CS2521: Weighted Graph

N. Oren
n.oren@abdn.ac.uk

University of Aberdeen

Where are we?

- We've examined graph traversal, a basic building block of most graph algorithms.
- We've considered both directed and undirected graphs.
- But what can we do with weighted graphs?
- For example, the shortest path for a weighted graph is not computed in terms of nodes visited, but the sum of weights of edges traveresed.

Problems of Weighted Graphs

- How can we connect all vertices at minimum cost (minimum spanning tree)?
- What is the shortest path between two vertices (shortest path problem)?
- What is the maximum amount we can transfer between two nodes (maximum flow problem)?

Minimum Spanning Tree

- A spanning tree of a graph $G = (V, E)$ is a subset of edges forming a tree connecting all vertices of V.
- A minimum spanning tree (MST) is a spanning tree whose sum of edge weights is minimal.
- Sample application: connect homes using smallest amount of wire possible.
- Can a graph have more than one MST?

Minimum Spanning Tree

- A spanning tree of a graph $G = (V, E)$ is a subset of edges forming a tree connecting all vertices of V.
- A minimum spanning tree (MST) is a spanning tree whose sum of edge weights is minimal.
- Sample application: connect homes using smallest amount of wire possible.
- Can a graph have more than one MST?
- Yes - consider the spanning trees of an unweighted (or equally weighted) graph; all contain exactly $n - 1$ equal weight edges. These can all be found using DFS or BFS.
- Finding a MST for weighted graphs is more complex.

Prim's Algorithm

- This algorithm starts with one vertex, and grows the rest of the tree one edge at a time until all vertices are included.
- It is an example of a greedy algorithm — one which selects the best local choice with no regard to global structure.
- Here, the greedy algorithm repeatedly selects the smallest weight edge that will increase the number of vertices in the tree.

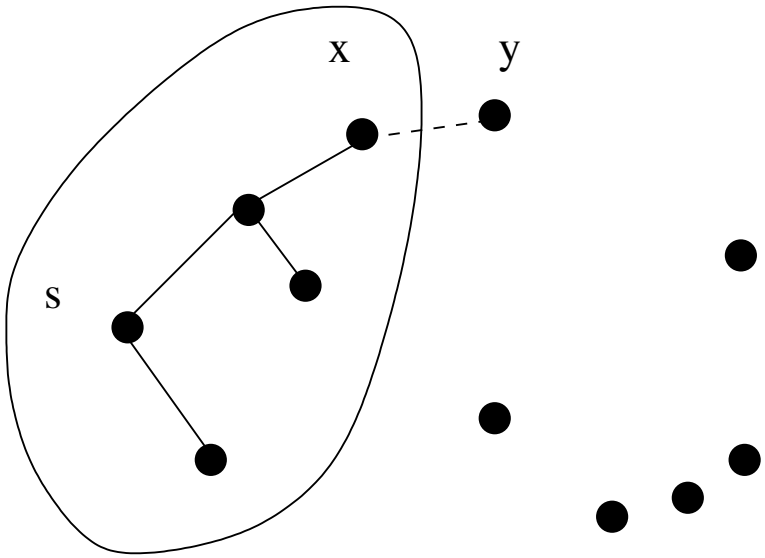
Prim's Algorithm

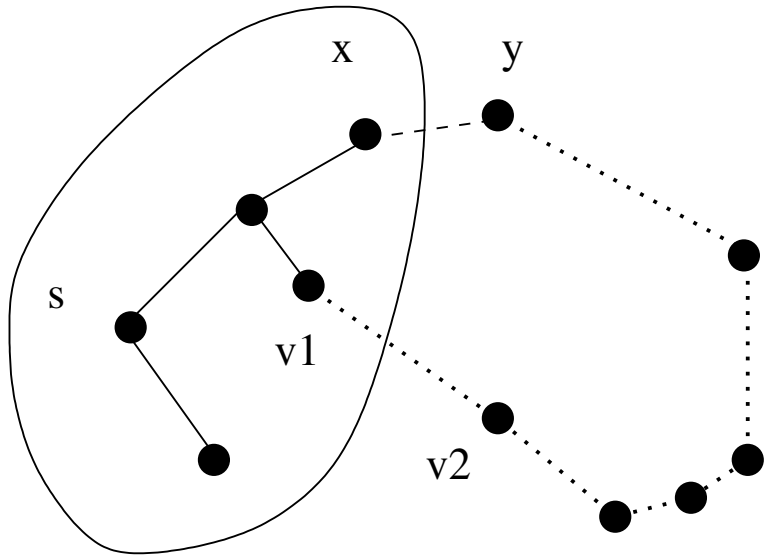
1: **function** PrimMST(G)
2: select a vertex s to start the tree from
3: **while** there are vertices not in the tree **do**
4: select the edge of minimum weight between a tree and non-tree vertex
5: add this edge and vertex to the tree

Prim's Algorithm

- The algorithm clearly creates a spanning tree — no cycle can be introduced by adding edges between tree and non-tree vertices.
- Proof (by contradiction): assume false. Then assume that adding an edge between tree and non-tree vertex creates a cycle. But then there must have been a connection to the non-tree vertex already, which is a contradiction.
- But how do we know that the result is a MST?

Proof





Proof

- Supposed that there is a graph G for which the algorithm didn't return a MST.
- So at some point we added an edge (x, y) which has greater weight than the one we should have added.
- Before this point, we had a subtree that was a MST.
- Since we are building a MST, let us assume that the correct path from x to y goes through an edge (v_1, v_2) instead, where v_1 is in the tree created by the algorithm, but v_2 is not.
- Edge (v_1, v_2) must have weight at least that of (x, y) as otherwise the algorithm would have selected it before (x, y).
- But inserting (x, y) and deleting (v_1, v_2) makes the weight of the tree no larger than before, and so selecting (x, y) is correct.
- But then we have a contradiction, and so Prim's algorithm must result in a MST.

Complexity

- The algorithm makes n iterations, considering all m edges at each iteration — $O(mn)$.
- But we can be more efficient.
- Assume that there is a vertex v not in the tree, with two edges (u, v) and (u', v) leading to it (u, u' are in the tree). If the weight of (u, v) is less than (u', v) then the latter would never be added.
- So we can ignore these "heavy" edges.
- Idea: for each vertex not in the tree, remember which vertex in the tree has the smallest weight that connects to it, and always connect this edge first. How?

Complexity

- The algorithm makes n iterations, considering all m edges at each iteration — $O(mn)$.
- But we can be more efficient.
- Assume that there is a vertex v not in the tree, with two edges (u, v) and (u', v) leading to it (u, u' are in the tree). If the weight of (u, v) is less than (u', v) then the latter would never be added.
- So we can ignore these "heavy" edges.
- Idea: for each vertex not in the tree, remember which vertex in the tree has the smallest weight that connects to it, and always connect this edge first. How?
- Priority queue

Algorithm

```
1:  function Prim(G, w, s)
2:      Q=vert(G)                                   ▷ Q is a priority queue
3:      set weights of all vertices in Q to ∞
4:      set weight of s in Q to 0                   ▷ Root of MST has min weight
5:      set parent[s]=∅                             ▷ Root has no parent
6:      while Q is not empty do
7:          u=extract-min(Q)
8:          for all v ∈ Adj[u] do                   ▷ Consider all neighbours
9:              if v ∈ Q and w(u, v) <weight of v in Q then
10:                 parent[v]=u
11:                 set weight of v in Q to w(u, v)
```

Algorithm

- Lines 2-5 initialise the algorithm
- Line 7 considers minimum weight node
- Lines 9-11 update the tree and weight of nodes (1) if they are not in MST, and (2) if they are the lowest weight node found so far adjacent to u.
- The weight of a node is the weight of the smallest edge from outside the tree (from v) to an element in the tree (u).
- Within the loop in line 8, we have a loop invariant — the weight of v is the smallest edge weight found so far.
- Given this invariant, we always select the edge of minimum weight between a tree and non-tree vertex, and hence duplicate Prim's algorithm from above.

Complexity

- Recall: priority queue's extract-min operation typically takes $\log n$ if there are n items in the queue.
- updating the key within the queue can be done in $O(1)$ (amortized).
- Given n nodes and m vertices, lines 2-5 is $O(n)$
- The while loop is repeated n times
  - $\log n$ for extract-min, so $n \log n$ in total.
  - What about the for loop (line 8)?

Complexity

- Recall: priority queue's extract-min operation typically takes $\log n$ if there are n items in the queue.
- updating the key within the queue can be done in $O(1)$ (amortized).
- Given n nodes and m vertices, lines 2-5 is $O(n)$
- The while loop is repeated n times
  - $\log n$ for extract-min, so $n \log n$ in total.
  - What about the for loop (line 8)? Since we only visit each edge once over all runnings of the for loop, and have a total of $2m$ edges in an undirected graph, total work done over while/for loop combination is $O(m)$.
- $O(m + n + n \log n) = O(m + n \log n)$
- `https://www.cs.usfca.edu/~galles/visualization/Prim.html`

Kruskal's Algorithm

- Another MST-finding algorithm, more efficient than Prim's on sparse graphs.
- Uses a priority queue Q of remaining edges, ordered by weight.
- Maintain a forest T of trees; at each step, reduce the number of trees by adding the lowest weight edge that connects two different trees.
- We must keep track of which tree the different edges belong to, and be able to merge these.
- Note, initially, every vertex belongs to its own tree.

Kruskal's Algorithm

```
1:  function Kruskal(G)
2:      T = ∅
3:      Add edges of G to Q
4:      count=0                                  ▷ number of vertices in T
5:      while count < (number of vertices in G)-1 do
6:          (u, v)=extract-min(Q)
7:          if u and v are in different trees then
8:              count++
9:              add (u, v) to T
10:             merge the trees of u and v
```

Kruskal's Algorithm

https://www.cs.usfca.edu/~galles/visualization/Kruskal.html

Algorithm Termination

- Suppose there are n vertices; note that Q is finite and an edge is eliminated at every iteration of the while loop. So termination occurs.
- Algorithm always connects vertices from separate components, so we never have a cycle.
- We also have an invariant — at the beginning and end of the loop, *count* holds the number of edges in T.
- Termination occurs when a spanning tree has been constructed.
- But is this a minimal tree?

Minimality

- Suppose we are at a point where (x, y) is added, x in tree C, and Y in tree D, giving tree T_k.
- But assume there is a MST T without (x, y) where $w(T) < w(T_k)$
- T must have a path p from x to y
- Adding (x, y) to T would give a cycle
- There is an edge (v_1, v_2) on the path which has greater weight than (x, y) (as otherwise it would have come earlier in the queue and been used in the algorithm).
- We can remove (v_1, v_2) and replace it with (x, y) to obtain a new MST T', where $w(T') < w(T)$
- So we have a contradiction.

Complexity

- Extract min: $\log m$
- Repeated m times in the while loop
- So total $O(m \log m)$
- Compare to $O(m + n \log n)$ for Prim
- Density of graph should be considered when choosing which algorithm to use

Partitioning into Components

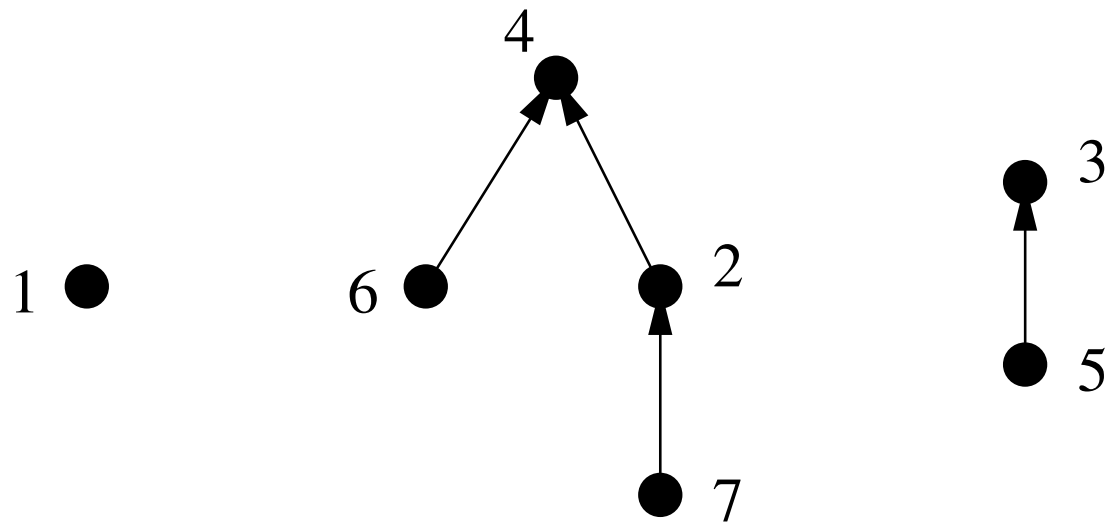
- Note that we must (1) be able to identify which tree, or component each vertex appears in within Kruskal's algorithm, and (2) be able to merge these components.
- More generally, we are considering set partitions — a partitioning of the elements of some universal set into a collection of disjoint subsets.
- What data structure can we use to determine whether two components lie in the same set, and to merge components?

The wrong approach

- If we explicitly label each element with its component number, we can test whether two elements are in the same component in constant time. But updating after a merger is a linear time operation.
- We can treat merging components as inserting an edge in a graph, but then identifying connected components requires graph traversal (linear time).

The Union Find Data Structure

- Represent each component as a backwards tree, with a pointer from a node to its parent.



- Operations:
  - Find(i) — find the root of the tree containing element i. Implemented by walking up parent pointers and returning label of root.
  - Union(i, j) — Link root of one tree (e.g., containing i) so that find(i) equals find(j).

The Union Find Data Structure

- We need to minimise tree height to ensure efficiency.
- To do so, when executing Union, it is better to make the smaller tree the subset of the big one.

The Union Find Data Structure

- We need to minimise tree height to ensure efficiency.
- To do so, when executing Union, it is better to make the smaller tree the subset of the big one.
- The height of all nodes in the "big"/root subtree stay the same, height of the nodes merged into the tree increase by one — merging the smaller tree leaves height unchanged on the larger set of vertices.

Union Find - analysis

- Consider smallest possible tree (height h). Single node tree has height 1; smallest height 2 tree has 2 nodes, obtained from union of single node trees.
- Merging in more single node trees won't increase height beyond 2. Only if two height 2 trees are merged do we get a height 3 tree, with (at least) 4 nodes.
- At worst, we double the number of nodes in the tree to get an extra unit of height.
- At worst, for n nodes, $\log n$ doublings are needed, so Union and Find will operate in $O(\log n)$. This is "good enough" for Kruskal's algorithm as extract-min is $O(m \log m)$.
- Can be done faster (but we won't consider how).

Where are we?

- We've considered MSTs, but other variants exist, e.g., maximum spanning tree; minimum product spanning tree; minimum bottleneck spanning tree. But we ignore these here (see Skiena for details).
- A MST connects all nodes as cheaply as possible.
- But what about getting from one point to another as cheaply as possible?
- This is the shortest path problem.

Shortest Path

- For an unweighted graph, shortest path from node s to node t can be found using BFS from s.
- This doesn't work for weighted graphs — shortest path might use many (cheap) edges, rather than one (expensive) edge.
- Dijkstra's algorithm is the classic approach to finding shortest path in weighted graphs.
- The algorithm finds the shortest path from s to every other vertex in the graph (including t).

Dijkstra's Algorithm

- Assume there is a shortest path from s to t going through x.
- This path contains the shortest path from s to x as a subpath.
  - Assume false. Then there is a shorter path from s to x; by using this path, the path from s to t will be shorter, contradiction.
- So Dijkstra's algorithm operates by, in each round, identifying a shortest path from s to some new vertex x.
- x is the vertex that minimises $dist(s, v_i) + w(v_i, x)$ where
  - $dist(s, v_i)$ is the (known) shortest distance from s to v_i.
  - $w(v_i, x)$ is the weight of the (v_i, x) edge.
- Thus, $dist(s, x) = dist(s, v_i) + w(v_i, x)$

Dijkstra's Algorithm

1: **function** ShortestPath-Dijkstra((V,E),s,t)
2: $known = \{s\}$ ▷ Nodes already visited
3: **for all** $i = 1$ to $|V|$ **do**
4: $dist[i] = \infty$ ▷ minimum distance to node i
5: **for all** edges (s, v) **do**
6: $dist[v] = w(s, v)$
7: $last = s$ ▷ last node considered
8: **while** $last \neq t$ **do**
9: select v_n, the vertex minimising $dist[v]$, such that $v_n \notin known$
10: **for all** edges (v_n, x) **do**
11: $dist[x] = min\{dist[x], dist[v_n] + w(v_n, x)\}$
12: $last = v_n$
13: $known = known \cup v_n$

Dijkstra's Algorithm

- There is a strong similarity between Dijkstra's algorithm and Prim's.
- In both we add one vertex every round, which is optimal.
- In both, we track the best path seen to date for all vertices outside the tree, adding them in order of increasing cost.
- Difference is how vertices are ranked.
- In MST, we care about weight of next potential tree edge.
- In shortest path, we include closest outside vertex.
- Note: Skiena shows how Dijkstra can be implemented by changing 3 lines in the implementaton of Prim's algorithm.

Dijkstra — Analysis

- Simple implementation is $O(n^2)$, as per simple Prim algorithm.
- $dist[x]$ stores distance from s to x. But how do we find the actual path?
- We can — as per Prim — introduce parent pointers and follow them backwards.
- The algorithm only works when no edges have negative weights.
- Such negative edges change the cheapest way to get from s to some other vertex already in the tree.

Floyd's Algorithm

- Suppose we want to find the shortest path between all pairs of vertices in a graph.
- This is useful when we are trying to work out average/worst case times to traverse a graph (e.g., to address congestion, or work out how long a letter might take to arrive).
- We could solve this all-pairs shortest path problem by calling Dijkstra's algorithm for each of the n starting vertices ($O(n^3)$).
- Floyd's algorithm is an alternative, constructing a $n \times n$ distance matrix.
- Best to run on an adjacency matrix rather than adjacency list. Weights of missing edges should be ∞ rather than 0.

Floyd's Algorithm

- Define $W[i,j]^k$ to be the length of the shortest path from i to j using only vertices $1, 2, \ldots k$ as intermediates.
- Note that $W[i,j]^0$ does not allow intermediate vertices, so this will be the adjacency matrix.
- For $W[i,j]^1$, we will also allow paths going through node 1; if a path between i and j going through this node is possible, it will replace edges with weight ∞.
- Similar argument applies for $W[i,j]^2$, $\ldots$ $W[i,j]^n$
- The update rule is thus

$$W[i,j]^k = \min(W[i,j]^{k-1}, W[i,k]^{k-1} + W[k,j]^{k-1})$$

Floyd's Algorithm

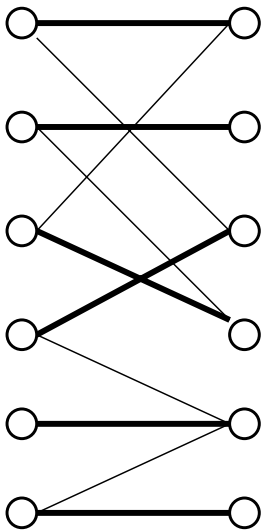
1: **function** FloydWarshall(W) ▷ W is an adjacency matrix representation of a weighted graph
2: **for all** $k = 1 \ldots n$ **do**
3: **for all** $i = 1 \ldots n$ **do**
4: **for all** $j = 1 \ldots n$ **do**
5: $W[i,j] = \min(W[i,j], W[i,k] + W[k,j])$

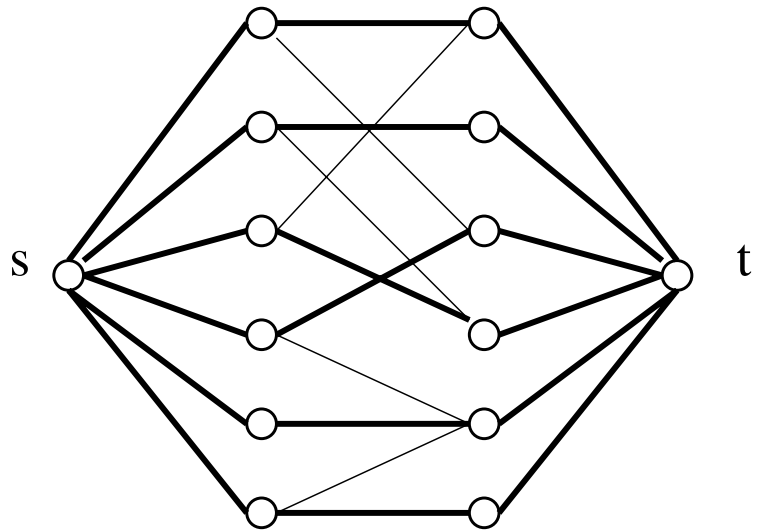
Floyd's Algorithm

- Complexity is still $O(n^3)$
- But loop is so tight, and program so short that in practice this usually runs faster than Dijkstra for all pairs.
- We need to retain another matrix to store parent choices if we want to recreate paths.
- But the algorithm is usually used to only determine distances.

Network Flow

- An edge weighted graph could represent a series of pipes with different capacities.
- The network flow problem asks what is the maximum possible amount of flow between s and t, respecting pipe capacities.
- While interesting, it is useful in solving other important graph problems, e.g., bipartite matching.
- A matching in a graph $G = (V, E)$ is a subset of edges $E' \subseteq E$ such that no two edges of E' share a vertex.





Bipartite Graphs and Matchings

- A graph is bipartite or two-colourable if its vertices can be divided into two disjoint sets where every edge within the graph links an element from one set to another.
- Many real world problems can be captured using bipartite graphs, e.g., allocating jobs to people, or identifying romantic partners. A matching represents these links.
- The largest pipartite matching can be found using network flow.
  - Create a source node s connected to every vertex in one set with an edge weight of 1.
  - Create a sink node t connected to every vertex in the other set with an edge weight of 1.
  - Assign each edge in the bipartite graph a weight of 1.
  - The maximum possible flow from s to t defines the largest matching in the graph.
- But we still need to identify the maximum flow.

Bipartite Graphs and Matchings

- A graph is bipartite or two-colourable if its vertices can be divided into two disjoint sets where every edge within the graph links an element from one set to another.
- Many real world problems can be captured using bipartite graphs, e.g., allocating jobs to people, or identifying romantic partners. A matching represents these links.
- The largest pipartite matching can be found using network flow.
  - Create a source node s connected to every vertex in one set with an edge weight of 1.
  - Create a sink node t connected to every vertex in the other set with an edge weight of 1.
  - Assign each edge in the bipartite graph a weight of 1.
  - The maximum possible flow from s to t defines the largest matching in the graph.
- But we still need to identify the maximum flow.
- Which we won't do here.

Where are we?

- We've examined forming MSTs over weighted
  - Prim's Algorithm
  - Kruskal's algorithm
- And examined some optimisations.
- We've also considered the problem of finding a shortest path between nodes
  - Dijkstra's algorithm
  - Floyd-Warshall algorithm (all shortest paths)
- Briefly looked at the importance of network flows.