Lecture XIV
MINIMUM COST PATHS

"The shortest path between two truths in the real domain passes through the complex domain."
– Jacques Salomon Hadamard (1865–1963)

Several problems which we studied under "pure graph problems" in Chapter (???) will now be generalized. Connectivity becomes considerably more interesting when we introduce cost functions. Connectivity has to do with paths. In the presence of cost functions, these paths has associated costs. We now study various problems related to paths of minimum cost. It is interesting to note that shortest algorithms can take advantage of the special nature of the input cost function in the following cases:

- When all the edges have unit cost.
- When the costs are symmetric (i.e., we are dealing with bigraphs).
- When the costs are positive.
- When the graph is sparse (i.e., $C(e) = \infty$ for most edges e).

Each of these cases will be illustrated below. Another direction is to generalize shortest path problems to computations over semirings. The transitive closure problem can then be viewed as such a generalization.

§1. Minimum Path Problems

Let $G = (V, E; C)$ be a simple 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'. Two special cost functions are worth noting: we say C is a **positive cost** if $C > 0$. The simplest positive cost is the **unit cost** where $C = 1$. In constrast to positive costs, we may speak of a "general" cost function to emphasize the possibility of negative costs.

Convention. As usual, let $n = |V| \geq 1$ and $m = |E|$. These are the size parameters for complexity considerations. We usually let $V = [1..n]$.

Minimum cost paths. Let $p = (v_0, \ldots, v_k)$ be a path of G, so $(v_{i-1}, v_i) \in E$ 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 $C(p) = 0$. We 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, denoted

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

For short, we say "minimum path" path instead of "minimum cost path". We also speak of **min-paths** or **min-costs**.

If there is no path from i to j, let $\delta(i, j) := \infty$. If there is a path from i to j but there does not exist a minimum path from i to j, then $\delta(i, j) := -\infty$. This situation obtains if for every negative number r, there is a path with cost less than r. Thus we can view δ as the **C-minimum cost matrix**

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

The reference to C may be omitted when understood or irrelevant.

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 the minimum path from s to t.
- **Single-source minimum paths** Given an edge-costed digraph $G = (V, E; C, s)$ with source $s \in V$, find the minimum paths from s to each $t \in V$.
- **All-pairs minimum paths** Given an edge-costed digraph $G = (V, E; C)$, find the minimum paths between from s to t for all $s, t \in V$.

When there does not exist a minimum path from i to j for one of the pairs (i, j) that is asked for, the problem requires that we detect this and distinguish between $\delta(i, j) = \infty$ or $\delta(i, j) = -\infty$; in the latter case, we want the algorithm to 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 presense of negative edges. Otherwise, any negative bi-directional edge immediately give us a negative cycle. Some special techniques can be brought into play for bigraphs, and these will be considered in §8 and §9.

Another remark is in order. Clearly the three problems are in order of increasing difficulty. But you will not encounter any algorithm that is expressedly designed for the first problem (single-pair case). This is because every *known* algorithm for the single-pair problem is essentially also a solution to the single-source problem! It would be nice to prove that this is necessarily so.

Minimum cost versions. There is a simpler version of each of the above problems, *viz.*, where we ask for the minimum cost $\delta(i, j)$ instead of the minimum path from i to j (for various i, j depending on the problem). We call this the **minimum cost version** of the corresponding shortest path problem. For instance, the "single-source minimum cost problem". It is easy to compute the minimum cost from a minimum path, so the minimum cost problems are reducible to the minimum path problems. It turns out that all algorithms for minimum path problems also compute the minimum cost as a by-product. Intuitively, this is necessarily so because the minimum costs constitute the critical information that drives these algorithms. It is pedagogically advantageous to present only the minimum cost version of these algorithms. *We adopt this strategy throughout.* It is usually a simple matter to insert some additional lines of code into the minimum cost algorithms to derive a minimum path algorithm, without changing the asymptotic complexity. Still, we will see exceptions to this remark (Exercise).

Dynamic programming principle. The dynamic programming principle applies to minimum paths: subpaths of minimum paths are minimum paths. In fact, the simplification from minimum paths to minimum costs is a general feature of dynamic programming solutions (see Lecture III).

Unit costs and shortest paths. If C is the unit cost then $C(p) = k$ is just the **length** of the path $p = (v_0, \ldots, v_k)$. Consistent with this "length" terminology, we could call any path of minimum length between its endpoints a **shortest path**. But this terminology can be quite confusing since the literature usually say "shortest path" instead of our "minimum path". To avoid any ambiguity, we could alternatively call $\delta(i,j)$ under the unit cost function the **link distance** between i and j. We say j is **reachable** from i if the link distance from i to j is finite. The single-source problem in this case can be solved by the Breadth First Search graph algorithm.

Truncated minimum paths. Let k be a non-negative integer. We define a path to be a k-**link minimum path** if it has minimum cost among all k-link paths from its source to its terminus. Let $\delta^{(k)}(i,j)$ denote the cost of a k-link minimum path from i to j and we again have the k-**link minimum cost matrix** $\delta^{(k)}$. We can also minimum cost among all paths of at most k links. The corresponding matrix is given by

$$\delta^{(\le k)}(i,j) = \min_{\ell=0}^{k} \delta^{(\ell)}(i,j).$$

Call $\delta^{(\le k)}$ the k-**truncated minimum cost matrix**. But unlike the δ matrix, $\delta^{(k)}$ never attain $-\infty$. If C is positive, it is easy to see that

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

Minimum path tree. Our single-source path algorithms construct a set of minimum paths that comes from a single tree rooted at the source. By a **minimum path tree** of $G = (V, E; C)$ we mean a finite rooted tree T such that the paths from the root to every vertex in the tree is a minimum path; moreover, every node reachable from the root appears in T. Note that if there is a path from s that contains a negative cycle, then there does not exist a minimum path tree rooted at s. The minimum path tree under the unit cost $C = 1$ is just the breadth first search (BFS) tree. The following is a characterization of minimum path trees.

Lemma 1 (minimum path tree). *Suppose that $T \subseteq E$ is a tree rooted at $s \in V$ and T spans the set of nodes reachable from s. For any node i in the tree, let $d(i)$ denote the cost from s to i along a path of T. Then T is a minimum path tree iff for all $(i,j) \in E$, $d(j) \le d(i) + C(i,j)$.*

Exercises

Exercise 1.1: Considers the following shortest path problem: each node u has a weight $W(u)$ and the cost of edge (u,v) is $W(v) - W(u)$. Give an $O(m)$ algorithm to solve the *minimum cost version* of the single source minimum path problem. Can you convert this algorithm into one that actually produce the minimum paths? ♢

Exercise 1.2: Another variation of shortest paths is to assign costs to the vertices. The cost of a path is the sum of the costs of the vertices along the path. Reduce this **vertex-costed** version of minimum paths to the original **edge-costed** version. ♢

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: Prove the minimum path tree lemma. ♢

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.

The Bellman-Ford algorithm is extremely simple, using only a single array $c[1..n]$ as datastructure. Assuming the source is vertex 1, we desire the algorithm to compute δ_1 which will be represented by $c[1..n]$. To bring out the main ideas, we first give a simplified version that is correct *provided no negative cycle is reachable from vertex* 1. In fact, we will say somewhat more about the output of the simplified algorithm in general (negative cycle or no):

Correctness Criteria: The array c at the end of the algorithm is a realizable lower bound on $\delta_1^{(n-1)}$.

In general, for any $k \geq 0$, we call $c[1..n]$ a **realizable lower bound on** $\delta^{(k)}$ if:
(a) (Lower bound) $c[i] \leq \delta_1^{(k)}(i)$, for all $i \in [1..n]$.
(b) (Realizability) There is a path from 1 to i with cost $c[i]$, for all $i \in [1..n]$.
Clearly we have

$$\delta_1^{(k)}(i) \geq c[i] \geq \delta_1(i), \qquad i \in [1..n].$$

From (a) and (b), we conclude that $c[i] = \infty$ means there is no path from 1 to i.

```
Simple Bellman-Ford Algorithm:
    Input:    $(V, E; C, s)$ where $V = [1..n]$ and $s = 1$.
    Output:   Array $c[1..n]$ as described above.
    INITIALIZATION:
        $c[1] \leftarrow 0$
        for all $i = 2$ to $n$, $c[i] \leftarrow \infty$
    MAIN LOOP:
        for $k = 1$ to $n - 1$ do
            Phase (* see below*)
```

The main loop consists of $n-1$ identical **phases** described as follows:

```
Phase:
    for all $(u,v) \in E$ do
        $c[v] \leftarrow \min\{c[v], c[u] + C(u,v)\}$
```

The initialization is regarded as the zeroth phase. It is clear that each phase takes $O(m)$ time for an overall complexity of $O(mn)$.

Lemma 2 (Invariance). *At the end of the kth phase ($k \geq 0$), the array $c[1..n]$ is a realizable lower bound on $\delta_1^{(k)}$.*

Proof. This is immediate for $k=0$ so assume $k \geq 1$. Let $v \in [1..n]$ and $c[v] < \infty$. First we show that $c[v]$ is realizable, *i.e.*, there is a path from 1 to v with cost $c[v]$. If $c[v]$ is unchanged in the kth phase, then this follows by induction. Otherwise it is updated as $c[u] + C(u,v)$ for some u. Clearly $c[u] < \infty$ and so it represents the cost of some path p from 1 to u. Thus $c[v]$ is now the cost of $p;(u,v)$. This proves realizability of c. Next we must show that $c[v] \leq \delta^{(k)}(v)$. If $\delta^{(k)}(v)$ represents the cost of a path from 1 to v of length less than k, then the desired inequality follows by induction: $c[v] \leq \delta^{(k-1)}(v) = \delta^{(k)}(v)$. Otherwise, $\delta^{(k)}(v)$ is the cost of a path of length k. Let this path be $p;(u,v)$ for some u. By induction, the previous value of $c[u]$ is $\leq C(p)$. Because of our update method, $c[v] \leq c[u] + C(u,v)$. Hence $c[v] \leq C(p) + C(u,v) = \delta^{(k)}(v)$. **Q.E.D.**

In the absence of negative cycles, $\delta = \delta^{(n-1)}$. Then the output array c represents δ_1, as desired.

Bellman-Ford in the presence of negative cycles. We now remove our assumption that there are no negative cycles.

Lemma 3 (Negative Cycle Test). *Let $c[1..n]$ be a realizable lower bound on $\delta_1^{(n-1)}$.*
(a) If there are no negative cycles reachable from 1 then for all $i,j \in [1..n]$, $c[j] \leq c[i] + C(i,j)$.
(b) If Z is a negative cycle reachable from 1 then $c[j] > c[i] + C(i,j)$ holds for some edge (i,j) in Z.

Proof. (a) If no negative cycle is reachable, then no optimum path from 1 has length more than $n-1$. Hence $c[i] \leq \delta_1^{(n-1)}(i)$ implies $c[i] = \delta_1^{(n-1)}(i) = \delta_1(i)$. The desired inequality follows from $\delta(j) \leq \delta(i) + C(i,j)$. (b) By way of contradiction, suppose $c[j] \leq c[i] + C(i,j)$ for all edges (i,j) in a reachable negative cycle Z. Summing over all edges in Z,

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

Cancelling the summation on each side, we see that $0 \leq C(Z)$, a contradiction. **Q.E.D.**

It is easy to use this lemma to detect if there are any negative cycles reachable from 1 in the simple Bellman-Ford algorithm. But we can also use it to justify a **general Bellman-Ford algorithm** which compute δ_1 for an arbitrary input graph.

```
General Bellman-Ford Algorithm:
    Input:   $(V, E; C, s)$ with $V = [1..n]$ and $s = 1$.
    Output:  Array $c[1..n]$ representing $\delta_1$.
    INITIALIZATION: (as in Simple Bellman-Ford Algorithm)
    MAIN LOOP: (as in Simple Bellman-Ford Algorithm)
    END LOOP:
        for $k = 1$ to $n$ do
            End Phase (* see below*)
```

The End-Phase is just a simple modification of the Phase computation:

```
End Phase:
    for all $(u, v) \in E$ do
        if $c[v] > c[u] + C(u, v)\}$ then $c[v] \leftarrow -\infty$.
```

After n iterations of this, it is easy to see that $c[1..n]$ represents δ_1. Moreover, the asymptotic complexity of the original algorithm is preserved.

Minimum paths. We indicate how the minimum paths can be computed by a simple modification to the above 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 $p[v] \leftarrow u$. It is easy to see that the set of edges $\{(v, p[v]) : v \in V, p[v] \neq \text{nil}\}$ forms a minimum path tree.

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: 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 of 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.

$\diamondsuit$

Exercise 2.4: 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. $\diamondsuit$

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.

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".

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 $d[i]$ is the current approximation to $\delta_1(u)$. Inductively, the array $d[1..n]$ satisfies the following invariant:

(A) $d[u] = \min_{v \in S}\{d[v] + C(v, u)\}$ for all $u \in V$.

(B) If $u \in S$ then $d[u] = \delta_1(u)$, the mininum cost from vertex 1 to u.

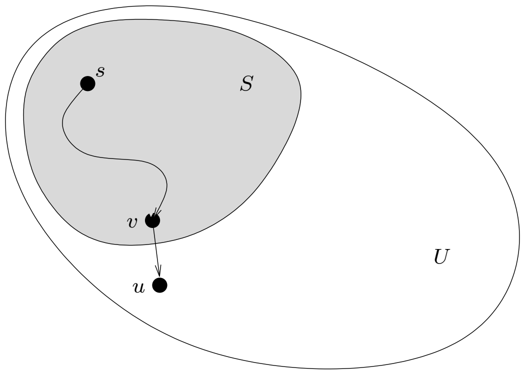

Figure 1: Illustrating Dijkstra's Invariant

We may interpret $d[u]$ as the minimum cost of all paths from 1 to u whose intermediate vertices are restricted to S. It follows from invariants (A) and (B) that

$$d[u] \geq \delta_1(u), \qquad (u \in V). \tag{1}$$

To see this, note that invariant (B) implies that for each $u \in S$, there is a path from 1 to u with cost $d[u]$. For all $v \in V$, invariant (A) says that $d[v]$ corresponds to the cost of an actual path from 1 to v. This cost cannot be less than $\delta_1(v)$, as claimed.

Lemma 4. *Assume invariants (A) and (B). Let $u_0 \in V \setminus S$ such that*

$$d[u_0] = \min\{d[i] : i \in V \setminus S\}.$$

Then $d[u_0] = \delta_1(u_0)$.

Proof. Suppose p is a minimum path from 1 to u_0. Then we can decompose p into the form

$$p = p'; (v,u); p''$$

where $v \in S$ and $u \in V \setminus S$. See figure 1. Note that $C(p') = \delta_1(v)$. Then

$$\begin{array}{rcll} d[u_0] & \le & d[u] & \text{(choice of } u_0\text{)} \\ & \le & d[v] + C(v,u) & \text{(invariant (B))} \\ & = & \delta_1(v) + C(v,u) & \text{(invariant (A))} \\ & = & C(p') + C(v,u) & \text{(dynamic programming principle)} \\ & \le & C(p) & \text{(since costs are positive)} \\ & = & \delta_1(u_0). & \text{(choice of } p\text{)} \end{array}$$

Combined with equation (1), we conclude that $d[u_0] = \delta_1(u_0)$. **Q.E.D.**

This lemma shows that if we extend S to $S' := S \cup \{u_0\}$, invariant (A) is preserved. It is easy to see invariant (B) can also be preserved by updating the value of $d[i]$ for each $i \in V \setminus S'$ using the following equation:

$$d[i] \leftarrow \min\{d[i], d[u_0] + C(u_0, i)\}. \tag{2}$$

Moreover, we only need update those i that are adjacent to u_0. The repeated extension of the set S while preserving invariants (A) and (B) constitutes Dijkstra's algorithm.

Let us now summarize the algorithm. First, let the dynamic set $U = V \setminus S$ be stored in a min-priority queue Q, using $d[i]$ as the priority of vertex $i \in U$. The queue is assumed[1] to support the DecreaseKey operation, which is needed in updating $d[i]$ *á la* equation (2).

[1]This assumption is equivalent to the ability to delete an arbitrary element from the queue. For, DecreaseKey of x can be viewed as a deletion of x followed by an re-insertion of x with the new priority. Conversely, if we have DecreaseKey, then we can delete an arbitrary element by decreasing its priority to $-\infty$ followed by a removeMin.

```
Dijkstra's Algorithm:
      Input:    $(V, E; C, s)$ where $V = [1..n]$ and $s = 1$.
      Output:   Array $d[1..n]$ with $d[i] = \delta_1(i)$.
      ▷ INITIALIZATION
1.          $d[1] = 0$; Initialize an empty queue $Q$.
2.          for $i = 2$ to $n$
3.                $d[i] \leftarrow \infty$,
4.                $Q$.Insert$(i, d[i])$.
      ▷ MAIN LOOP
5.          while $Q \neq \emptyset$ do
6.                $u_0 \leftarrow Q$.DeleteMin()
7.                for all $i$ adjacent to $u_0$ do
8.                      if $d[i] > d[u_0] + C(u_0, i)$ then
9.                            $d[i] \leftarrow d[u_0] + C(u_0, i)$
10.                           $Q$.DecreaseKey$(i, d[i])$
            end{while}
```

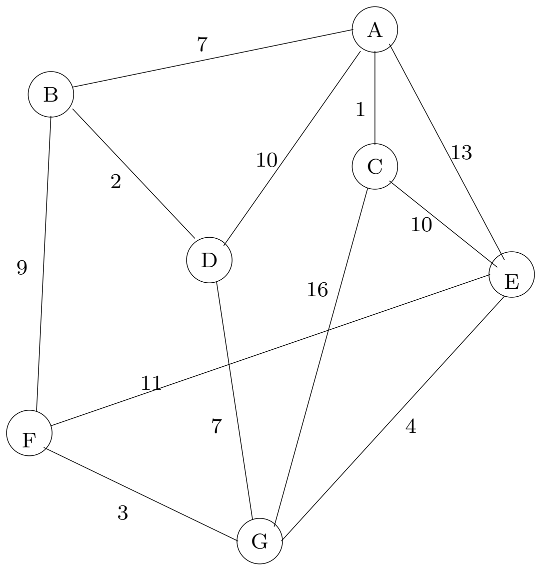

Figure 2: Illustrating Dijkstra's Algorithm

Hand Simulation. Let us perform a hand-simulation of this algorithm using the graph in figure 2. Let the source node be A. The array $d[i]$ is initialized to ∞ with $d[A] = 0$. It is updated at each stage: we have underlined the entry that is the minimum extracted for that stage, and only updated entries of that stage are explicitly indicated:

VERTICES	A	B	C	D	E	F	G
STAGE 0	0	∞	∞	∞	∞	∞	∞
STAGE 1	$\underline{0}$	7	1	10	11		
STAGE 2			$\underline{1}$				17
STAGE 3		$\underline{7}$		9		16	
STAGE 4				$\underline{9}$			16
STAGE 5					$\underline{11}$		15
STAGE 6						18	$\underline{15}$
STAGE 7						$\underline{18}$	

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.

We ought to note that if the graph is sparse (say, 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 $O(n^2)$. The details are left as an exercise.

Exercises

Exercise 3.1: Show that $c[v]$ is the minimum cost of paths from 1 to v whose intermediate vertices are restricted to S. ♢

Exercise 3.2: Show that Dijstra's algorithm may fail if G has negative edge weights (even without negative cycles). ♢

Exercise 3.3: 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 satisfy properties (A) and (B), so that the basic algorithm works). ♢

Exercise 3.4: Give the programming details for the "simple" $O(n^2)$ implementation of Dijstra's algorithm. ♢

Exercise 3.5: Convert Dijkstra's algorithm above into a minimum path algorithm. ♢

Exercise 3.6: Justify this remark: if every edge in the graph has weight 1, then the BFS algorithm is basically like Dijkstra's algorithm. ♢

Exercise 3.7: (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 9 in Dijkstra's algorithm). If $c[i]$ is unchanged, we do nothing, so suppose $c[i]$ is decreased. If i is in the queue, we do DecreaseKey on i as before; otherwise we reinsert i into Q. Prove that this modification terminates with the correct answer.
(iii) Choose the vertex u_0 carefully so that the algorithm in (ii) is $O(n^3)$. $\diamondsuit$

Exercise 3.8: 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. $\diamondsuit$

End Exercises

§4. Semirings

Before considering the all pairs minimum cost problems, let us recall some facts about matrix rings. All our matrices are square (n by n, for some $n \geq 1$). A 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 $A = [A_{ij}]_{i,j}$. This should not be confused with the notation $(A)_{ij}$ denoting the (i,j)-th entry of matrix A. Recall the usual multiplication of numerical matrices: if $A = [A_{ij}], B = [B_{ij}]$ then their product AB is $C = [C_{ij}]$ where

$$C_{ij} = \sum_{i=1}^{k} A_{ik} B_{kj}. \tag{3}$$

To generalize such matrices, consider a **ring with unity**

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

By definition[2] this means the set R satisfies the following axioms.
(i) $(R, +, 0)$ is an Abelian group,
(ii) $(R, \times, 1)$ is a monoid,
(iii) $\times$ distributes over $+$.

As usual, we simply refer to the set R as the ring if the other data $(+, \times, 0, 1)$ are understood and the product $a \times b$ (for $a, b \in R$) is written as ab or $a \cdot b$. For $n \geq 1$, we have another ring with unit,

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

where $M_n(R)$ is the set of n-square matrices with entries in R. We call $M_n(R)$ a **matrix ring** over R. Addition of matrices, $A +_n B$, is defined componentwise. The product $A \times_n B$ of matrices is defined as in equation (3). 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.

[2]Most of our rings have a multiplicative identity usually denoted 1: $x \cdot 1 = 1 \cdot x = x$. We usually call 1 the **unity** element. An algebraic structure $(M, +, 0)$ is a monoid if $+$ is an associative binary operation on M with 0 as an identity. A standard example of a monoid is the set of strings over an alphabet under the concatenation operation, with the empty string as identity. [Incidentally, dropping the identity of a monoid gives us a **semigroup**.] A group $(G, +, 0)$ is a monoid where $+$ has an inverse relative to 0, *i.e.*, for all x there is a 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 using '+' for the group operation, we denote the inverse of an element x by $-x$.

Let $\mathtt{MM}(n)$ denote the number of ring operations in R necessary to compute the product of two matrices in $M_n(R)$. The problem of determining $\mathtt{MM}(n)$ has been extensively studied ever since Strassen demonstrated that the obvious $\mathtt{MM}(n) = O(n^3)$ bound is suboptimal. The current record is from Coppersmith and Winograd:

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

Connection to shortest paths. Problems on minimum paths has an underlying algebraic structure that is similar to matrix multiplication. To see this connection, note that the cost of a 2-truncated minimum path path from vertex i to j is given by

$$\delta^{(2)}(i,j) = \min_{k=1}^{n} C(i,k) + C(k,j).$$

This expression is analogous to equation (3), except that we have replaced summation by minimization, and product by summation. Hence *computing the* 2*-truncated minimum costs between all pairs of vertices is equivalent to the problem of matrix multiplication* where the matrices have elements from a certain ring-like structure:

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

where ∞ and 0 are the respective identities for the minimization and addition operation. In fact, the only thing this structure lacks to make it a ring is *an inverse for minimization.* Such structures are pervasive enough to be studied abstractly:

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*.*
1) [Additive monoid] $(R, \oplus, 0)$ *is an Abelian monoid.*
2) [Multiplicative monoid] $(R, \otimes, 1)$ *is a monoid.*
3) [Annihilator] 0 *is the annihilator under multiplication:* $x \otimes 0 = 0 \otimes x = 0$.
4) [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.

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. Moreover, the axiom that 0 is a multiplicative annihilator must be explicitly stated, whereas it was previously implied by the ring axioms (exercise above). The following are examples of semirings that are not rings.

1. The "canonical example" of a semiring is the natural numbers $\mathbb{N}$ augmented with ∞. It is useful to test all concepts about semirings against this one.

2. The structure
$$(\mathbb{R} \cup \{\pm\infty\}, \min, +, \infty, 0) \tag{4}$$
noted above is a semiring. For reference, call this the **minimization semiring**. Note that the annihilator axiom implies $\infty + (-\infty) = \infty$. Any subring $S \subseteq \mathbb{R}$ induces a sub-semiring $S \cup \{\pm\infty\}$ of this real minimization semiring. We warn that the "multiplication" in the minimization semiring is ordinary addition! To avoid confusion, we will say "semiring multiplication" to refer to +, or "semiring addition" to refer to min, when viewing $\mathbb{R} \cup \{\pm\infty\}$ as a semiring.

3. Naturally, there is an analogous **(real) maximization semiring**,
$$(\mathbb{R} \cup \{\pm\infty\}, \max, +, -\infty, 0). \tag{5}$$
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**,
$$(\mathbb{R}_{\geq 0} \cup \{\infty\}, \min, +, \infty, 0). \tag{6}$$
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 $B_2 := \{0,1\}$.
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 $A, B \subseteq \Sigma^*$, we define their 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 5. *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 (3).*

The verification of this lemma is left to the reader. 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.

Complexity of multiplying Boolean matrices. For Boolean semiring matrices, we can obtain a sub-cubic 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
$$\mathtt{MM}_2(n) = O(\mathtt{MM}(n) \lg n). \tag{7}$$

§5. 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 $\bigoplus_{i\ge 1} a_i = 0$.
2) [Countable Associativity]

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

3) [Countable Commutativity]

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

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

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

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

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

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 $i \ge 2$.

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. But we continue to use "min", by a slight abuse of notation. 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 minimum 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.

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

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

is a closed semiring if we interpret $+, \times$ in the ordinary way. This 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

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

To check that this is a partial order: Clearly $x \leq x$. If $x \leq y$ and $y \leq x$ then $x = y$. 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, and $x \leq y, x' \leq y'$ implies $x \oplus y \leq x' \oplus y'$. But be warned that in the minimization semiring $\mathbb{R} \cup \{\pm\infty\}$, this definition "$\leq$" is the inverse of the usual ordering on reals! 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 [1]). This algebra can be defined independently from semirings.

Exercises

Exercise 5.1: Show that in a ring R: $-x = (-1) \cdot x$, and $x \cdot 0 = 0 \cdot x = 0$ for all $x \in R$. ♢

Exercise 5.2: 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 5.3: Under what conditions does the canonical construction of $\mathbb{Z}$ from $\mathbb{N}$ extend to give a ring from a semiring? ♢

Exercise 5.4: 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 5.5: Generalize the problem of optimal triangulation (lecture 3) 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

§6. 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.

For this problem, we shall represent the costed graph by its cost matrix $C = [C_{i,j}]_{i,j=1}^{n}$. The underlying semiring is assumed to the minimization semiring (see (4)). An easy generalization of an earlier observation (for the case $k = 2$) gives:

Lemma 6. *Let C be a cost matrix regarded as a matrix over the minimization semiring. If $C^k = [C_{ij}^{(k)}]$ is the the kth power of C then C^k is the matrix of the k-truncated minimum cost function $\delta^{(k)}$: for all i, j,*

$$\delta^{(k)}(i,j) = C_{ij}^{(k)}$$

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

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

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.

We next improve this bound using the **Floyd-Warshall algorithm**[3]. Another advantange to the Floyd-Warshall algorithm is that we do not need to assume the absense of negative cycles. To explain this algorithm, we need to define a ***k*-path** ($k \in [1..n]$) of a digraph: a path

$$p = (v_0, v_1, \ldots, v_\ell)$$

is called a k-path if the vertices in p, with the exception of v_0, v_ℓ, belong to the set $[1..k]$. Unlike the truncated cost function $\delta^{(k)}$, we impose no bound on the length ℓ of the path p. By extension, we may say that a 0-path is one 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 $k \ge 1$:

$$\delta^{[k]}(i,j) = \min\{\delta^{[k-1]}(i,j), \delta^{[k-1]}(i,k) + \delta^{[k-1]}(k,k)^* + \delta^{[k-1]}(k,j)\} \tag{8}$$

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

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

[3]The method is similar to the standard proof of Kleene’s characterization of regular languages.

Notice that $\delta^{[n]}(i,j)$ is precisely equal to $\delta(i,j)$. The Floyd-Warshall algorithm simply uses equation (8) to compute $\delta^{[k]}$ for $k = 1, \ldots, n$:

```
Floyd-Warshall Algorithm:
    Input:   Cost matrix $C$ which is $n$ by $n$.
    Output:  Matrix $c[1..n, 1..n]$ representing $\delta$.
    INITIALIZATION
        for all $i, j = 1$ to $n$ do
            $c[i,j] \leftarrow C_{ij}$
    MAIN LOOP
        for $k = 1$ to $n$ do
            for all $i, j = 1$ to $n$ do
(A)                 $c[i,j] \leftarrow \min\{c[i,j], c[i,k] + c[k,k]^* + c[k,j]\}$
```

This algorithm clearly takes $O(n^3)$ time. The correctness can be proved by induction. Note that line (A) in the algorithm is not an exact transcription of equation (8) because the matrix $c[1..n, 1..n]$ is used to store the values of $\delta^{[k]}$ as well as $\delta^{[k-1]}$. Nevertheless (as in the Bellman-Ford algorithm), we have the invariant that in the kth iteration,

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

Exercises

Exercise 6.1: 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 6.2: 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 6.3: 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 6.4: 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 6.5: Modify the Floyd-Warshall Algorithm so that it computes the lengths of the first and also the second shortest path. The second shortest path must be distinct from the shortest path. In particular, if the shortest path does not exist, or is unique, then the second shortest path does not exist. In this case, the length is ∞. ♢

End Exercises

§7. 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 $w = (i_0, \ldots, i_m) \in [1..n]^*$, define

$$C(w) := \bigotimes_{j=1}^{m} C(i_{j-1}, i_j), \quad m \geq 2.$$

If $m = 0$ or 1, $C(w) := 1$ (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 7.
(i) $C^{[0]} = C$ and for $k = 1, \ldots, n$,

$$C_{ij}^{[k]} = C_{ij}^{[k-1]} \oplus \left(C_{ik}^{[k-1]} \otimes (C_{kk}^{[k-1]})^* \otimes C_{kj}^{[k-1]} \right) \tag{9}$$

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

Proof. We only verify equation (9), 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 (9). 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

$$C_{ij} = C_{ij} \oplus (C_{ik} \otimes (C_{kk})^* \otimes C_{kj}) . \qquad (10)$$

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 (9) in line (A) of the Floyd-Warshall algorithm. The algorithm uses $O(n^3)$ operations of the underlying closed semiring operations.

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 (7)). *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 8. $\mathtt{TC}_2(n) = \Theta(\mathtt{MM}_2(n))$.

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

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}^* = \begin{pmatrix} E^* & E^*BD^* \\ D^*CE^* & D^* + D^*CE^*BD^* \end{pmatrix}, \tag{11}$$

where

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

This formidable-looking expression (11) 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 (11). 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 (11). 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 7.1: Rewrite update rule (9) that corresponds to the improved rule (10). In other words, show when the update of $C_{ij}^{[k]}$ is sometimes using an "advance value" on the right-hand side. ♢

Exercise 7.2: Give similar interpretations for the other three entries of the RHS of equation (11). ♢

Exercise 7.3: Express the RHS of equation (11) 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

§8. 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

$$\widehat{C}(i,j) = C(i,j) + \phi(i) - \phi(j). \tag{12}$$

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 minimum 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 9. *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 10. *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 $C'(i,j) = C(i,j)$, $C'(s,j) = 0$ and $C'(i,s) = \infty$. 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 minimum path in G iff it is a minimum 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 $\delta_s$.
        Abort if negative cycle discovered; else, 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 $\delta_v$.
```

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 $O(n^2 \log n + mn)$.

§9. All-pairs Minimum Link Paths in Bigraphs

We consider all-pairs minimum 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 [2] 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:

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

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

$$\ell \le d_{ij} \Longrightarrow \ell = d_{ik} = d_{ij} - d_{jk}. \tag{13}$$

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

Lemma 11.
0) $d'_{ij} = \left\lceil \frac{d_{ij}}{2} \right\rceil$.
1) d_{ij} *=even implies* $d'_{ik} \ge d'_{ij}$ *for all* k *adjacent to* j.
2) d_{ij} *=odd implies* $d'_{ik} \le d'_{ij}$ *for all* k *adjacent to* j*. Moreover, there is a* k *adjacent to* j *such that* $d'_{ik} < d'_{ij}$.

Proof. 0) We have $2d'_{ij} \ge d_{ij}$ 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 $2d'_{ij} \le d_{ij} + 1$ 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

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

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

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

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

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

Moreover, by equation (13), 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 12. *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 $\leq 2$,
              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
        $d_{ij} \leftarrow \begin{cases} 2d'_{ij} & \text{if } t_{ij} \geq \deg(j)d'_{ij} \\ 2d'_{ij} - 1 & \text{else.} \end{cases}$
```

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 12. 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.

Exercises

Exercise 9.1: 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] 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.

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