CITS2200 Data Structures and Algorithms

Topic 15

Shortest Path Algorithms

- Priority-first search
- Shortes path problems
- Dijkstra's algorithm
- The Bellman-Ford algorithm.
- The all pairs shortest path problem
- Dynamic programming method
- Matrix product algorithm
- Floyd-Warshall algorithm

Reading: Weiss Chapter 14

Priority-first search

Let us generalize the ideas behind the implementation of Prim's algorithm.

Consider the following very general graph-searching algorithm. We will later show that by choosing different specifications of the priority we can make this algorithm do very different things. This algorithm will produce a *priority-first search tree*.

The key-values or priorities associated with each vertex are stored in an array called key.

Initially we set $key[v]$ to ∞ for all the vertices $v \in V(G)$ and build a heap with these keys — this can be done in time $O(V)$.

Then we select the source vertex s for the search and perform **change(**s**,**0**)** to change the key of s to 0, thus placing s at the top of the priority queue.

The operation of PFS

After initialization the operation of PFS is as follows:

procedure PFS(s)
  change(s,0)
  while $Q \neq \emptyset$
    $u \leftarrow$ **Q.dequeue()**
    for each v adjacent to u **do**
      if $v \in Q \wedge$ *PRIORITY* $< key[v]$ **then**
        $\pi[v] \leftarrow u$
        change(Q,v,*PRIORITY*)
      end if
    end for
  end while

It is important to notice how the array π is managed — for every vertex $v \in Q$ with a finite key value, $\pi[v]$ is the vertex *not in* Q that was responsible for the key of v reaching the highest priority it has currently reached.

Complexity of PFS

The complexity of this search is easy to calculate — the main loop is executed V times, and each **extractmin** operation takes $O(\lg V)$ yielding a total time of $O(V \lg V)$ for the extraction operations.

During all V operations of the main loop we examine the adjacency list of each vertex exactly once — hence we make E calls, each of which may cause a **change** to be performed. Hence we do at most $O(E \lg V)$ work on these operations.

Therefore the total is

$$O(V \lg V + E \lg V) = O(E \lg V).$$

Prim's algorithms is PFS

Prim's algorithm can be expressed as a priority-first search by observing that the priority of a vertex is the weight of the shortest edge joining the vertex to the rest of the tree.

This is achieved in the code above by simply replacing the string *PRIORITY* by

$$weight(u, v)$$

At any stage of the algorithm:

• The vertices not in Q form the tree so far.

• For each vertex $v \in Q$, $key[v]$ gives the length of the shortest edge from v to a vertex in the tree, and $\pi[v]$ shows which tree vertex that is.

Shortest paths

Let G be a directed weighted graph. The *shortest path* between two vertices v and w is the path from v to w for which the sum of the weights on the path-edges is lowest. Notice that if we take an unweighted graph to be a special instance of a weighted graph, but with all edge weights equal to 1, then this coincides with the normal definition of shortest path.

The weight of the shortest path from v to w is denoted by $\delta(v, w)$.

Let $s \in V(G)$ be a specified vertex called the *source* vertex.

The *single-source shortest paths* problem is to find the shortest path from s to every other vertex in the graph (as opposed to the *all-pairs shortest paths problem*, where we must find the distance between every pair of vertices).

Dijkstra's algorithm

Dijkstra's algorithm is a famous single-source shortest paths algorithm suitable for the cases when the weights are all non-negative.

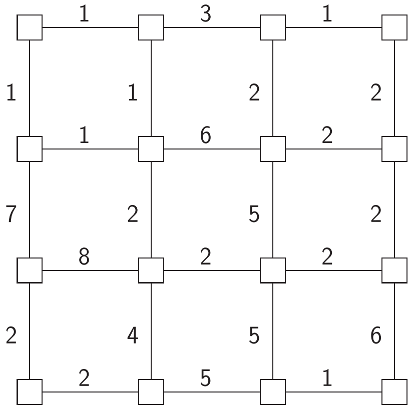
Dijkstra's algorithm can be implemented as a priority-first search by taking the priority of a vertex $v \in Q$ to be the shortest path from s to v that consists entirely of vertices in the priority-first search tree (except of course for v).

This can be implemented as a PFS by replacing PRIORITY with

$$key[u] + weight(u, v)$$

At the end of the search, the array $key[]$ contains the lengths of the shortest paths from the source vertex s.

Dijkstra's algorithm in action



Proof of correctness

It is possible to prove that Dijkstra's algorithm is correct by proving the following claim (assuming $T = V(G) - Q$ is the set of vertices that have already been removed from Q).

> *At the time that a vertex u is removed from Q and placed into T $key[u] = \delta(s, u)$.*

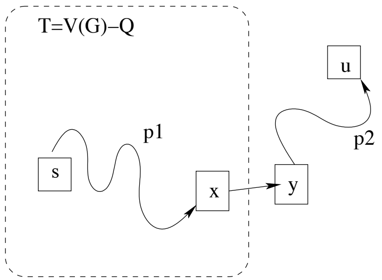
This is a proof by contradiction, meaning that we try to prove $key[u] \neq \delta(s, u)$ and if we fail then we will have proved the opposite.

Assuming $u \neq s$ then $T \neq \emptyset$ and there exists a path p from s to u. We can decompose the path into three sections:

1. A path p_1 from s to vertex x, such that $x \in T$ and the path is of length 0 or more.
2. An edge between x and y, such that $y \in Q$ and $(x, y) \in E(G)$.
3. A path p_2 from y to u of length 0 or more.

Proof (contd)

The decomposed path may be illustrated thus.



Firstly, we know $key[y] = \delta(s, y)$ since the edge (x, y) will have been examined when x was added to T.

Furthermore, we know that y is before u on path p and therefore $\delta(s, y) \leq \delta(s, u)$. This implies $key[y] \leq key[u]$ (inequality A).

Proof (contd)

But we also know that u was chosen from Q before y which implies $key[u] \leq key[y]$ (inequality B) since the priority queue always returns the vertex with the smallest key.

Inequalities A and B can only be satisfied if $key[u] = key[y]$ but this implies

$$key[u] = \delta(s, u) = \delta(s, y) = key[y]$$

But our initial assumption was that $key[u] \neq \delta(s, u)$ giving rise to the contradiction. Hence we have proved that $key[u] = \delta(s, u)$ at the time that u enters T.

Relaxation

Consider the following property of Dijkstra's algorithm.

- At any stage of Dijkstra's algorithm the following inequality holds:

$$\delta(s, v) \leq key[v]$$

This is saying that the *key[]* array always holds a collection of *upper bounds* on the actual values that we are seeking. We can view these values as being our "best estimate" to the value so far, and Dijkstra's algorithm as a procedure for systematically improving our estimates to the correct values.

The fundamental step in Dijkstra's algorithm, where the bounds are altered is when we examine the edge (u, v) and do the following operation

$key[v] \leftarrow \mathbf{min}(key[v], key[u] + weight(u, v))$

This is called *relaxing* the edge (u, v).

Relaxation schedules

Consider now an algorithm that is of the following general form:

- Initially an array $d[]$ is initialized to have $d[s] = 0$ for some source vertex s and $d[v] = \infty$ for all other vertices
- A sequence of edge relaxations is performed, possibly altering the values in the array $d[]$.

We observe that the value $d[v]$ is always an upper bound for the value $\delta(s, v)$ because relaxing the edge (u, v) will either leave the upper bound unchanged or replace it by a better estimate from an upper bound on a path from $s \rightarrow u \rightarrow v$.

Dijkstra's algorithm is a particular schedule for performing the edge relaxations that guarantees that the upper bounds converge to the exact values.

Negative edge weights

Dijkstra's algorithm cannot be used when the graph has some negative edge-weights (why not? find an example).

In general, no algorithm for shortest paths can work if the graph contains a cycle of negative total weight (because a path could be made arbitrarily short by going round and round the cycle). Therefore the question of finding shortest paths makes no sense if there is a negative cycle.

However, what if there are some negative edge weights but no negative cycles?

The Bellman-Ford algorithm is a relaxation schedule that can be run on graphs with negative edge weights. It will either *fail* in which case the graph has a negative cycle and the problem is ill-posed, or will finish with the single-source shortest paths in the array $d[]$.

Bellman-Ford algorithm

The initialization step is as described above. Let us suppose that the weights on the edges are given by the function w.

Then consider the following relaxation schedule:

for $i = 1$ **to** $|V(G)| - 1$ **do**
    for each edge $(u, v) \in E(G)$ **do**
        $d[v] \leftarrow \mathbf{min}(d[v], d[u] + w(u, v))$
    end for each
end for

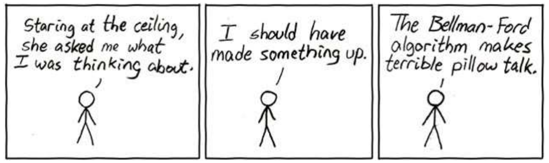
Finally we make a single check to determine if we have a failure:

for each edge $(u, v) \in E(G)$ **do**
    if $d[v] > d[u] + w(u, v)$ **then**
        FAIL
    end if
end for each

Complexity of Bellman-Ford

The complexity is particularly easy to calculate in this case because we know exactly how many relaxations are done — namely $E(V-1)$, and adding that to the final failure check loop, and the initialization loop we see that Bellman-Ford is $O(EV)$

There remains just one question — how does it work?



Correctness of Bellman-Ford

Let us consider some of the properties of relaxation in a graph with no negative cycles.

Property 1 Consider an edge (u, v) that lies on the shortest path from s to v. If the sequence of relaxations includes relaxing (u, v) at a stage when $d[u] = \delta(s, u)$, then $d[v]$ is set to $\delta(s, v)$ and never changes after that.

Once convinced that Property 1 holds we can show that the algorithm is correct for graphs with no negative cycles, as follows.

Consider any vertex v and let us examine the shortest path from s to v, namely

$$s \sim v_1 \sim v_2 \cdots v_k \sim v$$

Now at the initialization stage $d[s] = 0$ and it always remains the same. After one pass through the main loop the edge (s, v_1) is relaxed and by Property 1, $d[v_1] = \delta(s, v_1)$ and it remains at that value. After the second pass the edge (v_1, v_2) is relaxed and after this relaxation we have $d[v_2] = \delta(s, v_2)$ and it remains at this value.

As the number of edges in the path is at most $|V(G)| - 1$, after all the loops have been performed $d[v] = \delta(s, v)$.

Note that this is an inductive argument where the induction hyptohesis is "*after n iterations, all shortest paths of length n have been found*".

All-pairs shortest paths

Recall the Shortest Path Problem in Topic **??**.

Now we turn our attention to constructing a complete table of shortest distances, which must contain the shortest distance between any pair of vertices.

If the graph has no negative edge weights then we could simply make V runs of Dijkstra's algorithm, at a total cost of $O(VE \lg V)$, whereas if there are negative edge weights then we could make V runs of the Bellman-Ford algorithm at a total cost of $O(V^2E)$.

The two algorithms we shall examine both use the adjacency matrix representation of the graph, hence are most suitable for dense graphs. Recall that for a weighted graph the weighted adjacency matrix A has $weight(i, j)$ as its ij-entry, where $weight(i, j) = \infty$ if i and j are not adjacent.

A dynamic programming method

Dynamic programming is a general algorithmic technique for solving problems that can be characterised by two features:

- The problem is broken down into a collection of smaller subproblems
- The solution is built up from the stored values of the solutions to all of the subproblems

For the all-pairs shortest paths problem we define the simpler problem to be

"What is the length of the shortest path from vertex i to j that uses at most m edges?"

We shall solve this for $m = 1$, then use that solution to solve for $m = 2$, and so on . . .

The initial step

We shall let $d_{ij}^{(m)}$ denote the distance from vertex i to vertex j along a path that uses at most m edges, and define $D^{(m)}$ to be the matrix whose ij-entry is the value $d_{ij}^{(m)}$.

As a shortest path between any two vertices can contain at most $V - 1$ edges, the matrix $D^{(V-1)}$ contains the table of all-pairs shortest paths.

Our overall plan therefore is to use $D^{(1)}$ to compute $D^{(2)}$, then use $D^{(2)}$ to compute $D^{(3)}$ and so on.

The case $m = 1$

Now the matrix $D^{(1)}$ is easy to compute — the length of a shortest path using at most one edge from i to j is simply the weight of the edge from i to j. Therefore $D^{(1)}$ is just the adjacency matrix A.

The inductive step

What is the smallest weight of the path from vertex i to vertex j that uses at most m edges? Now a path using at most m edges can either be

(1) A path using less than m edges
(2) A path using exactly m edges, composed of a path using $m - 1$ edges from i to an auxiliary vertex k and the edge (k, j).

We shall take the entry $d_{ij}^{(m)}$ to be the lowest weight path from the above choices.

Therefore we get

$$\begin{aligned} d_{ij}^{(m)} &= \min\left(d_{ij}^{(m-1)}, \min_{1\le k\le V}\{d_{ik}^{(m-1)} + w(k,j)\}\right) \\ &= \min_{1\le k\le V}\{d_{ik}^{(m-1)} + w(k,j)\} \end{aligned}$$

Example

Consider the weighted graph with the following weighted adjacency matrix:

$$A = D^{(1)} = \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & \infty & \infty \\ 10 & \infty & 0 & \infty & \infty \\ \infty & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix}$$

Let us see how to compute an entry in $D^{(2)}$, suppose we are interested in the $(1, 3)$ entry:

$1 \to 1 \to 3$ has cost $0 + 11 = 11$ $\quad$ $1 \to 2 \to 3$ has cost $\infty + 4 = \infty$
$1 \to 3 \to 3$ has cost $11 + 0 = 11$ $\quad$ $1 \to 4 \to 3$ has cost $2 + 6 = 8$
$1 \to 5 \to 3$ has cost $6 + 6 = 12$

The minimum of all of these is 8, hence the $(1, 3)$ entry of $D^{(2)}$ is set to 8.

Computing $D^{(2)}$

$$\begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & \infty & \infty \\ 10 & \infty & 0 & \infty & \infty \\ \infty & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix} \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & \infty & \infty \\ 10 & \infty & 0 & \infty & \infty \\ \infty & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix} = \begin{pmatrix} 0 & 4 & 8 & 2 & 5 \\ 1 & 0 & 4 & 3 & 7 \\ 10 & \infty & 0 & 12 & 16 \\ 3 & 2 & 6 & 0 & 3 \\ 16 & \infty & 6 & \infty & 0 \end{pmatrix}$$

If we multiply two matrices $AB = C$, then we compute

$$c_{ij} = \sum_{k=1}^{k=V} a_{ik} b_{kj}$$

If we replace the multiplication $a_{ik}b_{kj}$ by addition $a_{ik} + b_{kj}$ and replace summation Σ by the minimum $\min$ then we get

$$c_{ij} = \min_{k=1}^{k=V} a_{ik} + b_{kj}$$

which is precisely the operation we are performing to calculate our matrices.

The remaining matrices

Proceeding to compute $D^{(3)}$ from $D^{(2)}$ and A, and then $D^{(4)}$ from $D^{(3)}$ and A we get:

$$D^{(3)} = \begin{pmatrix} 0 & 4 & 8 & 2 & 5 \\ 1 & 0 & 4 & 3 & \boxed{6} \\ 10 & \boxed{14} & 0 & 12 & \boxed{15} \\ 3 & 2 & 6 & 0 & 3 \\ 16 & \infty & 6 & \boxed{18} & 0 \end{pmatrix} \qquad D^{(4)} = \begin{pmatrix} 0 & 4 & 8 & 2 & 5 \\ 1 & 0 & 4 & 3 & 6 \\ 10 & 14 & 0 & 12 & 15 \\ 3 & 2 & 6 & 0 & 3 \\ 16 & \boxed{20} & 6 & 18 & 0 \end{pmatrix}$$

A new matrix "product"

Recall the method for computing $d_{ij}^{(m)}$, the (i, j) entry of the matrix $D^{(m)}$ using the method similar to matrix multiplication.

$d_{ij}^{(m)} \leftarrow \infty$
for $k = 1$ **to** V **do**
  $d_{ij}^{(m)} = \min(d_{ij}^{(m)}, d_{ik}^{(m-1)} + w(k, j))$
end for

Let us use $\star$ to denote this new matrix product.

Then we have

$$D^{(m)} = D^{(m-1)} \star A$$

Hence it is an easy matter to see that we can compute as follows:

$$D^{(2)} = A \star A \qquad D^{(3)} = D^{(2)} \star A \ldots$$

Complexity of this method

The time taken for this method is easily seen to be $O(V^4)$ as it performs V matrix "multiplications" each of which involves a triply nested **for** loop with each variable running from 1 to V.

However we can reduce the complexity of the algorithm by remembering that we do not need to compute *all* the intermediate products $D^{(1)}$, $D^{(2)}$ and so on, but we are only interested in $D^{(V-1)}$. Therefore we can simply compute:

$$D^{(2)} = A \star A$$
$$D^{(4)} = D^{(2)} \star D^{(2)}$$
$$D^{(8)} = D^{(4)} \star D^{(4)}$$

Therefore we only need to do this operation at most $\lg V$ times before we reach the matrix we want. The time required is therefore actually $O(V^3 \lceil \lg V \rceil)$.

Floyd-Warshall

The Floyd-Warshall algorithm uses a different dynamic programming formalism.

For this algorithm we shall define $d_{ij}^{(k)}$ to be the length of the shortest path from i to j whose intermediate vertices all lie in the set $\{1, \ldots, k\}$.

As before, we shall define $D^{(k)}$ to be the matrix whose (i, j) entry is $d_{ij}^{(k)}$.

The initial case

What is the matrix $D^{(0)}$ — the entry $d_{ij}^{(0)}$ is the length of the shortest path from i to j with *no* intermediate vertices. Therefore $D^{(0)}$ is simply the adjacency matrix A.

The inductive step

For the inductive step we assume that we have constructed already the matrix $D^{(k-1)}$ and wish to use it to construct the matrix $D^{(k)}$.

Let us consider all the paths from i to j whose intermediate vertices lie in $\{1, 2, \ldots, k\}$. There are two possibilities for such paths

(1) The path does not use vertex k
(2) The path does use vertex k

The shortest possible length of all the paths in category (1) is given by $d_{ij}^{(k-1)}$ which we already know.

If the path does use vertex k then it must go from vertex i to k and then proceed on to j, and the length of the shortest path in this category is $d_{ik}^{(k-1)} + d_{kj}^{(k-1)}$.

The overall algorithm

The overall algorithm is then simply a matter of running V times through a loop, with each entry being assigned as the minimum of two possibilities. Therefore the overall complexity of the algorithm is just $O(V^3)$.

$D^{(0)} \leftarrow A$
for $k = 1$ **to** V **do**
  for $i = 1$ **to** V **do**
    for $j = 1$ **to** V **do**
      $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$
    end for j
  end for i
end for k

At the end of the procedure we have the matrix $D^{(V)}$ whose (i, j) entry contains the length of the shortest path from i to j, all of whose vertices lie in $\{1, 2, \ldots, V\}$ — in other words, the shortest path in total.

Example

Consider the weighted directed graph with the following adjacency matrix:

$$D^{(0)} = \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & \infty & \infty \\ 10 & \infty & 0 & \infty & \infty \\ \infty & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix} \qquad D^{(1)} = \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & & \\ 10 & \infty & 0 & & \\ \infty & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix}$$

To find the $(2, 4)$ entry of this matrix we have to consider the paths through the vertex 1 — is there a path from 2 – 1 – 4 that has a better value than the current path? If so, then that entry is updated.

The entire sequence of matrices

$$D^{(2)} = \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & \boxed{3} & \boxed{7} \\ 10 & \infty & 0 & \boxed{12} & \boxed{16} \\ \boxed{3} & 2 & 6 & 0 & 3 \\ \infty & \infty & 6 & \infty & 0 \end{pmatrix} \qquad D^{(3)} = \begin{pmatrix} 0 & \infty & 11 & 2 & 6 \\ 1 & 0 & 4 & 3 & 7 \\ 10 & \infty & 0 & 12 & 16 \\ 3 & 2 & 6 & 0 & 3 \\ \boxed{16} & \infty & 6 & \boxed{18} & 0 \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & \boxed{4} & \boxed{8} & 2 & \boxed{5} \\ 1 & 0 & 4 & 3 & \boxed{6} \\ 10 & \boxed{14} & 0 & 12 & \boxed{15} \\ 3 & 2 & 6 & 0 & 3 \\ 16 & \boxed{20} & 6 & 18 & 0 \end{pmatrix} \qquad D^{(5)} = \begin{pmatrix} 0 & 4 & 8 & 2 & 5 \\ 1 & 0 & 4 & 3 & 6 \\ 10 & 14 & 0 & 12 & 15 \\ 3 & 2 & 6 & 0 & 3 \\ 16 & 20 & 6 & 18 & 0 \end{pmatrix}$$

Finding the actual shortest paths

In both of these algorithms we have not addressed the question of actually finding the paths themselves.

For the Floyd-Warshall algorithm this is achieved by constructing a further sequence of arrays $P^{(k)}$ whose (i, j) entry contains a predecessor of j on the path from i to j. As the entries are updated the predecessors will change — if the matrix entry is not changed then the predecessor does not change, but if the entry does change, because the path originally from i to j becomes re-routed through the vertex k, then the predecessor of j becomes the predecessor of j on the path from k to j.

Summary

1. Priority first search generalizes Prim's algorithm
2. Dijkstra's Algorithm is a priority-first search that can solve the shortest path problem in time $O(E \lg V)$, provided all graph edges have non-negative edge weights.
3. The Bellman-Ford algorithm can solve all shortest path problems and runs in time $O(EV)$.
4. Dynamic Programming is a general approach for solving problems which can be decomposed into sub-problems and where solutions to sub-problems can be combined to solve the main problem.
5. Dynamic Programming can be used to solve the shortest path problem directly or via the Floyd-Warshall formulation.