cisc3665, fall 2011 / prof sklar
Lab III.3: Path Planning

Instructions

This is the third lab for unit III, Game Physics, Motion and Perception.
You do not have to submit this lab. The purpose of this lab is to give you a chance to put into practice some concepts concerning *Path Planning*.

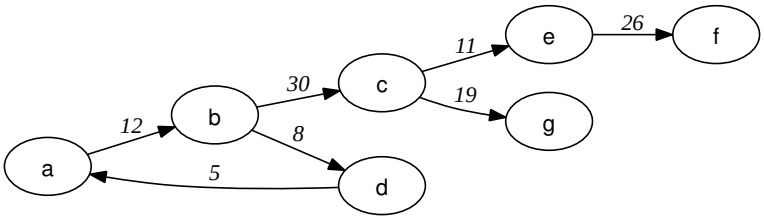
1 Graphs

In Computer Science, we often use a type of formal model called a *graph*. This is not the same thing as an $x-y$ graph or a bar graph or a pie chart. This type of graph consists of *nodes* and *edges*. The formal definition of graph, G, is given as:

$$G = \{N, E\}$$

where N is the set of nodes in the graph and E is the set of edges.

An example of such a graph is shown below:



The set of nodes is $N = \{a, b, c, d, e, f, g\}$. The set of edges is $E = \{ab, bc, bd, ce, cg, da, ef\}$. Each edge is labeled with a number. In the examples discussed in this lab, we define that number to represent the *cost* associated with that edge. So, the cost of going from node a to node b is 12, and the cost of going from node b to node d is 8. The cost of going from node a to node d via node b is $12 + 8 = 20$, whereas the cost of going from node d to node a is 5.

A graph is *connected* if every node has at least one edge. The above graph is connected.

If the edges in a graph have arrows on one (or both) end(s), the graph is called a *directed* graph (or a "digraph"). If the edges do not have arrows on them, then it is called an *undirected* graph. The graph above is a directed graph. The nodes pointed to by an edge are called the *children* of the node from which the edge came; conversely, that node is called the *parent*.

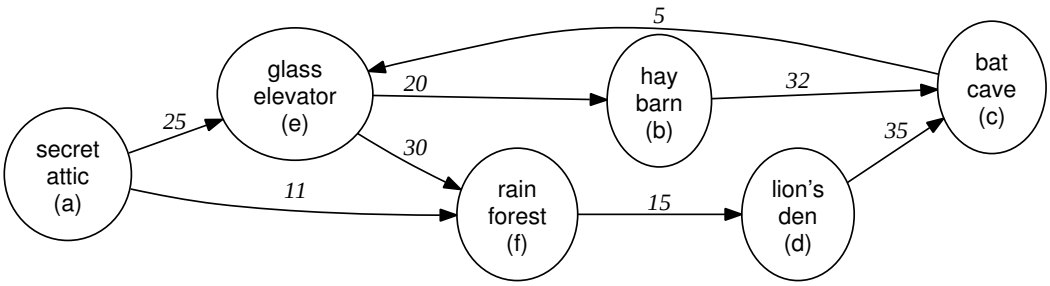
A *subgraph* is a subset of the nodes in a graph, and the corresponding edges that connect them. For example, $\{a, b, d\}$ is a subgraph of the graph above, with edges $\{ab, bd, da\}$.

A *path* is a sequence of edges. For example, $\{ab, bc, ce\}$ is a path going from node a to node e.

A path that leads back to (at least) one of the nodes along that same path is called a *cycle*. For example, the path $\{ab, bd, da\}$) is a cycle, because it starts and ends with node a. A graph with at least one cycle in it is called a *cyclic* graph. A graph with no cycles is called an *acyclic* graph. A directed acyclic graph is sometimes referred to as a *DAG*.

A *tree* is an acyclic graph. The *root* of a tree has no edges going into it. The *leaf* of a tree has no edges leaving it. Typically, a tree has one root and multiple leaves.

The graph below represents places within a game environment, where each node represents a different place. Each edge represents a route that an agent can take from one place to another. The cost along each edge is the cost to the agent of traveling along that route.



Using this graph, answer the following questions:

1.1 What is the set of nodes, N, in the graph? (You can use the letter abbreviations instead of the place names.)

1.2 What is the set of edges, E, in the graph?

1.3 What is the cost associated with each edge?

1.4 Which edge has the smallest cost?

1.5 Which edge has the largest cost?

1.6 Is this graph connected or unconnected?

1.7 Is this graph directed or undirected?

1.8 Is this graph cyclic or acyclic?

1.9 How many ways are there to go from the secret attic to the hay barn?

1.10 What is the cost of each route, or *path* (from the secret attic to the hay barn)?

1.11 Which path (from the secret attic to the hay barn) is the *shortest* (i.e., has the least cost)?

2 Shortest Path — Dijkstra's Algorithm

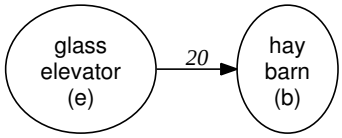
It is common to use a graph to represent different ways to get from one place to another, as in our example, above. So, it is common to want to find the *shortest path* from one node in particular to another node in particular, as you just did above. It is also common to find the shortest path between *any* two nodes. If the graph is large, you can see where that might become really time-consuming to figure out—because you have to calculate all the possible routes from each node to every other node. In fact, even enumerating the routes to check in the above graph with 6 nodes would be time-consuming. The paths are from a to b, a to c, a to d, a to e, a to f, b to a, b to c, b to d, b to e, b to f, c to a, c to b, c to d, c to e, c to f, d to a, d to b, d to c, d to e, d to f, e to a, e to b, e to c, e to d, e to f, f to a, f to b, f to c, f to d, and f to e. That is 30 paths to compute the cost of!

In addition, each path can be constructed in multiple ways. For example, the path ac (from the secret attic, a, to the bat cave, c), can be constructed as $\{ae, be, bc\}$ or $\{af, fd, dc\}$.

Luckily, there is a handy algorithm invented by a man called Dijkstra for computing the shortest path within a graph. We use the algorithm to construct a "shortest path tree", which helps us find the shortest path in the graph.

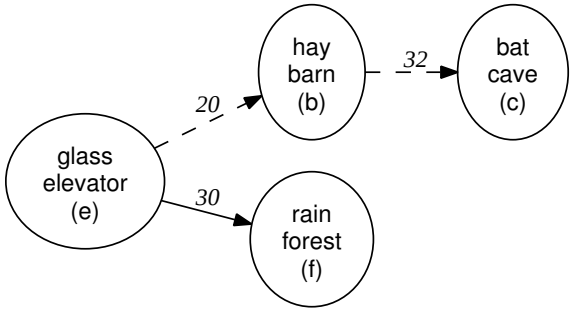
Go through the step-by-step instructions, below, in order to learn **Dijkstra's algorithm** and find the shortest path in our graph.

2.1 Start by designating one node in the graph as the *source* node. This is the node from which we will compute the shortest path. Let's use the glass elevator, node e, as our source. Node e has two edges leaving it (eb and ef). Compare their costs—which one is less? Edge eb is less because its cost is 20, whereas the cost of ef is 30. So we begin creating our shortest path tree by putting e and b in the tree:



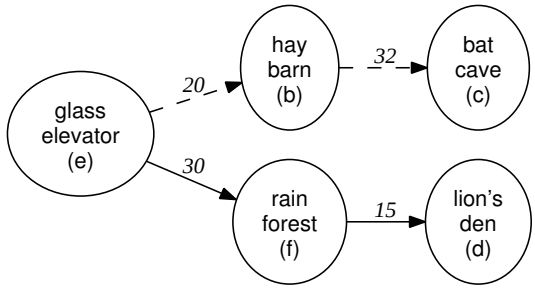
2.2 Next look at the edges leaving node b. Which edges leave node b and what are their associated costs?

2.3 There is only one edge leaving node b, and that is bc with a cost of 32. If we add that path, bc, to our shortest path tree, then we need to add the cost of bc to the cost of eb: $20 + 32 = 52$.
However, since we are looking for the shortest path within the graph that starts at node e, we need to go back and compare the cost of this path $\{eb, bc\}$ with other paths that start from e. This means that we need to go back to node e and see if any other edge leaving e has a cost less than our current shortest path cost (52). Indeed, the edge that we first ignored, ef, which has a cost of 30 is less than the cost of 50 which we have so far. So now we add node f to the shortest path tree:



2.4 Again, look at the edges leaving the node we just added to the tree, f. Which edges are those and what are their costs?

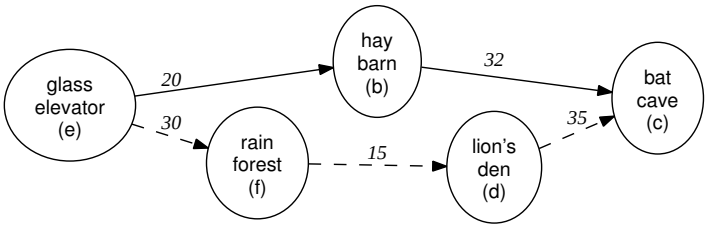
2.5 The shortest path tree now looks like this:



The shortest path is $\{ef, fd\}$ at a cost of 45, as compared to the other path we have explored, $\{eb, bc\}$, at a cost of 52.

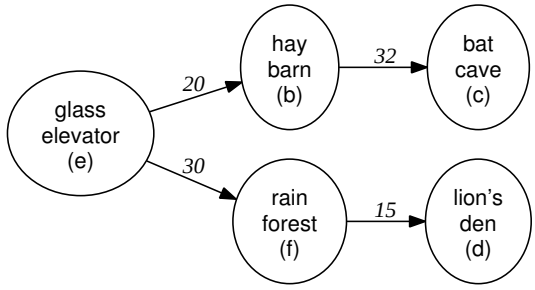
2.6 Again, look at the edges leaving the node we just added to the tree, d. Which edges are those and what are their costs?

2.7 There is only one edge leaving node d, and that goes to node c, a node we already visited when considering a different path: $\{eb, bc\}$. So now we compare the cost of getting to node c using the newly-added edge, dc, to the current shortest path: $\{ef, fd, dc\}$, in comparison with the already considered path $\{eb, bc\}$.
What is the cost of path $\{ef, fd, dc\}$?
What is the cost of path $\{eb, bc\}$?
Which path is shortest?

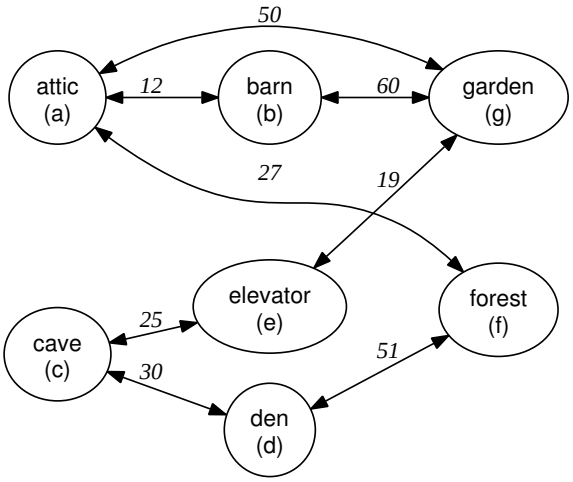


We remove the edge dc from the tree, since we now know that the shortest route to get from e to c does not go through dc. Cross out edge dc from the tree above.

2.8 The remaining tree, see below, now contains the shortest path from the source node, e, to every other node in the graph that can be reached by e. Note that node a is not in the tree; this is because there is no edge that leads into node a from any of the nodes reachable from e.



2.9 Now try it on your own. Given the graph below:

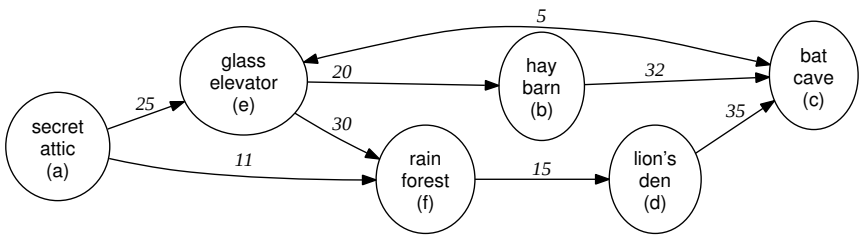


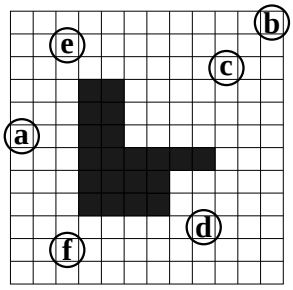
Find the shortest path from the attic to every other reachable node in the graph. Notice that all the edges are *bi-directional* (arrows at both ends). Treat the bi-directional edges as if there were two edges with the same cost, one going in each direction.

3 A* Algorithm

Dijkstra's algorithm works well when you really know the cost of going from one node to another. But often, you don't know the actual cost until you have done the traveling from one node to another. A very popular algorithm for computing a path when you don't know the actual cost is called **A*** (pronounced "A-star"). This algorithm is based on two principles: first, that you know the cost of how far you have traveled (i.e., from the source node to your current node); and second, that you can estimate somehow the cost from your current node to the final node in your journey.

Let's use A* on a slightly modified version of the graph from page 2, repeated below with the modification which allows travel between nodes (c) and (e) in both directions instead of just one direction. For the A* algorithm, assume that we do not know ahead-of-time the costs of each edge (like we do on the graph). Instead, we'll show the places that the graph represents on a map of the game environment—such as the map below, shown to the right of the graph. We'll use A* to figure out how to get from the secret attic (a) to the bat cave (b).



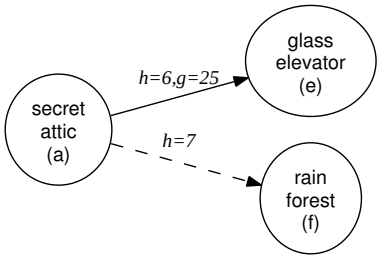


3.1 The A* algorithm works by constructing a tree, like we did with the shortest path calculation. Start by putting the starting location, or source, in this case a at the root of the tree. Then look at the places that can be reached by a: in this case, these are places e and f. Since we don't know the actual cost of traveling on path ae or path af, we have to estimate.

This is called using a *heuristic*, i.e., an educated guess. We will use the Manhattan distance on the map to estimate the cost; and we will use the numbers on the graph on page 2 as the actual costs of traveling between each place on the map.

The Manhattan distance for path $ae = 6$ and for path $af = 7$. So we put ae on the shortest cost tree.

3.2 Now, once we have reached the place, we know what the actual cost was. Since we used the heuristic to select e to travel to, we don't know what the actual cost of traveling to f would be—but we do know the cost to get to e. The tree below shows the heuristic cost, h, for both paths, and the actual cost, g, for the path we travelled. By convention in defining the A* algorithm, the variable h is used to represent the heuristic (estimated) cost and the variable g is used to represent the actual cost.



3.3 The next step, as with the Dijkstra algorithm, is to consider the next places we can go to from e, the glass elevator. As above, this would be either hay barn, b, or the bat cave, c.

Again, we use our heuristic to guestimate the cost of going from e to b and c.

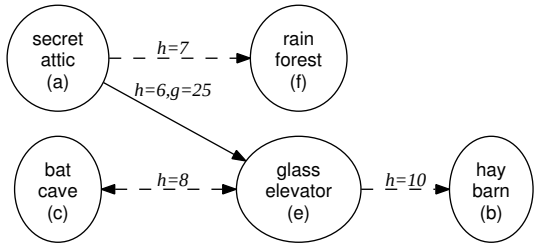
What is the Manhattan distance of ec?

What is the Manhattan distance of eb?

3.4 Here is where the A* algorithm differs from the Dijkstra algorithm. Moving forward, we add the *heuristic* cost of going to the next place to the *actual* cost of getting to the current place:

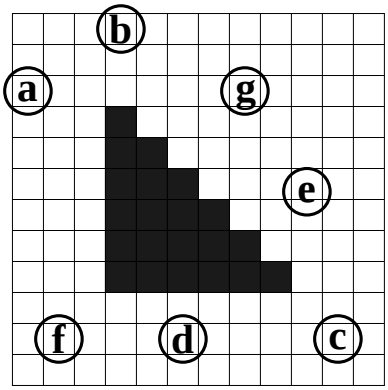
$$cost = h + g$$

So, given the shortest cost tree below, what does that say about the A* estimated cost of getting from a to b and a to c?



The actual cost of getting from a to e is 25 (taken from the graph on the previous page). The heuristic, estimated cost, for going on to the next place is 10 to go to the hay barn and 8 to go to the bat cave: i.e., $25 + 10$ versus $25 + 8$. So we choose the bat cave. In addition, the algorithm recognizes that our ultimate destination (as stated above) is the bat cave. So we're done.

3.5 Now try it on your own. Given the graph below:



Using the A* algorithm, find the shortest-cost path from the attic to the cave. Use the Manhattan distance for the heuristic measure, and the costs on the graph on page 5 for the actual costs.