September 2, 2013

Graphs 2: Networks

Undirected or directed graphs often model problems by enhancing themselves with useful structural information, such as coloring of vertices (bipartite graphs and the map coloring problem are two famous examples here) and adding weight or "cost" to edges. This chapter concerns the latter cases, which loosely form the study of (undirected or directed) networks. We will first consider the undirected case and look at two algorithms for finding a minimum spanning tree (which is a spanning tree with minimal total weight). Kruskal's MST algorithm ties to the spanning tree algorithm introduced as an application of the Partition/Union-Find ADT.

Prim's MST algorithm starts us on a more general path that can be generalized to directed networks yielding Dijkstra's single-source-shortest-paths algorithm along the way to the A* algorithm of AI fame.

Outside the Box

Best Lecture Slides	PU1
	PU2
Best Video	UCB
Kruskal Demo	whfreeman

Textbook deconstruction. This notes chapter corresponds roughly to Chapter 23 and 24 in [Cormen et al 3e]. What is missing there and here is a wide description of applications and accompanying illustrations. The Sedgewick-Wayne slides from Princeton are excellent resources in general, but particularly for applications and graphics. The Berkeley lectures start out with material from breadth-first search, but there is also a nice discussion of Kruskal in them.

Weighted graphs are good models for many kinds of systems and networks, such as:

Transportation networks: roads, airline routes, UPS delivery routes

Transmission networks: pipelines, electrical grid, telephone, TV Cable, hydraulic control systems

Communication networks: CPU chips, ethernet, cell phone wireless, Internet...

plus many applications in other areas where graphs model things like pixels in an image and points in cluster analysis. We will study just a sample of important algorithms, finding a spanning tree of a weighted graph whose total cost or weight is minimized. Such a minimum spanning tree would, for example, represent the lowest resistance power supply to a collection of points on a grid or the shortest communication pathways among a set of nodes.

1 Weighted Graphs

A *weighted graph* is a directed or undirected graph $G = (V, E)$ and a real-valued function $w : E \to \mathbb{R}$ defined on E. The value $w(e)$ is called the *weight* of the edge e. If $S \subset E$ is any set of edges of G, the *weight* $w(S)$ of S is the defined as the total weight of the edges of S:

$$w(S) = \sum_{e \in S} w(e).$$

When the subset is a path $P \subset E$, the weight $w(P)$ is often called the *length* of P. Section 2 concerns minimum-weight spanning trees of weighted undirected graphs. Section 3 concerns shortest (minimum-weight) paths in weighted directed graphs.

Before looking at the runtime of MST algorithms, here are a couple of observations that simplify statements of runtime for connected graphs

(1) If $G = (V, E)$ is a connected graph, then $O(|V|) \leq O(|E|)$.
(2) If $G = (V, E)$ is a connected graph, then $\Theta(\log |V|) = \Theta(\log |E|)$.

(1) follows from the fact that $|V| \leq 1+|E|$ and (2) from the observation that $|V|-1 \leq |E| \leq |V|^2$.

2 Minimum Spanning Trees

Suppose $G = (V, E)$ is an *undirected* graph and $w : E \to \mathbb{R}$ is a weight function for G. A *spanning tree* of G is a subgraph T that is connected and has no cycles (i.e., a tree that touches all of the vertices of G). A *minimum spanning tree [MST]* is a spanning tree whose edge set has minimal weight among all spanning trees of G: if T' is any other spanning tree, then $w(T) \leq w(T')$

We discussed spanning trees briefly in Section 5.2 of Disjoint Sets Union/Find Algorithms as an application of the Partition data structure, and again in Section 4 of Graphs 1: Representation, Search, and Traverse as applications of graph survey algorithms. Here again is the Spanning Tree Algorithm:

```
Spanning Tree Algorithm [STA]
given: Graph     G = (V,E)
use:   Queue     q of edges in E // in any order
       Partition p               // connectivity bookkeeping

initialize p(|E|)      // partition initialized to singletons
initialize F = empty   // forest initialized to vertices
for each edge e=[x,y] in q
{
  if (!p.Find(x,y)) // x and y are in different trees in F
  {
    add e to F      // joins the two trees in F
    p.Union(x,y)    // records the join in the partition
  }
}
output (V,F)
```

We showed that

$$\text{STA Runtime } \leq O(|E| \log^* |V| + |E| \times ARQ)$$

which reduces to

$$\text{STA Runtime } \leq O(|E| \log^*(|V|))$$

if we make no restrictions on the order maintained by the queue on the edges, where ARQ is the amortized runtime cost of the queue operations. Recall that $\log^*(n) \leq 4$ for $n < 2^{1024}$, which means in practical terms that the STA runtime is $O(|E| \times ARQ)$.

2.1 Kruskal's MST Algorithm

Kruskal's algorithm for finding a MST for a weighted graph builds on the STA algorithm by the additional assumption that we consider the edges in order of increasing weight:

```
Kruskal's Minimum Spanning Tree Algorithm [MSTA]
given: WeightedGraph  G = (V,E),w
use:   Queue          q of edges in E, sorted by increasing weight
       Partition      p // connectivity bookkeeping

initialize q;           // ARQ1
initialize p(|E|)
initialize F = empty
while (!q.Empty())
{
```

```
  // loop invariant: (V,F) contains no cycles
  [x,y] = q.Front();    // ARQ2
  q.Pop();              // ARQ3
  if (!p.Find(x,y))
  {
    F = F union {e};
    p.Union(x,y);
  }
}
output (V,F)
```

Because the STA was shown to produce a spanning tree for any connected graph and any ordering of edges, the same is true for Kruskals MSTA. The question remaining is one of minimality, which is quite plausible given that we consider the edges in order of increasing weight.

Suppose that `F` as exported from Kruskal's MSTA is not minimal among spanning trees. Then some other MST `F1` has weight less than that of `F`. Let `e` be the first least cost edge in `F1` that is not in `F`. Then up to considering `e`, `F` would be the same as `F1`, because at each stage the existence of `F1` implies acceptance of the edges for `F`.[1] But then `F` would add `e` because it meets the criteria for inclusion, contradiction the assumption and proving the correctness of Kruskal's MSTA. □

The runtime of Kruskal's MSTA is $O(|E|\log^*|V| + |E| \times ARQ)$, and the bound is tight if ARQ dominates $\log^*|V|$. In order to consider the edges in order of increasing weight, we have prepared the queue by increasing weight, which gives the following estimates for ARQ.

ARQ1	queue initialization	$\Theta(\|E\|\log\|E\|)$
ARQ2	q.Front()	$\Theta(1)$
ARQ3	q.Pop()	$\Theta(1)$

yielding the estimate for $O(ARQ)$ to be $O(|E|\log|E|)$. Using the observations (1) and (2) above, we see that $O(ARQ) = O(|E|\log|V|)$. Putting this estimate together with the general estimate for STA yields the following estimate for the asymptotic runtime of Kruskal's MSTA:

$$\text{Runtime of KMSTA} \quad \leq O(|E|\log|V|).$$

This runtime can be improved by using a faster sort algorithm to prepare the queue. Perhaps the fastest would be to use a numerical (non-comparison) sort such as `byte_sort` to

[1] This argument assumes no two edges have the same weight. The proof in the general case is somewhat more elaborate. See Textbook.

initialize the queue to be completely sorted by increasing weight.[2] Making this substitution yields an improved runtime estimation table:

ARQ1	queue initialization	$\Theta(\|E\|)$
ARQ2	q.Front()	$\Theta(1)$
ARQ3	q.Pop()	$\Theta(1)$

and a runtime estimate of $O(|E| \log^* |V|)$. These runtimes are summarized as follows:

THEOREM 1C. The runtime of Kruskal's MST Algorithm, using a $\Theta(n \log n)$-time comparison sort to prepare the edge queue, is $O(|E| \log |V|)$.
THEOREM 1N. The runtime of Kruskal's MST Algorithm, using a linear-time sort to prepare the edge queue, is $O(|E| \log^* |V|)$.

In practical terms, this last runtime is $\Theta(|E|)$.

2.2 Prim's Algorithm

Prim takes a similar approach - build the MST by greedily adding edges meeting an acceptance criterion. The difference is that Prim maintains the property that the growing collection of edges is acyclic *and* connected, so that it is always a tree.

```
Prim's Minimum Spanning Tree Algorithm [MSTA]
given: WeightedGraph  G = (V,E),w
use:   PriorityQueue  q of DIRECTED edges in E

start vertex s
initialize W = {s}     // vertices of Prim tree
initialize T = empty   // edges of Prim tree
for each neighbor n of s,
  q.Push((s,n));       // ARQ1

while (!q.Empty())
{
  // loop invariant 1: (W,T) contains no cycles
  // loop invariant 2: (W,T) is connected
  (x,y) = q.Front();   // ARQ2
  q.Pop();             // ARQ3
  if (y is not in T)
  {
    W = W union {y};
    T = T union {[x,y]};
```

[2]This would require assuming edge weights are integers, or at least that there is a convenient $O(|E|)$ algorithm monotonically mapping edge weights to integers.

```
    for each neighbor z of y,
      if (z is not in T)
        q.Push((y,z)); // ARQ4
  }
}
output (W,T)
```

Many descriptions of Prim's algorithm use a priority queue of vertices. We use a priority queue of directed edges, which amounts to the same thing, except that edges have a built-in priority `w(e)`. In this way we do not have to invoke a key (priority) change in the queue once an edge has been inserted. We could as easily think of the queue as containing the "to" vertex of the directed edge. (See Section 3.5 below for more discussion of this topic.)

It is straightforward to verify loop invariants (1) and (2) and conclude that `(W,T)` is a tree. The two remaining issues are spanning and minimality.

Suppose some vertex of `G` is not in `W`. Then there must be an edge with one end in `W` and the other not. Let `e = [x,y]` be such an edge with minimal weight. Then when `x` is added to `W`, the edge `e` is added to the priority queue. At some future time, `e` will pass the test and be added to `T` and at the same time `y` will be added to `W`, a contradiction. Therefore `(W,T)` spans `G`.

Suppose that `W1` is a MST with weight less than `W`. Let `e` be the least-weight edge in `W1` that is not in `W`. Then up to considering `e` the edges of `W` are the same as those of `W1`. But then `e` would be at the front of the control queue, so `e` would be added to `W`. □

Runtime of Prim's MSTA using comparison-based $O(n \log n)$ priority queue is estimated in the following table:

ARQ1	q.Push(e)	$O(\log q.size)$
ARQ2	q.Front()	$\Theta(1)$
ARQ3	q.Pop()	$O(\log q.size)$
ARQ4	q.Push(e)	$\Theta(1)$

Aggregating the cost of all queue operations yields $O(|E| \log |E|) = O(|E| \log |V|)$ runtime, the same as that of Kruskal (Theorem 1c). Though Prim is not using a Partition data structure, that small superlinear component would be masked by the queue runtime in any case. Prim's queue operational requirements are not readily available from a simple initial sort, so the numerical sort optimization we used to reduce Kruskal's runtime (Theorem 1n) is not available for Prim. However, Prim's algorithm is a harbinger of other increasingly advanced search algorithms.

3 Shortest Paths

The context for this section is weighted *directed* graphs. Assume that $G = (V, E)$ is a directed graph with weight function $w : E \to \mathbb{R}$.

3.1 Weighted Cycles

In the various search algorithms studied so far, including Breadth-First and Depth-First search in either directed or undirected graphs and MST algorithms in undirected graphs, we ensured that cycles were avoided by some mechanism that guaranteed the final result was acyclic by simply never visiting a vertex more than once.

When designing algorithms for discovering shortest paths, in contrast, a vertex may be visited more than once while incrementally minimizing shortest path distances, and this introduces the possibility of traversing a cycle during the run of the algorithm. Consider the following weighted directed graph $G2$:

```
0: <2,1> <3,1>
1: <2,1> <3,1>        digraph G2 in adjacency list representation
2: <1,1> <3,1>
3: <1,-3>
```

Note the directed path from v_0 to v_3: $P = \{v_0, v_2, v_3\}$ with total weight $1 + 1 = 2$. But also note the cycle $C = \{v_1, v_2, v_3, v_1\}$ with total weight $1 + 1 + (-3) = -1$. And finally, note that $P \cup C = \{v_0, v_2, v_3, v_1, v_2, v_3\}$ is another path from v_0 to v_3 with total weight $1 + 1 + (-3) + 1 + 1 = 1$. In fact we could splice as many copies of C into P as we like, say k copies, to get a path of total weight $1 + 1 - k = 2 - k$.

Lemma 1. A minimum weight path from x to y cannot contain a negative-weight cycle.

Proof. Suppose P is a path from x to y that contains a negative-weight cycle L. Then $P' = P \cup L$ is another path from x to y that has weight $w(P') = w(P) + w(L) < w(P)$. Therefore there is no minimum weight path from x to y. □

Lemma 2. A minimum weight path from x to y cannot contain a positive-weight cycle.

Proof. Suppose P is a path from x to y that contains a positive-weight cycle L. Then $P' = P - L$ is another path from from x to y with weight $W(P') = w(P) - w(L) < w(P)$, so P does not have minimum weight. □

If P is a path and some vertex along P belongs to a zero-weight cycle L, then we could add or subtract L to P without changing the path weight, so a minimum weight path

might contain a zero-weight cycle. However, it makes sense to eliminate that possibility by insisting that a minimum-weight path be *simple* - that is, contain no redundancies in its vertex set.

3.2 Relaxation

Minimum-weight path algorithms maintain two entities associated with a vertex x: an evolving estimate $d[x]$ of the minimum path distance $D(s, x)$ from the start vertex s to x and a predecessor vertex $p[x]$ for x in a minimum-weight path. At the conclusion of the run of the algorithm, $d[x] = D(s, x)$ and $p[x]$ reconstructs a minimum-weight path. However during the run of the algorithm, both $d[x]$ and $p[x]$ may be revised a number of times.

Suppose for example that we have an estimate of $d[x]$ for the distance from a starting vertex s to x and at some point we discover an edge $e = (x, y)$. Then a path from s to y can be pieced together from a shortest path from s to x and the edge e, so that the path distance from s to y can be no greater than $d(x) + w(e)$. If we discover that the estimate $d[y] > d[x] + w(e)$ then we revise the estimate for $d[y]$ and modify the current search path to y in a process called *relaxation*:

```
Relax(x,y,w) // e = (x,y), w is the weight function
  if (d[x] + w(e) < d[y])
  {
    d[y] = d[x] + w(e);  // update path distance estimate d[y]
    parent[y] = x;       // update path to go through x
  }
```

Dijkstra, Bellman-Ford, and A* all use relaxation to refine paths during the search.[3]

Another routine that single source shortest path algorithms have in common is the way search is initialized:

```
Initialize-single-source (G,s)
  for each vertex x of G
  {
    distance[x] = infinity
    parent[x]   = NULL
  }
  d[s] = 0;
```

Dijkstra, Bellman-Ford, and A* all use this initialization procedure.

[3] In breadth-first search, we did not need relaxation to adjust the path weight because weight equals edge count in a graph or digraph whose edge weights are all 1.

3.3 Single-Source Shortest Path Algorithms

All of the algorithms in this section solve the Single-Source Shortest-Path [SSSP] problem, which can be posed as follows: Given a weighted directed graph $G = (V, E), w$ and a starting vertex $s \in V$, construct functions $d : V \to \mathbb{R}$ and $p : V \to V^*$ such that

(1) $d(x)$ is the length of a shortest path from s to x; and
(2) $p(x) \neq$ NULL is the predecessor of x in a shortest path from s to x

where $V^* = V \cup \{\text{NULL}\}$. The path $P(x)$ reconstructed using p is a simple path, and the totality of paths $P(x)$ form a tree rooted at s. Moreover, $p[x] \neq$ NULL iff x is reachable from s. An SSSP algorithm solves the SSSP problem.

3.4 Dijkstra's SSSP Algorithm

The following Single-Source Shortest-Path algorithm was published by Edger Dijkstra in 1959. Assume that $G = (V, E)$ is a directed graph with non-negative weight function $w : E \to [0, +\infty)$.

```
Dijkstra's SSSP Algorithm
Input: weighted digraph (G,w), w is non-negative
       source vertex s
Uses:  Set X of vertices

Initialize-single-source (G,s);
Initialize X = V;

while (X is not empty)
{
  x = vertex in X with minimal distance d;
  X.Remove(x);
  for each vertex y adjacent from x
  {
    Relax(x,y,w);
  }
}
```

LEMMA 3. In Dijkstra's SSSP algorithm, when a vertex `x` is removed from the set `X`, `d[x]` is the minimal path distance from `s` to `x` and `p[]` reconstructs a path of weight d[x] from `s` to `x`.

Proof. We use induction on the number of "black" vertices - those that have been removed from the set `X`. Since the first vertex removed is `s` and `d[s] = 0, p[s] = NULL` the base case is verified.

For the inductive step, assume the result is true for all current black vertices, and consider a vertex `y` in `X` with minimal `d[y]`. Suppose that `d[y]` is not the minimum path distance of `y`, and let `P` be a minimal weight path from `s` to `y`. Then `w(P) < d[y]`. Let `x` be the penultimate vertex on `P` and let `P1` be `P` with the edge `(x,y)` removed. Then `w(P1) = w(P) - w(x,y) <= w(P)` (because `w(x,y) >= 0`). Note also that `d[x] <= w(P1)`, so that `d[x] < d[y]`, hence `x` is black. In summary: `x` is black and

```
d[x] + w(x,y) <= w(P) < d[y].
```

But `x` being black means that `d[y]` would have been updated to `d[y] = d[x] + w(x,y)` when `x` was removed from `X`, a contradiction. Therefore `d[y]` is the minimal path distance of `x`.

Note also that when `y` is removed from `X`, `p[y] = x` and `d[y] = d[x] + w(x,y)`. Invoking the induction hypothesis, `p[]` reconstructs a minimum-weight path `Q1` from `s` to `x`. Note that the path `Q` obtained by appending `(x,y)` to `Q1` is the path from `s` to `y` constructed by `p[]`, and that

```
w(Q) = w(Q1) + w(x,y) = d[x] + e(x,y) = d[y].
```

Therefore `Q` is a minimum-weight path from `s` to `y`. □

The original version of the algorithm used an array as the set `X`. There are various strategies one could use to implement the first two lines of the while loop:

(1) Use sequential search in the array to find `x` and "leapfrog" removal.
(2) Sort the array, use binary search to find `x`, and leapfrog removal.
(3) Replace the array with a linked list, sequential search, and unlink-to-remove.

All of these result in $O(|V|)$ cost for the find and remove steps, inside a loop that runs $|V|$ times. Therefore this version of Dijkstra's SSSP algorithm runs in time $O(|V|^2)$.

Using modern priority queue technology, we can improve the algorithm runtime for all but the most densely connected digraphs.

```
Dijkstra's SSSP Algorithm v2
Input: weighted digraph (G,w)
       source vertex s
Uses:  Priority queue Q of vertices with priority d[]
       Vector<bool> finished[]

Initialize-single-source (G,s);
Initialize finished to false
Initialize Q = V with priority d
while (!Q.Empty())
{
  x = Q.Front();
  Q.Pop();
  if (finished[x]) break;
  for each vertex y adjacent from x
  {
    Relax2(x,y,w,Q);
  }
  finished[x] = true;
}
```

with the following refinement of the relaxation process:

```
Relax2(x,y,w,Q) // e = (x,y), w is the weight function
{
  if (d[x] + w(e) < d[y])
  {
    d[y] = d[x] + w(e);  // update path distance estimate d[y]
    parent[y] = x;       // update path to go through x
    Q.Push(y,d[y]);      // (re)insert y in Q
  }
}
```

Note that the queue may contain several copies of a vertex with differing priorities. Only the highest priority entry is processed through the Relax loop. A functionally equivalent version keeps fewer vertices in memory (fewer "gray" vertices) and has a smaller queue:

```
Dijkstra's SSSP Algorithm v3
Input: weighted digraph (G,w)
       source vertex s
Uses:  Priority queue Q of vertices with priority d[]

Initialize-single-source (G,s);
Q.Push(s);
while (!Q.Empty())
```

```
{
  x = Q.Front();
  Q.Pop();
  for each vertex y adjacent from x
  {
    Relax3(x,y,w,Q);
  }
}


Relax3(x,y,w,Q) // e = (x,y), w is the weight function
{
  if (d[y] == infinity)    // (or parent[y] == NULL); y is a white vertex
  {
    d[y] = d[x] + w(e);    // estimate path distance d[y]
    parent[y] = x;         // define path from s through x to y
    Q.Push(y);             // insert y into Q; y is now a gray vertex
  }
  else                     // y is a gray vertex
  if (d[x] + w(e) < d[y])
  {
    d[y] = d[x] + w(e);    // update path distance estimate d[y]
    parent[y] = x;         // update path to go through x
    Q.DecreaseKey(y,d[y]); // upgrade y's place in Q
  }
}
```

Note that Lemma 3 applies to all three versions of Dijkstra's algorithm, because it makes no structural constraints on the control set.

THEOREM 2 (CORRECTNESS OF DIJKSTRA SSSP). Suppose (G, w) is a weighted directed graph with $w(e) \geq 0$ for all edges e of G. Let s be a vertex of G and run Dijkstra's SSSP algorithm (any version). Then for any vertex x that is reachable from s, (1) $d(x)$ is the weight of a shortest path from s to x and (2) p reconstructs a shortest path from s to x.

Proof. Let x be a reachable vertex, that is, a vertex with a path P from s to x. Then x is discovered (placed in the control set X or Q) no later than when the penultimate edge (w, x) of P is traversed. Therefore x is eventually removed from the control set. By Lemma 3, $d(x)$ is the weight of a shortest path from s to x after x has been removed from the control set (X or Q) and p reconstructs a shortest path from s to x. □

THEOREM 3 (RUNTIME OF DIJKSTRA'S SSSP). Suppose (G, w) is a weighted directed graph with $w(e) \geq 0$ for all edges e of G. Let s be a vertex of G. The runtime of Dijkstra's SSSP algorithm is:
Version 1: $O(|V|^2)$

Version 2: $O(|E| \log |V|)$
Version 3: $O(|E| \log |V|)$.

Proof. The following table captures the cost of various components of the algorithm:

operation	**v1**	**v2**	**v3**
initialize X	$\|V\|$	$\|V\|$	1
find light edge	$\|V\|$	$\log \|E\|$	$\log \|E\|$
process edge	$\|V\|$	$\log \|E\|$	$\log \|E\|$
aggregate	$\|V\| \times \|V\|$	$\|E\| \times \log \|E\|$	$\|E\| \times \log \|E\|$

Aggregating over the main loop which has length $|V|$ and the processing of edges yields the counts at the bottom row of the table. Substituting $\log |E| = \log |V|$ completes the proof. □

3.5 Priority Queue Technology

Both the Prim and Dijkstra algorithms use priority queues to keep track of vertices prioritized by current value of $d[x]$. The implementation details of the algorithms and the priority queues are bound together, and the nuances are subtle but also important. We explore these details here.

3.5.1 Profligate Queue Strategy

This strategy uses a standard heap-based priority queue in the algorithm and the following version of `Relax`:

```
Relax-Profligate(x,y,w,Q)
{
  if (d[x] + w(e) < d[y])
  {
    d[y] = d[x] + w(e);
    parent[y] = x;
    Q.Push(y,d[y]); // insert y with priority d
                    // previous entry remains, but ignored
  }
}
```

The typical priority queue API does not have an efficient (or public) way to locate and/or remove an element that is somewhere in the queue. This strategy just doesn't bother and leaves the obsolete entry in the queue. Dijkstra's SSP would just add a check to detect black (obsolete) vertices when they appear at the front of the queue and remove them.

This strategy has the advantages of simpler code and the use of standard priority queue technology. The disadvantage is the control queue has worst-case size $\Theta(|E|)$.

3.5.2 Frugal Queue Strategy

In order to keep the size of the control queue as small as possible, we need to upgrade the status of vertices already in the queue with a change of priority key value and a restructure operation that moves the entry to a new location in the queue based on its new priority. This strategy uses the following version of `Relax`:

```
Relax-Frugal(x,y,w,Q)
{
  if (d[y] == infinity)    // y is a white vertex
  {
    d[y] = d[x] + w(e);
    parent[y] = x;
    Q.Push(y,d[y]);
  }
  else                     // y is a gray vertex
  if (d[x] + w(e) < d[y])
  {
    d[y] = d[x] + w(e);
    parent[y] = x;
    Q.DecreaseKey(y,d[y]); // adjust y's place in Q
                           // based on changed priority d[y]
  }
}
```

The new feature is a DecreaseKey operation in the (min) priority queue, and also an implied ability to directly access a location in the priority queue to make the key change. The DecreaseKey operation is easily implemented as long as we know where to decrease the key.

The priority queue would need to maintain an indexing of vertices to their location in the queue, possibly a vector `index_` such that `index_[x]` is the location of `x` in the underlying heap. This mapping would need to be updated in parallel with the heap, so that it remains current. Then the DecreaseKey operation can be implemented as

```
DecreaseKey(x,d)
{
  index_[x]->key_ = d;                        // change key value in heap
  g_push_heap(heap_.Begin(), 1+index_[x]);  // fix range [Begin,index[x]]
}
```

where `heap_` is the underlying heap structure.

This enhanced priority-queue-with-locator-index has the same asymptotic runtime as a standard priority queue, but there is significantly more overhead involved in maintaining the locator index. The payoff is reduced size of the queue, which also affects runtime of the queue operations. The queue size is $O(|V|)$ worst case.

3.6 Bellman-Ford SSSP Algorithm

The algorithms studied so far in this chapter, Spanning Tree, Kruskal, Prim, and Dijkstra, have all been greedy algorithms. Bellman-Ford is an SSSP algorithm that is classifiable as a dynamic programming algorithm, with optimal substructure. It is also an algorithm that does not require the assumption of non-negative weights and can detect negative-weight cycles (which make the SSSP problem unsolvable).

```
Bellman-Ford SSSP Algorithm

returns: BOOL
// true iff G has no negative-weight cycles

Initialize-single-source(G,s)
repeat (|V| - 1) times
{
  for each edge (x,y)
  {
    Relax(x,y,d);
  }
}
for each edge (x,y)
  if (d[y] > d[x] + w(x,y))
    return false;
return true
```

The runtime of Bellman-Ford is straightforward to estimate. Initialization requires $|V|$ steps. The main loop structure is a nested pair of fixed-length loops of lengths $|V|$ and $|E|$, respectively, giving $|V| \times |E|$ steps. And the closing loop requires $|E|$ steps. The runtime is therefore $\Theta(|V||E|)$.

THEOREM 4. Assume G, w is a weighted directed graph and s is a vertex of G. Let b be the return value after running the Bellman-Ford algorithm on G, w, s. If b is false then G has a negative-weight cycle reachable from s. If b is true then d, p solves the SSSP problem in G, w starting at s.

4 Introduction to Algorithm A*

Algorithm A* was introduced by Peter Hart, Nils Nilsson and Bertram Raphael in 1968. It has been in heavy use ever since in both research and practical applications. There is at least one web site maintained specifically to explain algorithm A* and keep up with its evolving applications and descendants.

Algorithm A* uses two notions of distance between vertices x, y in a weighted graph: actual (path) distance as the weight of a shortest path from x to y, and heuristic distance, defined by a distance estimator heuristic. Let $D(x, y)$ denote the path distance from x to y, that is, the weight of a shortest path from x to y, where path weight is defined to be the total weight of the path, and assume we have a heuristic distance estimator $H(x, y)$. We will further assume that the heuristic function H satisfies both of the following properties:

(1) H is *admissible* iff for any two vertices $x, y \in V$, $H(x, y) \leq D(x, y)$.
(2) H is *monotonic* iff for any edge $e = (x, y)$ and any vertex z, $H(x, z) \leq w(e) + H(y, z)$.

That is, H never overestimates distances and H obeys a "triangle inequality".[4] From (2) we can prove:

Lemma 4 (Admissibility). If H is monotonic and $H(x, x) = 0$ for all x then H is admissible.

Proof. Let $P = \{x, v_1, v_2, \ldots, v_{k-1}, y\}$ be a shortest path from x to y. Then

$$\begin{aligned} H(x, y) &\leq w(x, v_1) + H(v_1, y) \\ &\leq w(x, v_1) + w(v_1, v_2) + H(v_2, y) \\ &\leq w(x, v_1) + w(v_1, v_2) + w(v_2, v_3) + H(v_3, y) \\ &\leq \ldots \\ &\leq \sum_{e \in P} w(e) + H(y, y) \\ &= w(P) + H(y, y). \end{aligned}$$

Observing that $H(y, y) = 0$ and $w(P) = D(x, y)$ completes the proof. □

We now set up the notation for Algorithm A* as it is traditionally done: Suppose s is a start vertex and *goal* is a goal vertex, and define

$g(x)$	$= D(s, x)$	= path distance from start to x
$h(x)$	$= H(x, goal)$	= heuristic estimate of distance from x to goal
$f(x)$	= g(x) + h(x)	= A* search priority

[4] We use the notation of directed edges, but the context can be either directed or undirected.

We will use the following example to motivate and illustrate algorithm A*: The graph represents a roadmap, with vertices being cities or other road intersections and edges being road segments directly connecting two vertices. The distances are $w(x, y)$ = the weight of the road segment directly connecting x to y; $D(x, y)$ = the shortest roadway distance from x to y; and $H(x, y)$ = the straight-line (euclidean) distance between x and y. We have a start city s and a destination city *goal* and wish to calculate the driving distance from s to *goal*. A* sets up as follows:

vertex x	city
edge $e = (x, y)$	road between cities x and y
$w(e)$	length of road between x and y
$g(x)$	$D(s, x)$ = shortest roadway distance from s to x
$h(x)$	$H(x, goal)$ = straight-line distance from x to *goal*
$f(x)$	$D(s, x) + H(x, goal)$

The euclidian distance function is admissible and monotonic, so our assumptions are met by the example. Note in passing that $H(x, y)$ is easily calculated directly from GPS data, whereas $w(x, y)$ requires a database of road segment lengths and $D(x, y)$ can be calculated but is not readily available.

Algorithm A* uses the same search pattern as that of Prim and Dijkstra, using $f(x)$ as priority in the queue. The queue strategy can be either profligate or frugal. (See section 5.5.) The queue is called the OPEN set in the following statement of the algorithm. The CLOSED set consists of the black vertices with which the algorithm is finished. The white vertices have not yet been encountered.

```
Algorithm A*
find a shortest path from start to goal

Uses:   ordered set OPEN of vertices, ordered by F
        cost functions G, H, F = G + H

OPEN   = set of gray vertices
CLOSED = set of black vertices

initialize visited[x] = 0, parent[x] = NULL, color[x] = white

visited[start] = 1;
parent[start]  = NULL;
color[start]   = gray;
OPEN.Insert(start,F(start));

while (!q.Empty())
{
  f = q.Front();
```

```
  if (f == goal) break;
  for each adjacent n from f
  {
    cost = G(f) + w(f,n);
    if (color[n] = white)
    {
      G(n) = cost;
      F(n) = cost + H(n);
      parent[n] = f;
      OPEN.Insert(n,F(n));
    }
    else if (color[n] = gray)
    {
      if (cost < G(n)) // path through n is not optimal
      { // Relax
        G(n) = cost;
        F(n) = cost + H(n);
        parent[n] = f;
        OPEN.DecreaseKey(n, F(n));
      }
    }
    else // color[n] = black
    {
      // no action needed when H is admissable and monotonic
      // otherwise it is possible n needs to be reconsidered
    }
  }
  OPEN.Remove(f);
  color[f] = black;
}
reconstruct reverse path from goal to start by following parent pointers
```

THEOREM 5 (CORRECTNESS OF A*). Using an admissible monotonic heuristic, if there is a path from start to goal, A* will always find one with minimum weight.