Weighted Graph Algorithms

Beyond DFS/BFS exists an alternate universe of algorithms for *edge-weighted graphs*. Our adjacency list representation quietly supported these graphs:

typedef struct {
    int y;
    int weight;
    struct edgenode *next;
} edgenode;
Minimum Spanning Trees

A tree is a connected graph with no cycles. A spanning tree is a subgraph of $G$ which has the same set of vertices of $G$ and is a tree.

A minimum spanning tree of a weighted graph $G$ is the spanning tree of $G$ whose edges sum to minimum weight. There can be more than one minimum spanning tree in a graph — consider a graph with identical weight edges.
Why Minimum Spanning Trees?

The minimum spanning tree problem has a long history – the first algorithm dates back at least to 1926!. Minimum spanning tree is always taught in algorithm courses since (1) it arises in many applications, (2) it is an important example where greedy algorithms always give the optimal answer, and (3) Clever data structures are necessary to make it work.

In greedy algorithms, we make the decision of what next to do by selecting the best local option from all available choices – without regard to the global structure.
Applications of Minimum Spanning Trees

Minimum spanning trees are useful in constructing networks, by describing the way to connect a set of sites using the smallest total amount of wire. Minimum spanning trees provide a reasonable way for clustering points in space into natural groups. What are natural clusters in the friendship graph?
One of the war stories in the text describes how to partition a graph into compact subgraphs by deleting large edges from the minimum spanning tree.
Minimum Spanning Trees and TSP

When the cities are points in the Euclidean plane, the minimum spanning tree provides a good heuristic for traveling salesman problems.
The optimum traveling salesman tour is at most twice the length of the minimum spanning tree.

Note: There can be more than one minimum spanning tree considered as a group with identical weight edges.
Prim’s Algorithm

If $G$ is connected, every vertex will appear in the minimum spanning tree. If not, we can talk about a minimum spanning forest.

Prim’s algorithm starts from one vertex and grows the rest of the tree an edge at a time.

As a greedy algorithm, which edge should we pick? The cheapest edge with which can grow the tree by one vertex without creating a cycle.
Prim’s Algorithm (Pseudocode)

During execution each vertex $v$ is either in the tree, *fringe* (meaning there exists an edge from a tree vertex to $v$) or *unseen* (meaning $v$ is more than one edge away).

**Prim-MST(G)**

- Select an arbitrary vertex $s$ to start the tree from.
- While (there are still non-tree vertices)
  - Select the edge of minimum weight between a tree and a non-tree vertex.
  - Add the selected edge and vertex to the tree $T_{prim}$.

This creates a spanning tree, since no cycle can be introduced, but is it minimum?
Prim’s Algorithm in Action

G

A

7 9 4 2 3 7 12

5 7 4

1

A

Prim(G,A)

1 2 6

4 3

5

A

Kruskal(G)

6 2 1

3 4

5
Why is Prim Correct?

We use a proof by contradiction:
Suppose Prim’s algorithm does not always give the minimum cost spanning tree on some graph.
If so, there is a graph on which it fails.
And if so, there must be a first edge \((x, y)\) Prim adds such that the partial tree \(V'\) cannot be extended into a minimum spanning tree.
But if $(x, y)$ is not in $MST(G)$, then there must be a path in $MST(G)$ from $x$ to $y$ since the tree is connected. Let $(v, w)$ be the first edge on this path with one edge in $V'$. Replacing it with $(x, y)$ we get a spanning tree with smaller weight, since $W(v, w) > W(x, y)$. Thus you did not have the MST!!
Prim’s Algorithm is correct!

Thus we cannot go wrong with the greedy strategy the way we could with the traveling salesman problem.
How Fast is Prim’s Algorithm?

That depends on what data structures are used. In the simplest implementation, we can simply mark each vertex as tree and non-tree and search always from scratch:

Select an arbitrary vertex to start.
While (there are non-tree vertices)
    select minimum weight edge between tree and fringe
    add the selected edge and vertex to the tree

This can be done in $O(nm)$ time, by doing a DFS or BFS to loop through all edges, with a constant time test per edge, and a total of $n$ iterations.
Prim’s Implementation

To do it faster, we must identify fringe vertices and the minimum cost edge associated with it fast.

```c
prim(graph *g, int start)
{
    int i; (* counter *)
    edgenode *p; (* temporary pointer *)
    bool intree[MAXV]; (* is the vertex in the tree yet? *)
    int distance[MAXV]; (* distance vertex is from start *)
    int v; (* current vertex to process *)
    int w; (* candidate next vertex *)
    int weight; (* edge weight *)
    int dist; (* best current distance from start *)

    for (i=1; i<=g->nvertices; i++) {
        intree[i] = FALSE;
        distance[i] = MAXINT;
        parent[i] = -1;
    }

    distance[start] = 0;
}```
v = start;

while (intree[v] == FALSE) {
    intree[v] = TRUE;
    p = g -> edges[v];
    while (p != NULL) {
        w = p -> y;
        weight = p -> weight;
        if ((distance[w] > weight) && (intree[w] == FALSE)) {
            distance[w] = weight;
            parent[w] = v;
        }
        p = p -> next;
    }
}

v = 1;
dist = MAXINT;
for (i=1; i<= g -> nvertices; i++)
    if ((intree[i] == FALSE) && (dist > distance[i])) {
        dist = distance[i];
        v = i;
    }
}
Prim’s Analysis

Finding the minimum weight fringe-edge takes $O(n)$ time – just bump through fringe list.
After adding a vertex to the tree, running through its adjacency list to update the cost of adding fringe vertices (there may be a cheaper way through the new vertex) can be done in $O(n)$ time.
Total time is $O(n^2)$. 
Kruskal’s Algorithm

Since an easy lower bound argument shows that every edge must be looked at to find the minimum spanning tree, and the number of edges $m = O(n^2)$, Prim’s algorithm is optimal in the worst case. Is that all she wrote? The complexity of Prim’s algorithm is independent of the number of edges. Can we do better with sparse graphs? Yes! Kruskal’s algorithm is also greedy. It repeatedly adds the smallest edge to the spanning tree that does not create a cycle.
Kruskal’s Algorithm in Action
Why is Kruskal’s algorithm correct?

Again, we use proof by contradiction. Suppose Kruskal’s algorithm does not always give the minimum cost spanning tree on some graph. If so, there is a graph on which it fails. And if so, there must be a first edge \((x, y)\) Kruskal adds such that the set of edges cannot be extended into a minimum spanning tree. When we added \((x, y)\) there previously was no path between \(x\) and \(y\), or it would have created a cycle. Thus if we add \((x, y)\) to the optimal tree it must create a cycle. At least one edge in this cycle must have been added after \((x, y)\), so it must have a heavier weight.
Deleting this heavy edge leave a better MST than the optimal tree? A contradiction!
How fast is Kruskal’s algorithm?

What is the simplest implementation?

- Sort the $m$ edges in $O(m \lg m)$ time.
- For each edge in order, test whether it creates a cycle the forest we have thus far built – if so discard, else add to forest. With a BFS/DFS, this can be done in $O(n)$ time (since the tree has at most $n$ edges).

The total time is $O(mn)$, but can we do better?
Fast Component Tests Give Fast MST

Kruskal’s algorithm builds up connected components. Any edge where both vertices are in the same connected component create a cycle. Thus if we can maintain which vertices are in which component fast, we do not have test for cycles!

- **Same component**\((v_1, v_2)\) – Do vertices \(v_1\) and \(v_2\) lie in the same connected component of the current graph?
- **Merge components**\((C_1, C_2)\) – Merge the given pair of connected components into one component.
Fast Kruskal Implementation

Put the edges in a heap

\[ \text{count} = 0 \]

while (\text{count} < n - 1) do
  get next edge \((v, w)\)
  if (\text{component}(v) \neq \text{component}(w))
    add to T
    \text{component}(v) = \text{component}(w)

If we can test components in \(O(\log n)\), we can find the MST in \(O(m \log m)\)!

Question: Is \(O(m \log n)\) better than \(O(m \log m)\)?
Union-Find Programs

We need a data structure for maintaining sets which can test if two elements are in the same and merge two sets together. These can be implemented by \textit{union} and \textit{find} operations, where

- \textit{Find}(i) – Return the label of the root of tree containing element \textit{i}, by walking up the parent pointers until there is no where to go.

- \textit{Union}(i,j) – Link the root of one of the trees (say containing \textit{i}) to the root of the tree containing the other (say \textit{j}) so \textit{find}(i) now equals \textit{find}(j).
Same Component Test

\[ \text{Is } s_i \equiv s_j \]
\[ t = \text{Find}(s_i) \]
\[ u = \text{Find}(s_j) \]

Return (Is \( t = u \)?)
**Merge Components Operation**

Make $s_i \equiv s_j$

\[
t = d(s_i)
\]

\[
u = d(s_j)
\]

Union($t, u$)
We are interested in minimizing the time it takes to execute any sequence of unions and finds. A simple implementation is to represent each set as a tree, with pointers from a node to its parent. Each element is contained in a node, and the name of the set is the key at the root:
Worst Case for Union Find

In the worst case, these structures can be very unbalanced:

For $i = 1$ to $n/2$ do
  UNION($i$, $i+1$)

For $i = 1$ to $n/2$ do
  FIND(1)
Who’s The Daddy?

We want the limit the height of our trees which are effected by union’s. When we union, we can make the tree with fewer nodes the child.

Since the number of nodes is related to the height, the height of the final tree will increase only if both subtrees are of equal
height!
If $\text{Union}(t, v)$ attaches the root of $v$ as a subtree of $t$ iff the number of nodes in $t$ is greater than or equal to the number in $v$, after any sequence of unions, any tree with $h/4$ nodes has height at most $\lceil \lg h \rceil$. 
Proof

By induction on the number of nodes $k$, $k = 1$ has height 0. Let $d_i$ be the height of the tree $t_i$

If $(d_1 > d_2)$ then $d = d_1 \leq \lceil \log k_1 \rceil \leq \lceil \log(k_1 + k_2) \rceil = \lceil \log k \rceil$

If $(d_1 \leq d_2)$, then $k_1 \geq k_2$.

$d = d_2 + 1 \leq \lceil \log k_2 \rceil + 1 = \lceil \log 2k_2 \rceil \leq \lceil \log(k_1 + k_2) \rceil = \log k$
Can we do better?

We can do unions and finds in $O(\log n)$, good enough for Kruskal’s algorithm. But can we do better? The ideal Union-Find tree has depth 1:

On a find, if we are going down a path anyway, why not change the pointers to point to the root?
This path compression will let us do better than $O(n \log n)$ for $n$ union-finds.

$O(n)$? Not quite … Difficult analysis shows that it takes $O(n \alpha(n))$ time, where $\alpha(n)$ is the inverse Ackerman function and $\alpha($number of atoms in the universe$)= 5.$
Problem of the Day

Suppose we are given the minimum spanning tree $T$ of a given graph $G$ (with $n$ vertices and $m$ edges) and a new edge $e = (u, v)$ of weight $w$ that we will add to $G$. Give an efficient algorithm to find the minimum spanning tree of the graph $G + e$. Your algorithm should run in $O(n)$ time to receive full credit, although slower but correct algorithms will receive partial credit.
Finding the shortest path between two nodes in a graph arises in many different applications:

- Transportation problems – finding the cheapest way to travel between two locations.
- Motion planning – what is the most natural way for a cartoon character to move about a simulated environment.
- Communications problems – how long will it take for a message to get between two places? Which two locations are furthest apart, i.e., what is the diameter of the network.
Shortest Paths and Sentence Disambiguation

In our work on reconstructing text typed on an (overloaded) telephone keypad, we had to select which of many possible interpretations was most likely. We constructed a graph where the vertices were the possible words/positions in the sentence, with an edge between possible neighboring words.
GIVE ME A RING.
The weight of each edge is a function of the probability that these two words will be next to each other in a sentence. ‘hive me’ would be less than ‘give me’, for example.

The final system worked extremely well – identifying over 99% of characters correctly based on grammatical and statistical constraints.
Dynamic programming (the Viterbi algorithm) can be used on the sentences to obtain the same results, by finding the shortest paths in the underlying DAG.
Shortest Paths: Unweighted Graphs

In an unweighted graph, the cost of a path is just the number of edges on the shortest path, which can be found in $O(n+m)$ time via breadth-first search.

In a weighted graph, the weight of a path between two vertices is the sum of the weights of the edges on a path. BFS will not work on weighted graphs because sometimes visiting more edges can lead to shorter distance, i.e., $1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 < 10$.

Note that there can be an exponential number of shortest paths between two nodes – so we cannot report all shortest paths efficiently.
Negative Edge Weights

Note that negative cost cycles render the problem of finding the shortest path meaningless, since you can always loop around the negative cost cycle more to reduce the cost of the path. Thus in our discussions, we will assume that all edge weights are positive. Other algorithms deal correctly with negative cost edges. Minimum spanning trees are uneffected by negative cost edges.
Dijkstra’s Algorithm

The principle behind Dijkstra’s algorithm is that if \( s, \ldots, x, \ldots, t \) is the shortest path from \( s \) to \( t \), then \( s, \ldots, x \) had better be the shortest path from \( s \) to \( x \).

This suggests a dynamic programming-like strategy, where we store the distance from \( s \) to all nearby nodes, and use them to find the shortest path to more distant nodes.
Initialization and Update

The shortest path from $s$ to $s$, $d(s, s) = 0$. If all edge weights are positive, the *smallest* edge incident to $s$, say $(s, x)$, defines $d(s, x)$.

We can use an array to store the length of the shortest path to each node. Initialize each to $\infty$ to start. Soon as we establish the shortest path from $s$ to a new node $x$, we go through each of its incident edges to see if there is a better way from $s$ to other nodes thru $x$. 
Pseudocode: Dijkstra’s Algorithm

\[
\begin{align*}
\text{known} & = \{s\} \\
\text{for } i = 1 \text{ to } n, \quad \text{dist}[i] & = \infty \\
\text{for each edge } (s, v), \quad \text{dist}[v] & = d(s, v) \\
\text{last} & = s \\
\text{while } (\text{last} \neq t) \\
& \quad \text{select } v \text{ such that } \text{dist}(v) = \min_{\text{unknown}} \text{dist}(i) \\
& \quad \text{for each } (v, x), \quad \text{dist}[x] = \min(\text{dist}[x], \text{dist}[v] + w(v, x)) \\
& \quad \text{last} = v \\
\text{known} & = \text{known} \cup \{v\}
\end{align*}
\]

Complexity \( \rightarrow O(n^2) \).

This is essentially the same as Prim’s algorithm.
Dijkstra Example
Dijkstra’s Implementation

See how little changes from Prim’s algorithm!

```c
void dijkstra(graph *g, int start) (* WAS prim(g,start) *)
{
    int i; (* counter *)
    edgenode *p; (* temporary pointer *)
    bool intree[MAXV]; (* is the vertex in the tree yet? *)
    int distance[MAXV]; (* distance vertex is from start *)
    int v; (* current vertex to process *)
    int w; (* candidate next vertex *)
    int weight; (* edge weight *)
    int dist; (* best current distance from start *)

    for (i=1; i<=g->nvertices; i++) {
        intree[i] = FALSE;
        distance[i] = MAXINT;
        parent[i] = -1;
    }

    distance[start] = 0;
    v = start;
```
while (intree[v] == FALSE) {
    intree[v] = TRUE;
    p = g->edges[v];
    while (p != NULL) {
        w = p->y;
        weight = p->weight;
        (* CHANGED *) if (distance[w] > (distance[v]+weight)) {
            (* CHANGED *) distance[w] = distance[v]+weight;
            (* CHANGED *) parent[w] = v;
        }
        p = p->next;
    }
    v = 1;
    dist = MAXINT;
    for (i=1; i <= g->nvertices; i++)
        if (((intree[i] == FALSE) && (dist > distance[i]))) {
            dist = distance[i];
            v = i;
        }
}
}
Prim’s/Dijkstra’s Analysis

Finding the minimum weight fringe-edge takes $O(n)$ time – just bump through fringe list. After adding a vertex to the tree, running through its adjacency list to update the cost of adding fringe vertices (there may be a cheaper way through the new vertex) can be done in $O(n)$ time. Total time is $O(n^2)$. 
Improved Time Bounds

An $O(m \lg n)$ implementation of Dijkstra’s algorithm would be faster for sparse graphs, and comes from using a heap of the vertices (ordered by distance), and updating the distance to each vertex (if necessary) in $O(\lg n)$ time for each edge out from freshly known vertices.

Even better, $O(n \lg n + m)$ follows from using Fibonacci heaps, since they permit one to do a decrease-key operation in $O(1)$ amortized time.
All-Pairs Shortest Path

Notice that finding the shortest path between a pair of vertices \((s, t)\) in worst case requires first finding the shortest path from \(s\) to all other vertices in the graph. Many applications, such as finding the center or diameter of a graph, require finding the shortest path between all pairs of vertices.

We can run Dijkstra’s algorithm \(n\) times (once from each possible start vertex) to solve all-pairs shortest path problem in \(O(n^3)\). Can we do better?
Dynamic Programming and Shortest Paths

The four-step approach to dynamic programming is:

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution.
3. Compute this recurrence in a bottom-up fashion.
4. Extract the optimal solution from computed information.
Initialization

From the adjacency matrix, we can construct the following matrix:

\[
D[i, j] = \begin{cases} 
\infty, & \text{if } i \neq j \text{ and } (v_i, v_j) \text{ is not in } E \\
\omega(i, j), & \text{if } (v_i, v_j) \in E \\
0, & \text{if } i = j 
\end{cases}
\]

This tells us the shortest path going through no intermediate nodes.
There are several ways to characterize the shortest path between two nodes in a graph. Note that the shortest path from \( i \) to \( j \), \( i \neq j \), using at most \( M \) edges consists of the shortest path from \( i \) to \( k \) using at most \( M - 1 \) edges plus \( W(k, j) \) for some \( k \).

This suggests that we can compute all-pair shortest path with an induction based on the number of edges in the optimal path.
Recurrence on Path Length

Let $d[i, j]^m$ be the length of the shortest path from $i$ to $j$ using at most $m$ edges.

What is $d[i, j]^0$?

$$d[i, j]^0 = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$$

What if we know $d[i, j]^{m-1}$ for all $i, j$?

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

$$= \min(d[i, k]^{m-1} + w[k, j]), 1 \leq k \leq i$$

since $w[k, k] = 0$
Not Floyd Implementation

This gives us a recurrence, which we can evaluate in a bottom up fashion:

\[
\text{for } i = 1 \text{ to } n \\
\quad \text{for } j = 1 \text{ to } n \\
\quad d[i, j]^m = \infty \\
\quad \text{for } k = 1 \text{ to } n \\
\quad \quad d[i, j]_0^0 = \min( d[i, k]^m, d[i, k]^{m-1} + d[k, j] )
\]
Time Analysis – Bummer

This is an $O(n^3)$ algorithm just like matrix multiplication, but it only goes from $m$ to $m + 1$ edges. Since the shortest path between any two nodes must use at most $n$ edges (unless we have negative cost cycles), we must repeat that procedure $n$ times ($m = 1$ to $n$) for an $O(n^4)$ algorithm. Although this is slick, observe that even $O(n^3 \log n)$ is slower than running Dijkstra’s algorithm starting from each vertex!
The Floyd-Warshall Algorithm

An alternate recurrence yields a more efficient dynamic programming formulation. Number the vertices from 1 to \( n \).

Let \( d[i, j]^k \) be the shortest path from \( i \) to \( j \) using only vertices from 1, 2, ..., \( k \) as possible intermediate vertices.

What is \( d[j, j]^0 \)? With no intermediate vertices, any path consists of at most one edge, so \( d[i, j]^0 = w[i, j] \).
Recurrence Relation

In general, adding a new vertex $k + 1$ helps iff a path goes through it, so

$$d[i, j]^k = w[i, j] \text{ if } k = 0$$

$$= \min(d[i, j]^{k-1}, d[i, k]^{k-1} + d[k, j]^{k-1}) \text{ if } k \geq 1$$

Although this looks similar to the previous recurrence, it isn’t.
Implementation

The following algorithm implements it:

\[ d^0 = w \]

for \( k = 1 \) to \( n \)

for \( i = 1 \) to \( n \)

for \( j = 1 \) to \( n \)

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

This obviously runs in \( \Theta(n^3) \) time, which is asymptotically no better than \( n \) calls to Dijkstra’s algorithm. However, the loops are so tight and it is so short and simple that it runs better in practice by a constant factor.
Problem of the Day

The *single-destination shortest path* problem for a directed graph is to find the shortest path *from* every vertex to a specified vertex $v$. Give an efficient algorithm to solve the single-destination shortest paths problem.
Problem of the Day

Let $G$ be a weighted directed graph with $n$ vertices and $m$ edges, where all edges have positive weight. A directed cycle is a directed path that starts and ends at the same vertex and contains at least one edge. Give an $O(n^3)$ algorithm to find a directed cycle in $G$ of minimum total weight. Partial credit will be given for an $O(n^2m)$ algorithm.