TopSort. SP (G, s) // G is a DAG: (V, E), s ∈ V.  
// Finds shortest path distances from s to all other nodes.  
1. Q = TopSort(V, E) // store vertices in top sort order.
2. for all v ∈ V, v.dist = ∞, v.pred = null. (no more processed)
3. s.dist = 0
4. while (!Q.empty())
5. u = Q.dequeue()
6. for each edge (u, v)
7. update (u, v) // if (v.dist > u.dist + w_{uv})
          v.dist = u.dist + w_{uv}
          v.pred = u

ex: TopSort. SP (G, s).

r is "thrown" away (can't get to it):
RunTime: Step 1 (Top-Sort) : \( O(IV + 1E1) \)
Steps 2-3 : \( O(IV) \)
Steps 4-7: \( O(IV) + (\text{not *) } O(1E) \)

Total: \( O(2(IV + 1E)) = O(IV + 1E) \)

vs. \( O(IV \log IV + 1E \log IV) \) for Dijkstra's, \( O(IV + 1E) \) for BF.

Correctness: Similar to Dijkstra's correctness proof.

When node \( u \) is dequeued, \( u \text{-dist} \) is optimal distance to \( u \).

1. Only edges \((v,u)\) where \( v \) comes before \( u \) in the top sort order can be in the optimal \( s \rightarrow u \) path.
2. Algorithm finds optimal distance to all such nodes before setting distance for \( u \).
   Distance to \( u \) must be optimal.

Why 'known' not needed:

Cyclic:
- Node most recently deleted

Acyclic (nodes deleted in top sort order):
- Node most recently deleted

\( (5') \quad 3 \quad 1 \quad 2 \quad 0 \)

edge \((u,v)\) exists where \( u \text{-dist} < v \text{-dist} \) so don't update \( v \).

All nodes adjacent to \( u \) will have \( dist > u \text{-dist} \)
Minimum Spanning Tree

Suppose we have set of cities, roads.

- Big snowstorm hits.
- Can't afford to have all roads open, but need access b/w every pair of cities.
- Roads have costs.

Goal: Find set of roads of minimum total cost s.t. there is a path b/w every pair of cities.

What can we say about this set?

⇒ Will make up a tree ⇒ No cycles!

Why?

All three edges cannot be in minimum cost set since any two allows access to all three nodes.
\[ \text{not min cost} \quad \text{min cost} \]

\[ \begin{array}{c}
\circ \quad 5 \quad \circ \\
\circ \quad 10 \quad \circ \\
\circ \quad 2 \quad \circ
\end{array} \quad \begin{array}{c}
\circ \quad 5 \quad \circ \\
\circ \quad 1 \quad \circ \\
\circ \quad 2 \quad \circ
\end{array} \]

Cost = 24 \quad Cost = 15

Minimum Spanning Tree (MST)

\[ G = (V, E) \]
\[ V = \{ v_1, v_2, \ldots, v_n \} \]
\[ E = \{ (v_i, v_j) : v_i, v_j \in V \} \]

For \((v_i, v_j) \in E\), \(c(v_i, v_j) > 0\) is cost of \((v_i, v_j)\).

Goal: Find a subset of edges \( T \subseteq E \) such that:

\((V, T)\) is connected and total cost \( \sum_{(v_i, v_j) \in T} c(v_i, v_j) \) is minimized.

Two Greedy Approaches work!

1. Start with any node, continuously choose the cheapest node to add to current connected graph.
2. Continuously choose the cheapest edge that doesn't cause a cycle.

Start with 1) \( \Rightarrow \) how to implement?
(1) Very similar to Dijkstra's!

What changes?

Prim's-MST \((G = V, E)\)  // \(G\) is undirected weighted graph.
1. Create empty set \(T\)
2. For all \(v \in V\): \(v\.cost = \infty\), \(v\.spanned = \text{false}\), \(v\.pred = \text{null}\)
3. Choose random vertex \(s\): \(s\.cost = 0\), \(s\.spanned = \text{true}\).
4. \(Vheap\.insert(s)\)  // Min-Heap ordered by cost.

5. for \(i = 1\) to \(|V|\)
6. \(v = Vheap\.delete\text{Min}()\)
7. \(v\.spanned = \text{true}\).
8. for each neighbor \(u\) of \(v\):
9. \(\text{if } (\neg u\.spanned \text{ and } u\.cost > w_{v,u})\)
10. \(u\.cost = w_{v,u}\)  // \(Vheap\) is updated.
11. \(u\.pred = v\)
12. for all \(v\):
13. \(\text{if } (v\.pred, v) \text{ and } (v, v\.pred) \in T \text{ add } (v\.pred, v) \text{ to } T.\)

\(v = s \rightarrow c \rightarrow d \rightarrow b \rightarrow e\)

\(\text{pred} = \emptyset \quad s \rightarrow c \rightarrow d \rightarrow b \rightarrow e\)

6 edges in \(T\):
\((s, c) \quad (c, d) \quad (d, b) \quad (d, e)\).
Run Time

Same as Dijkstra's: $O(1V^110g(1V1 + 1E110g1V1))$
Prim's spans a vertex $v$ using edge $(u,v)$ such that $u \in T$ and $w_{u,v}$ is minimum over all edges $(x,v)$ such that $x \in T$. (So $v$ cost is min over all unspanned nodes).

Claim: $(u,v)$ is the best edge to span $v$.

Proof: Suppose byoc that there is a better edge $(x,v)$. Either:

(i) $x \in T$ : Then $w_{u,v} < w_{x,v}$ so $(u,v)$ is better $\Rightarrow$ contradiction.

(ii) $x \notin T$ : Then $w_{u,v} > w_{x,v}$ so $(u,v)$ is better $\Rightarrow$ contradiction.
(2) \( x \in T \):

Then must also span \( x \).

Suppose do so with cost \( c_{T,x} \). Two cases:

(a) \( c_{T,x} > w_{u,v} \)

Cost to span both \( u \) and \( x \):

\[
\text{cost using } c_{T,x} = c_{T,x} + w_{x,v}.
\]

\[
\text{where } w_{u,v} \leq w_{u,v} + w_{x,v}.
\]

Since \( c_{T,x} > w_{u,v} \)

\[
c_{T,x} + w_{x,v} > w_{u,v} + w_{x,v}.
\]
(2) $c_{T/x} < w_{u,v}$

4.99

Then $x \cdot \text{cost} < v \cdot \text{cost} \Rightarrow \text{contradiction}$.
Kruskal's (V, E)
1. Create empty set T.
2. EdgeHeap = BuildHeap(E)
3. while ( |T| < |V| - 1 )
   (u, v) = EdgeHeap. deleteMin()
   add (u, v) to T if it does not create a cycle
      (via Disjoint Sets Data Structure)

Time:
O(1E1 log(1E1)) w/o D.Sets
O(1V1 (log(1E1) + 1V1 + 1E1))
= O(1V1^2 + 1V1 1E1)

Optimized ?
Similar proof applies.
Always choosing min cost edge that connects tree to non-tree.

T = \{ (c, d), (b, d), \uparrow (s, c), (d, e) \} \neq 
not (b, c)