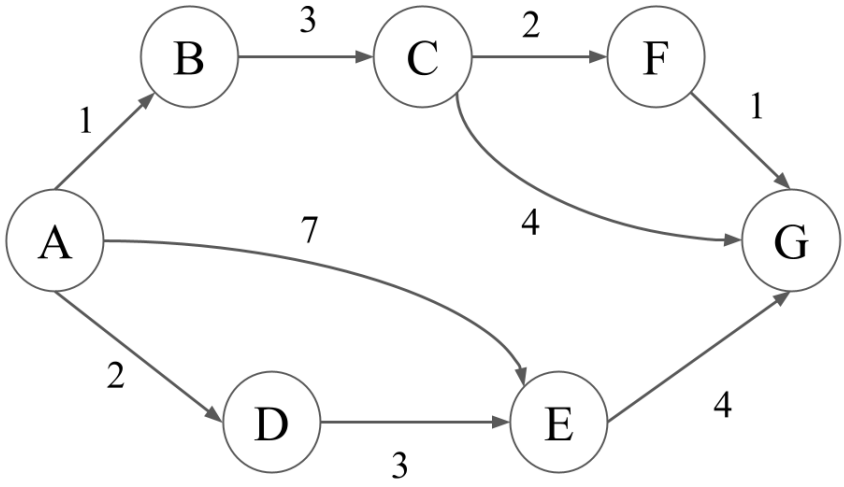
CS 61B Fall 2023

Shortest Paths, MSTs

Discussion 10: October 23, 2023

1 The Shortest Path To Your Heart

For the graph below, let `g(u, v)` be the weight of the edge between any nodes `u` and `v`. Let `h(u, v)` be the value returned by the heuristic for any nodes `u` and `v`.



Below, the pseudocode for Dijkstra's and A* are both shown for your reference throughout the problem.

Dijkstra's Pseudocode

```
PQ = new PriorityQueue()
PQ.add(A, 0)
PQ.add(v, infinity) # (all nodes except A).

distTo = {} # map
edgeTo = {} # map
distTo[A] = 0
distTo[v] = infinity # (all nodes except A).

while (not PQ.isEmpty()):
  poppedNode, poppedPriority = PQ.pop()

  for child in poppedNode.children:
    if PQ.contains(child):
      potentialDist = distTo[poppedNode] +
        edgeWeight(poppedNode, child)
      if potentialDist < distTo[child]:
        distTo.put(child, potentialDist)
        PQ.changePriority(child, potentialDist)
        edgeTo[child] = poppedNode
```

A* Pseudocode

```
PQ = new PriorityQueue()
PQ.add(A, h(A))
PQ.add(v, infinity) # (all nodes except A).

distTo = {} # map
distTo[A] = 0
distTo[v] = infinity # (all nodes except A).

while (not PQ.isEmpty()):
  poppedNode, poppedPriority = PQ.pop()
  if (poppedNode == goal): terminate

    for child in poppedNode.children:
      if PQ.contains(child):
        potentialDist = distTo[poppedNode] +
          edgeWeight(poppedNode, child)

        if potentialDist < distTo[child]:
          distTo.put(child, potentialDist)
          PQ.changePriority(child, potentialDist + h(child))
          edgeTo[child] = poppedNode
```

(a) Run Dijkstra's algorithm to find the shortest paths from A to every other vertex. You may find it helpful to keep track of the priority queue. We have provided a table to keep track of best distances, and the adjacent vertex that has an edge going to the target vertex in the current shortest paths tree so far.

Solution:

`B = 1 ; C = 4 ; D = 2 ; E = 5 ; F = 6 ; G = 7`

Explanation:

For the best explanation, it is recommended to check the slideshow linked on the website or watch the walkthrough video, as the text explanation is verbose.

We will maintain a priority queue and a table of distances found so far, as suggested in the problem and pseudocode. We will use {} to represent the PQ, and (()) to represent the distTo array.

`{A:0, B:inf, C:inf, D:inf, E:inf, F:inf, G:inf}. (()).`

`Pop A.`
`{B:inf, C:inf, D:inf, E:inf, F:inf, G:inf}. ((A: 0)).`

`changePriority(B, 1). changePriority(D, 2). changePriority(E, 7).`
`{B:1, D:2, C:inf, E:7, F:inf, G:inf}. ((A: 0)).`

`Pop B.`
`{D:2, C:inf, E:7, F:inf, G:inf}. ((A: 0, B: 1)).`

`changePriority(C, 4).`
`{D:2, C:4, E:7, F:inf, G:inf}. ((A: 0, B: 1)).`

`Pop D.`
`{C:4, E:7, F:inf, G:inf}. ((A: 0, B: 1, D: 2)).`

`changePriority(E, 5).`
`{C:4, E:5, F:inf, G:inf}. ((A: 0, B: 1, D: 2)).`

`Pop C.`
`{E:5, F:inf, G:inf}. ((A: 0, B: 1, D: 2, C: 4)).`

`changePriority(F, 6). changePriority(G, 8).`
`{E:5, F:6, G:8}. ((A: 0, B: 1, D: 2, C: 4)).`

`Pop E.`
`{F:6, G:8}. ((A: 0, B: 1, D: 2, C: 4, E: 5)).`

potentialDistToG = 9, which is worse than our current best known distance to G. No updates made.

Pop F.
{G:8}. ((A: 0, B: 1, D: 2, C: 4, E: 5, F: 6)).

potentialDistToG = 7. changePriority(G, 7).
{G:7}. ((A: 0, B: 1, D: 2, C: 4, E: 5, F: 6)).

Pop G.
{}. ((A: 0, B: 1, D: 2, C: 4, E: 5, F: 6, G: 7)).

(b) Given the weights and heuristic values for the graph above, what path would A* search return, starting from A and with G as a goal?

Edge weights	Heuristics
$g(A,B) = 1$	$h(A,G) = 7$
$g(B,C) = 3$	$h(B,G) = 6$
$g(C,F) = 2$	$h(C,G) = 3$
$g(C,G) = 4$	$h(F,G) = 1$
$g(F,G) = 1$	$h(D,G) = 6$
$g(A,D) = 2$	$h(E,G) = 3$
$g(D,E) = 3$	
$g(E,G) = 4$	
$g(A,E) = 7$	

Solution: A* would return $A - B - C - F - G$. The cost here is 7.

Explanation: A* runs in a very similar fashion to Dijkstra's. We got the same answer for the shortest path to G, though we actually explored less unnecessary nodes in the process (we never popped D and E off the queue). The main difference is the priority in the priority queue. For A*, whenever computing the priority (for the purposes of the priority queue) of a particular node n, always add $h(n)$ to whatever you would use with Dijkstra's. Additionally, note that A* will be run to find the shortest path to a particular goal node (as our heuristic is calculated as our estimate to our specific goal node), whereas Dijkstra's may be run with a specific goal, or it may be run to find the shortest paths to ALL nodes. In the solutions above, we found the shortest paths to all nodes, but if we only needed to know the shortest path to E, for example, we could have stopped after visiting E.

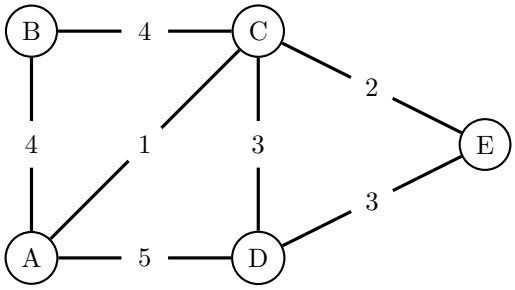
(c) Based on the heuristics for part b, is the A* heuristic for this graph good? In other words, will it always give us the actual shortest path from A to G? If it is good, give an example of a change you would make to the heuristic so that it is no longer good. If it is not, correct it.

Solution: The heuristic is admissible: for every node, the heuristic value is less than or equal to the shortest distance path from that node to the target node. It is also consistent: each estimate is less than or equal to the estimated distance from any neighboring vertex to the goal, plus the cost of reaching

that neighbor. Because it is both admissible and consistent, we can say that the heuristic is good. If we changed the heuristic from F to G to be 2 (ie. h(F, G) = 2), then the overall heuristic for the graph would no longer be admissible.

2 Minimalist Moles

Circle the mole wants to dig a network of tunnels connecting all of his secret hideouts. There are a set few paths between the secret hideouts that Circle can choose to possibly include in his tunnel system, shown below. However, some portions of the ground are harder to dig than others, and Circle wants to do as little work as possible. In the diagram below, the numbers next to the paths correspond to how hard that path is to dig for Circle. Lucky for us, he knows how to use MSTs to optimize the tunnel paths!



Below, the pseudocode for Kruskal's and Prim's are shown for your reference throughout the problem.

Kruskal's Pseudocode

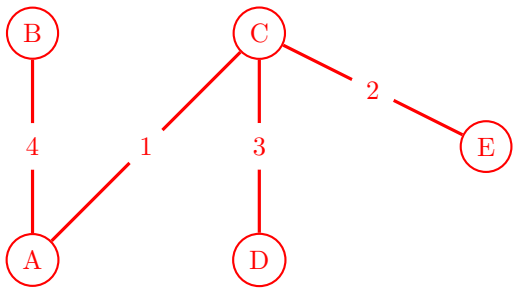
```
while there are still nodes not in the MST:
        Add the lightest edge
            that does not create a cycle.
        Add the new node to the
            set of nodes in the MST.
```

Prim's Pseudocode

```
Start with any node.
Add that node to the set of nodes in the MST.
While there are still nodes not in the MST:
    Add the lightest edge from a node in the MST
        that leads to a new node that is unvisited.
    Add the new node to the set of
        nodes in the MST.
```

(a) Find a valid MST for the graph above using Kruskal's algorithm, then Prim's. For Prim's algorithm, take A as the start node. In both cases, if there is ever a tie, choose the edge that connects two nodes with lower alphabetical order.

Solution: Both Prim's and Kruskal's give the MST below.



(b) Are the above MSTs different or the same? If different, describe a tie-breaking scheme that would make them the same. If the same, describe a tie-breaking scheme that would make them different.

Solution: In this particular case, the trees for Prim's and Kruskal's are the same. However, because our graph has edges with duplicate weights, then **it would be possible for Prim's and Kruskal's to give different answers with a different tiebreaking scheme**. For example, if in the graph

above, depending on which node we start Prim's from, as well as which tiebreaking scheme we use (ie. instead of lower alphabetical order, use higher alphabetical order, or randomness, or by most recently added node(s)), we could get other perfectly valid MSTs, like: