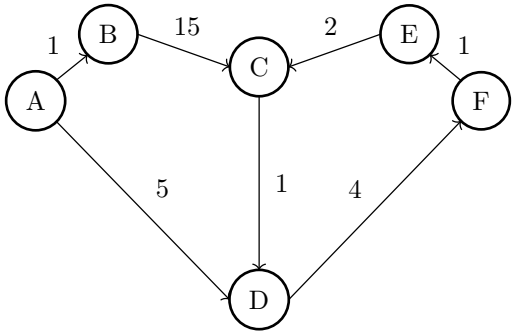
CS 61B Fall 2026

Shortest Paths, MSTs, DAGs

Discussion 09: October 26, 2026

1 The Shortest Path to Your Heart



Dijkstra's Pseudocode Dijkstra's finds the shortest paths from a starting node v to all other nodes. The overall strategy is to process nodes in order of distance from v, starting from the closest.

```
PQ = new PriorityQueue()
distTo = {}                              // Map of vertex to best known distance from v.
edgeTo = {}                              // Map of vertex to the vertex we came from.

PQ.add(v, 0)                             // Distance from the starting node v to itself is
0.
distTo[v] = 0
for all other vertices u:
  PQ.add(u, infinity)                    // Distance infinity because we don't yet
  distTo[u] = infinity                   // know any paths to u.

while (not PQ.isEmpty()):
  u, dist = PQ.pop()                     // u is the closest unprocessed vertex.

  for all vertices w adjacent to u:
    potentialDist = dist + edgeWeight(u, w)

    if potentialDist < distTo[w]:        // See if there's a shorter path to
      distTo.put(w, potentialDist)       // each neighbor via u. If so, update
      PQ.changePriority(w, potentialDist)  // distTo, edgeTo, and the weight in
      edgeTo[w] = u                      // PQ accordingly.
```

(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						

Step 0: Initialize PQ: {A:0, B:∞, C:∞, D:∞, E:∞, F:∞}

	A	*B*	*C*	*D*	*E*	*F*
distTo	0	∞	∞	∞	∞	∞
edgeTo	–	–	–	–	–	–

Step 1: Pop A, update B and D PQ: {**B:1**, **D:5**, C:∞, E:∞, F:∞}

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

Step 2: Pop B, update C PQ: {D:5, **C:16**, E:∞, F:∞}

	A	*B*	*C*	*D*	*E*	*F*
distTo	0	1	**16**	5	∞	∞
edgeTo	–	A	**B**	A	–	–

Step 3: Pop D, update F PQ: {**F:9**, C:16, E:∞}

	A	*B*	*C*	*D*	*E*	*F*
distTo	0	1	16	5	∞	**9**
edgeTo	–	A	B	A	–	**D**

Step 4: Pop F, update E PQ: {**E:10**, C:16}

	A	*B*	*C*	*D*	*E*	*F*
distTo	0	1	16	5	**10**	9
edgeTo	–	A	B	A	**F**	D

Step 5: Pop E, update C PQ: {**C:12**}

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

Step 6: Pop C (done) PQ: {}

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

(b) Answer the following questions about the runtime of Dijkstra's algorithm. Write your answers in terms of the number of vertices V and the number of edges E.

i. How many total calls are made to `PQ.add()` on lines 5 and 8?

Solution: V calls total. Line 5 is called once (for the starting vertex), and line 8 is called $V-1$ times (for all other vertices).

ii. How many total calls are made to `PQ.pop()` on line 12?

Solution: V calls. Each vertex is popped exactly once from the priority queue.

iii. How many total calls are made to `PQ.changePriority()` on line 19?

Solution: $O(E)$ calls. In the worst case, we call `changePriority` once per edge, since we check each edge exactly once (when we pop its source vertex).

iv. All priority queue operations take $O(\log N)$ time, where N is the number of items in the priority queue. Given that, what's the total runtime? Simplify as much as possible, using the assumption that $V < E$. Your answer should contain only E and V (no N).

Solution: The PQ contains at most V items, so $N = V$. We have V adds, V pops, and $O(E)$ changePriority calls, each taking $O(\log V)$ time. Total: $O(V \log V + V \log V + E \log V) = O(E \log V)$ (since $V < E$).

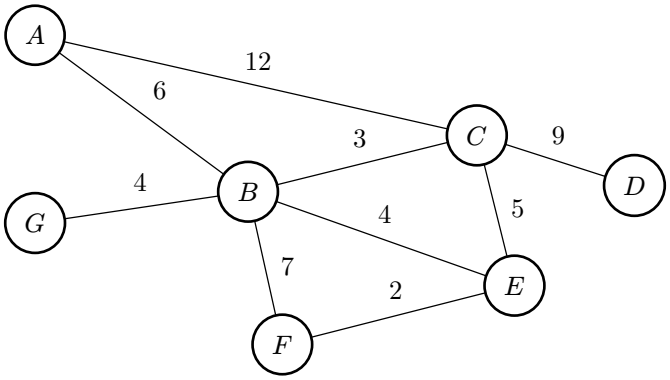
2 MST Algorithms

- **Prim's algorithm:** Start from any vertex. Repeatedly add the shortest edge that connects a vertex in the tree to a vertex not yet in the tree.
- **Kruskal's algorithm:** Sort all edges by weight. Add edges in order from smallest to largest, skipping any edge that would create a cycle.

(a) Fill in the blank with an expression using V (the number of vertices) and/or E (the number of edges):

Both algorithms stop when all V vertices are connected to the tree, which occurs when the tree contains _____ edges.

$V-1$ edges. A tree with V vertices always has exactly $V-1$ edges.



(b) For the graph above, list the edges in the order they're added to the MST by Kruskal's and Prim's algorithm. Assume Prim's algorithm starts at vertex A. Assume ties are broken in alphabetical order. Denote each edge as a pair of vertices (e.g. AB is the edge from A to B). The first edge is shown for you.

Prim's algorithm order: AB,

Kruskal's algorithm order: EF,

Solution: Prim's algorithm order: **AB, BC, BE, EF, BG, CD**
Kruskal's algorithm order: **EF, BC, BE, BG, AB, CD**

(c) True/False: Adding 1 to the smallest edge of a graph G with unique edge weights must change the total weight of its MST.

Solution: True, either this smallest edge (now with weight +1) is included, or this smallest edge is not included and some larger edge takes its place since there was no other edge of equal weight. Either way, the total weight increases.

(d) True/False: If all edge weights in a graph are unique, there is only one possible MST.

Solution: True, the cut property states that the minimum weight edge in a cut must be in the MST. Since all weights are unique, the minimum weight edge is always unique, so there is only one possible MST.

(e) True/False: The shortest path from vertex u to vertex v in a graph G is the same as the shortest path from u to v using only edges in T, where T is the MST of G.

Solution: False, consider vertices C and E in the graph above. The shortest path between C and E uses the edge CE, but it is not part of the MST of the graph.

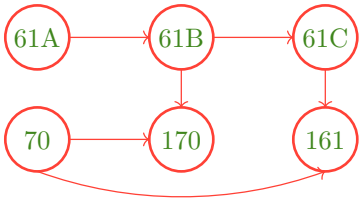
3 Class Enrollment

You're planning your CS classes for the upcoming semesters, but it's hard to keep track of all the prerequisites! Let's figure out a valid ordering of the classes you're interested in. A valid ordering is an ordering of classes such that every prerequisite of a class is taken before the class itself. Assume we're taking one CS class per semester.

(a) The list of prerequisites for each course is given below (not necessarily accurate to actual courses!). Draw a graph to represent our scenario.

- CS 61A: None
- CS 61B: CS 61A
- CS 61C: CS 61B
- CS 70: None
- CS 170: CS 61B, CS 70
- CS 161: CS 61C, CS 70

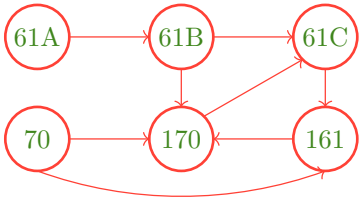
Solution:



(b) Suppose we added a new prerequisite where the student must take CS 161 before CS 170 and CS 170 before CS 61C. Is there still a valid ordering of classes such that no prerequisites are broken? If no, explain.

Solution:

The new graph looks like this:



There exists a cycle between 161 → 170 → 61C → 161, so a valid ordering does not exist. Our graph must be directed and acyclic for a topological sort to work.

(c) With the original graph, perform a topological sort to find a valid ordering of the 6 classes. Break ties by going to the lower course number first.

Solution:

With topological sorting, if an edge from vertex u to vertex v exists in the graph, then u must come before v in the sorted order. Every edge in our graph represents a prerequisite where class u must be taken before v. So, our topologically sorted order will ensure that we meet all prerequisites!

To topological sort on a graph, perform DFS from each vertex with indegree 0 (no incoming edges), but don't clear the node's we've marked between each new DFS traversal. Afterwards, we reverse the postorder to get our topologically sorted order.

In our graph, there are two vertices with indegree 0: CS 61A and CS 70. If we DFS from CS 61A then CS 70, we get a postorder of: [CS 161, CS 61C, CS 170, CS 61B, CS 61A, CS 70]. Reversing this order gives a valid ordering of: [**CS 70, CS 61A, CS 61B, CS 170, CS 61C, CS 161**].

If we DFS from CS 70 then CS 61A, we get a postorder of: [CS 161, CS 170, CS 70, CS 61C, CS 61B, CS 61A]. Reversing this order gives a different but still valid ordering of: [**CS 61A, CS 61B, CS 61C, CS 70, CS 170, CS 161**].

4 Shortest Paths Algorithm Design

Two cities, Chicago and Berkeley, are located in the United States. The railroad system connecting them can be modeled as a **weighted directed graph**, with V vertices, E edges, and weights representing the length of the railway. Ethan wishes to take a railway from Chicago to Berkeley, and needs to determine the shortest railway distance between them.

Define the set C to be all cities in Chicago, and B to be all cities in Berkeley. There can be cities that belong to neither region along the way. The shortest distance between the two cities is the shortest distance between any city c_C in Chicago and c_B in Berkeley.

Describe an algorithm that computes the minimum railway distance from Chicago to Berkeley, in $O((V + E) \log V)$ time. You are able to utilize all graph algorithms you learned in class.

Hint: Consider modifying the graph so that running a graph algorithm yields an equivalent answer to solving the original problem.

Solution: Create a dummy vertex d. For each city in **Chicago**, create an edge from d to that city with edge weight 0. This creates a new graph G'. Run Dijkstra's algorithm, starting from the dummy vertex d, on the graph G'. This yields the distance from the dummy vertex d to all cities in the network. Now, because we are only interested in the minimum distance to any city in **Berkeley** (B), we can iterate through all cities in B and take the one with the minimum distance.