Solutions to Homework Practice Problems

Problem 1: Dijkstra's Algorithm

This is not covered in the lectures because it's the sort of thing many of you have seen before, possibly multiple times (GT undergrads see it at least 3 times in their algorithms, data structures, and combinatorics class). If you haven't seen it, or need a refresher, look at Chapter 4.4 of [DPV].

(1) What is the input and output for Dijkstra's algorithm?

The inputs for Dijkstra's algorithm are a graph $G = (V, E)$ with positive weights l_e for each edge $e \in E$, along with a source vertex s. (The weights must be positive in order for the algorithm to work.)
The outputs of Dijkstra's algorithm are the shortest paths from the source vertex to all other vertices of the graph. Specifically, Dijkstra's algorithm will output a `dist` array (containing the shortest distance from the source to each vertex) and a `prev` array (indicating the previous vertex that the shortest path uses to get to each vertex). The `prev` array can be used to construct the shortest paths.
(As a historical note, Dijkstra's original formulation of the algorithm found the shortest path between two input nodes, but the algorithm can easily find the shortest path to all vertices from a source by continuing the search until no more outgoing edges exist, instead of stopping after the target vertex is found. By now, Dijkstra's algorithm usually refers to the source-to-all variant, since it also has the same runtime.)

(2) What is the running time of Dijkstra's algorithm using min-heap (aka priority queue) data structure? (Don't worry about the Fibonacci heap implementation or running time using it.)

The running time of Dijkstra's algorithm using a min-heap is $O((|V| + |E|) \log |V|)$. This comes from the binary heap requiring $O(\log |V|)$ operations to either insert a key or remove the minimum key. Dijkstra's requires $O(|V|)$ key removals and $O(|V| + |E|)$ key insertions, yielding the $O((|V| + |E|) \log |V|)$ runtime.

(3) What is the main idea for Dijkstra's algorithm?

The main idea of Dijkstra's algorithm is to find the distances to the nearest nodes, then gradually expand the frontier of the search and determine the shortest paths to nodes that are further and further away. Since edge weights must be positive, there will never be a case where the shortest path must first go through a far away vertex (the path would already be longer than another one that had been found).

[DPV] 4.14 shortest path through a given vertex

> You are given a strongly connected directed graph $G = (V, E)$ with positive edge weights along with a particular node $v_0 \in V$. Give an efficient algorithm for finding shortest paths between all pairs of nodes, with the one restriction that these paths must all pass through v_0.

Run Dijkstra's algorithm from v_0 to get all distances from v_0 to all vertices in V. Reverse the graph and run Dijkstra's again from v_0. Observe that the output corresponds to the distances of the path from all other vertices to v_0 in the original graph. For any pair of vertices $u, w \in V$, the shortest distance from u to w using a path through v_0 will be the sum of the two distances, u to v_0 in the reverse graph and v_0 to w in the original graph. Any single path $u \rightsquigarrow v_0 \rightsquigarrow w$ may be recovered using the $prev[]$ arrays which result from each run of Dijkstra.

Why this works: our graph is strongly connected, meaning that a path exists between every pair of vertices. We also know that Dijkstra's, given a starting vertex, will find the shortest path from that starting point to every other vertex. When we reverse a directed graph we are essentially reversing the direction of the path between any pair of vertices. So the shortest path from u to w which includes v_0 is the combination of the shortest path from u to v_0 in the reversed graph plus the shortest distance from v_0 to w in the original graph.

Each round of Dijkstra's takes $O((m+n)\log(n))$ - this runtime can be simplified to $O(m \log n)$ since the graph is strongly connected. Building the reverse graph takes $(On + m)$ time. Explicitly calculating the pairwise shortest distances between each pair of vertices would take $O(n^2)$ time, resulting in an overall runtime of $O(n^2 + (m+n)log(n))$, or $O(n^2 + m \log n)$ (if simplified) .

[DPV] Problems 5.1, 5.2 (Practice fundamentals of MST designs)

5.1
(a) the cost is 19
(b) there are 2 possible MSTs
(c)

Edge included	Cut
AE	{A,B,C,D} & {E,F,G,H}
EF	{A,B,C,D,E} & {F,G,H}
BE	{A,E, F,G,H} & {B,C,D}
FG	{A,B,E,F} & {C,D,G,H}
GH	{A,B,E,F,G} & {C,D,H}
CG	{A,B,E,F,G,H} & {C,D}
GD	{A,B,C,E,F,G,H} & {D}

5.2 (a)

Vertex included	Edge included	Cost
A		0
B	AB	1
C	BC	3
G	CG	5
D	GD	6
F	GF	7
H	GH	8
E	AE	12

5.2 (b)
Here are the values for the parent pointer π at each iteration of Kruskals. From this you should be able to deduce the disjoint-sets.

Union	Values of π for each vertex
Start	[A, B, C, D, E, F, G, H]
(A,B)	[B, B, C, D, E, F, G, H]
(F,G)	[B, B, C, D, E, G, G, H]
(D,G)	[B, B, C, G, E, G, G, H]
(G,H)	[B, B, C, G, E, G, G, G]
(C,G)	[B, B, G, G, E, G, G, G]
(B,C)	[B, G, G, G, E, G, G, G]
(A,E)	[G, G, G, G, G, G, G, G]

[DPV] Problem 5.9

(a) **False**. Consider a graph where a vertex is adjacent to a single edge
(b) **True**. Consider the order in which edges would be processed by Kruskal's
(c) **True**. A minimum weight edge would be a candidate for at least one possible MST
(d) **True**. The *Cut Property* assures this