"The shortest path between two truths in the real domain passes through the complex domain."

– Jacques Salomon Hadamard (1865–1963)

Lecture XIV
MINIMUM COST PATHS

We study digraphs with edge cost functions. Some problems studied under "pure" graphs in Chapter 4 can now be viewed as the special case of unit cost. Connectivity has to do with paths. Connectivity becomes considerably more interesting in the presence of cost functions. The basic problem is this: given vertices u and v, find a path from u to v with the minimum cost. The dynamic programming principle is at work in such problems: if $p = p'; p''$ is a minimum cost path, then p' and p'' must also be minimum cost paths. Minimum cost path algorithms can take advantage of the special nature of the cost function in the following cases:

- All edges have unit cost
- Positive edge costs
- Sparse graph (i.e., most edges have cost ∞)
- Edge costs are symmetric (i.e., we are dealing with bigraphs)
- Graphs that are implicitly or explicitly geometric in origin, or in abstract metric spaces.
- Graphs from real world applications such as road networks or communication networks or biology.

It is clear from this list that the available techniques can be extremely diverse. We have already studied the case of unit edge costs — the algorithm here is breadth first search (BFS). The key algorithmic feature of BFS is the use of a FIFO queue. When generalized to arbitrary positive edge costs, we must replace this FIFO queue by a priority queue.

Minimum cost path problems are usually called "shortest path problems" in the literature. But we shall reserve the term "shortest path" to refer to paths that minimizes the length of the path. We shall generalize such problems to computations over semirings. The important problem of transitive closure problem arises through this generalization.

§1. Minimum Path Problems

¶1. Costed Graphs. Unless otherwise noted, the graphs in this Lecture are digraphs. Let $G = (V, E; C)$ be a digraph with edge cost function

$$C \ : \ E \to \mathbb{R}.$$

We may extend the cost function C to the **cost matrix** $C' : V^2 \to \mathbb{R} \cup \{\infty\}$ where

$$C'(u,v) = \begin{cases} C(u,v) & \text{if } (u,v) \in E, \\ 0 & \text{if } u = v, \\ \infty & \text{else.} \end{cases}$$

Normally, we continue to write C for C'. The simplest cost function is **unit cost** where $C(e) = 1$ for all $e \in E$; this can be generalized to **positive cost functions** where $C(e) > 0$. In constrast to positive costs, we may speak of "general" cost functions to emphasize the possibility of negative costs.

¶2. Convention. The size parameters for complexity considerations are, as usual, $n = |V|$ and $m = |E|$. We usually let $V = [1..n] := \{1, \ldots, n\}$.

¶3. Minimum Cost Paths. Let $p = (v_0 - \cdots - v_k)$ be a path of G, i.e., $(v_{i-1}, v_i) \in E$ for $i = 1, \ldots, k$. The **C-cost of p** is defined to be

$$C(p) := \sum_{i=1}^{k} C(v_{i-1}, v_i).$$

In case of the empty path ($k = 0$), we define $C(p) = 0$. Call p a **C-minimum cost path** if there are no other paths from v_0 to v_k with smaller cost; in this case, $C(p)$ is the **C-minimum cost** from v_0 to v_k. We use the notation $\delta^C(v_0, v_k)$ for this cost:

$$\delta^C(v_0, v_k) := C(p).$$

Reference to C may be omitted when it is understood or irrelevant. For short, we say **minimum path** or **min-paths** instead of "minimum cost path". We view δ as a matrix, the **C-minimum cost matrix**

$$\delta^C : V^2 \to \mathbb{R} \cup \{\pm\infty\}.$$

The special values of positive and negative infinity require clarification: if there is no path from i to j, then we define $\delta^C(i,j) := +\infty$. But there is no path p with cost $C(p) = -\infty$ since the cost of any edge is greater than $-\infty$. Nevertheless, we define $\delta^C(i,j) = -\infty$ in case there exist paths from i to j with arbitrarily negative costs. How could this arise? A cycle $[v_0 - \cdots - v_k]$ is called a **negative cycle** if $\sum_{i=0}^{k} C(v_i, v_{i+1}) < 0$ (here, $v_{k+1} = v_0$). The situation arises if there is a path from i to j that contains a negative cycle. Then there are path that can pass through this cycle an indefinite number of times. This is captured by following basic fact:

Fact 1. *Let $i, j \in [1..n]$. The following are equivalent:*
(i) $\delta(i,j) = -\infty$.
(ii) There exist $k \in [1..n]$ and path $p = p_1; p_2; p_3$, such that $p_1 = (i - \cdots - k)$ and $p_3 = (k - \cdots - j)$ are simple paths, and $p_2 = (k - \cdots - k)$ is a closed path with negative cost.

Note that i, j, k in this basic fact need not be distinct (i.e., we may have $|\{i,j,k\}| < 3$).

¶4. Minimum Path Problems. There are three basic versions:

- **Single-pair minimum paths** Given an edge-costed digraph $G = (V, E; C, s, t)$ with source and sink $s, t \in V$, find any min-path from s to t. This is sometimes called the Point-to-Point (P2P) problem.
- **Single-source minimum paths** Given an edge-costed digraph $G = (V, E; C, s)$ with source $s \in V$, find for each $t \in V$ a min-path from s to t.
- **All-pairs minimum paths** Given an edge-costed digraph $G = (V, E; C)$, for all $s, t \in V$, find a min-path from s to t.

These problems require us to compute min-paths. But there are two cases when there is no min-path from i to j, when $\delta(i, j) = \infty$ or $\delta(i, j) = -\infty$. There does not seem to be an easy way to output a min-path substitute when $\delta(i, j) = \infty$. But when $\delta(i, j) = -\infty$, we could output a path from i to j containing a negative cycle.

Usually, these problems are stated for digraphs. Although the bigraphs can be viewed as special cases of digraphs for the purposes of these problems, we need to be careful in the presence of negative edges. Otherwise, any negative bi-directional edge immediately give us a negative cycle. Special techniques can be used for bigraphs (see §8 and §9).

Clearly the three problems are in order of increasing difficulty, and each is reducible to the other. For example, the single source problem can be reduced to n calls to the single-pair problem.

¶5. Dynamic programming principle and Min-Cost Problems. The dynamic programming principle (Chapter 7) applies to min-paths: subpaths of min-paths are min-paths. A common feature of dynamic programming problems is that we can often simplify the problem of computing an optimal object (e.g., a min-cost path) to computing the optimal value (e.g., min-cost). In other words, for each min-path problem, we can study the corresponding **min-cost version** of the problem. Usually[1] the min-cost algorithms can be simply modified to also compute the min-path(s) as a by-product. Intuitively, this is because the min-costs constitute the critical information that drives these algorithms. So it is pedagogically advantageous to present only the min-cost version of these algorithms.

We adopt this strategy.

¶6. Path Length and Link Distance. If C is the unit cost then $C(p) = k$ is just the **length** of the path $p = (v_0 - \cdots - v_k)$. Consistent with this "length" terminology, we will a path of minimum length a **shortest path**. Sometimes we need to discuss shortest paths as well as minimum cost paths in the same context. This can be confusing since the literature often use "shortest path" for the general minimum cost path. It is therefore useful to call the minimum length of a path from i to j the **link distance** from i to j. Say j is **reachable** from i if the link distance from i to j is finite.

i.e., shortest path ≡ minimum unit cost path

Let k be a non-negative integer. A path p is called a **k-link min-path** if it has minimum cost among all paths with at most k links, from its source to its terminus. Note that p may have less than k links. If p is such a k-line min-path from i to j, we denote the cost of this path

[1] See the Exercises for exceptions to this remark.

$\delta^{(k)}(i,j)$. The case $k=0$ and $k=1$ are simple:

$$\delta^{(0)}(i,j) = \begin{cases} 0 & \text{if } i=j, \\ \infty & \text{else.} \end{cases} \tag{1}$$

For $k \geq 1$, we have this recurrence equation:

$$\delta^{(k)}(i,j) = \min\left\{\delta^{(k-1)}(i,k) + C(k,j) : j \in V\right\}. \tag{2}$$

For instance, when $k=1$, (2) yields:

$$\delta^{(1)}(i,j) = \begin{cases} 0 & \text{if } i=j, \\ C(i,j) & \text{else.} \end{cases}$$

From (2), and using the fact that $C(j,j)=0$, we have:

$$\delta(i,j) \leq \delta^{(k+1)}(i,j) \leq \delta^{(k)}(i,j).$$

Let $\delta^{(k)}$ be the corresponding (at most) **k-link minimum cost matrix**. Unlike the δ matrix, $\delta^{(k)}$ never attain $-\infty$. If there are no negative cycles, it is easy to see that

$$\delta^{(n-1)} = \delta.$$

¶7. Vertex Costed Graphs. Our cost function assigns costs to edges. But it is also reasonable in some applications to assign costs to vertices. In this case we have a **vertex-costed graph**. To convert between edge-costed and vertex-costed graphs, we introduce a graph $b(G)$ with vertex set $V \cup E$ and edges of the form $u{-}e$ and $e{-}v$ whenever $e = u{-}v \in E$. Note that $b(G)$ is a bipartite graph and is sparse (vertex set size $n+m$, edge set size $2m$). Also, $b(G)$ is a bi- or digraph, according as G. Using $b(G)$, we can convert an edge cost on G into a vertex cost of $b(G)$ (since an edge in G is a vertex in $b(G)$) in the obvious way. Conversely, given an vertex cost $C(v)$ on G, we can introduce the edge cost $C(v{-}e)$ on $b(G)$. We have thus shown that *an edge-costed minimum cost problem can be reduced to a vertex-costed minimum cost problem, and vice-versa.* In the theory of algorithms, we are interested in efficient reduction among problems – our reductions is "linearly efficient" assuming sparse graph representations. See Exercises.

¶8. Minimum path tree. A **min-path tree** of $G=(V,E;C)$ is any subgraph T of G such that
(1) T is a rooted tree,
(2) every maximal path in T is either a min-cost path or contains a negative cycle.
(3) for every node u in T, there is a unique maximal path $p(u)$ that contains u.

Maximal paths in (2) refers to paths from root to a leaf or infinite paths. In case there are no negative cycles, condition (2) ensures that every path is finite, and condition (3) ensures that T has size n. If there are negative cycles, condition (2) allows T to have infinite paths, but condition (3) ensures that there are at most n infinite paths.

Call T a **maximal** min-path tree if it is not a proper subgraph of another min-path tree. Thus a maximal min-path tree rooted at x is finite iff there are no negative cycles reachable from x. For example, under unit cost, any BFS tree is a maximal min-path tree. Maximal min-path tree (for a given root x) may not be unique because min-paths between two nodes may be non-unique.

Min-cost trees is an important data structure in min-path algorithms. With a size of $O(n)$, it gives a compact (yet explicit) encoding of up to n min-paths from the root to each of the other nodes. Using n such trees, the output of the all-pairs min-paths problem can be encoded in $O(n^2)$ space.

In the following, consider a subgraph T of $G = (V, E; C)$ with vertex set $U \subseteq V$. Further, assume T is a tree rooted at $s \in V$, and for each $i \in U$, let $d(i)$ denote the cost of the tree path in T from s to each i.

PROPERTY(T): For all $i, j \in U$, we have

$$d(j) \leq d(i) + C(i, j) \tag{3}$$

with equality if $(i{-}j)$ is a tree edge.

We will call the inequality (3) the $i{-}j$ **constraint**. If $i{-}j$ is not an edge, then $C(i, j) = \infty$ and (3) is automatically true. Otherwise, if $i{-}j$ is an edge, the constraint may either be **violated** or **satisfied**. Note that PROPERTY(T) allows the $i{-}j$ constraint to be an equality (thus satisfied) for a non-tree $i{-}j$, but does not require it.

Lemma 1 (Min-path Tree).
(a) If T is a min-path tree, then PROPERTY(T) holds.
(b) If PROPERTY(T) holds, and there are no edges $(i{-}j) \in E$ of the form $i \notin U$ and $j \in U$, then T is a min-path tree.

Proof. (a) The inequality (3) is clearly necessary if $d(j)$ is a min-cost from the root to j. Note that this includes the case where $d(i) = \infty$ or $C(i, j) = \infty$. Moreover, by the dynamic programming principle, this must be an equality when $(i{-}j)$ is a tree edge.
(b) Conversely, assume PROPERTY(T). The additional requirement that there are no edges from outside U into U, implies that all paths from the root to a tree node belongs to the subgraph induced by U. The edges in this subgraph are either tree edges or non-tree edges. The inequality (3) says that non-tree edges are not essential for min-paths. It follows that the tree paths are all min-paths. So T is a min-path tree. **Q.E.D.**

Part (b) gives a sufficient condition for min-path trees.

¶9. Programming Note. Most algorithms in this chapter are quite easy to program. Several of the algorithms require handling $+\infty$ and $-\infty$. Students often represent these values by some large positive or negative numbers such as $+10000$ or -10000. Of course, the largest or smallest representable machine double might play this role as well. In `Java`, a more elegant solution is `Double.POSITIVE_INFINITY` and `Double.NEGATIVE_INFINITY`

Exercises

Exercise 1.1: In min-cost tree, for each node u reachable from the root, we require that there is exactly one maximal path containing that u. Suppose we allow more than one maximal path to any node u. More precisely, let us define **min-path DAG** to be a rooted DAG in which every maximal path in the DAG is either a min-cost path or contains a negative cycle. (We drop the uniqueness property (3) in the min-path tree definition.) Suppose edge costs are non-negative. Is the size of a *maximal* min-path DAG $O(n)$? ♢

Exercise 1.2: Let T_x be any maximal min-path tree rooted at x. This tree may be infinite. Construct from T_x another tree T'_x that has size $O(n)$ and which encodes (in a suitable sense) a min-path to each node reachable from x. ♢

Exercise 1.3: Let $B := \min\{C(e) : e \in E\} < 0$ and let p be a path with cost $C(p) < (n-1)B$. Show the following:
(a) The path p contains a negative cycle.
(b) The bound $(n-1)B$ is the best possible.
(c) If Z is a negative cycle then Z contains a simple negative subcycle. The same is true of positive cycles. ♢

Exercise 1.4: Considers the following min-path problem: each node u has a weight $W(u)$ and the cost of edge (u, v) is $W(v) - W(u)$. (a) Give an $O(n+m)$ algorithm to solve the *minimum cost version* of the single source min-path problem. (b) Convert this algorithm into one that produces the min-paths. ♢

Exercise 1.5: We develop the notion of reduction among graph problems: Let P and Q be two problems. We view a problem P as a binary relation such that if $(I, O) \in P$, it means that the output O is a solution for input I. In general, the relation P need not be functional, so there could be more than one acceptable answer for a given input. We say P is **reducible** to Q if there exist functions T_{in} and T_{out} such that: given any input I to P, if O satisfies $(T_{in}(I), O) \in Q$ then $(I, T_{out}(O)) \in P$ We write P is reduced to Q **via** (T_{in}, T_{out}). The reduction is **linearly efficient** if T_{in} and T_{out} are linear time computable. Note that if I is a graph of size n, m, the linear time means $\Theta(n+n)$. (a) Show that the reduction from vertex-costed problems to edge-costed computation is linear time. (b) Show that the reduction from edge-costed problems to vertex-costed computation is linear time. ♢

End Exercises

§2. Single-source Problem: General Cost

We begin with an algorithm for general cost functions, due to Bellman (1958) and Ford (1962). We assume that the input digraph has the adjacency-list representation. Assuming $V = \{1, \ldots, n\}$ and 1 is the source, we want to compute $\delta(1, i) = \delta_1(i)$ for each $i = 1, \ldots, n$.

¶10. Simple Bellman-Ford Algorithm. The Bellman-Ford algorithm can be implemented using just an array $c[1..n]$ as data structure. At the conclusion of the algorithm, $c[i] = \delta_1(i)$. To bring out the main ideas, we first give a simple version that is correct *provided no negative cycle is reachable from vertex* 1.

The key concept in this algorithm may be captured as follows: let $k \geq 0$.

A number r is called a **k-bound** for (i, j) if

$$\delta(i, j) \leq r \leq \delta^{(k)}(i, j).$$

An array $c[1..n]$ is called a **k-bound** if each $c[i]$ is a k-bound for $(1, i)$. Thus, if we initialize

$$c[i] \leftarrow \begin{cases} 0 & \text{if } i = 1, \\ \infty & \text{if } i > 1, \end{cases}$$

then the array c is a 0-bound. For example, the array $c[1, x, y, z, t] = [0, 1, 3, \infty, 7]$ is a 1-bound for the digraph in Figure 1. But so is $c[1, x, y, z, t] = [0, 1, 3, 4, 5]$. If there are no negative cycles, then a $(n-1)$-bound c will yield the min-cost paths from vertex 1 to all the other vertices. E.g., for Figure 1, the 4-bound $c[1, x, y, z, t] = [0, 1, 2, 3, 4]$ yields the min-costs from 1 to each i.

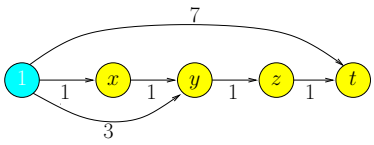


Figure 1: k-paths

So the idea of Bellman-Ford is to start with 0-bound and iteratively compute a $(k+1)$-bound from a k-bound. Each iteration is called a "phase". If there are no negative cycles, then $c[1..n]$ yields the shortest path distances after $n-1$ phases. Here is the definition of a phase:

```
PHASE():
    for each (u−v) ∈ E,
(*)     c[v] ← min{c[v], c[u] + C(u,v)}
```

tightening an inequality is "relaxing"?

Step 2 which updates $c[v]$ in PHASE() is called a u−v **relaxation step**. This terminology might sound odd, but the correct perspective is to imagine that we are constructing a min-path tree: the inequality $c[v] \leq c[u] + C(u,v)$ is just the u−v constraint of (3). After the relaxation step, the constraint is no longer violated, i.e., the constraint has been "relaxed". But notice that by relaxing the u−v constraint, we may have violated a v−w constraint for some w, and in this way, the u−v constraint might later be violated again. Thus the ability to relax all constraints simultaneously is not trivial.

In order to show that some progress is being made, we prove:

LEMMA 2 (Progress in k-bound). *Suppose $c[1..n]$ is a k-bound, and $i \in \{1, \ldots, n\}$.*
(i) For any j, $\min\{c[i], c[j] + C(j,i)\}$ is a k-bound for $(1,i)$.
(ii) $\min\{c[j] + C(j,i) : j = 1, \ldots, n\}$ is a $(k+1)$-bound for $(1,i)$.

Proof. (i) Let $r = c[j] + C(j,i)$. There is nothing to prove if $c[i] \leq r$, since $c[i]$ is already a k-bound for $(1,i)$. So assume $c[i] > r$. We must prove that

$$\delta(1,i) \overset{(*)}{\leq} r \overset{(**)}{\leq} \delta^{(k)}(1,i).$$

The second inequality (**) is immediate from the fact that $r < c[i]$ and $c[i]$ is a k-bound for $(1,i)$. The first inequality (*) follows from $\delta(1,i) \leq \delta(1,j) + C(j,i) \leq c[j] + C(j,i) = r$.
(ii) Let $r = \min\{c[j] + C(j,i) : j = 1, \ldots, n\}$. Note that $r \leq c[i]$ because the number $c[i] = c[i] + C(i,i)$ is included in the minimization that defines r. Thus, r is a k-bound for $(1,i)$ follows from part(i). But why is r a $(k+1)$-bound? For this, we only need to prove the analogue of (**) in part (i):

$$r \leq \delta^{(k+1)}(1,i).$$

There are two possibilities. If $\delta^{(k+1)}(1,i) = \delta^{(k)}(1,i)$, then there is nothing to prove since r is a k-bound. So assume $\delta^{(k+1)}(1,i) < \delta^{(k)}(1,i)$. This implies that

$$\delta^{(k+1)}(1,i) = \delta^{(k)}(1,j) + C(j,i)$$

for some j. Therefore

$$r \leq c[j] + C(j,i) \leq \delta^{(k)}(1,j) + C(j,i) = \delta^{(k+1)}(1,i).$$

Q.E.D.

As corollary, we see that an application of PHASE() will convert a k-bound $c[1..n]$ into a $(k+1)$-bound. The simplified algorithm is just a repeated call to PHASE():

```
Simple Bellman-Ford Algorithm:
    Input:   (V, E; C, s) where V = [1..n] and s = 1.
    Output:  Array c[1..n] representing a (n − 1)-bound.
    ▷ INITIALIZATION
        c[1] ← 0
        for i ← 2 to n
            c[i] ← ∞
    ▷ MAIN LOOP
        for k ← 1 to n − 1
            Phase()   ◁ see above
```

The initialization is regarded as the zeroth phase. It is clear that each phase takes $O(m)$ time for an overall complexity of $O(mn)$. In the absence of negative cycles, $\delta_1 = \delta_1^{(n-1)}$. This implies a $(n-1)$-bound is c in fact equal to δ_1.

We can improve this simple algorithm in two ways. (1) We can try to terminate in less than $n-1$ phases, by detecting whether the array c changed in a Phase. If there is no change, we can stop at once because all the constraints have been relaxed. (2) Second, we need not do an arbitrary order of relaxation steps: we can follow the order produced by a BFS tree. If there are few or no cycles, this will also lead to fast termination.

¶11. Bellman-Ford for Negative Cycles. To remove our assumption about no negative cycles, we need to detect their presence.

Lemma 3 (Negative Cycle Test).
Let $c[1..n]$ be an arbitrary array of finite numbers. Let Z be a negative cycle. Then for some edge $(i-j)$ in Z, we have

$$c[j] > c[i] + C(i,j).$$

In other words, the $i-j$ constraint is violated.

Proof. By way of contradiction, suppose $c[j] \leq c[i] + C(i,j)$ for all edges (i,j) in a negative cycle Z. Summing over all edges in Z,

$$\begin{aligned} \sum_{(i-j)\in Z} c[j] &\leq \sum_{(i-j)\in Z} (c[i] + C(i,j)) \\ &\leq C(Z) + \sum_{(i-j)\in Z} c[i]. \end{aligned}$$

But the summation $\sum_{(i-j)\in Z} c[j]$ appears on both sides of this inequality. Canceling, we see that $0 \leq C(Z)$, a contradiction. **Q.E.D.**

We can easily use this lemma to detect if there are any negative cycles reachable from 1 in the simple Bellman-Ford algorithm. A more interesting application is derive an efficient algorithm for the single source min-cost problem for input graphs $G = (V, E; C, s)$ that may have negative cycles. In this case, we could modify the Relaxation Step (*) in the Phase computation as following:

```
NegativePhase():
    for each (u−v) ∈ E
(**)    If(c[v] > c[u] + C(u, v)) then
            c[v] ← −∞
```

Then we just run this "Negative Phase" computation for another n times. This would propagate the $-\infty$ values to all the needed vertices. But since the negative phase computation is relatively expensive, costing $O(mn)$. We now provide a direct solution with overall complexity $O(m)$..

Let $N \subseteq V$ comprise the vertices with $\delta_s(i) = -\infty$. Clearly, $v \in N$ iff there exists a path from some node in a negative cycle to v. At the end of the Bellman-Ford algorithm, we can gather the set N_0 comprising those v such that $u - v$ constraint is violated for some u at c, i.e., $c[v] > c[u] + C(u, v)$.

CLAIM A: $N_0 \subseteq N$. To see this, note that $v \notin N$ iff $\delta(1, v) = \delta^{(n-1)}(1, v)$. But the later equality implies that the $u - v$ constraint cannot be violated at c. Therefore, any violation of $u - v$ constraint in c implies $v \in N$.

CLAIM B: For each $v \in N$, there is a path from some $v_0 \in N_0$ to v. To see this, note that $v \in N$ iff there exists a path from 1 to v which contains a negative cycle. But this negative cycle must contain a node in N_0 because of Lemma 3.

Using CLAIMS A and B, we now have an $O(m + n)$ method to identify each element in the set N: First, detect the set N_0 by looking for $i-j$ violations. Next, we compute all nodes reachable from N_0 using a simple BFS algorithm. In the usual BFS algorithm, we initialize a FIFO queue Q with a single node s. But now, we initialize the Q with all the nodes of N_0.

```
Subroutine AB:
    Input:   The array c[1..n] from Simple Bellman Ford
    Output:  The set N where i ∈ N iff c[i] = −∞
        ▷ Initialization — populating Q with N_0
        Initialize a queue Q.
        for each (i−j) ∈ E
            If (c[j] > c[i] + C(i, j)) then
                c[j] ← −∞,  Q.enqueue(j)
        ▷ Main Loop — BFS
        While Q ≠ ∅
            i ← Q.dequeue()
            for each (j adjacent to i)    ◁ Inner Loop
                If (c[j] ≠ −∞) then
                    c[j] ← −∞ and Q.enqueue(j).
```

```
General Bellman-Ford Algorithm:
    Input:   (V, E; C, s) with V = [1..n] and s = 1.
    Output:  Array c[1..n] representing δ_1, the min-costs from 1.
1.      Run the Simple Bellman-Ford Algorithm to compute the (n − 1)-bound c[1..n]
2.      Run Subroutine AB to detect the set N; now c[i] = −∞ iff i ∈ N
        Return the array c.
```

The correctness of this algorithm is already shown in the development of this algorithm. The running time of $O(mn)$ comes from the Simple Bellman-Ford algorithm.

¶12. Minimum paths. We now show how min-paths can be computed by a simple modification to the above min-cost algorithm. We maintain another array $p[1..n]$, initialized to `nil`. Each time we update $c[v]$ to some $c[u] + C(u, v)$, we also update $p[v] \leftarrow u$. Assuming there are no negative cycles, it is easy to see that the set of edges $\{(v, p[v]) : v \in V, p[v] \neq \texttt{nil}\}$ forms a min-path tree. We leave it as an exercise to fix this in case of negative cycles.

Exercises

Exercise 2.1: After phase k in the simple Bellman-Ford algorithm, $c[v]$ is the cost of a path from 1 to v of length at most km ($m = |E|$). ♢

Exercise 2.2:
(a) Show that using $n-1$ phases, followed by n end phases in the general Bellman-Ford algorithm is the best possible.
(b) Suppose we mark a vertex j to be **active** (for the next phase) if the value $c[j]$ is decreased during a phase. In the next phase, we only need to look at those edges out of active vertices. Discuss how this improvement affect the complexity of the Bellman-Ford algorithm. ♢

Exercise 2.3: Modify the General Bellman-Ford Algorithm so that we can represent all min-cost paths from vertex 1 to every other $i \in V$. ♢

Exercise 2.4: Suppose R is an $n \times n$ matrix where $R_{i,j} > 0$ is the amount of currency j that you can buy with 1 unit of currency i. E.g., if i represents British pound and j represents US dollar then $R_{i,j} = 1.8$ means that you can get 1.8 US dollars for 1 British pound. A **currency transaction** is a sequence $c_0, c_1, \ldots, c_m$ of $m \geq 1$ currencies such that you start with one unit of currency c_0 and use it to buy currency c_1, then use the proceeds (which is a certain amount in currency c_1) to buy currency c_2, etc. In general, you use the proceeds of the ith transaction (which is a certain amount of currency c_i) to buy currency c_{i+1}. Finally, you obtain a certain amount $T(c_0, c_1, \ldots, c_m)$ of currency c_m.
(a) We call $(c_0, c_1, \ldots, c_m)$ an **arbitrage situation** if $c_m = c_0$ and $T(c_0, c_1, \ldots, c_m) > 1$. Characterize an arbitrage situation in terms of the matrix R.
(b) Give an efficient algorithm to detect an arbitrage situation from an input matrix R. What is the complexity of your algorithm? NOTE: Assuming no transaction costs, it is clear that international money bankers can exploit arbitrage situations.
♢

Exercise 2.5: In the previous question, the algorithm outputs any arbitrage situation. Let $(i_0, i_1, \ldots, i_m)$ be an arbitrage situation where $i_m = i_0$ and $T(i_0, i_1, \ldots, i_m) < 1$ as before. We define the **inefficiency** of this arbitrage situation to be the product $(m \times T(i_0, i_1, \ldots, i_m)$. Thus the large m or $T(i_0, \ldots, i_m)$ is, the less efficient is the arbitrage situation. Give an efficient algorithm if detect the most efficient arbitrage situation. ♢

End Exercises

§3. Single-source Problem: Positive Costs

We now solve the single-source minimum cost problem, *assuming the costs are positive*. The algorithm is from Dijkstra (1959). The input graph is again assumed to have adjacency-list representation.

¶13. Dijkstra's Algorithm: two invariants The idea is to grow a set S of vertices, with S initially containing just the source node, 1. The set S is the set of vertices whose minimum cost from the source is known (as it turns out). Let $U := V \setminus S$ denote the complementary set of "unknown" vertices.

S = *source or stable*,
U = *unknown*

This algorithm has the same abstract structure as Prim's algorithm for minimum spanning tree. We maintain an array $d[1..n]$ of real values where $d[j]$ is the current approximation to $\delta_1(j)$. Initially, the array is given $d[j] = C(1, j)$. In particular, $d[1] = 0$ and $d[j] = \infty$ if $(1{-}j) \notin E$. We require the the array $d[1..n]$ to satisfy the two invariants:

Invariant (S)
If $u \in S$ then $d[u]$ is equal to $\delta_1(u)$, the minimum cost from 1 to u.
This invariant is only concerned about the value of d on S.

Invariant (U)
For each $u \in U$, we have

$$d[u] = \min_{v \in S}\{d[v] + C(v, u)\}. \tag{4}$$

This invariant is only concerned about the value of d on U, but the minimization (4) is over $v \in S$.

Both invariants are initially true, with $S = \{1\}$ and $d[j] = C(1, j)$. Combined with Invariant (S), Equation (4) in Invariant (U) says that $d[u]$ is *the minimum cost ranging over all paths from 1 to u whose intermediate vertices are restricted to S*. Since $d[u]$ is the cost of an actual path, it must be at least the minimum cost, i.e.,

$$d[u] \geq \delta_1(u), \qquad (u \in V). \tag{5}$$

We next see how to extend these invariants when S is expanded to include new elements.

Lemma 4 (Update Lemma).
Under Invariants (S) and (U), we have $d[u_0] = \delta_1(u_0)$ where

$$u_0 := \text{argmin}\,\{d[i] : i \in U\}\,.$$

Slick notation!

The definition of u_0 in this lemma uses the increasingly common "argmin" notation: for any non-empty set X, and function $f : X \to \mathbb{R}$, the notation $\text{argmin}\,\{f(x) : x \in X\}$ refers to any element $x_0 \in X$ that satisfies the equation $f(x_0) = \min\{f(x) : x \in X\}$. Naturally, this is only one of a whole family of "arg" notations (argmax for instance).

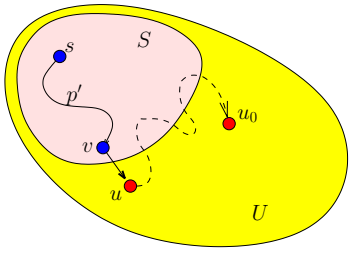


Figure 2: Dijkstra's Invariant

Proof. First, we take care of the trivial case when $\delta_1(u_0) = \infty$. In this case, the lemma is true because (5) implies $d[u] = \infty$.

Otherwise, there exists a min-path p from 1 to u_0. Then we can decompose p into the form

$$p = p'; (v{-}u); p''$$

where $v \in S$ and $u \in U$. See Figure 2. This decomposition exists because the first vertex 1 in p is in S and the last vertex u_0 in p is in U, and so there must be a first time when an edge $(v{-}u)$ exits from S into U. Note that $C(p') = \delta_1(v)$. Then

$$\begin{array}{rcll} d[u_0] & \leq & d[u] & \text{(choice of } u_0 \text{ as minimum)} \\ & \leq & d[v] + C(v, u) & (u \in U \text{ and Invariant (U))} \\ & = & \delta_1(v) + C(v, u) & (v \in S \text{ and Invariant (S))} \\ & = & C(p') + C(v, u) & \text{(dynamic programming principle)} \\ & \leq & C(p) & \text{(since costs are non-negative)} \\ & = & \delta_1(u_0). & \text{(choice of } p) \end{array}$$

Combined with equation (5), we conclude that $d[u_0] = \delta_1(u_0)$. **Q.E.D.**

This Update Lemma shows that if we update S to $S' := S \cup \{u_0\}$, Invariant (S) is preserved. What of Invariant (U)? Well, we need to update the value of $d[i]$ for those $i \in V \setminus S'$ that might be affected by the addition of u_0:

$$d[i] \leftarrow \min\{d[i], d[u_0] + C(u_0, i)\}. \tag{6}$$

This is of course a relaxation step. Moreover, we only need to perform this step if i that are adjacent to u_0. The repeated updates of the set S while preserving Invariants (S) and (U) constitutes Dijkstra's algorithm. The algorithm halts with at at most $n-1$ updates until either the set U is empty or the $\min\{d[i] : i \in U\} = \infty$.

OK, be sure to use the variables `i, j, k` *with the array* `d`*: after all, this is Dijkstra's algorithm*

¶14. Implementation. We are ready to implement Dijkstra's algorithm. The main data structure is a min-priority queue Q to store elements of the set $U = V \setminus S$. Each $x \in U$ is stored in Q with $d[x]$ as its priority. Recall (III.¶6) that min-priority queues normally supports two operations, insertion and deleting the minimum item. But we shall require a third operation called "decreaseKey". These operations are specified as follows:

- `insert`(x, k): this inserts an item x into the queue using priority k.
- `deleteMin`$()$: this returns the item with the minimum priority.
- `decreaseKey`(x, k'): this replace the previous priority k of x with a new priority k' which is is guaranteed to be smaller than k.

The reason we need `decreaseKey` is to support the update of $d[i]$ according to equation (6). Note that `decreaseKey` is essentially equivalent to having a `delete`(x) operation. If we could delete any key, then `decreaseKey`(x, k') amounts to `delete`(x) followed by `insert`(x, k'). Conversely, if we have `decreaseKey` then `delete`(x) amounts to `decreaseKey`$(x, -\infty)$ followed by `deleteMin`$()$.

```
Dijkstra's Algorithm:
    Input:    (V, E; C, s) where V = [1..n] and s = 1.
    Output:   Array d[1..n] with d[i] = δ_1(i).
    ▷ INITIALIZATION
1.      d[1] ← 0; Initialize an empty queue Q.
2.      for i ← 2 to n, d[i] ← ∞,
3.      for i ← 1 to n, Q.insert(i, d[i]).
    ▷ MAIN LOOP
4.      While Q ≠ ∅ do
5.          u_0 ← Q.deleteMin()
6.          for each i adjacent to u_0 do
7.              If ((i ∉ Q) ∧ (d[i] > d[u_0] + C(u_0, i)) then
8.                  d[i] ← d[u_0] + C(u_0, i)
9.                  Q.DecreaseKey(i, d[i])
        end{while}
```

¶15. Hand Simulation. Let us perform a hand-simulation of this algorithm using the graph shown next to Table 1. Such hand simulations are expected in homework sets. Let the source node be A. The array $d[i]$ is initialized to ∞ with $d[A] = 0$. It is updated at each stage: first, we circle the entry that is the extracted minimum for that stage. Then only updated entries of that stage are explicitly indicated. E.g., in Stage 4, the extracted minimum is 9, and the only updated value is 16.

VERTICES	A	B	C	D	E	F	G
STAGE 0	0	∞	∞	∞	∞	∞	∞
STAGE 1	(0)	7	1	10	11		
STAGE 2			(1)				17
STAGE 3		(7)		9		16	
STAGE 4				(9)			16
STAGE 5					(11)		15
STAGE 6							(15)
STAGE 7						(16)	

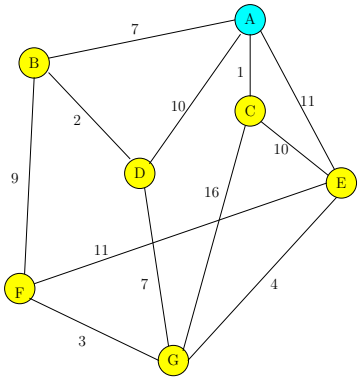


Table 1: Hand Simulation of Dijkstra's Algorithm

¶16. Complexity. Assume Q is implemented by Fibonacci heaps. The initialization (including insertion into the queue Q) takes $O(n)$ time. In the main loop, we do $n-1$ DeleteMins and at most m DecreaseKeys. [To see this, we may charge each DecreaseKey operation to the edge (u_0, i) used to test for adjacency in step 8.] This costs $O(m + n \log n)$, which is also the complexity of the overall algorithm.

If the graph is relatively dense, with $\Omega(n^2/\log n)$ edges, then a more straightforward algorithm might be used that dispenses with the queue. Instead, to find the next minimum for the while loop, we just use an obvious $O(n)$ search. The resulting algorithm has complexity $\Theta(n^2)$.

Exercises

Exercise 3.1: Suppose we have a *vertex-costed* graph $G = (V, E; C, s_0)$. So the cost function is $C : V \to \mathbb{R}$. The cost of a path is just the sum of the costs $C(v)$ of vertices v along the path. Note that all our algorithms for min-cost path problems assume *edge-costed* graphs.
(a) The text gave a detailed version of Dijkstra's algorithm. Please modify this algorithm for our vertex-costed graphs. $C(v) \geq 0$ for each $v \in V$.
(b) Prove that your algorithm in (a) is correct, assuming that that cost function is non-negative, ♢

Exercise 3.2: Carry the hand-simulation of Dijkstra's algorithm for the graph in Table 1, but using the edge costs $C_9(e)$ defined as follows: $C_9(e) = C(e) + 9$ if $C(e) < 9$, and $C_9(e) = C(e) - 9$ if $C(e) \geq 9$. ♢

Exercise 3.3: Show that Dijstra's algorithm may fail if G has negative edge weights (even without negative cycles). ♢

Exercise 3.4: Show that the set S satisfies the additional property that each node in $U = V \setminus S$ is at least as close to the source 1 as the nodes in S. Discuss potential applications where Dijkstra's algorithm might be initialized with a set S that does not satisfy this property (but still satisfying invariants (S) and (U), so that the basic algorithm works). ♢

Exercise 3.5: Give the programming details for the "simple" $O(n^2)$ implementation of Dijstra's algorithm. ♢

Exercise 3.6: Convert Dijkstra's algorithm above into a min-path algorithm. ♢

Exercise 3.7: Justify this remark: if every edge in the graph has weight 1, then the BFS algorithm is basically like Dijkstra's algorithm. ♢

Exercise 3.8: (D.B. Johnson) Suppose that G have negative cost edges, but no negative cycle.
(i) Give an example that cause Dijstra's algorithm to break down.
(ii) Modify Dijstra's algorithm so that each time we delete a vertex u_0 from the queue Q, we look at *all* the vertices of V (not just the vertices adjacent to u_0). For each $i \in V$, we update $c[i]$ in the usual way (line 8 in Dijkstra's algorithm). Prove that this modification terminates with the correct answer.
(iii) Choose the vertex u_0 carefully so that the algorithm in (ii) is $O(n^3)$. ♢

Exercise 3.9: Let C_1, C_2 be two positive cost matrices on $[1..n]$. Say a path p from i to j is (C_1, C_2)-**minimum** if for all paths q from i to j, $C_1(q) \geq C_1(p)$, and moreover, if $C_1(q) = C_1(p)$ then $C_2(q) \geq C_2(p)$. E.g., if C_2 is the unit cost function then a (C_1, C_2)-minimum path between u and v is a C_1-minimum cost path such that its length is minimum among all C_1-minimum paths between u and v. Solve the single-source minimum cost version of this problem. ♢

End Exercises

§4. Goal-Directed Dijkstra

Using any GPS Car Navigation system, you can type in any address and get "a shortest path" from a current location to that address. It takes a few noticeable seconds to do this. This is clearly an important application of minimum cost path computation. But using a plain Dijkstra algorithm will not do: the underlying road network can be pretty large so that even a linear time worst case solution is unacceptable. Nevertheless, Dijkstra's algorithm can serve as the basis for several important and useful extensions. We will look at some of these.

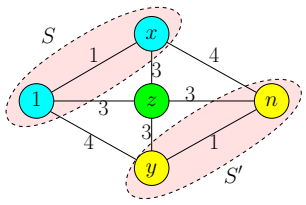


Figure 3: Bidirectional search from 1 to n

¶17. Bidirectional Search. It is interesting to observe that Dijkstra's algorithm has no particular goal, in the sense that we seek a path from the source to *any target*. In many natural settings such as the GPS navigation system application, we have a specific target, say vertex n. An obvious way to speed up Dijkstra is to simultaneously conduct a similar search "backwards" from the target n. The forward search from source 1 maintains a set $S \subseteq V$ with the property that $\delta_1(v)$ is known for each $v \in S$; the backward search from target n maintains a similar set $S' \in V$ for which $\delta_v(n)$ is known for each $v \in S'$. We alternately grow the sets S and S' one element at a time, terminating the moment $S \cap S'$ is non-empty. Let z be the first vertex found by this bidirectional search to be in $S \cap S'$. According to Goldberg and Harrelson [1], the "standard mistake" in bidirectional search is to assume that $\delta_1(n) = \delta_1(z) + \delta_z(n)$.

To illustrate this standard mistake, consider the graph in Figure 3: initially, $S = \{1\}$ and $S' = \{n\}$. After growing the sets S and S' in one round, we get $S = \{1, x\}$ and $S' = \{n, y\}$. Next, $S = \{1, x, z\}$ and $S' = \{n, y, z\}$, and we stop since $S \cap S'$ is non-empty. At this point, $\delta_1(z) = 3$ and $\delta_z(n) = 3$. The standard mistake is to conclude $\delta_1(n) = \delta_1(z) + \delta_z(n) = 3 + 3 = 6$. Of course, we can see that $\delta_1(n)$ is really 5.

What then is the correct bidirectional search algorithm? The above outline and stopping condition is actually correct. However, we need to track the potential values of $\delta_1(n)$ using a global "relaxation variable" Δ. Initially, let $\Delta = \infty$. Each time we add a vertex v to S, for each $v' \in S'$ that is adjacent to v, we relax Δ as follows:

$$\begin{aligned}\Delta(v, v') &\leftarrow \delta_1(v) + \delta_{v'}(n) + C(v, v');\\ \Delta &\leftarrow \min\left\{\Delta, \Delta(v, v')\right\}.\end{aligned}$$

Symmetrically, we must update Δ whenever we add a vertex to S'. At the end of the algorithm, when we first found a $z \in S \cap S'$, we perform one final relaxation step,

$$\Delta \leftarrow \min\left\{\Delta, \delta_1(z) + \delta_z(n)\right\}.$$

We claim that the final value of Δ is the desired $\delta_1(n)$: It is easy to see that $\Delta \leq \delta_1(n)$. The converse is also easily verified because the minimum distance from 1 to n must either have the form $\delta_1(v) + \delta_{v'}(n) + C(v, v')$ or is equal to the $\delta_1(z) + \delta_z(n)$ used in our relaxation. In the example Figure 3, we would have updated Δ from ∞ to 5 when we first added x to S. The algorithm would correctly terminate with $\Delta = 5$.

¶18. A* Search. What is important in the bi-directional search is the additional information from the existence of a goal, or a target vertex n. In general, the goal need not be a single vertex but a set of vertices, In this section, we extend Dijkstra's algorithm by another "goal-directed" heuristic. This idea becomes even more important when the underlying graph G is implicitly defined and possibly infinite, so that termination can only be defined by having attained some goal (E.g. in subdivision algorithms in robot motion planning).

Figure 4. US Road Map

Consider this: let the bigraph $G = (V, E; C)$ represent the road network of the United States with $V = \{1, \ldots, n\}$ representing cities and cost $C(i, j)$ representing the road distance between cities i and j. Again we start from city 1 but our goal set is some $W \subseteq V$. That is, we want the minimum cost path from 1 to any $j \in W$. Let

$$\delta(j, W) := \min\left\{\delta(j, i) : i \in W\right\}.$$

Suppose city 1 is Kansas City (Kansas/Missouri), near the geographical center of the US, and W is the set of cities on the West Coast. A standard Dijkstra search would fan out from Kansas City equally in all directions. Intuitively, our goal-directed search ought to explore the graph G with a westward bias.

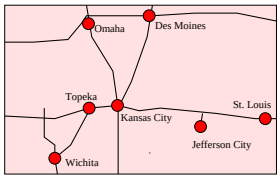
"California or bust" trip

How can Dijkstra's algorithm be modified to serve this purpose? An important heuristic here is the **A* Search** (read "A-star") from Hart, Nilsson and Raphael [2] in the artificial intelligence literature. See Goldberg and Harrelson [1] for an updated algorithmic treatment.

Its justification requires only a slight extension of Dijkstra's algorithm. A function $h : V \to \mathbb{R}$ is called a **heuristic cost function**. We say h is **admissible** if each $h(j)$ $(j \in V)$ is a lower bound on the minimum cost from j to W:

$$0 \leq h(j) \leq \delta(j, W). \tag{7}$$

In our road network example, suppose $c(i, j)$ denote the "crow distance" between cities i and j. This is the distance "as a crow flies", or on a flat earth, it is the Euclidean distance between i and j. Then we define $h(j) = \min\{c(j, i) : i \in W\}$. It is clear that this particular choice of $h(j)$ is admissible. If W is small (e.g., $|W| = 1$), then $h(j)$ is easy to compute. Recall that in Dijkstra's algorithm, we maintain an array $d[1..n]$. We now add the value of $h(j)$ to the value of $d[j]$ in doing our minimizations and comparisons.



Search from Kansas City

For instance, suppose we want to find the shortest path from Kansas City to San Francisco. First, consider the four cities adjacent to Kansas City: Topeka to the west, Omaha and Des Moines to the north, and Jefferson City to the east. Here are the distances (obtained with Map Quest queries):

Distance (in miles)	Topeka	Omaha	Des Moines	Jefferson City	Wichita
Kansas City	61	184	197	161	197
San Francisco	1780	1669	1800	1968	1680
Estimated Distance	1841	1853	1997	2129	1877

Ordinary Dijkstra would begin by considering the distance of Kansas City to the four neighboring cities (Topeka, Omaha, Des Moines, Jefferson City). Since Topeka is the closest of the three cities at 61 miles, we would expand the set $S = \{KansasCity\}$ to $S = \{KansasCity, Topeka\}$. For our A* search, suppose our goal is to reach San Francisco, i.e., $W = \{SanFrancisco\}$. Then we must add to these distances an additional "heuristic distance" (i.e., their estimated distances to San Francisco). For the sake of argument, suppose we added the actual distance of these cities to San Francisco. This is given in the second row in the above table. We see that A* would still choose Topeka to be added to S because its value of 1841 is still minimum. In the next iteration, the ordinary Dijkstra algorithm would choose Jefferson City (at 161 miles). But when the heuristic distance is taking into account, we see that Omaha is the next added city: $S = \{KansasCity, Topeka, Omaha\}$. A* will choose Wichita next. The obvious bias of the search towards the west coast is thus seen.

¶19. Heuristic Cost Function. We now justify the use of heuristic functions (also known as **potential functions**). We need a crucial property which is best approached as follows: suppose our original cost function $C : E \to \mathbb{R}_{\geq 0}$ is modified by h to become

$$C^h(i, j) := C(i, j) - h(i) + h(j). \tag{8}$$

Let $\delta^h(i, j)$ denote the minimum cost from i to j using the modified cost function C^h. Comparing this with the original minimum cost $\delta(i, j)$, we claim:

$$\delta^h(i, j) = \delta(i, j) - h(i) + h(j). \tag{9}$$

This relation is immediate, by telescopy. It follows that a path is minimum cost under C iff it is minimum cost under C^h.

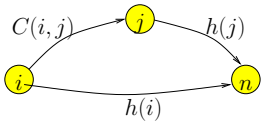
By the **A* Algorithm** with cost function C and heuristic function h, we mean the algorithm that runs Dijkstra's algorithm using C^h as cost function. Clearly, A* Algorithm is a generalization of Dijkstra's algorithm since Dijkstra amounts to using the identically 0 heuristic function.

From our preceding discussion, we know that the minimum cost path will also be found by the A* algorithm. But the order in which we add vertices to the set S can be rather different! There

is one important caveat: since Dijkstra's algorithm is only justified when the cost function is non-negative, the A* Algorithm is only justify if C^h is non-negative. This amounts to the requirement $C(i,j) - h(i) + h(j) \geq 0$, or we prefer to write this as a kind of triangular inequality:

$$C(i,j) + h(j) \geq h(i). \tag{10}$$

This property has various names in the literature: Following Goldberg and Harrelson [1], we say h is **feasible** if (10) holds. The literature also calls h monotone or consistent instead of feasible.



triangular inequality

We note some basic properties of feasible potential functions. We call the set $S^h \subseteq V$ constructed by the A* Algorithm the **scanned set**.

Lemma 5. *Let h and h' be two feasible heuristic functions.*
(i) If $h(j) \leq 0$ for all $j \in W$, then h is admissible.
(ii) The function $\max\{h, h'\}$ is also feasible.
(iii) Let S (resp. S') be the final scanned set when the A Algorithm is searching for the target node n using the heuristic function h (resp. h'). If $h \geq h'$ and $h(n) = h'(n) = 0$ then $S \subseteq S'$.*

¶20. Subdivision Robot Motion Planning. We can apply the goal directed search to the problem of robot motion planning: finding a path from an initial position α to a final position β amidst obstacles. The space we are searching in is now a continuum, not a graph. Nevertheless, we can superimpose a hierarchical grid in the form of a quadtree (in 2-D) or a subdivision tree in general. We can keep track of adjacent boxes that are free, and those that are blocked. Those "mixed" boxes can be further expanded. We can keep track of the connected components of free boxes, and of the "holes" of the blocked boxes. How can we include this heuristic into A^* search?

¶* 21. TIGER Dataset. Suppose you want to carry out experiments with real roadmaps. Where could you get such data? Fortunately, there is a free source for a roadmap of the whole USA in a famous data set called the TIGER Dataset. TIGER stands for *Topologically Integrated Geographic Encoding and Referencing*. The system was developed by the U.S. Bureau of the Census. The data set contains the geographic encoding of the whole USA that includes not only a roadmap but various information such as zip code, landmarks, indicators for metropolitan regions, etc, (presumably important for census purposes). But we may ignore all the other data for our purposes. The data set is organized into files, and files are grouped by counties. The entire United States and its territories are divided into over 3200 counties. Each county may contain up to 17 ASCII files, representing 17 different record types. For road maps, we are interested in three main record types: (1) line features, such as roads and railroads; (2) landmarks, such as schools and parks; and (3) polygon information for area boundaries. The line features and polygon information form the bulk of the data. Each county is given a 6 digit identifier called an FIPS. For instance, the FIPS for New York County (i.e., Manhattan) is 36061 while Kings County (i.e., Brooklyn) is 36047. The first two digits of the FIPS identifies the state (New York State is 36). The files for Manhattan are named TGR36061.RT1, TGR36061.RT2, TGR36061.RTA, etc. The suffix RT1, RT2, etc, tells us that the file contains "Record Type 1", "Record Type 2", etc. The file suffixes are RTn where n is one of the characters 1, 2, ..., 9, A, C, H, I, P, R, S, Z. Each record is stored in one line of the file, and each line has a fixed length. Each record also has a fixed number of fields, and fields occupy predetermined positions in its line.

Coordinate System and Accuracy. The coordinate system is based on Longitude and Latitudes. Each Longitude is given as a signed 9 digit sequence, with an implied 6 decimal places. Thus -123456789 really represents -123.456789. Each latitude is given as a signed 8 digit sequence, also with an implied 6 decimal places. Thus $+12345678$ really represents $+123.45678$. For instance, the bounding box for New York State is (minLon, maxLon, minLat, maxLat) = $(-79.762418, -71.778137, 40.477408, 45.010840)$. The accuracy of the map is that of a 1:100,000 scale map, which is correct up to ± 167 feet. However, the relative positions of any specified point (with respect to the plane subdivision of the Tiger data) is correct. In other words, the geometry of lines and polygons are topologically correct.

Exercises

Exercise 4.1: In the worst case sense, the improvement of bi-directional Dijkstra's algorithm is at most a factor of 2. Construct instances where this improvement factor is arbitrarily large. ♢

Exercise 4.2: Construct an example where the A* Algorithm is incorrect when the heuristic function is infeasible. ♢

Exercise 4.3: Prove Lemma 5. ♢

Exercise 4.4: *The Concept of Reach.* Let $G = (V, E; C)$ be a digraph with positive cost function C. For any simple path $p = p'; u; p''$ that passes through u, define the **reach** of u relative to p to be $R(u; p) = \min\{C(p'), C(p'')\}$. The **reach** of u is then $\max_p R(u; p)$ where p ranges over all shortest paths that passes through u. Suppose we can compute an upper bound $\overline{R}(u)$ on $R(u)$ for all $u \in V$. We use $\overline{R}$ to speed up the Bidirectional Dijkstra's algorithm as follows: whenever ♢

Exercise 4.5: (Project) Download the TIGER Dataset for Manhattan, and construct a graph representing its roads. Carry out experiments including computing min-cost paths using the algorithms in this section. This project can be done in Java, and is best coupled with some visualization capability. ♢

End Exercises

§5. The Algebraic Structure of Min-Cost Paths

It is interesting to note that the all-pairs minimum cost has a algebraic structure that reminds one of matrix multiplication. Thus, the product of two real matrices $A = [A_{ij}]_{i,j=1}^n$ and $B = [B_{ij}]_{i,j=1}^n$, is $C = [C_{ij}]_{i,j=1}^n$ where

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}. \tag{11}$$

Suppose we interpret the multiplication $A_{ik}B_{kj}$ as ordinary addition "$A_{ik} + B_{kj}$", and the summation $\sum_{k=1}^n$ as minimization "$\min_{k=1}^n$":

$$C_{ij} = \min_{k=1}^{n} A_{ik} + B_{kj}. \tag{12}$$

Then we see that C_{ij} is the min-cost 2-link distance from i to j. So the matrix C^2 is the min-cost 2-link distance function $\delta^{=2}$ corresponding to the original cost matrix C. This connection to matrix multiplication is no accident. The goal of this section is to expose some of this algebraic structure of min-cost paths.

What does "algebraic" mean? We are familiar with algebra being the abstraction of elementary arithmetic $+, -, \times, \div$ with special constants like 0 and 1. Here is the big picture when we generalize these ideas: if S is any set, an **algebraic operator** on S is a partial function $\circ : S^k \to S$ for some $k \in \mathbb{N}$ (k is the **arity** of the operator). For instance, division ($\div$) is a partial operator on real numbers. If $k = 0$, we interpret $\circ$ as specifying a constant element of S. An **algebraic structure** is a set S together with one or more operators $\circ_1, \circ_2, \ldots, \circ_k$ defined on S. The structure is denoted $(S, \circ_1, \ldots, \circ_k)$, but we may simply call S the algebraic structure if the operators are understood. Already in these notes, you will encounter several kinds of algebraic structures: rings, monoids, groups, semigroups, fields, semirings, etc. You will see we will recycle our familiar symbols $+$, $\times$, 0, 1, etc, to denote the operators of our algebraic structures. This is for a very good reason: they evoke familiar or expected properties. On the flip side, you also have to be careful not to infer more properties that we explicitly provide.

Let us first review some college algebra: informally, a "ring" is a set R with two special values $0, 1 \in R$ and three binary operations $+, -, \times$ defined satisfying certain axioms. The integers $\mathbb{Z}$ is the simplest example of a ring. Indeed, a ring basically obeys all the algebraic laws you expect to hold for integers $\mathbb{Z}$ under the usual $+/-/\times$ operations. E.g., the distributive law $x(y+z) = xy + xz$ holds for integers, and it is an axiom for rings. The *only* exception is the commutative law for multiplication, $xy = yx$. This law need not hold in rings. A ring that satisfies this law is called a **commutative ring**. All our rings have a multiplicative identity, usually denoted 1, that satisfies $x \cdot 1 = 1 \cdot x = x$. Call 1 the **unity** element. Algebraists sometimes consider rings without a unity element. The set of square $n \times n$ matrices whose entries are integers forms another ring, the **integer matrix ring** $M_n(\mathbb{Z})$. Note that $M_n(\mathbb{Z})$ is no longer commutative for $n \geq 2$. An $n \times n$ matrix A whose (i,j)-th entry is $A_{i,j}$ will be written $A = [A_{i,j}]_{i,j=1}^n$. We often simplify this to $A = [A_{i,j}]$ or $A = [A_{ij}]$ or $A = [A_{ij}]_{i,j}$ when n is understood.

Actually, a ring is a interplay of two underlying simpler algebraic structures. These simpler structures are pervasive, so it is useful to extract them. An algebraic structure

$$(M, +, 0)$$

is a **monoid** if $+$ is an associative binary operation on M with 0 as an identity element. In Computer Science, a good example of a monoid is the set of strings over an alphabet under the concatenation operation, with the empty string as identity. If we dropping the identity element of a monoid, the resulting structure is called a **semigroup**. If we require that the $+$ operator in a monoid has an **inverse** relative to 0, then the algebraic structure is called a **group**. More precisely, in a group we require that every element x has an inverse element y such that $x + y = 0$. We write $-x$ for the inverse of x. A monoid or group is **Abelian** when its operation is commutative. When denoting the group operator by '$\times$' instead of '$+$', we will write inverse of x as x^{-1}.

Formally, then, a **ring** R is an algebraic structure

$$(R, +, \times, 0, 1)$$

that satisfies the following axioms.
(i) $(R, +, 0)$ is an Abelian group,
(ii) $(R, \times, 1)$ is a monoid,
(iii) $\times$ distributes over $+$:

$$(x + y)(x' + y') = xx' + xy' + yx' + yy'. \tag{13}$$

For any ring R and $n \geq 1$, we can derive another ring

$$(M_n(R), +_n, \times_n, 0_n, 1_n)$$

called the **matrix ring of order** n over R. This matrix ring is the set of n-square matrices with entries in R. Addition of matrices, $A +_n B$, is defined componentwise. The product $A \times_n B$ of matrices is defined as in equation (12). The additive and multiplicative identities of $M_n(R)$ are (respectively) the matrix 0_n with all entries 0 and the matrix 1_n of 0's except the diagonal elements are 1's.

Complexity of algebraic computation is a major topic in theoretical computer science. A key problem here is the complexity of matrix multiplication: let $\texttt{MM}(n)$ denote the number of ring operations in R necessary to compute the product of two matrices in $M_n(R)$. The determination of $\texttt{MM}(n)$ has been extensively studied ever since Strassen (1969) demonstrated that the obvious $\texttt{MM}(n) = O(n^3)$ bound is suboptimal. The current record is from Coppersmith and Winograd (1987):

$$\texttt{MM}(n) = O(n^{2.376}).$$

You will see that this bound will control the complexity of some of our algorithms for shortest path problems.

¶22. Connection to shortest paths. Let $V = [1..n]$ and C be a cost matrix $C : V^2 \to \mathbb{R} \cup \{\infty\}$. Recall that the k-link min-cost function $\delta^{(k)}$ gives us the minimum cost of a path with at most k-links between any pair $(i, j) \in V^2$. Consider the variant $\delta^{(=k)}$ where

$$\delta^{(=k)}(i, j) \tag{14}$$

denote the minimum cost of a path from i to j with exactly k-links. In our introduction we show that $\delta^{(=2)}$ is just C^2 when we replace summation by minimization, and product by sum. In other words, C^2 is referring to matrix multiplication in a certain ring-like structure which we denote by

$$(\mathbb{R} \cup \{\pm\infty\}, \min, +, \infty, 0).$$

Here, ∞ and 0 are the respective identities for the minimization and sum operators. In fact, the only property that this ring-like structure lacks to make it a ring is *an inverse for minimization*. Such structures are quite pervasive, and is studied abstractly as "semirings":

Definition 1. *A* **semiring** $(R, \oplus, \otimes, 0, 1)$ *is an algebraic structure satisfying the following properties. We call* $\oplus$ *and* $\otimes$ *the additive and multiplicative operations of* R.

[Additive monoid] $(R, \oplus, 0)$ *is an Abelian monoid.*

[Multiplicative monoid] $(R, \otimes, 1)$ *is a monoid.*

[Annihilator] 0 *is the annihilator under multiplication:* $x \otimes 0 = 0 \otimes x = 0$.

[Distributivity] Multiplication distributes over addition:

$$(a \oplus b) \otimes (x \oplus y) = (a \otimes x) \oplus (a \otimes y) \oplus (b \otimes x) \oplus (b \otimes y)$$

The reader may check that semirings are indeed rings save for the additive inverse. We may call $\otimes$ and $\oplus$ the **semiring multiplication** and **semiring addition** operators, to distinguish them from ordinary multiplication and addition.

¶23. Examples of semirings. Of course, a ring R is automatically a semiring. When viewing R as a semiring, instead of the Abelian group axioms for $(R, +, 0)$, we simply require that it be a monoid with commutativity. The following are examples of semirings that are not rings.

1. The "natural example" of a semiring is the natural numbers $(\mathbb{N}, +, \times, 0, 1)$. It is useful to test all concepts about semirings against this one.
2. We already noted the all important semiring

$$(\mathbb{R} \cup \{\pm\infty\}, \min, +, +\infty, 0) \tag{15}$$

which we may call the **minimization semiring**. Note that the semiring product in this case is denoted $+$, and it is commutative in this case. The $+$ operator is the usual addition except

that we must address the special elements $\pm\infty$. In ordinary extensions of the real numbers to $\pm\infty$, it is stipulated that $\infty + (-\infty)$ is undefined. But here, we have the rule that for any $x \in \mathbb{R}$, $\infty + x = \infty$ and $-\infty + x = -\infty$. However,

$$(-\infty) + \infty = \infty.$$

This is remarkable because in mathematics, $-\infty + \infty$ is usually regarded as undefined. If it is to be defined, it is unclear why the result is $+\infty$ rather than $-\infty$. This ambiguity is resolved by the annihilator axiom which says that the identity element $(+\infty)$ of the semiring addition (min) is the annihilator element for semiring multiplication $(+)$.

Any subring $S \subseteq \mathbb{R}$ induces a sub-semiring $S \cup \{\pm\infty\}$ of this real minimization semiring.

3. As expected, there is an analogous **(real) maximization semiring**,

$$(\mathbb{R} \cup \{\pm\infty\}, \max, +, -\infty, 0). \tag{16}$$

But in this semiring, $\infty + (-\infty) = -\infty$.

4. If we restrict the costs to be non-negative, we get a closely-related **positive minimization semiring**,

$$(\mathbb{R}_{\geq 0} \cup \{\infty\}, \min, +, \infty, 0). \tag{17}$$

5. The **Boolean semiring** is $(\{0,1\}, \vee, \wedge, 0, 1)$ where $\vee$ and $\wedge$ is interpreted as the usual Boolean-or and Boolean-and operations. We sometimes write $\mathbb{B}_2 := \{0,1\}$.
6. The **powerset semiring** is $(2^S, \cup, \cap, \emptyset, S)$ where S is any set and 2^S is the power set of S.
7. The **language semiring** is $(2^{\Sigma^*}, \cup, \cdot, \emptyset, \{\epsilon\})$ where Σ is a finite alphabet and $2^{\Sigma*}$ is the power set of the set Σ^* of finite strings over Σ, and ϵ is the empty string. For sets $A, B \subseteq \Sigma^*$, the semiring addition is set union $A \cup B$, while the semiring multiplication is concatenation $A \cdot B = \{a \cdot b : a \in A, b \in B\}$.
8. The **min-max semiring** is $([0,1], \min, \max, 1, 0)$ with the obvious interpretation. Of course, the max-min semiring is similar.

We let the reader verify that each of the above structures are semirings. As for rings, we can generate infinitely many semirings from an old one:

Lemma 6. *If R is a semiring, then the set $M_n(R)$ of n-square matrices with entries in R is also a semiring with componentwise addition and multiplication analogous to equation (12).*

We call $M_n(R)$ a **matrix semiring** (over R). Note that the multiplication of two matrices in $M_n(R)$ takes $O(n^3)$ semiring operations; in general, nothing better is known because the sub-cubic bounds on $\mathtt{MM}(n)$ which we noted above exploits the additive inverse of the underlying ring.

¶24. Complexity of multiplying Boolean matrices. For Boolean semiring matrices, we can obtain a subcubic bound by embedding their multiplication in the ring of integer matrices. More precisely, if A, B are Boolean matrices, we view them as integer matrices where the Boolean values $0, 1$ are interpreted as the integers $0, 1$. If AB denotes the product over $\mathbb{Z}$, it is easy to see that if we replace each of the non-zero elements in AB by 1, we obtain the correct Boolean product. To bound the bit complexity of this embedding, we must ensure that the intermediate integers do not get large. Note that each entry in AB can be computed in $O(\log n)$ bit operations. Thus, if $\mathtt{MM}_2(n)$ denotes the bit complexity of Boolean matrix multiplication, we have

$$\mathtt{MM}_2(n) = O(\mathtt{MM}(n) \lg n). \tag{18}$$

§6. Closed Semirings

The non-ring semirings we have introduced above can be extended as follows:

Definition 2. *A semiring $(R, \oplus, \otimes, 0, 1)$ is said to be* **closed** *if for any countably infinite sequence $a_1, a_2, a_3, \ldots$ in R, the* **countably infinite sum**

$$\bigoplus_{i \geq 1} a_i$$

is defined, and satisfies the following properties:
0) [Compatibility]

$$a_0 \oplus \left(\bigoplus_{i \geq 1} a_i \right) = \bigoplus_{j \geq 0} a_j.$$

1) [Countable Zero] The a_i's are all zero iff $\bigoplus_{i \geq 1} a_i = 0$.
2) [Countable Associativity]

$$\bigoplus_{i \geq 1} a_i = \bigoplus_{i \geq 1} (a_{2i-1} \oplus a_{2i}).$$

3) [Countable Commutativity]

$$\bigoplus_{i \geq 1} \bigoplus_{j \geq 1} a_{ij} = \bigoplus_{j \geq 1} \bigoplus_{i \geq 1} a_{ij}.$$

4) [Countable Distribution] Multiplication distributes over countable sums:

$$\left(\bigoplus_{i \geq 1} a_i\right) \otimes \left(\bigoplus_{j \geq 1} b_j\right) = \bigoplus_{i,j \geq 1} (a_i \otimes b_j).$$

Let us note some consequences of this definition.
1. By the compatibility and countable zero properties, we can view an element a as the countable sum of $a, 0, 0, 0, \ldots$.
2. Using compatibility and associativity, we can embed each finite sum into a countable sum. E.g., $a \oplus b$ is equal to the countable sum of $a, b, 0, 0, 0, \ldots$. Henceforth, we say **countable sum** to cover both the countably infinite and the finite cases.
3. If σ is any permutation of the natural numbers then

$$\bigoplus_{i \geq 0} a_i = \bigoplus_{i \geq 0} a_{\sigma(i)}.$$

To see this, define $a_{ij} = a_i$ if $\sigma(j) = i$, and $a_{ij} = 0$ otherwise. Then $\bigoplus_i a_i = \bigoplus_i \bigoplus_j a_{ij} = \bigoplus_j \bigoplus_i a_{ij} = \bigoplus_j a_{\sigma(j)}$.
4. If $b_1, b_2, b_3, \ldots$ is a sequence obtained from $a_1, a_2, a_3, \ldots$ in which we simply replaced some pair a_i, a_{i+1} by $a_i \oplus a_{i+1}$, then the countable sum of the b's is equal to the countable sum of the a's. E.g., $b_1 = a_1 \oplus a_2$ and $b_i = a_{i+1}$ for all $i \geq 2$.

All our examples of non-ring semirings so far can be viewed as closed semirings by an obvious extension of the semiring addition to the countably infinite case. Note that "min" in the real semirings should really be "inf" when viewed as closed semiring. A similar remark applies for "max" versus "sup".

The definition of countable sums in the presence of commutativity and associativity is quite non-trivial. For instance, in the ring of integers, the infinite sum $1 - 1 + 1 - 1 + 1 - 1 + \cdots$ is undefined because, by exploiting commutativity, we can make it equal to any integer we like. In terms of min-paths, closed semirings represent our interest in finding the minimum costs of paths of *arbitrary length* rather than paths *up to some finite length*.

For any closed semiring $(R, \oplus, \otimes, 0, 1)$, we introduce an important unary operation: for $x \in R$, we define its **closure** to be

$$x^* := 1 \oplus x \oplus x^2 \oplus x^3 \oplus \cdots$$

where x^k, as expected, denotes the k-fold self-application of $\otimes$ to x. We call x^k the kth **power** of x. Note that $x^* = 1 \oplus (x \otimes x^*)$. For instance, in the real minimization semiring, we see that x^* is 0 and

$-\infty$, depending on whether x is non-negative or negative. When R is a matrix semiring, the closure of $x \in R$ is usually called **transitive closure**. Computing the transitive closures is an important problem. In particular, this is a generalization of the all-pairs minimum cost problem. The transitive closure of Boolean matrices corresponds to the all-pairs reachability problem of graphs.

¶25. Idempotent Semirings. In all our examples of closed semirings, we can verify that the semiring addition $\oplus$ is **idempotent**:

$$x \oplus x = x$$

for all ring elements x. Some authors include idempotence as an axiom for semirings. To show that this axiom is non-redundant, observe that the following structure

$$(\mathbb{N}, +, \times, 0, 1)$$

is a closed semiring if we interpret $+, \times$ in the ordinary way. The semiring addition is, of course, not idempotent. For a finitary example of a closed semiring that is not idempotent, consider

$$(\{0, 1, \infty\}, +, \times, 0, 1).$$

Under idempotence, countable sums is easier to understand. In particular, $\oplus_{i \geq 1} a_i$ depends only on the set of distinct elements among the a_i's.

We can introduce a partial order $\leq$ in an idempotent semiring $(R, \oplus, \otimes, 0, 1)$ by defining

$$x \leq y \quad \text{iff} \quad (x \oplus y) = y.$$

Let us check that this defines a partial order: Clearly $x \leq x$ follows from $(x \oplus x) = x$. If $x \leq y$ and $y \leq x$ then $(x \oplus y) = y$ and $(y \oplus x) = x$. By commutativity of $\oplus$, this shows that $y = (x \oplus y) = (y \oplus x) = x$. Finally, $x \leq y$ and $y \leq z$ implies $x \leq z$ (since $x \oplus z = x \oplus (y \oplus z) = (x \oplus y) \oplus z = y \oplus z = z$). Note that 0 is the minimum element in the partial order since $(0 \oplus x) = x$, and $x \leq y, x' \leq y'$ implies $x \oplus y \leq x' \oplus y'$. Instead of defining the closure a^* operation via countable sum, we can now directly introducing the closure operation to satisfy the axiom

$$ab^*c = \sup_{n \geq 0} ab^n c.$$

An idempotent semiring with such a closure operation is called a **Kleene algebra** (see [3]). This algebra can be defined independently from semirings.

Exercises

Exercise 6.1: The axioms for a ring did include an annihilator axiom. Why is this? ♢

Exercise 6.2: Recall that the minimization semiring $\mathbb{R} \cup \{\pm\infty\}$ (15). Check that it is idempotent, and recall the definition of "$\leq$" for such rings. What is unusual about this definition? ♢

Exercise 6.3: Show that in a ring R: $-x = (-1) \cdot x$, and $x \cdot 0 = 0 \cdot x = 0$ for all $x \in R$. ♢

Exercise 6.4: Give examples of groups that are not Abelian. HINT: consider words over the alphabet $\{x_i, \bar{x}_i : i = 1, \ldots, n\}$ with the cancellation law $x_i \bar{x}_i = \bar{x}_i x = \epsilon$. ♢

Exercise 6.5: Under what conditions does the canonical construction of $\mathbb{Z}$ from $\mathbb{N}$ extend to give a ring from a semiring? ♢

Exercise 6.6: Which of the following is true for the closure operator?
(i) $(x^*)^2 = x^*$.
(ii) $(x^*)^* = x^*$.
(iii) For all x, $y = x^*$ is the only solution to the equation $y = 1 \oplus (x \otimes y)$. ♢

Exercise 6.7: Generalize the problem of optimal triangulation (§VII.7) so that the weight function has values in an idempotent semiring. If the semiring product is not commutative, how do you make the problem meaningful? ♢

End Exercises

§7. All-Pairs Minimum Cost: Dense Case

The input digraph G has a general cost function. Informally, we may take "dense" to mean that G satisfies $m = \Theta(n^2)$. To solve the all-pairs problem for G, we could, of course, run Bellman-Ford's algorithm for a total of n times, for an overall complexity of $O(n^2 m) = O(n^4)$. We shall improve on this.

In this problem, we shall represent the costed graph by its cost matrix $C = [C_{i,j}]_{i,j=1}^n$. This representation is justified since G is dense. The underlying semiring is assumed to be the minimization semiring (see (15)).

¶26. Reduction to Transitive Closure. Let C be a cost matrix of real numbers. Recall that (**??**) shows that the exact 2-link min cost function $\delta^{=2}$ is the the square $C^2 = C \cdot C$ where the matrix multiplication is performed over the minimization semiring (15). In general, we have:

Lemma 7. *Let C be cost matrix regarded as a matrix over the minimization semiring. Then the k-th power C^k of C gives the k-link minimum cost function $\delta^{(k)}$:*

$$\delta^{(=k)}(i,j) = C^k_{ij}.$$

where $C^k = [C^k_{ij}]_{i,j}$.

As corollary, the all-pairs min-path problem is equivalent to the problem of computing the transitive closure C^* of C since for all i, j:

$$(C^*)_{ij} = \inf_{k \geq 0} \{C^{(=k)}_{ij}\}.$$

Since semiring matrix multiplication takes $O(n^3)$ time, it follows that we can determine C^k by $k-1$ matrix multiplications, taking time $O(n^3 k)$. But this can be improved to $O(n^3 \log k)$ by exploiting associativity. The method is standard: to compute C^k, we first compute the sequence

$$C^1, C^2, C^4, \ldots, C^{2^\ell},$$

where $\ell = \lfloor \lg k \rfloor$. This costs $O(n^3 \ell)$ semiring operations. By multiplying together some subset of these matrices together, we obtain C^k. This again takes $O(n^3 \ell)$. This gives a complexity of $O(n^3 \log n)$ when $k = n$. In case C has no negative cycles, $C^* = C^{n-1}$ and so the transitive closure can be computed in $O(n^3 \log n)$ time.

¶27. The Floyd-Warshall Algorithm. We now improve the $O(n^3 \log n)$ bound to $O(n^3)$. This algorithm is usually attributed[2] to Robert Floyd (1962) and Stephen Warshall (1962). An advantage to the Floyd-Warshall algorithm is that it does need not to assume the absence of negative cycles. To explain this algorithm, we define a k-**path** ($k \in [1..n]$) of a digraph to be any path

$$p = (v_0 - v_1 - \cdots - v_\ell)$$

whose the vertices in p, with the exception of v_0, v_ℓ, belong to the set $[1..k]$. Unlike the k-link cost function $\delta^{(k)}$, we impose no bound on the length ℓ of the path p. By extension to the case $k = 0$, we may say that a 0-path is any path of length at most 1. Let

$$\delta^{[k]}(i, j)$$

denote the cost of the minimum cost k-path from i to j. For instance $\delta^{[0]}(i, j) = C_{ij}$. It follows that the following equation holds for $k \geq 1$:

$$\delta^{[k]}(i, j) = \min\{\delta^{[k-1]}(i, j), \quad \delta^{[k-1]}(i, k) + \delta^{[k-1]}(k, k)^* + \delta^{[k-1]}(k, j)\} \tag{19}$$

where we define for any $r \in \mathbb{R} \cup \{\pm\infty\}$,

$$r^* = \begin{cases} 0 & \text{if } r \geq 0, \\ -\infty & \text{if } r < 0. \end{cases}$$

Notice that $\delta^{[n]}(i, j)$ is precisely equal to $\delta(i, j)$. The Floyd-Warshall algorithm simply uses equation (19) to compute $\delta^{[k]}$ for $k = 1, \ldots, n$:

```
FLOYD-WARSHALL ALGORITHM:
    Input:   An n × n cost matrix C representing a digraph.
    Output:  Matrix c[1..n, 1..n] representing δ.
    ▷ Initialization
        for (i = 1 to n)
            for (j = 1 to n)
                c[i, j] ← C_ij
    ▷ Main Loop
        for (k = 1 to n)    ◁ This is "Stage k"
            for (i = 1 to n)
                for (j = 1 to n)
(A)                 c[i, j] ← min{c[i, j],   c[i, k] + (c[k, k])* + c[k, j]}
        Return(matrix c)
```

Note that the outermost for-loop determines the "kth stage" of the algorithm ($k = 1, \ldots, n$). Step (A) is intended to be an implementation of (19). But it is not an exact transcription of equation (19) because the value of $c[i, j]$ in the kth stage is not necessarily equal to $\delta^{[k-1]}(i, j)$, but some intermediate value in the range

$$\delta^{[k]}(i, j) \leq c[i, j] \leq \delta^{[k-1]}(i, j).$$

There is a similar situation in the Bellman-Ford Algorithm; the overall correctness is not affected.

But here is a subtlety concerning the proper interpretation of Step (A), in particular, of the expression:

$$c[i, k] + (c[k, k])^* + c[k, j]. \tag{20}$$

When $c[k, k] < 0$, we have $(c[k, k])^* = -\infty$. Then (20) becomes

$$c[i, k] + (-\infty) + c[k, j].$$

What happens if $c[i, k]$ or $c[k, j]$ is $+\infty$? We have to evaluate the expression "$+\infty + (-\infty)$". Although this expression is regarded as undefined in mathematics, in this context, our application forces us to

[2] It is also called the Roy-Floyd-Warshall Algorithm since Bernard Roy (1959) had also proposed a similar algorithm. The method is also similar to the standard proof of Kleene's characterization of regular languages.

equate this expression to $+\infty$. Why? Because this case arises if there is no edge from i to k (or from k to j), and so to define a value other than $+\infty$ would be most counter intuitive. Of course, we already know that our annihilator axiom requires the $+\infty$ interpretation.

The Floyd-Warshall algorithm clearly takes $\Theta(n^3)$ time because of its triply nested loop. The correctness can be proved by induction (provided we interpret the expression (20) appropriately).

¶28. Incremental Floyd-Warshall. We now consider an "incremental approach" to solving the all-pairs min-cost problem Suppose G_k is the subgraph of G induced by $V_k = \{1, \ldots, k\}$. The idea of the incremental algorithm is to solve the all-pairs problem for G_k, assuming that the solution is known for G_{k-1}. Call this the kth Update Step. If we can do the kth update in time $O(k^2)$, we would have an alternative $O(n^3)$ algorithm. Such an incremental algorithm was first defined by J. Jaffar, M. J. Maher, P. J. Stuckey, and R. H. C. Yap in the paper "Beyond finite domains", Lecture Notes in Computer Science, Springer-Verlag, pages 86–94, 1994.

In preparation for this development, we must modify the concept of a k-path. We call a path in G_k a **strict k-path**. Thus a strict k-path is also a k-path, but not vice-versa in general. Suppose p is a simple path in G_k. Recall that a path is simple if no vertex is duplicated, except possibly for the first and last. The following is immediate:

Lemma 8. *Suppose p is a simple path in G_k i to j, for some $k \geq 2$. If p is not a path in G_{k-1} then there exists paths p' and p'' in G_{k-1} and vertices $i', j' \in V_{k-1}$ such that one of the following holds:*

- *CASE 1: $i = k = j$, and $p = (k{-}i'); p'; (j'{-}k)$.*
- *CASE 2: $i \neq k = j$, and $p = p'; (j'{-}k)$.*
- *CASE 3: $i = k \neq j$, and $p = (k{-}i'); p'$.*
- *CASE 4: i, j, k are all distinct, and $p = p'; (i'{-}k{-}j'); p''$.*

Based on these four cases, we can now provide the incremental Floyd-Warshall Algorithm. A matrix $c[1..n, 1..n]$ has **PROPERTY**(k) if the following holds:

$$c[i,j] = \begin{cases} \delta(i,j) & \text{if } i,j \in V_{k-1} \\ C(i,j) & \text{else.} \end{cases}$$

The Update Step amounts to converting a matrix with PROPERTY$(k-1)$ into one with PROPERTY(k).

```
Floyd-Warshall Update Step
    Input: k and matrix c[1..n, 1..n]
        where c has PROPERTY(k − 1) and k = 2, ..., n.
    Output: matrix c with PROPERTY(k).
        ▷ CASE 1
        Done ← false
        for i' = 1, ..., k − 1
            If (Done) break
            for j' = 1, ..., k − 1
                If (Done) break
                If (C(k, i') + δ(i', j') + C(j', k) < 0) then
                    δ(k, k) = −∞
                    Done ← true
        ▷ CASES 2 and 3
        for i = 1, ..., k − 1
            for j' = 1, ..., k − 1
                δ(i, k) ← min {δ(i, k), δ(i, j') + C(j', k) + δ(k, k)}
                δ(k, i) ← min {δ(k, i), δ(k, k) + C(k, j') + δ(j', i)}
        ▷ CASE 4
        for i = 1, ..., k − 1
            for j = 1, ..., k − 1
                δ(i, j) ← min {δ(i, j), δ(i, k) + δ(k, k) + δ(k, j)}
```

Correctness Proof. There are three double for-loops in the algorithm: CASE 1, CASES 2& 3, and CASE 4. The sequencing of these three parts are important:

- The first double for-loop detects whether there is a negative cycle through k. Once this is detected, we can break out of the double loop, as seen in the tests for the Boolean variable *Done*. It is clear that if $\delta(k,k) = -\delta$, we have a negative cycle. Conversely, if there is a negative cycle through k, then for some $i', j' \in V_{k-1}$, we obtain $\delta(k,i') + \delta(i',j') + \delta(j',k) < 0$. This proves that $\delta(k,k)$ is correctly determined.
- The second double for-loop determines $\delta(i,k)$ and $\delta(k,i)$ for each $i \in V_{k-1}$. By symmetry, we only discuss $\delta(i,k)$. Note that if i can reach k, then there are two cases: either $\delta(i,k) = \delta(i,j') + C(j',k)$ for some j', or $\delta(i,k) = -\infty$. In either case, we obtain $\delta(i,k) = \delta(i,j') + C(j',k) + \delta(k,k)$ for some j'. Such a j' will be discovered by the inner for-loop.
- The last double for-loop determines $\delta(i,j)$ for $i, j < k$. We already have the value of $\delta(i,j)$ when restricted to strict $(k-1)$-paths. The remaining cases to minimize over are paths from i to k and from k to j. These have been captured the previous double loop, and we are checking these.

¶29. What is Floyd-Warshall for bigraphs? Floyd-Warshall is traditionally defined for digraphs. Suppose we let C be a symmetric matrix representing a bigraph. There is an issue in evaluating the expression (20): when $i = j$, then it becomes

$$c[i,k] + (c[k,k])^* + c[k,i] = 2 \cdot c[i,k] + (c[k,k])^*$$

since $c[i,k] = c[k,i]$. If the edge $(i{-}k)$ is negative, then $c[i,k] = C(i,j) < 0$ implies $c[i,i] < 0$, i.e., vertex i is regarded as part of a negative cycle. This happens for every i that is incident to a negative edge. For bigraphs, this is not an interesting interpretation of negative cycles. The idea is to discount cycles of the form $i{-}j{-}i$; in ¶IV.9, we call a path of the form $p = p'; (i{-}j{-}i); p''$ reducible. We must define $\delta(i,j)$ to mean the minimum cost over *irreducible* paths from i to j.

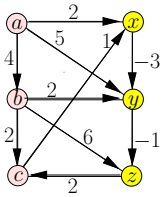
We can use the above incremental Floyd-Warshall algorithm. We just have to ensure that in CASE 1, we avoid checking reducible paths, by requiring the additional condition $i' \neq j'$ when we update $\delta(k,k) = -\infty$ in Step (*). Here is the replacement for Step (*):

(*)	If $(i' \neq j')$ and $(\delta(k,i') + \delta(i',j') + \delta(j',k) < 0)$ then $\delta(k,k) = -\infty$

We leave as an exercise to show the correctness of this algorithm for bigraphs.

Exercises

Exercise 7.1: (a) Run the Floyd-Warshall Algorithm on the digraph in Figure 5(a). Note that there is a negative cycle.
(b) Suppose the digraph is made into a bigraph by omitting directions on each edge. By symmetry, just the upper triangular entries of the cost matrix need to be shown, as in Figure 5(b). Show 6 more such matrices. ♢



(a)

C:	a	b	c	x	y	z
a	0	4		2	5	
b		0	2		2	6
c			0	1		2
x				0	−3	
y					0	−1
z					0	

(b)

Figure 5: Digraph and bigraph

Exercise 7.2: Run the Floyd-Warshall Algorithm on the following matrix: $C^{(0)} = \begin{bmatrix} 0 & -1 & \infty \\ \infty & 0 & -1 \\ -1 & \infty & 0 \end{bmatrix}$. ♢

Exercise 7.3: The transitive closure of the cost matrix C was computed as C^{n-1} in case C has no negative cycles. Extend this methods to the case where C may have negative cycles. ♢

Exercise 7.4: Modify the Floyd-Warshall Algorithm in order to compute a representative min-cost path for each $i, j \in [1, \ldots, n]$. ♢

Exercise 7.5: Consider the min-cost path problem in which you are given a digraph $G = (V, E; C_1, \Delta)$ where C_1 is a positive cost function on the edges and Δ is a positive cost function on the vertices. Intuitively, $C_1(i,j)$ represents the time to fly from city i to city j and $\Delta(i)$ represents the time delay to stop over at city i. A jet-set business executive wants to construct matrix M where the (i,j)th entry $M_{i,j}$ represents the "fastest" way to fly from i to j. This is defined as follows. If $\pi = (v_0, v_1, \ldots, v_k)$ is a path, define

$$C(\pi) = C_1(\pi) + \sum_{j=1}^{k-1} \Delta(v_j)$$

and let $M_{i,j}$ be the minimum of $C(\pi)$ as π ranges over all paths from i to j. Please show how to compute M for our executive. Be as efficiently as you can, and argue the correctness of your algorithm. ♢

Exercise 7.6: Same setting as the previous exercise, but Δ can be negative. (There might be "negative benefits" to stopping over at particular cities). For simplicity, assume no negative cycles. ♢

Exercise 7.7: An edge $e = (i,j)$ is **essential** if $C(e) = \delta(i,j)$ and there are no alternative paths from i to j with cost $C(e)$. The subgraph of G comprising these edges is called the **essential subgraph** of G, and denoted G^*. Let m^* be the number of edges in G^*.
(i) For every i, j, there exists a path from i to j in G^* that achieves the minimum cost $\delta_G(i,j)$.
(ii) G^* is the union of the n single-source shortest path trees.

(iii) Show some $C > 0$ and an infinite family of graphs G_n such that G_n^* has $\geq Cn^2$ edges.
(iv) (Karger-Koller-Phillips, C. McGeoch) Assume positive edge costs. Solve the all-pairs minimum cost problem in $O(nm^* + n^2 \log n)$. HINT: From part (ii), we imagine that we are constructing G^* by running n copies of Dijkstra's algorithm simultaneously. But these n copies are coordinated by sharing one common Fibonacci heap. ♢

Exercise 7.8: Modify the Floyd-Warshall Algorithm so that it computes the lengths of the first and also the second minimum path. The second min-path must be distinct from the min-path. In particular, if the min-path does not exist, or is unique, then the second min-path does not exist. In this case, the length is ∞. ♢

End Exercises

§8. Transitive Closure

The Floyd-Warshall algorithm can also be used to compute transitive closures in $M_n(R)$ where $(R, \oplus, \otimes, 0, 1)$ is a closed semiring. For any sequence $w = (i_0, \ldots, i_m) \in [1..n]^*$ $(m \geq 1)$, define

$$C(w) := \bigotimes_{j=1}^{m} C(i_{j-1}, i_j).$$

If $m = 0$, $C(w) := 1$ (the identity for $\otimes$). For each $k = 0, \ldots, n$, we will be interested in sequences in $w \in i[1..k]^* j$, which may be identified with k-paths. We define the matrix $C^{[k]} = [C_{ij}^{[k]}]$ where

$$C_{ij}^{[k]} = \bigoplus_{w \in i[k]^* j} C(w).$$

Lemma 9.
(i) $C^{[0]} = C$ *and for* $k = 1, \ldots, n$,

$$C_{ij}^{[k]} = C_{ij}^{[k-1]} \oplus \left(C_{ik}^{[k-1]} \otimes (C_{kk}^{[k-1]})^* \otimes C_{kj}^{[k-1]} \right) \tag{21}$$

(ii) $C^{[n]} = C^*$.

Proof. We only verify equation (21), using properties of countable sums:

$$\begin{aligned}
C_{ij}^{[k]} &= \left(\bigoplus_{w \in i[1..k-1]^* j} C(w) \right) \oplus \left(\bigoplus_{w \in i[1..k-1]^* k[1..k]^* j} C(w) \right) \\
&= C_{ij}^{[k-1]} \oplus \left(\left(\bigoplus_{w' \in i[1..k-1]^* k} C(w') \right) \otimes \left(\bigoplus_{w'' \in k[1..k]^* j} C(w'') \right) \right) \\
&= C_{ij}^{[k-1]} \oplus \left(C_{ik}^{[k-1]} \otimes \left(\bigoplus_{w' \in k[1..k]^* k} C(w') \right) \otimes \left(\bigoplus_{w'' \in k[1..k-1]^* j} C(w'') \right) \right) \\
&= C_{ij}^{[k-1]} \oplus \left(C_{ik}^{[k-1]} \otimes \left(\bigoplus_{w \in k[1..k]^* k} C(w) \right) \otimes C_{kj}^{[k-1]} \right).
\end{aligned}$$

It remains to determine the element $x = \bigoplus_{w \in k[1..k]^* k} C(w)$. It follows from countable commutativity that

$$x = 1 \oplus C_{kk}^{[k-1]} \oplus (C_{kk}^{[k-1]})^2 \oplus (C_{kk}^{[k-1]})^3 \oplus \cdots = (C_{kk}^{[k-1]})^*,$$

as desired. **Q.E.D.**

In practice, we can actually do better than (21). Suppose we do not keep distinct copies of the $C^{[k]}$ matrix for each k, but have only one C matrix. Then we can use the update rule

$$C_{ij} = C_{ij} \oplus (C_{ik} \otimes (C_{kk})^* \otimes C_{kj}) . \tag{22}$$

It may be verified that this leads to the same result. However, we may be able to terminate earlier.

We use the analogue of equation (21) in line (A) of the Floyd-Warshall algorithm. The algorithm uses $O(n^3)$ operations of the underlying closed semiring operations.

¶30. Boolean transitive closure. We are interested in computing transitive closure in the matrix semiring $M_n(B_2)$, where $B_2 = \{0, 1\}$ is the closed Boolean semiring. Let $\mathtt{TC}_2(n)$ denote the bit complexity of computing the transitive closure in $M_n(B_2)$. Here "complexity" refers to the number of operations in the underlying semiring B_2. The Floyd-Warshall algorithm shows that

$$\mathtt{TC}_2(n) = O(n^3).$$

We now improve this bound by exploiting the bound

$$\mathtt{MM}_2(n) = O(\mathtt{MM}(n) \log n) = o(n^3)$$

(see equation (18)). *We may assume that* $\mathtt{MM}_2(n) = \Omega(n^2)$ *and* $\mathtt{TC}_2(n) = \Omega(n^2)$. This assumption can be verified in any reasonable model of computation, but we will not do this because it would involve us in an expensive detour with little insights for the general results. This assumption also implies that $\mathtt{MM}_2(n)$ is an upper bound on addition of matrices, which is $O(n^2)$. Our main result will be:

Theorem 10. $\mathtt{TC}_2(n) = \Theta(\mathtt{MM}_2(n))$.

In our proof, we will interpret a matrix $A \in M_n(B_2)$ as the adjacency matrix of a digraph on n vertices. So the transitive closure A^* represents the **reachability matrix** of this graph:

$$(A^*)_{ij} = 1 \text{ iff vertex } j \text{ is reachable from } i.$$

We may assume n is a power of 2. To show that $\mathtt{TC}_2(n) = O(\mathtt{MM}_2(n))$, we simply note that if $A, B \in M_n(B_2)$ then the reachability interpretation shows that if

$$C = \begin{pmatrix} 0 & A & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{pmatrix}$$

then

$$C^* = I + C + C^2 = \begin{pmatrix} I & A & AB \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}.$$

Thus, we can reduce computing the product AB to computing the transitive closure of $C \in M_{3n}(B_2)$:

$$\mathtt{MM}(n) = O(\mathtt{TC}_2(3n)) + O(n^2) = O(\mathtt{TC}_2(n)).$$

Now we show the converse. Assuming that $A, B, C, D \in M_n(B_2)$, we claim that

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}^* = \begin{pmatrix} E^* & E^*BD^* \\ D^*CE^* & D^* + D^*CE^*BD^* \end{pmatrix}, \tag{23}$$

where

$$E := A + BD^*C.$$

This formidable-looking expression (23) has a relatively simple combinatorial explanation using the reachability interpretation. Assume the matrix of interest has dimensions $2n \times 2n$ and it has been partitioned evenly into A, B, C, D. If the vertices of the corresponding graph G is $[1..2n]$ then A represents the subgraph induced by $[1..n]$, D the subgraph induced by $[n+1..2n]$, B the bipartite graph comprising edges from vertices in $[1..n]$ to those in $[n+1..2n]$, and C is similarly interpreted. Now E represents the reachability relation on $[1..n]$ determined by paths of G that makes *at most one detour outside* $[1..n]$. It is then clear that E^* represents the reachability relation of G, restricted to those vertices in $[1..n]$. This justifies the top-left submatrix in the RHS of equation (23). We leave it to the reader to similarly justify the other three submatrices on the RHS.

Thus, the RHS is obtained by computing, in this order:

$$\begin{array}{lll} D^* & \text{(costing} & \mathtt{TC}_2(n)), \\ E & \text{(costing} & O(\mathtt{MM}_2(n))), \\ E^* & \text{(costing} & \mathtt{TC}_2(n)), \end{array}$$

and finally, the remaining three submatrices on the RHS of equation (23). The total cost of this procedure is

$$\mathtt{TC}_2(2n) = 2\mathtt{TC}_2(n) + O(\mathtt{MM}_2(n))$$

which has solution $\mathtt{TC}_2(2n) = O(\mathtt{MM}_2(n))$. This shows $\mathtt{TC}_2(n) = O(\mathtt{MM}_2(n))$, as desired.

Exercises

Exercise 8.1: Rewrite update rule (21) that corresponds to the improved rule (22). In other words, show when the update of $C_{ij}^{[k]}$ is sometimes using an "advance value" on the right-hand side. ♢

Exercise 8.2: Give similar interpretations for the other three entries of the RHS of equation (23). ♢

Exercise 8.3: Express the RHS of equation (23) as a product of three matrices

$$\begin{pmatrix} I & 0 \\ D^*C & I \end{pmatrix} \begin{pmatrix} E^* & 0 \\ 0 & D^* \end{pmatrix} \begin{pmatrix} I & BD^* \\ 0 & I \end{pmatrix},$$

and give an interpretation of the three matrices as a decomposition of paths in the underlying graph. ♢

End Exercises

§9. All-pairs Minimum Cost: Sparse Case

Donald Johnson gave an interesting all-pairs minimum cost algorithm that runs in $O(n^2 \log n + mn)$ time. This improves on Floyd-Warshall when the graph is sparse (say $m = o(n^2)$). *Assume that there are no negative cycles in our digraph* $G = (V, E; C)$. The idea is to introduce a **potential function**

$$\phi : V \to \mathbb{R}$$

and to modify the cost function to

$$\widehat{C}(i,j) = C(i,j) + \phi(i) - \phi(j). \tag{24}$$

(Cf. (8) in A-star search.) We want the modified cost function $\widehat{C}$ to be non-negative so that Dijkstra's algorithm is applicable on the modified graph $\widehat{G} = (V, E; \widehat{C})$.

But how are min-paths in $\widehat{G}$ and in G related? Notice that if p, p' are two paths from a common start to a common final vertex then

$$\widehat{C}(p') - \widehat{C}(p) = C(p') - C(p).$$

This proves:

Lemma 11. *A path is a minimum cost path in $\widehat{G}$ iff it is minimum cost path in G.*

Suppose s is a vertex that can reach all the other vertices of the graph. In this case, we can define the potential function to be

$$\phi(v) := \delta(s, v).$$

Note that $\phi(v) \neq -\infty$ since we stipulated that G has no negative cycle. Also $\phi(v) \neq \infty$ since s can reach v. The following inequality is easy to see:

$$\phi(j) \leq \phi(i) + C(i, j)$$

Thus we have:

Lemma 12. *Assuming there are no negative cycles, and $s \in V$ can reach all other vertices, the above modified cost function $\widehat{C}$ is non-negative,*

$$\widehat{C}(i, j) \geq 0.$$

In particular, there are no negative cycles in $\widehat{G}$. To use the suggested potential function, we need a vertex that can reach all other vertices. This is achieved by introducing an artificial vertex $s \notin V$ and using the graph $G' = (V \cup \{s\}, E'; C')$ where $E' = E \cup \{(s, v) : v \in V\}$ and for all $i, j \in V$, let

$$C'(i, j) = \begin{cases} 0 & \text{if } i = s \\ \infty & \text{if } j = 2 \\ C(i, j) & \text{else.} \end{cases}$$

Call G' the **augmentation of G with s**. Note that G' has no negative cycle iff G has no negative cycle; furthermore, for a path p between two vertices in V, p is a min-path in G iff it is a min-path in G'. This justifies the following algorithm.

Johnson's Algorithm:
Input: Graph $(V, E; C)$ with general cost, no negative cycle.
Output: All pairs minimum cost matrix.
▷ *INITIALIZATION*
Let $(V', E'; C')$ be the augmentation of $(V, E; C)$ by $s \notin V$.
Invoke Bellman-Ford on $(V', E'; C', s)$ to compute δ_s.
For all $u, v \in V$,
let $\widehat{C}(u, v) \leftarrow C(u, v) + \delta(s, u) - \delta(s, v)$
▷ *MAIN LOOP*
For each $v \in V$, invoke Dijkstra's algorithm on $(V, E; \widehat{C}, v)$
to compute δ_v.

The complexity of initialization is $O(mn)$ and each invocation of Dijkstra in the main loop is $O(n \log n + m)$. Hence the overall complexity is $O(n(n \log n + m))$.

§10. All-pairs Minimum Link Paths in Bigraphs

We consider all-pairs min-paths in bigraphs with unit costs. Hence we are interested in minimum length paths. Let G be a bigraph on vertices $[1..n]$ and A be its adjacency matrix. For our purposes, we will assume that the diagonal entries of A are 1. Let d_{ij} denote the minimum length of a path between i and j. Our goal is to compute the matrix $D = [d_{ij}]_{i,j=1}^{n}$. We describe a recent result of Seidel [4] showing how to reduce this to integer matrix multiplication. For simplicity, we may assume that G is a connected graph so $d_{ij} < \infty$.

In order to carry out the reduction, we must first consider the "square of G". This is the graph G' on $[1..n]$ such that (i, j) is an edge of G' iff there is a path of length at most 2 in G between i and j. Let A' be the corresponding adjacency matrix and d'_{ij} denote the minimum length of a path in G' between i and j. Note that $A' = A^2$, where the matrix product is defined over the underlying Boolean semiring.

The following lemma relates d_{ij} and d'_{ij}. But first, note the following simple consequence of the triangular inequality for bigraphs:

$$d_{ik} - d_{jk} \leq d_{ij} \leq d_{ik} + d_{jk}, \qquad \forall i, j, k.$$

Moreover, for all i, j, ℓ, there exists k such that

$$\ell \leq d_{ij} \Longrightarrow \ell = d_{ik} = d_{ij} - d_{jk}. \tag{25}$$

In our proof below, we will choose $\ell = d_{ij} - 1$ and so k is adjacent to j.

Lemma 13.
0) $d'_{ij} = \left\lceil \frac{d_{ij}}{2} \right\rceil$.
1) d_{ij} *=even implies* $d'_{ik} \geq d'_{ij}$ *for all* k *adjacent to* j.
2) d_{ij} *=odd implies* $d'_{ik} \leq d'_{ij}$ *for all* k *adjacent to* j. *Moreover, there is a* k *adjacent to* j *such that* $d'_{ik} < d'_{ij}$.

*Proof.*0) We have $2d'_{ij} \geq d_{ij}$ because given any path in G' of length d'_{ij}, there is one in G between the same end points of length at most $2d'_{ij}$. We have $2d'_{ij} \leq d_{ij} + 1$ because given any path in G of length d_{ij}, there is one in G' of length at most $(d_{ij} + 1)/2$ between the same end points. This shows

$$d_{ij} \leq 2d'_{ij} \leq d_{ij} + 1,$$

from which the desired result follows.
1) If k is adjacent to j then $d_{ik} \geq d_{ij} - d_{jk} = d_{ij} - 1$. Hence

$$d'_{ik} \geq \left\lceil \frac{d_{ij} - 1}{2} \right\rceil = \left\lceil \frac{d_{ij}}{2} \right\rceil = d'_{ij}.$$

2) If k is adjacent to j then $d_{ik} \leq d_{ij} + 1$ and hence

$$d'_{ik} \leq \left\lceil \frac{d_{ij} + 1}{2} \right\rceil = \left\lceil \frac{d_{ij}}{2} \right\rceil = d'_{ij}.$$

Moreover, by equation (25), there is a k adjacent to j such that $d_{ik} = d_{ij} - 1$. Then

$$d'_{ik} = \left\lceil \frac{d_{ij} - 1}{2} \right\rceil = \left\lceil \frac{d_{ij}}{2} \right\rceil - 1 = d'_{ij} - 1.$$

Q.E.D.

As a corollary of 1) and 2) above:

Corollary 14. *For all* i, j, *the inequality*

$$\sum_{k : d_{kj} = 1} d'_{ik} \geq \deg(j) \cdot d'_{ij}$$

holds if and only if d_{ij} *is even.*

Notice that $\sum_{k:d_{kj}=1} d'_{ik}$ is equal to the (i,j)th entry in the matrix $T = D' \cdot A$. So to determine the parity of d_ij we simply compare T_{ij} to $\deg(j) \cdot d'_{ij}$.

We now have a simple algorithm to compute $D = [d_{ij}]$. The **diameter** $diam(G)$ is the maximum value in the matrix D. Let E be the matrix of all 1's. Clearly $diam(G) = 1$ iff $D = E$. Note that the diameter of G' is $\lceil r/2 \rceil$.

Seidel Algorithm
Input: A, the adjacency matrix of G.
Output: The matrix $D = [d_{ij}]$.
1) Compute $A' \leftarrow A^2$, the adjacency matrix of G'.
2) If $A' = E$ then the diameter of G is ≤ 2,
and return $D \leftarrow 2A' - A - I$ where I is the identity matrix.
3) Recursively compute the matrix $D' = [d'_{ij}]$ for A'.
4) Compute the matrix product $[t_{ij}] \leftarrow D' \cdot A$.
5) Return $D = [d_{ij}]$ where

$$d_{ij} \leftarrow \begin{cases} 2d'_{ij} & \text{if } t_{ij} \geq \deg(j) d'_{ij} \\ 2d'_{ij} - 1 & \text{else.} \end{cases}$$

¶31. Correctness. The correctness of the output when A' has diameter 1 is easily verified. The inductive case has already been justified in the preceding development. In particular, step 5 implements the test for the parity of d_{ij} given by corollary 14. Each recursive call reduces the diameter of the graph by a factor of 2 and so the depth of recursion is at most $\lg n$. Since the work done at each level of the recursion is $O(\mathtt{MM}(n))$, we obtain an overall complexity of

$$O(\mathtt{MM}(n) \log n).$$

We remark that, unlike the other minimum cost algorithms, it is no simple matter to modify the above algorithm to obtain the minimum length paths. In fact, it is impossible to output these paths explicitly in subcubic time since this could have $\Omega(n^3)$ output size. But we could encode these paths as a matrix N where $N_{ij} = k$ if some shortest path from i to j begins with the edge (i,k). Seidel gave an $O(\mathtt{MM}(n) \log^2 n)$ expected time algorithm to compute N.

¶32. Final Remarks. The topic of shortest paths is a important topic in many areas, and capable of many generalizations. We have just touched the tip of such problems, focusing on basic combinatorial formulations. We hinted at applications in road networks which brings in other metrics for "shortest". Another direction is to consider **dynamic** shortest path problems, where the underlying graph can change. Yet another direction is to look at geometric shortest path problems, and in robotics motion planning. The algorithmic techniques become quite different when we take into account the geometry. An intermediate step between going from combinatorial graphs to continuous geometry is to focus on embedded graphs of a given genus (e.g., "Multiple-Source Shortest Paths in Embedded Graphs" S. Cabello, E. Wolf Chambers, and J. Erickson (SODA 2007 and SiCOMP 2012).

Exercises

Exercise 10.1: We consider the same problem but for digraphs:
(a) Show that if we have a digraph with unit cost then the following is true for all $i \neq j$: d_{ij} is even if and only if $d'_{ik} \geq d'_{ij}$ holds for all k such that $d_{kj} = 1$.
(b) Use this fact to give an algorithm using $O(\mathtt{MM}(n) \log n)$ arithmetic $(+, -, \times)$ operations on integers. HINT: replace $D' = [d'_{ij}]$ by $E = [e_{ij}]$ where $e_{ij} = n^{n - d'_{ij}}$. ♢

End Exercises

References

[1] A. V. Goldberg and C. Harrelson. Computing the shortest path: A* search meets graph theory. In *Proc. 16th ACM-SIAM Symp. on Discrete Algorithms (SODA '05)*, Jan. 2005.

[2] P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Trans. on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[3] D. Kozen. On Kleene algebras and closed semirings. In *Proc. Math. Foundations of Computer Sci.*, pages 26–47. Springer-Verlag, 1990. Lecture Notes in C.S., No.452.

[4] R. Seidel. On the all-pairs-shortest-path problem. *ACM Symp. Theory of Comput.*, 24:745–749, 1992.