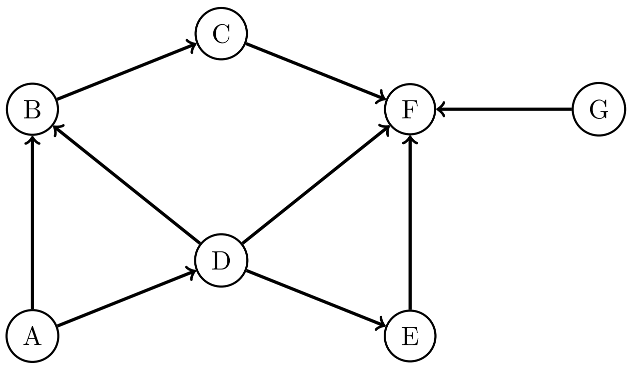
CS 61B
Fall 2020

DFS, BFS, Shortest Paths

Discussion 10: October 26, 2020

1 Graphs



(a) Write the graph above as an adjacency matrix, then as an adjacency list. What would be different if the graph were undirected instead?

Matrix:

```
  A B C D E F G <- end node
A 0 1 0 1 0 0 0
B 0 0 1 0 0 0 0
C 0 0 0 0 0 1 0
D 0 1 0 0 1 1 0
E 0 0 0 0 0 1 0
F 0 0 0 0 0 0 0
G 0 0 0 0 0 1 0
ˆ start node
```

List:

```
A: {B, D}
B: {C}
C: {F}
D: {B, E, F}
E: {F}
F: {}
G: {F}
```

For the undirected version of the graph, the representations look a bit more symmetric. For your reference, the representations are included below:

Matrix:

```
  A B C D E F G <- end node
A 0 1 0 1 0 0 0
B 1 0 1 1 0 0 0
C 0 1 0 0 0 1 0
D 1 1 0 0 1 1 0
E 0 0 0 1 0 1 0
F 0 0 1 1 1 0 1
G 0 0 0 0 0 1 0
ˆ start node
```

List:

```
A: {B, D}
B: {A, C, D}
C: {B, F}
D: {A, B, E, F}
E: {D, F}
F: {C, D, E, G}
G: {F}
```

(b) Give the DFS preorder, DFS postorder, and BFS order of the graph traversals starting from vertex A. Break ties alphabetically.

```
DFS preorder: ABCFDE (G)
DFS postorder: FCBEDA (G)
```

`BFS: ABDCEF (G)`

Explanations

DFS preorder and postorder: To compute this, we maintain a stack of nodes, and a marked set. As soon as we add something to our stack, we note the down for preorder. The top node in our stack represents the node we are currently on, and the marked set represents nodes that have been visited. After we add a node to the stack, we visit its lexicographically next unmarked child. If there is none, we pop the topmost node from the stack and note it down for postorder. *Note that there are two ways DFS could run: with restart or without; DFS with restart is the version where if we have exhausted our stack, and still have unmarked nodes left, we restart on the next unmarked node.*

Stack (bottom-top), MarkedSet, Preorder, Postorder.

```
A. {A}. A. -
AB. {AB}. AB. -
ABC. {ABC}. ABC. -
ABCF. {ABCF}. ABCF. -
ABC. {ABCF}. ABCF. F
AB. {ABCF}. ABCF. FC.
A. {ABCF}. ABCF. FCB.
AD. {ABCFD}. ABCFD. FCB.
ADE. {ABCFDE}. ABCFDE. FCB.
AD. {ABCFDE}. ABCFDE. FCBE.
A. {ABCFDE}. ABCFDE. FCBED.
\-. {ABCFDE}. ABCFDE. FCBEDA.
```

If DFS restarts on unmarked nodes, the following happens in the last line. Otherwise, we do not proceed further.

```
G. {ABCFDEG}. ABCFDEG. FCBEDAG.
```

BFS: Start at the provided start node. Note it down, and mark it. Now, consider all nodes that are 1-hop (i.e. one edge) away from the start node. Write all of them down, and mark all of them. Next, consider all unmrked nodes that are 1-hop away from the nodes that were 1-hop away from the start (i.e., 2 hops away from the start). And so on. Note that unlike DFS, BFS uses a queue.

BFS, MarkedSet.

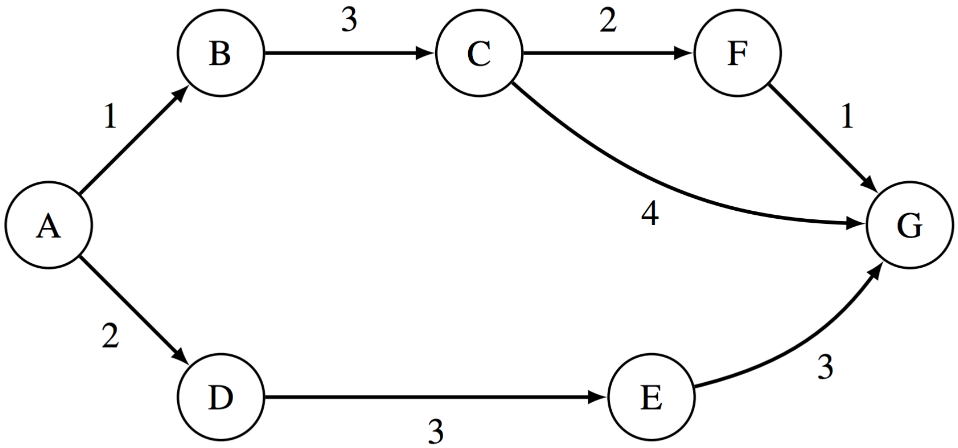
```
A. {A}.
A BD. {ABD}.
A BD CEF. {ABDCEF}.
```

If BFS restarts, the following happens at the end. Otherwise, we do not proceed further..

```
A BD CEF (G). {ABDCEFG}.
```

2 Dijkstra's Algorithm

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`.



Edge weights	Heuristics
$g(A,B)=1$	$h(A,G)=8$
$g(B,C)=3$	$h(B,G)=6$
$g(C,F)=4$	$h(C,G)=5$
$g(C,G)=4$	$h(F,G)=1$
$g(F,G)=1$	$h(D,G)=6$
$g(A,D)=2$	$h(E,G)=3$
$g(D,E)=3$	
$g(E,G)=3$	

(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 and make a table of current distances.

Pseudocode

```
PQ = new PriorityQueue()
PQ.add(A, 0)
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()

    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)
```

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

Explanation: 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 table of distances.

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

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

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

Pop D.
{C:4, E:inf, 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 = 8, which is the same. No change priority.

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 below, what path would A* search return, starting from A and with G as a goal?

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))
```

A* would return $A - D - E - G$. The cost here is $2 + 3 + 3 = 8$.

Explanation: A* runs in a very similar fashion to Dijkstra's. The only 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.

(c) Is the heuristic admissible? Why or why not?

A heuristic is admissible if all of its estimations `h(x)` are optimistic. No it's not, because the actual shortest path from $A \to G$ is of cost 7 if we take the northern route, but the heuristic estimates it will cost 8.