Lecture 22 Exercise Solutions

Mark Eramian

Exercise 1

a) Worst case analysis: If we are trying to compute $C = A \cdot B$, we find $C(i, j)$ by finding the sum of the products of the elements of the i-th row of A and the j-th row of B. If the matrices A and B are square, then this requires $O(n)$ multiplications and $O(n)$ additions, where n is the size of the matrices. So it costs $O(n)$ time to compute $C(i, j)$. Since C is also n by n, we have to do this n^2 times for a total time of $n^2 O(n)$ or $O(n^3)$. Best case: same as the worst case. Thus, matrix multiply is $\Theta(n^3)$.

b) The algorithm just computes the adjacency matrix A to the r-th power (best case = worst case). That's exactly $r - 1$ matrix multiplies, each of which costs $\Theta(n^3)$, for a total of $\Theta(rn^3)$.

Exercise 2

a) Worst case: we have to compute each power of A from A^1 to A^{n-1}. With one matrix multiply we can obtain A^i from A^{i-1}. Each time we get the next power, we add it to B. Thus we can obtain B in exactly $n - 2$ matrix multiplies. Each matrix multiply costs $O(n^3)$. Thus, the total time is $(n - 2)O(n^3)$ or $O(n^4)$. Best case: same as the worst case, so the algorithm is $\Theta(n^4)$ (n is the number of nodes).

b) Change the last line to simply: `return B`.

c) We still need to compute B, so the time complexity is the same as in part a).

Exercise 3

Warshall's algorithm has three nested loops. In the worst case, the body of the innermost loop is $O(1)$. The inner loop body runs n times and therefore the entire inner loop costs $O(n)$. The middle loop runs the inner loop n times, each run costs $O(n)$ so the total cost of the middle loop is $O(n^2)$. The outer loop runs the middle loop n times, and each such run of the middle loop costs $O(n^2)$ time so the outer loop costs $O(n^3)$. Beyond the outer loop there are just two additional statements, so the total time is $O(n^3) + 2$ or just $O(n^3)$. The best case is the same as the worst case, so Warshall's algorithm is $\Theta(n^3)$.

Exercise 4

a) True. In the case of a directed graph, the adjacency matrix is not symmetrical, but this poses no problems for computing powers of the adjacency matrix.

b) Probably not. If we only need to know if there is a path between two specific nodes, then Warshall's algorithm does much more work than we need. Warhsall's algorithm will give us the answer we want, but at a cost of $O(n^3)$ time where $n = |V|$. We can solve the same problem using a breadth-first search which costs at most $O(n^2)$, or possibly only $O(|E| + |V|)$ if we are using an adjacency list representation.

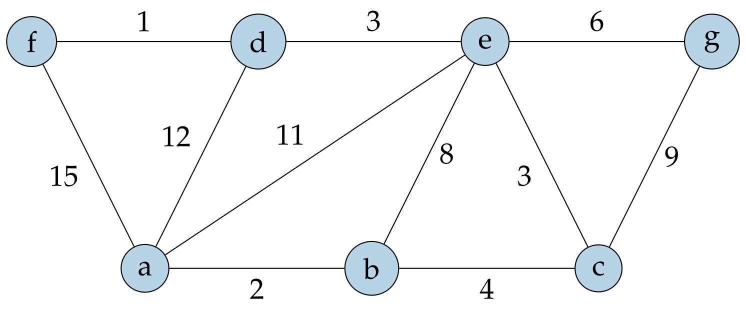
Exercise 5

a) The timing analysis of Floyd's algorithm is identical to that of Warshall's algorithm (remember, it's essentially the same algorithm!).

b) Yes. The negative weights will just make the paths cheaper.

c) No. If the sum of the weights along a cycle is negative, then there is no shortest path between two nodes on a negative cycle. You can always make a path between such a pair of nodes even shorter by just going around the cycle again. However, Floyd's algorithm can **detect** negative cycles. If any diagonal entry of the matrix is negative, then that means there is a path from a node v to itself with negative weight, which is the definition of a negative cycle!

1 Exercise 6



```
Algoirthm dijkstra(G, s)
G is a weighted graph with non-negative weights.
s is the start vertex.

Let V be the set of vertices in G.

For each v in V
    v.tentativeDistance = infinity
    v.visited = false
    v.predecessorNode = null

s.tentativeDistance = 0

while there is an unvisited vertex
    cur = the unvisited vertex with the smallest tentative distance.
    cur.visited = true

    // update tentative distances for adjacent vertices if needed
    // note that w(i,j) is the cost of the edge from $i$ to $j$.
    For each z adjacent to cur
         if (z is unvisited and z.tentativeDistance >
                               cur.tentativeDistance + w(cur,z) )
            z.tentativeDistance = cur.tentativeDistance + w(cur,z)
            z.predecessorNode = cur
```

For this trace through, we will demonstrate that we don't really need to keep drawing it as a graph as we go. All we really need is the adjacency list or matrix and to keep track of the tentative distances, predecessor nodes, and visited status for each node in the graph. We will represent each with an "array" of values indexed by node identifier. Initially, all vertices are unvisited, no nodes have known predecessor, and all tentative distances are ∞ except the tentative distance to node a. This is the state of things just before we enter the while loop of the algorithm for the first time. We represent it thusly:

	a	b	c	d	e	f	g
Visited:	F	F	F	F	F	F	F

	a	b	c	d	e	f	g
Predecessor:	-	-	-	-	-	-	-

	a	b	c	d	e	f	g
Tentative Distance:	0	∞	∞	∞	∞	∞	∞

Now we enter the loop and need to pick the unvisited vertex with the smallest tentative distance, which is a. We mark a as visited and then process each unvisited adjacent node. The unvisited nodes adjacent to a are b, d, and f. For node b we now have a path from a to b with cost 2, on which the predecessor of b is a. Likewise, for node d we have a path from a to d via node a with a cost of 12, for node e we have a path from a to e with cost 11, and for node f a path from a to f via node a with a cost of 15. We record all of this:

	a	b	c	d	e	f	g
Visited:	T	F	F	F	F	F	F

	a	b	c	d	e	f	g
Predecessor:	-	a	-	a	a	a	-

	a	b	c	d	e	f	g
Tentative Distance:	0	2	∞	12	11	15	∞

We are now finished processing node a. The node with the smallest tentative distance is b, so we mark b as visited and consider its unvisited adjacent nodes, which are e and c. The tentative distance to c is equal to the tentative distance to b (2) plus the cost of the edge from b to c (4), or 6, which is smaller than the existing tentative distance to c (∞), so we replace it and update predecessor of c to be b. Similarly, the existing tentative distance to e of 11 is replaced by the tentative distance to b (2) plus the cost of the edge from b to e (8), or 10. The predecessor of e is updated to b. So our data become:

	a	b	c	d	e	f	g
Visited:	T	T	F	F	F	F	F

	a	b	c	d	e	f	g
Predecessor:	-	a	b	a	b	a	-

	a	b	c	d	e	f	g
Tentative Distance:	0	2	6	12	10	15	∞

We are now finished processing b, so we start the next iteration of the while loop. The unvisited node with the smallest tentative distance is c. We mark c as visited, and proceed with the for loop. The unvisited vertices adjacent to c are e and g. The current path to e via c is the tentative distance to c (6) plus the cost of the edge from c to e (3) which is 9. This is smaller than the existing tentative distance for e (10). So we update e's tentative distance to 9, and it's predecessor to c. For node g, the existing tentative distance is ∞, so we update it's tentative distance to the tentative distance to c (6) plus the cost of the edge from c to g (9) or 15, and update g's predecessor to c.

	a	b	c	d	e	f	g
Visited:	T	T	T	F	F	F	F

	a	b	c	d	e	f	g
Predecessor:	-	a	b	a	c	a	c

	a	b	c	d	e	f	g
Tentative Distance:	0	2	6	12	9	15	15

We are now done processing c and start a new iteration of the while loop. The unvisited vertex with the smallest tentative distance is now e. We mark e as visited, and proceed with the for loop. The unvisited nodes adjacent to e are d and g. The current path to d via e has cost equal to the tentative distance to e (9) plus the cost of the edge from e to d (3) or 12, which is **not** less than the existing tentative distance to d of 12, so nothing gets updated. The current path to g via e has cost equal to the tentative distance to e (9) plus the cost of the edge from e to g (6), or 15. Again, this is not strictly less than the existing tentative distance to g, so nothing gets updated.

	a	b	c	d	e	f	g
Visited:	T	T	T	F	T	F	F

	a	b	c	d	e	f	g
Predecessor:	-	a	b	a	c	a	c

	a	b	c	d	e	f	g
Tentative Distance:	0	2	6	12	9	15	15

Now we are finished processing *e* and are back at the top of the while loop. The unvisited vertex with the smallest tentative distance is now *d*. We mark *d* as visited and start the for loop. The only unvisited vertex adjacent to *d* is *f*. The cost of the current path to *f* via *d* is equal to the tentative distance to *d* (12) plus the cost of the edge from *d* to *f* (1), or 13. Thus we have found a shorter path to *f* that goes via *d*. Consequently, the tentative distance to *f* gets updated to 13 and the predecessor of *f* gets updated to *d*.

	a	b	c	d	e	f	g
Visited:	T	T	T	T	T	F	F

	a	b	c	d	e	f	g
Predecessor:	-	a	b	a	c	d	c

	a	b	c	d	e	f	g
Tentative Distance:	0	2	6	12	9	13	15

We are now done processing *d*. We come back to the top of the while loop and select *f* as the unvisited vertex with the smallest tentative distance. We mark *f* as visited, and begin the for loop. Since there are no unvisited vertices adjacent to *f*, nothing happens in the for loop.

	a	b	c	d	e	f	g
Visited:	T	T	T	T	T	T	F

	a	b	c	d	e	f	g
Predecessor:	-	a	b	a	c	d	c

	a	b	c	d	e	f	g
Tentative Distance:	0	2	6	12	9	13	15

We are now finished processing f, and return to the top of the while loop. The only remaining unvisited vertex is g. We mark g as visited and begin the for loop. g has no adjacent vertices, so the for loop does nothing.

	a	b	c	d	e	f	g
Visited:	T	T	T	T	T	T	T

	a	b	c	d	e	f	g
Predecessor:	-	a	b	a	c	d	c

	a	b	c	d	e	f	g
Tentative Distance:	0	2	6	12	9	13	15

We now return to the while loop condition which is now false, because there are no unvisited vertices. So the algorithm ends. From the final arrays we can easily extract not only the costs of the shortest path from a to every other node (which are stored in the "tentative distance" array), but also, using the "predecessor" array, we can reconstruct, in reverse, the actual shortest path from any node back to a.

Exercise 7

We'll start with the worst-case analysis for an undirected graphs $G = (V, E)$.

Time Complexity of Dijkstra's algorithm for Undirected Graphs

The inner for loop executes for each vertex adjacent to `cur`. For any given vertex `cur`, the inner loop executes exactly $degree(cur)$ times. In the worst case, every vertex adjacent to `cur` is unvisited, and the if-statement is always true. Thus, the inner loop requires $O(degree(cur))$ time in the worst case.

The outer loop executes exactly $|V|$ times because initially all vertices are unvisited, and each iteration of the while loop causes exactly one unvisited vertex to be marked as visited. The inner loop will therefore be executed once for each vertex. This will require $O(degree(a)) + O(degree(b)) + O(degree(c)) + \cdots + O(degree(g))$ time. This can be rewritten as:

$$O(degree(a) + degree(b) + \cdots + degree(g))$$

Since each edge in an undirected graph is incident on exactly two vertices, the sum of the degrees of the vertices in an undirected graph is $2|E|$. Thus, the total cost of **all** executions of the for-loop is $O(|E|)$.

But, the cost of finding the unvisited vertex with the smallest tentative distance is not constant! So we have to take that into account too. If we do this naïvely using a linear search, this adds a cost of $O(|V|)$ to each iteration of the while loop, which has a total cost of $O(|V|^2)$ over all iterations of the while loop. This results in a total cost of the while loop of $O(|V|^2 + |E|)$. Since $|E|$ is at most $|V|^2$ (every possible edge is present), this is the same as $O(|V|^2)$.

However, we could maintain a priority queue (implemented by an arrayed heap) of the unvisited nodes where their priorities are their tentative distance. Each time a tentative distance changes, we can re-heapify the heap underlying the queue at the cost of $O(\log |V|)$ time. Without proof, this incurs a total cost of updating tentative distance of at worst $O(|E| \log |V|)$ but also incurs a cost of removing each from the heap, which is $O(|V| \log |V|)$ in total for the entire while loop. In this way, the total cost of the while loop is reduced to $O(|V| \log |V| + |E| \log |V| + |E|) = O((|V| + |E|) \log |V|)$.

If, however, we use a more advanced type of heap called a Fibonacci heap instead, the cost of updating the smallest tentative distances in the for-loop while retaining the heap property is reduced to $O(1)$, leaving only the cost of removing nodes from the heap, which remains $O(|V| \log |V|)$, making the total for the while loop $O(|V| \log |V| + |E|)$

Finally, we have to add the cost of the initialization, which is clearly $O(|V|)$. Thus the total cost of the algorithm becomes:

- $O(|V|^2 + |E| + |V|) = O(|V|^2)$ for the naïve linear search implementation;
- $O((|V| + |E|) \log |V| + |V|) = O((|V| + |E|) \log |V|)$ for the priority queue/arrayed heap implementation;
- $O(|V| \log |V| + |E| + |V|) = O(|V| \log |V| + |E|)$ for the Fibonacci heap implementation.

Time Complexity of Dijkstra's algorithm for Directed Graphs

The time complexity analysis of Dijkstra's algorithm for directed graphs is nearly identical to the undirected graph case. The main difference is that the total cost of all executions of the inner for-loop becomes

$$O(outdegree(a) + outdegree(b) + \cdots + outdegree(g))$$

The sum of all of the outdegrees of all nodes in a directed graph is exactly $|E|$ (each edge is outgoing from exactly one vertex) so the total cost of all executions of the for-loop is $O(|E|)$, which is the same as for the undirected graph. The remainder of the analysis remains the same as in the undirected case.