal, so it is returned as the solution.

```
5 states visited; Solution cost: 4
[(None, 'S'), (0, 'A'), (1, 'D')]
```

Dynamic programming

Now, we just need to add dynamic programming back in, but we have to do it slightly differently. We promise that, once we have *expanded* a node, that is, taken it out of the agenda, then we have found the shortest path to that state, and we need not consider any further paths that go through that state. So, instead of remembering which nodes we have visited (put onto the agenda) we will remember nodes we have *expanded* (gotten out of the agenda), and never visit or expand a node that has already been expanded. In the code below, the first test ensures that we don't expand a node that goes to a state that we have already found the shortest path to, and the second test ensures that we don't put any additional paths to such a state into the agenda.

```
def ucSearch(initialState, goalTest, actions, successor):
    startNode = SearchNode(None, initialState, None, 0)
    if goalTest(initialState):
        return startNode.path()
    agenda = PQ()
    agenda.push(startNode, 0)
    expanded = { }
    while not agenda.isEmpty():
        n = agenda.pop()
        if not expanded.has_key(n.state):
            expanded[n.state] = True
            if goalTest(n.state):
                return n.path()
            for a in actions:
                (newS, cost) = successor(n.state, a)
                if not expanded.has_key(newS):
                    newN = SearchNode(a, newS, n, cost)
                    agenda.push(newN, newN.cost)
    return None
```

Here is the result of running this version of uniform cost search on our bigger city graph with distances:

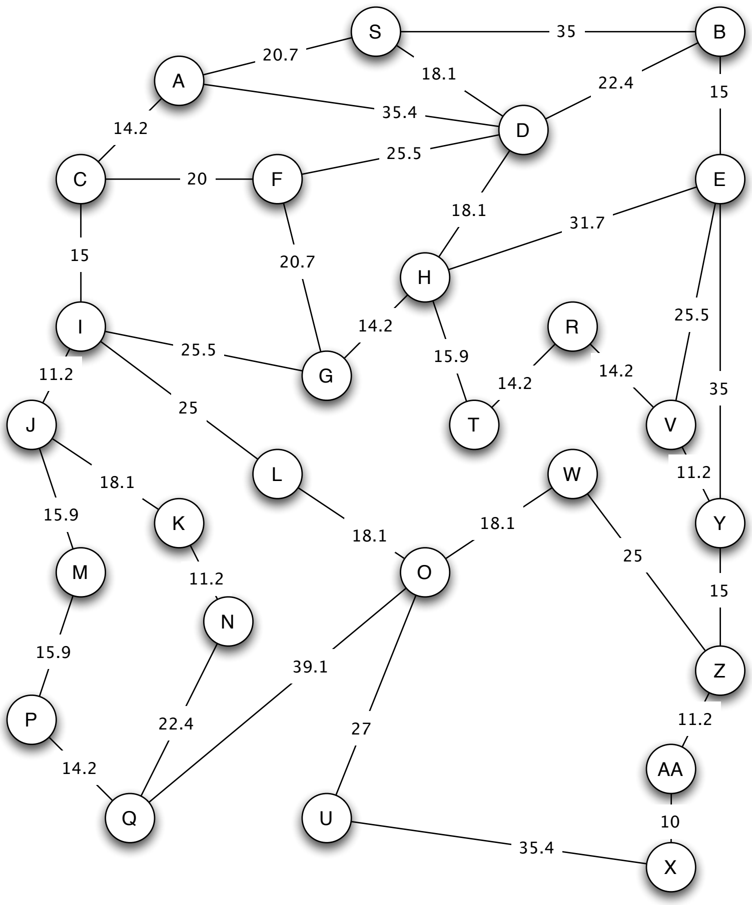
```
mapDistTest(map1dist,'S', 'G')
agenda:  PQ([(0, S)])
    0 :   expanding:  S
agenda:  PQ([(2, S-0->A), (1, S-1->B)])
    1 :   expanding:  S-1->B
agenda:  PQ([(2, S-0->A), (3, S-1->B-1->D), (4, S-1->B-2->E)])
    2 :   expanding:  S-0->A
agenda:  PQ([(3, S-1->B-1->D), (4, S-1->B-2->E), (5, S-0->A-1->C), (4, S-0->A-2->D)])
    3 :   expanding:  S-1->B-1->D
agenda:  PQ([(4, S-1->B-2->E), (5, S-0->A-1->C), (4, S-0->A-2->D), (7, S-1->B-1->D-2->F),
(9, S-1->B-1->D-3->H)])
    4 :   expanding:  S-1->B-2->E
agenda:  PQ([(5, S-0->A-1->C), (4, S-0->A-2->D), (7, S-1->B-1->D-2->F), (9, S-1->B-1->D-3-
>H), (6, S-1->B-2->E-1->H)])
agenda:  PQ([(5, S-0->A-1->C), (7, S-1->B-1->D-2->F), (9, S-1->B-1->D-3->H), (6, S-1->B-2-
>E-1->H)])
    5 :   expanding:  S-0->A-1->C
agenda:  PQ([(7, S-1->B-1->D-2->F), (9, S-1->B-1->D-3->H), (6, S-1->B-2->E-1->H), (6, S-0-
>A-1->C-1->F)])
    6 :   expanding:  S-1->B-2->E-1->H
agenda:  PQ([(7, S-1->B-1->D-2->F), (9, S-1->B-1->D-3->H), (6, S-0->A-1->C-1->F), (10, S-1-
>B-2->E-1->H-2->G)])
    6 :   expanding:  S-0->A-1->C-1->F
agenda:  PQ([(7, S-1->B-1->D-2->F), (9, S-1->B-1->D-3->H), (10, S-1->B-2->E-1->H-2->G), (7,
S-0->A-1->C-1->F-2->G)])
agenda:  PQ([(9, S-1->B-1->D-3->H), (10, S-1->B-2->E-1->H-2->G), (7, S-0->A-1->C-1->F-2-
>G)])
13 states visited; Solution cost: 7
[(None, 'S'), (0, 'A'), (1, 'C'), (1, 'F'), (2, 'G')]
```

8.5.1 Connection to state machines

When we use a state machine to specify a domain for a cost-based search, we only need to make a small change: the `getNextValues` method of a state machine can still serve as the `successor` function in a search (the inputs to the machine are the actions). We usually think of `getNextVal-ues` as returning the next state and the output: now, we will modify that interpretation slightly, and think of it as returning the next state and the incremental cost of taking the action that transitions to that next state. This has the same form as the the `ucSearch.search` procedure expects a successor function to have, so we don't need to change anything about the `smSearch` procedure we have already defined.

8.6 Search with heuristics

Ultimately, we'd like to be able to solve huge state-space search problems, such as those solved by a GPS that can plan long routes through a complex road network. We'll have to add something to uniform-cost search to solve such problems efficiently. Let's consider the city below, where the actual distances between the intersections are shown on the arcs:



If we use uniform cost search to find a path from `G` to `X`, we expand states in the following order (the number at the beginning of each line is the length of the path from `G` to the state at the end of the path:

```
>>> bigTest('G', 'X')
    0 :   expanding:  G
    14.2 :   expanding:  G-2->H
    20.7 :   expanding:  G-1->F
    25.5 :   expanding:  G-0->I
    30.1 :   expanding:  G-2->H-2->T
    32.3 :   expanding:  G-2->H-0->D
    36.7 :   expanding:  G-0->I-3->J
    40.5 :   expanding:  G-0->I-0->C
    44.3 :   expanding:  G-2->H-2->T-1->R
    45.9 :   expanding:  G-2->H-1->E
    50.4 :   expanding:  G-2->H-0->D-0->S
    50.5 :   expanding:  G-0->I-2->L
    52.6 :   expanding:  G-0->I-3->J-2->M
    54.7 :   expanding:  G-2->H-0->D-2->B
    54.7 :   expanding:  G-0->I-0->C-0->A
    54.8 :   expanding:  G-0->I-3->J-1->K
    58.5 :   expanding:  G-2->H-2->T-1->R-1->V
    66.0 :   expanding:  G-0->I-3->J-1->K-1->N
    68.5 :   expanding:  G-0->I-3->J-2->M-1->P
    68.6 :   expanding:  G-0->I-2->L-1->O
    69.7 :   expanding:  G-2->H-2->T-1->R-1->V-2->Y
    82.7 :   expanding:  G-0->I-3->J-2->M-1->P-1->Q
    84.7 :   expanding:  G-2->H-2->T-1->R-1->V-2->Y-2->Z
    86.7 :   expanding:  G-0->I-2->L-1->O-1->W
    95.6 :   expanding:  G-0->I-2->L-1->O-2->U
    95.9 :   expanding:  G-2->H-2->T-1->R-1->V-2->Y-2->Z-2->AA
39 nodes visited; 27 states expanded; solution cost: 105.9
[(None, 'G'), (2, 'H'), (2, 'T'), (1, 'R'), (1, 'V'), (2, 'Y'), (2, 'Z'), (2, 'AA'), (1,
'X')]
```

This search process works its way out, radially, from `G`, expanding nodes in contours of increasing path length. That means that, by the time the search expands node `X`, it has expanded every single node. This seems kind of silly: if you were looking for a good route from `G` to `X`, it's unlikely that states like `S` and `B` would ever come into consideration.

Heuristics

What is it about state `B` that makes it seem so irrelevant? Clearly, it's far away from where we want to go. We can incorporate this idea into our search algorithm using something called a *heuristic function*. A heuristic function takes a state as an argument and returns a numeric estimate of the total cost that it will take to reach the goal from there. We can modify our search algorithm to be biased toward states that are closer to the goal, in the sense that the heuristic function has a smaller value on them.

In a path-planning domain, such as our example, a reasonable heuristic is the actual Euclidean distance between the current state and the goal state; this makes sense because the states in this domain are actual locations on a map.

A*

If we modify the uniform-cost search algorithm to take advantage of a heuristic function, we get an algorithm called A^* (pronounced 'a star'). It is given below, with the differences highlighted in red. The *only* difference is that, when we insert a node into the priority queue, we do so with a cost that is `newN.cost + heuristic(newS)`. That is, it is the sum of the actual cost of the path from the start state to the current state, and the estimated cost to go from the current state to the goal.

```
def ucSearch(initialState, goalTest, actions, successor, heuristic):
    startNode = SearchNode(None, initialState, None, 0)
    if goalTest(initialState):
        return startNode.path()
    agenda = PQ()
    agenda.push(startNode, 0)
    expanded = { }
    while not agenda.isEmpty():
        n = agenda.pop()
        if not expanded.has_key(n.state):
            expanded[n.state] = True
            if goalTest(n.state):
                return n.path()
            for a in actions:
                (newS, cost) = successor(n.state, a)
                if not expanded.has_key(newS):
                    newN = SearchNode(a, newS, n, cost)
                    agenda.push(newN, newN.cost + heuristic(newS))
    return None
```

Example

Now, we can try to search in the big map for a path from G to X, using, as our heuristic function, the distance between the state of interest and X. Here is a trace of what happens (with the numbers rounded to increase readability):

- We get the start node out of the agenda, and add its children. Note that the costs are the actual path cost *plus* the heuristic estimate.

```
    0 :   expanding:  G
agenda:  PQ([(107, G-0->I), (101, G-1->F), (79, G-2->H)])
```

- The least cost path is G-2->H, so we extract it, and add its successors.

```
    14.2 :   expanding:  G-2->H
agenda:  PQ([(107, G-0->I), (101, G-1->F), (109, G-2->H-0->D), (116, G-2->H-1->E), (79,
G-2->H-2->T)])
```

- Now, we can see the heuristic function really having an effect. The path G-2->H-2->T has length 30.1, and the path G-1-F has length 20.7. But when we add in the heuristic cost estimates, the path to T has a lower cost, because it seems to be going in the right direction. Thus, we select G-2->H-2->T to expand next:

```
    30.1 :   expanding:  G-2->H-2->T
agenda:  PQ([(107, G-0->I), (101, G-1->F), (109, G-2->H-0->D), (116, G-2->H-1->E), (100,
G-2->H-2->T-1->R)])
```

- Now the path G-2->H-2->T-1->R looks best, so we expand it.

```
    44.3 :   expanding:  G-2->H-2->T-1->R
agenda:  PQ([(107, G-0->I), (101, G-1->F), (109, G-2->H-0->D), (116, G-2->H-1->E),
(103.5, G-2->H-2->T-1->R-1->V)])
```

- Here, something interesting happens. The node with the least estimated cost is G-1->F. It's going in the wrong direction, but if we were to be able to fly straight from F to X, then that would be a good way to go. So, we expand it:

```
    20.7 :   expanding:  G-1->F
agenda:  PQ([(107, G-0->I), (109, G-2->H-0->D), (116, G-2->H-1->E), (103.5, G-2->H-2->T-
1->R-1->V), (123, G-1->F-0->D), (133, G-1->F-1->C)])
```

- Continuing now, basically straight to the goal, we have:

```
    58.5 :   expanding:  G-2->H-2->T-1->R-1->V
agenda:  PQ([(107, G-0->I), (109, G-2->H-0->D), (116, G-2->H-1->E), (123, G-1->F-0->D),
(133, G-1->F-1->C), (154, G-2->H-2->T-1->R-1->V-1->E), (105, G-2->H-2->T-1->R-1->V-2-
>Y)])
    69.7 :   expanding:  G-2->H-2->T-1->R-1->V-2->Y
agenda:  PQ([(107, G-0->I), (109, G-2->H-0->D), (116, G-2->H-1->E), (123, G-1->F-0->D),
(133, G-1->F-1->C), (154, G-2->H-2->T-1->R-1->V-1->E), (175, G-2->H-2->T-1->R-1->V-2->Y-
0->E), (105, G-2->H-2->T-1->R-1->V-2->Y-2->Z)])
    84.7 :   expanding:  G-2->H-2->T-1->R-1->V-2->Y-2->Z
agenda:  PQ([(107, G-0->I), (109, G-2->H-0->D), (116, G-2->H-1->E), (123, G-1->F-0->D),
(133, G-1->F-1->C), (154, G-2->H-2->T-1->R-1->V-1->E), (175, G-2->H-2->T-1->R-1->V-2->Y-
0->E), (151, G-2->H-2->T-1->R-1->V-2->Y-2->Z-1->W), (106, G-2->H-2->T-1->R-1->V-2->Y-2-
>Z-2->AA)])
    95.9 :   expanding:  G-2->H-2->T-1->R-1->V-2->Y-2->Z-2->AA
agenda:  PQ([(107, G-0->I), (109, G-2->H-0->D), (116, G-2->H-1->E), (123, G-1->F-0->D),
(133, G-1->F-1->C), (154, G-2->H-2->T-1->R-1->V-1->E), (175, G-2->H-2->T-1->R-1->V-2->Y-
0->E), (151, G-2->H-2->T-1->R-1->V-2->Y-2->Z-1->W), (106, G-2->H-2->T-1->R-1->V-2->Y-2-
>Z-2->AA-1->X)])
18 nodes visited; 10 states expanded; solution cost: 105.9
[(None, 'G'), (2, 'H'), (2, 'T'), (1, 'R'), (1, 'V'), (2, 'Y'), (2, 'Z'), (2, 'AA'), (1,
'X')]
```

Using A* has roughly halved the number of nodes visited and expanded. In some problems it can result in an enormous savings, but, as we'll see in the next section, it depends on the heuristic we use.

Good and bad heuristics

In order to think about what makes a heuristic good or bad, let's imagine what the perfect heuristic would be. If we were to magically know the distance, via the shortest path in the graph, from each node to the goal, then we could use that as a heuristic. It would lead us directly from start to goal, without expanding any extra nodes. But, of course, that's silly, because it would be at least as hard to compute the heuristic function as it would be to solve the original search problem.

So, we would like our heuristic function to give an estimate that is as close as possible to the true shortest-path-length from the state to the goal, but also to be relatively efficient to compute.

An important additional question is: if we use a heuristic function, are we still guaranteed to find the shortest path through our state space? The answer is: yes, if the heuristic function is *admissible*. A heuristic function is admissible if it is guaranteed to be an *underestimate* of the actual cost of the optimal path to the goal. To see why this is important, consider a state s from which the goal can actually be reached in 10 steps, but for which the heuristic function gives a value of 100. Any path to that state will be put into the agenda with a total cost of 90 more than the true cost. That means that if a path is found that is as much as 89 units more expensive that the optimal path, it will be accepted and returned as a result of the search.

It is important to see that if our heuristic function always returns value 0, it is admissible. And, in fact, with that heuristic, the A^* algorithm reduces to uniform cost search.

In the example of navigating through a city, we used the Euclidean distance between cities, which, if distance is our cost, is clearly admissible; there's no shorter path between any two points.

Exercise 8.1. Would the so-called 'Manhattan distance', which is the sum of the absolute differences of the x and y coordinates be an admissible heuristic in the city navigation problem, in general? Would it be admissible in Manhattan?

Exercise 8.2. If we were trying to minimize travel time on a road network (and so the estimated time to travel each road segment was the cost), what would be an appropriate heuristic function?