"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. Several problems studied under "pure" graphs in Chapter 4 is thereby generalized. Connectivity becomes considerably more interesting in the presence of cost functions. Connectivity has to do with paths. Suppose $\Pi(u, v)$ denote the set of all paths from vertex u to vertex v. The basic problem here is to find a path $\pi \in \Pi(u, v)$ whose cost is minimum. The dynamic programming principle is at work in such problems. 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)

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. We will also see how to take advantage of sparse graphs as well as bigraphs.

We can generalize shortest path problems to computations over semirings. The important problem of transitive closure problem arises through this generalization.

§1. Minimum Path Problems

¶1. Costed Graphs. 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 $V = \{1, \ldots, n\}$.

¶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". Although it is very common to say "shortest path" for min-paths, but we prefer to restrict this usage only to the case of unit cost. If there is no path from i to j, let $\delta(i, j) := \infty$. A cycle $[v_0 -, \ldots, -v_k]$ is called a **negative cycle** if $\sum_{i=0}^{k} C(v_i, v_{i+1}) < 0$ (here, $v_{k+1} = v_0$). In case there exist paths from i to j with arbitrarily negative costs, we define $\delta(i, j) := -\infty$. This situation obtains if there is a path from i to j that contains a negative cycle. Thus we can view δ as a matrix, the C**-minimum cost matrix**

shortest path = min cost + unit cost

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

¶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 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 is no minimum path from i to j for one of the pairs (i, j) that is asked for, we are expected to detect this and output $\delta(i, j) = \infty$ or $\delta(i, j) = -\infty$; in the latter case, we further 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. 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.

¶5. 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 **min-cost version** of the corresponding shortest path problem. Usually[1] the min-cost algorithms can easily be modified to also compute the min-path as a by-product, without affecting the asymptotic complexity. Intuitively, this is because the minimum 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 generally adopt this strategy.*

¶6. Dynamic programming principle. The dynamic programming principle (Chapter 7) applies to minimum paths: subpaths of minimum paths are minimum paths. Indeed, the simplification from minimum solution instances to minimum costs is also a feature of dynamic programming.

¶7. Path Length and Link Distance. If C is the unit cost then $C(p) = k$ is just the **length** of the path $p = (v_0, \ldots, v_k)$. Consistent with this "length" terminology, we might call paths of minimum length a "shortest path". Unfortunately, the literature also use "shortest path" for the general min-path. To avoid ambiguity, we adopt another terminology found in the literature: the minimum length of a path from i to j may be called the **link distance** from i to j. Say j is **reachable** from i if the link distance from i to j is finite.

¶8. Link-bounded minimum paths. Let k be a non-negative integer. We define a path to be the **exact** 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 an exact k-link minimum path from i to j and we again have the **exact** k**-link minimum cost matrix** $\delta^{(=k)}$. We can also consider *at most* k links: the corresponding matrix is given by

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

Call $\delta^{(k)}$ the 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

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

¶9. 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. Under unit cost, this tree is just the a breadth first search (BFS) tree. If s can reach a negative cycle, then the minimum path tree rooted at s is not defined. 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 $(i, j) \in E$, $d(j) \leq d(i) + C(i, j)$.*

EXERCISES

Exercise 1.1: Considers the following minimum 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? $\diamondsuit$

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

Exercise 1.2: Another variation of minimum 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. 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 is simple, and uses only 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 simplified version that is correct *provided no negative cycle is reachable from vertex* 1. In fact, we can 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 $(n-1)$-bound.

For any $k \geq 0$, we call $c[1..n]$ a **realizable k-bound** if for each $i \in [1..n]$,
(a) (Lower bound) $c[i] \leq \delta_1^{(k)}(i)$.
(b) (Upper bound/Realizability) There is a path from 1 to i with cost $c[i]$.
Thus a realizable k-bound is both a lower bound and an upper bound, given by

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

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 $i \leftarrow 2$ to n, $c[i] \leftarrow \infty$
▷ *MAIN LOOP*
for $k \leftarrow 1$ to $n-1$
PHASE() ◁ *see below*

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

```
PHASE()
    for all (u, v) ∈ E
        c[v] ← 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 ($k \ge 0$), the array $c[1..n]$ is a realizable k-bound.*

Proof. This is immediate for $k=0$ so assume $k \ge 1$. 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 $c[v] \le \delta^{(k)}(v)$. 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: $c[v] \le \delta^{(k-1)}(v) = \delta^{(k)}(v)$. 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 $\le C(p)$. Because of our update method, $c[v] \le c[u] + C(u,v)$. Hence $c[v] \le C(p) + C(u,v) = \delta^{(k)}(v)$. **Q.E.D.**

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

¶11. Bellman-Ford with negative cycles. We now remove our assumption about no negative cycles.

LEMMA 3 (Negative Cycle Test). *Let $c[1..n]$ be a realizable $(n-1)$-bound.*
(a) If there are no negative cycles reachable from 1 then for all $i,j \in [1..n]$, $c[j] \le c[i] + C(i,j)$.
(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 $c[i] \le \delta_1^{(n-1)}(i)$ implies $c[i] = \delta_1^{(n-1)}(i) = \delta_1(i)$. The desired inequality follows from $\delta(j) \le \delta(i) + C(i,j)$. (b) By way of contradiction, suppose $c[j] \le c[i] + C(i,j)$ for all edges (i,j) in a reachable negative cycle Z. Summing over all edges in Z,

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

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

We can use this lemma to detect if there are any negative cycles reachable from 1 in the simple Bellman-Ford algorithm. 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 δ_1.
    ▷ INITIALIZATION
        (as in Simple Bellman-Ford Algorithm)
    ▷ MAIN LOOP
        (as in Simple Bellman-Ford Algorithm)
    ▷ END LOOP:
        for k ← 1 to n
            PHASE()      ◁ as before
        END PHASE()      ◁ see below
```

The END PHASE is a simple modification of the PHASE computation:

```
END PHASE()
    for all (u, v) ∈ E
        If (c[v] > c[u] + C(u, v)) then c[v] ← −∞.
```

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.

¶12. 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 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.

$\diamondsuit$

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.

¶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 $(1-j) \notin E$. Inductively, the array $d[1..n]$ satisfies the following invariants:

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

$$d[u] = \min_{v \in S}\{d[v] + C(v, u)\}. \tag{1}$$

Note that this invariant holds initially since $S = \{1\}$ and $d[1] = 0$.

Invariant (B) If $u \in S$ then $d[u]$ is equal to $\delta_1(u)$, the minimum cost from 1 to u. Again, this is initially true with $S = \{1\}$ and $d[1] = 0 = \delta_1(1)$.

From Invariant (B), we can interpret (1) in Invariant (A) as saying that $d[u]$ is *the minimum cost ranging over all paths from* 1 *to* u *whose intermediate vertices are restricted to* S. This implies

$$d[u] \geq \delta_1(u), \qquad (u \in V). \tag{2}$$

LEMMA 4. *Assume Invariants (A) and (B). Let $u_0 \in U = V \setminus S$ such that*

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

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

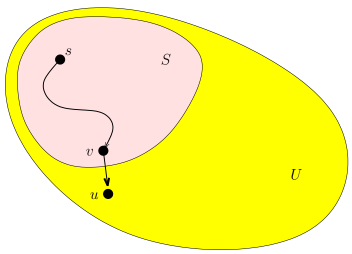


Figure 1: Dijkstra's Invariant

Proof. First, we take care of the trivial case when $\delta_1(u_0) = \infty$. In this case, the lemma is true because (2) implies $d[u] = \infty$.

Otherwise, there exists a minimum 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 $u \in U$. This decomposition exists because the first vertex 1 in p is in S and the last vertex u_0 in p is in U. See figure 1. Note that $C(p') = \delta_1(v)$. Then

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

Combined with equation (2), 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:

$$d[i] \leftarrow \min\{d[i], d[u_0] + C(u_0, i)\}. \tag{3}$$

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[2] to support the DecreaseKey operation, which is needed in updating $d[i]$ *á la* equation (3).

[2] 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*
  $d[1] \leftarrow 0$; Initialize an empty queue Q.
  for $i \leftarrow 2$ to n, $d[i] \leftarrow \infty$,
  for $i \leftarrow 1$ to n, Q.Insert$(i, d[i])$.
▷ *MAIN LOOP*
  while $Q \neq \emptyset$ do
    $u_0 \leftarrow Q$.DeleteMin()
    for all i adjacent to u_0 do
      If $d[i] > d[u_0] + C(u_0, i)$ then
        $d[i] \leftarrow d[u_0] + C(u_0, i)$
        Q.DecreaseKey$(i, d[i])$
  end{while}

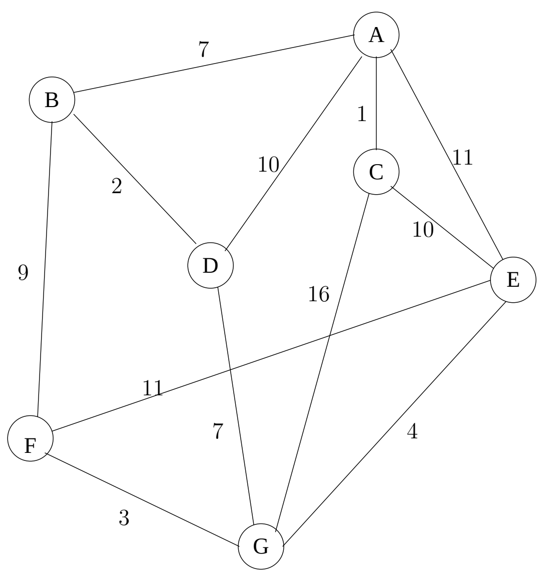


Figure 2: Illustrating Dijkstra's Algorithm

¶14. 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							$\underline{15}$
STAGE 7						$\underline{16}$	

¶15. 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: Carry the hand-simulation of Dijkstra's algorithm for the graph in Figure 2, but using the edge costs $C_9(e)$ defined as follows: $C_9(e) = C(e) + 9$ if $C(e) \le 9$, and $C_9(e) = C(e) - 9$ if $C(e) > 9$. ♢

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 satisfy properties (A) and (B), 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 minimum 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 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 $O(n^3)$. ♢

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

Dijkstra's algorithm can serve as the basis for several important and useful extensions.

¶16. 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, we have a specific target, say vertex n. In this case, 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 $S \subseteq V$ for which $\delta_1(v)$ is known for each $v \in S$; the backward search from target n maintains a similar set $S' \in V$ for which $\delta_n(v)$ 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 $S \cap S'$. Now, the "standard mistake" according to [1] is to assume that $\delta_1(n) = \delta_1(z) + \delta_n(z)$.

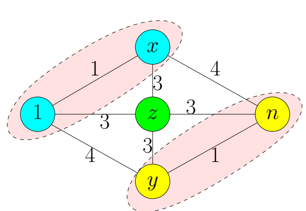


Figure 3: Bidirectional search from 1 to n

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. At this point, $\delta_1(z) = 3$ and $\delta_n(z) = 3$. The standard mistake is to conclude $\delta_1(n) = \delta_1(z) + \delta_n(z) = 3 + 3 = 6$. 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 correct, but we need to track the potential values of $\delta_1(n)$ using a "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:

$$\Delta' \leftarrow \delta_1(v) + \delta_n(v') + C(v, v');$$
$$\Delta \leftarrow \min\{\Delta, \Delta'\}.$$

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

$$\Delta \leftarrow \min\{\Delta, \delta_1(z) + \delta_n(z)\}.$$

We claim that the final value of Δ is $\delta_1(n)$: It is easy to see that $\Delta \le \delta_1(n)$. The converse is also easily verified because the minimum distance from 1 to n must have the form $\delta_1(v) + \delta_n(v') + C(v, v')$ or equal to the $\delta_1(z) + \delta_n(z)$ 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 $\Delta = 5$.

¶17. 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).

Figure 4. US Road Map

Consider a concrete example. Suppose the graph $G = (V, E)$ represents the road network of the United States with $V = \{1, \ldots, n\}$ representing cities and cost $C(i, j)$ representing the minimum distance road 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

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

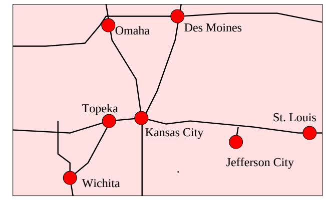
Suppose city 1 is Kansas City (Kansas/Missouri), near the geographical center of the US, and W is the set of cities on the West Coast. A standard Dijkstra search would fan out from Kansas City equally in all directions. Intuitively, our goal-directed search ought to explore the graph G with a westward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

How can Dijkstra's algorithm be modified to serve this purpose? This is the important heuristic called **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:

$$0 \le h(j) \le \delta(j, W). \qquad (4)$$

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 (from Map Quest):

Distance (in miles)	Topeka	Omaha	Des Moines	Jefferson City	Wichita
Kansas City	61	184	197	161	197
San Francisco	1780	1669	1800	1968	1680
	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., namely their respectively distance to San Francisco). For the sake of argument, suppose we added the actual distance of these cities to San Francisco. We see that A* would still choose Topeka to be added to S because its value of 1841 is still minimum. Next, ordinary Dijkstra 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.

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 $C : E \to \mathbb{R}_{\ge 0}$ is modified by h to become

$$C^h(i, j) := C(i, j) - h(i) + h(j). \tag{5}$$

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:

$$\delta^h(i, j) = \delta(i, j) - h(i) + h(j). \tag{6}$$

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** for 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. What is new is that the order in which we add nodes 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 $C(i, j) - h(i) + h(j) \ge 0$, or we prefer to write this as a kind of triangular inequality:

$$C(i, j) + h(h) \ge h(i). \tag{7}$$

This property has various names in the literature: Following Goldberg and Harrelson [1], we say h is **feasible** if (7) holds. The literature also calls h monotone or consistent instead of feasible.

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 $S \subseteq S'$.*

¶18. 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?

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. ♢

END EXERCISES

§5. Semirings

Before considering the all pairs minimum cost problems, let us recall some facts about matrix rings.

Let us first informally review some college algebra: 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. The rings that satisfy this law is called a commutative ring.

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 $M_n(R)$ is no longer commutative for $n \geq 2$. 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

$$C_{ij} = \sum_{i=1}^{k} A_{ik} B_{kj}. \tag{8}$$

Let us now proceed somewhat more formally: a **ring**

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

By definition[3] 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 $+$.

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 also written as ab or $a \cdot b$. For $n \geq 1$, we have another ring with unity,

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

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 (8). 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.

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 (1969) demonstrated that the obvious $\mathtt{MM}(n) = O(n^3)$ bound is suboptimal. The current record is from Coppersmith and Winograd (1987):

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

¶19. 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 an exact 2-link 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 (8), except that we have replaced summation by minimization, and product by summation. Hence *computing the exact 2-link 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. Also,

$$(-\infty) + x = \begin{cases} -\infty & \text{if } x \neq \infty, \\ \infty & \text{if } x = \infty. \end{cases}$$

In fact, the only thing this structure lacks to make it a ring is *an inverse for minimization*. Such structures are quite pervasive, and is studied abstractly as semirings:

[3] All our rings have a multiplicative identity usually denoted 1: $x \cdot 1 = 1 \cdot x = x$. We call 1 the **unity** element. Algebraists sometimes consider rings with such a unity element. Algebraic structures such as rings, groups, etc, are sets S together with operations o_1, o_2, etc, and are written $(S, o_1, o_2, \ldots)$. A constant is just a 0-ary operation. 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 $-x$.

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.

¶20. 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 $(\mathbb{N}, +, \times, 0, 1)$. It is useful to test all concepts about semirings against this one.

2. Another important semiring is
$$(\mathbb{R} \cup \{\pm\infty\}, \min, +, +\infty, 0) \tag{9}$$
as noted above. For reference, call this the **minimization semiring**. Note[4] 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. Be careful that the "multiplication" in the minimization semiring is ordinary addition! To avoid confusion, we may 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**,
$$(\mathbb{R} \cup \{\pm\infty\}, \max, +, -\infty, 0). \tag{10}$$
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**,
$$(\mathbb{R}_{\geq 0} \cup \{\infty\}, \min, +, \infty, 0). \tag{11}$$

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 $A, B \subseteq \Sigma^*$, 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.

[4] In standard extensions of the real numbers to $\pm\infty$, it is stipulated that $\infty + (-\infty)$ is undefined.

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 (8).*

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.

¶21. 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

$$\mathtt{MM}_2(n) = O(\mathtt{MM}(n) \lg n). \tag{12}$$

§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**

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

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

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

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:

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

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

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

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 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.

¶22. 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 $\le$ in an idempotent semiring $(R, \oplus, \otimes, 0, 1)$ by defining

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

To check that this is a partial order: Clearly $x \le x$. If $x \le y$ and $y \le x$ then $x = y$. Finally, $x \le y$ and $y \le z$ implies $x \le z$ (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 $x \le y, x' \le y'$ implies $x \oplus y \le x' \oplus y'$. But be warned that in the minimization semiring $\mathbb{R} \cup \{\pm\infty\}$, this definition "$\le$" 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

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

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

Exercise 6.2: 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 $x_i \bar{x}_i = \bar{x}_i x = \epsilon$. $\diamondsuit$

Exercise 6.3: Under what conditions does the canonical construction of $\mathbb{Z}$ from $\mathbb{N}$ extend to give a ring from a semiring? $\diamondsuit$

Exercise 6.4: 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 $y = 1 \oplus (x \otimes y)$. $\diamondsuit$

Exercise 6.5: 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? $\diamondsuit$

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.

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 (9)). An easy generalization of an earlier observation (for the case $k = 2$) gives:

LEMMA 7. *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 exact k-link 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**[5]. Another advantage to the Floyd-Warshall algorithm is that we do not need to assume the absence 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 k-link 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 $k \ge 1$:

$$\delta^{[k]}(i,j) = \min\{\delta^{[k-1]}(i,j), \delta^{[k-1]}(i,k) + \delta^{[k-1]}(k,k)^* + \delta^{[k-1]}(k,j)\} \tag{13}$$

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

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

Notice that $\delta^{[n]}(i,j)$ is precisely equal to $\delta(i,j)$. The Floyd-Warshall algorithm simply uses equation (13) to compute $\delta^{[k]}$ for $k = 1, \ldots, n$:

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

Floyd-Warshall Algorithm:
Input: Cost matrix C which is n by n.
Output: Matrix $c[1..n, 1..n]$ representing δ.
```
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 (13) 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 7.1: 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 7.2: 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 7.3: 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 7.4: 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 $\ge Cn^2$ 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. $\diamondsuit$

Exercise 7.5: Modify the Floyd-Warshall Algorithm so that it computes the lengths of the first and also the second minimum path. The second minimum path must be distinct from the minimum path. In particular, if the minimum path does not exist, or is unique, then the second minimum path does not exist. In this case, the length is ∞. $\diamondsuit$

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 $w = (i_0, \ldots, i_m) \in [1..n]^*$, 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 8.
(i) $C^{[0]} = C$ *and for* $k = 1, \ldots, n$,

$$C_{ij}^{[k]} = C_{ij}^{[k-1]} \oplus \left(C_{ik}^{[k-1]} \otimes (C_{kk}^{[k-1]})^* \otimes C_{kj}^{[k-1]} \right) \tag{14}$$

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

Proof. We only verify equation (14), 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 (14). 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

$$C_{ij} = C_{ij} \oplus (C_{ik} \otimes (C_{kk})^* \otimes C_{kj})\,. \tag{15}$$

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

¶23. 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 $\mathsf{TC}_2(n)$ 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

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

We now improve this bound by exploiting the bound

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

(see equation (12)). *We may assume that* $\mathsf{MM}_2(n) = \Omega(n^2)$ *and* $\mathsf{TC}_2(n) = \Omega(n^2)$. 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 $\mathsf{MM}_2(n)$ is an upper bound on addition of matrices, which is $O(n^2)$. Our main result will be:

THEOREM 9. $\mathsf{TC}_2(n) = \Theta(\mathsf{MM}_2(n))$.

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 $\mathsf{TC}_2(n) = O(\mathsf{MM}_2(n))$, 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

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix}^* = \begin{pmatrix} E^* & E^*BD^* \\ D^*CE^* & D^* + D^*CE^*BD^* \end{pmatrix}, \tag{16}$$

where

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

This formidable-looking expression (16) 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 (16). 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 (16). 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 (14) that corresponds to the improved rule (15). 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 (16). ♢

Exercise 8.3: Express the RHS of equation (16) 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

$$\widehat{C}(i,j) = C(i,j) + \phi(i) - \phi(j). \tag{17}$$

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 10. *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 11. *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 ∉ V.
        Invoke Bellman-Ford on (V', E'; C', s) to compute δ_s.
        Abort if negative cycle discovered; else, for all u, v ∈ V,
            let Ĉ(u, v) ← C(u, v) + δ(s, u) − δ(s, v)
    MAIN LOOP
        For each v ∈ V, invoke Dijkstra's algorithm on (V, E; Ĉ, 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 $O(n^2 \log n + mn)$.

§10. 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 [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:

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

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

$$\ell \le d_{ij} \Longrightarrow \ell = d_{ik} = d_{ij} - d_{jk}. \tag{18}$$

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

LEMMA 12.
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 (18), 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 13. *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 $d_i j$ 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

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

¶24. 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 13. 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 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 $e_{ij} = n^{n-d'_{ij}}$. $\diamondsuit$

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.