CS 61B Fall 2025

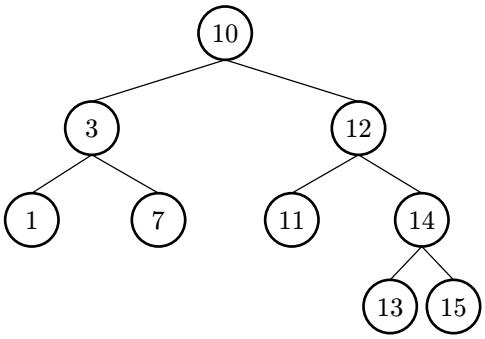
Tree Traversals, Graphs, and Shortest Paths

Discussion 08: October 20, 2025

1 Trees, Graphs, and Traversals

(a) Write the following traversals of the BST below.

Pre-order:
In-order:
Post-order:
Level-order (BFS):



Solution:

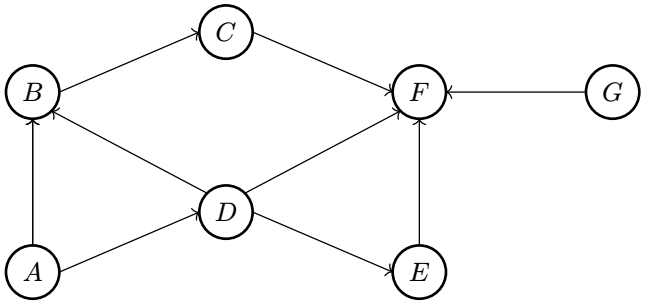
Pre-order: 10 3 1 7 12 11 14 13 15

In-order: 1 3 7 10 11 12 13 14 15

Post-order: 1 7 3 11 13 15 14 12 10

Level-order (BFS): 10 3 12 1 7 11 14 13 15

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



Solution:

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
^ 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
^ 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}
```

(c) Write the order in which (1) DFS pre-order, (2) DFS post-order, and (3) BFS would visit nodes in the same directed graph above, starting from vertex A. Break ties alphabetically.

Pre-order:

ABCFDE (G)

Post-order:

FCBEDA (G)

BFS:

ABDCEF (G)

To compute DFS, we maintain a stack of nodes, and a visited set. As soon as we add something to our stack, we note it 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)	VisitedSet	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

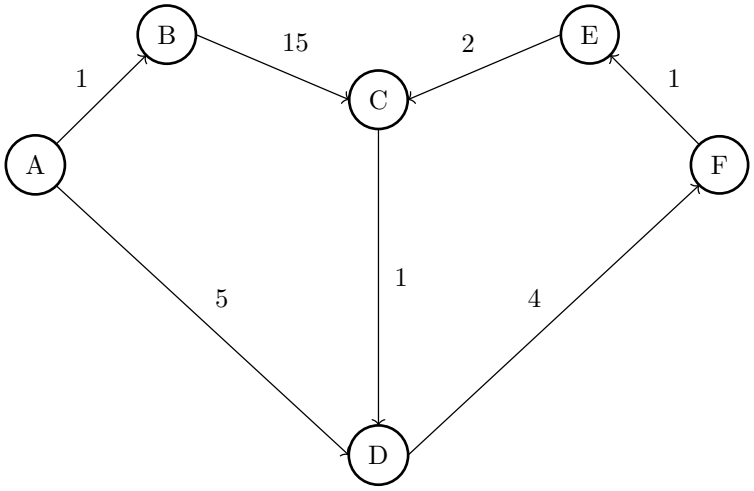
If DFS restarts on unmarked nodes, the following happens in the last line. Otherwise, we do not proceed further.

Stack (bottom-top)	VisitedSet	Preorder	Postorder
G	{ABCFDEG}	ABCFDEG	FCBEDAG

For BFS, we use a queue instead of a stack. BFS does not have the notion of in-order and post-order, so we only visit it when we remove it from the queue.

2 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:
    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, goal))
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:
    potentialDist = distTo[poppedNode] +
      edgeWeight(poppedNode, child)

    if potentialDist < distTo[child]:
      distTo.put(child, potentialDist)
      PQ.changePriority(child, potentialDist
        + h(child, goal))
      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.

	A	*B*	*C*	*D*	*E*	*F*
distTo						
edgeTo						

Solution:

B = 1 ; D = 5 ; F = 9 ; E = 10 ; C = 12

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}. (()).

Pop A.

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

changePriority(B, 1). changePriority(D, 5).

{B:1, D:5, C:inf, E:inf, F:inf}. ((A: 0)).

Pop B.

{D:5, C:inf, E:inf, F:inf}. ((A: 0, B: 1)).

changePriority(C, 16).

{D:5, C:16, E:inf, F:inf}. ((A: 0, B: 1)).

Pop D.

{C:16, E:inf, F:inf}. ((A: 0, B: 1, D: 5)).

changePriority(F, 9).

{F: 9, C:16, E:inf, F:inf}. ((A: 0, B: 1, D: 5)).

Pop F.

{C:16, E:inf}. ((A: 0, B: 1, D: 5, F: 9)).

changePriority(E, 10).

{E:10, C:16}. ((A: 0, B: 1, D: 5, F: 9)).

Pop E.

{C:16}. ((A: 0, B: 1, D: 5, F: 9, E: 10)).

changePriority(C, 12).

{C:12}. ((A: 0, B: 1, D: 5, F: 9, E: 10)).

Pop C.

{}. ((A: 0, B: 1, D: 5, F: 9, E: 10, C: 12)).

At the end, our table looks like this:

	A	*B*	*C*	*D*	*E*	*F*
distTo	0	1	12	5	10	9
edgeTo	-	A	E	A	F	D

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

Edge Weights	**Heuristics**
$g(A, B) = 1$	$h(A, F) = 8$
$g(A, D) = 5$	$h(B, F) = 16$
$g(B, C) = 15$	$h(C, F) = 4$
$g(C, D) = 1$	$h(D, F) = 4$
$g(D, F) = 4$	$h(E, F) = 5$
$g(F, E) = 1$	
$g(E, C) = 2$	

	A	*B*	*C*	*D*	*E*	*F*
distTo						
edgeTo						

Solution: A∗ would return *A-D-F* . The cost here is 9.

	A	*B*	*C*	*D*	*E*	*F*
distTo	0	1	∞	5	∞	9
edgeTo	-	A	-	A	-	D

Explanation: A∗ runs in a very similar fashion to Dijkstra's. We got the same answer for the shortest path to F, though we actually explored less unnecessary nodes in the process (we never popped B, C, or 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 F? 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 D to F to be 6 (i.e. `h(D, F)` = 6), then the overall heuristic for the graph would no longer be admissible.

3 Extra: Graph Conceptuals

(a) Answer the following questions as either **True** or **False** and provide a brief explanation:

1. If a graph with n vertices has $n-1$ edges, it **must** be a tree.

   Solution: False. The graph **must** be connected.

2. Every edge is looked at exactly twice in each full run of DFS on a connected, undirected graph.

   Solution: True. Say an edge connects u and v. Both u and v will look at the other one through this edge when it's their turn.

3. In BFS, let $d(v)$ be the minimum number of edges between a vertex v and the start vertex. For any two vertices u, v in the fringe (recall that the fringe in BFS is a queue), $|d(u) - d(v)|$ is **always less than** 2.

   Solution: True. Suppose this was not the case. Then, we could have a vertex 2 edges away and a vertex 4 edges away in the fringe at the same time. But, the only way to have a vertex 4 edges away is if a vertex 3 edges away was removed from the fringe. We see this could never occur because the vertex 2 edges away would be removed before the vertex 3 edges away!

(b) Given an undirected graph, provide an algorithm that returns true if a cycle exists in the graph, and false otherwise. Also, provide a Θ bound for the worst case runtime of your algorithm.

Solution: We do a depth first search traversal through the graph. While we recurse, if we visit a node that we visited already, then we've found a cycle. Assuming integer labels, we can use something like a `visited` boolean array to keep track of the elements that we've seen, and while looking through a node's neighbors, if `visited` gives true, then that indicates a cycle.

However, since the graph is undirected, if an edge connects vertices `u` and `v`, then `u` is a neighbor of `v`, and `v` is a neighbor of `u`. As such, if we visit `v` after `u`, our algorithm will claim that there is a cycle since `u` is a visited neighbor of `v`. To address this case, when we visit the neighbors of `v`, we should ignore `u`. To implement this in code, we could add the parent as another parameter in the method call. In the worst case, we have to explore at most V edges before finding a cycle (number of edges doesn't matter). So, this runs in $\Theta(V)$.

Pseudocode is provided below (for a disconnected graph, we should call `find_cycle` on each component).

```
find_cycle(v, parent=-1):
    visited[v] = true
    for (v, w) in G:
        if !visited[w]:
            if find_cycle(w, v):
                return True
        else if w != parent:
            return True
    return False
```