CSE 3101Z Design and Analysis of Algorithms
Professor: James Elder
Winter 2009
Assignment 3
Due 11:59pm Monday May 4

1. (10 marks) **Breadth- and Depth-First Search**

Trace BFS and DFS on the following two graphs. Hand in this page as Page 2 of your assignment. For each do the following:

(a) (2 marks) Start at node $s$ and when there is a choice follow edges from left to right. **Number** the nodes $1, 2, 3, \ldots$ in the order that they are found, starting with Node $s = 1$.

(b) (2 marks) **Darken** the edges of the Tree specified by the predecessor array $\pi$.

(c) (1 mark) What is the data structure used by BFS to store nodes that are found but not yet handled?

**Answer:** A queue.

(d) (2 marks) Circle the nodes that are in this data structure when Node 8 is first found.

(e) (1 mark) What is the data structure used by DFS to store nodes that are found but not yet handled?

**Answer:** A stack.

(f) (2 marks) Circle the nodes that are in this data structure when Node 8 is first found.

**Answer:**

A) Breadth First Search (BFS)  
B) Depth First Search (DFS)

2. (10 marks) **Dijkstra’s Algorithm.** Hand in this page as Page 3 of your assignment.

(a) (5 marks) Consider running Dijkstra’s algorithm on the following graph from source node $a$. Suppose we are at the stage of the computation when only nodes $a$, $b$ and $d$ have been handled
(in other words, we are just about to dequeue another node). Please indicate on the graph the current value of the distance estimate \( d \) for each node in the graph.

![Graph showing distance estimates]

Answer:

\[
\begin{align*}
\text{d=0} & : a \\
\text{d=16} & : b \\
\text{d=34} & : c \\
\text{d=20} & : d \\
\text{d=27} & : h \\
\end{align*}
\]

(b) (5 marks) Now consider running the next iteration of Dijkstra’s algorithm on this graph. On the graph below, show the new values of \( d \) for each node and any changed values of the predecessor array \( \pi \).

![Graph showing updated distance estimates]

Answer:

\[
\begin{align*}
\text{d=0} & : a \\
\text{d=16} & : b \\
\text{d=34} & : c \\
\text{d=20} & : d \\
\text{d=27} & : h \\
\end{align*}
\]

\( \pi = e \)

3. (15 marks) **Network flow**

Consider the flow network shown below, with source \( s \) and terminal (or sink) \( t \). The labels on the edges are of the form \( f/c \) where \( f \) is the flow and \( c \) is the capacity of the edge. This flow is feasible.
Consider one iteration of the network flow algorithm (i.e., the Ford-Fulkerson algorithm) on this flow network.

(a) (5 marks) Show the residual network for this flow. Identify an augmenting path on this network.

Answer:

One augmenting path is: \( s \to c \to d \to a \to b \to t \).

(b) (3 marks) Draw the new flow network obtained by adding the flow along this path.

Answer:

(c) (2 marks) What is the value of this flow?

Answer: The value of this flow value is \( f(s, a) + f(s, c) = 6 + 8 = 14 \).

(d) (5 marks) Prove that this flow is a maximum flow.

Answer: The min-cut \( (C, \overline{C}) \), where \( C = \{s, a, c, d\} \), is also shown on the right part of the above figure. The min-cut capacity is \( c(a, b) + c(d, b) + c(d, t) = 7 + 2 + 5 = 14 \). No flow can exceed this cut.
4. (20 marks) **Fixed edge on a shortest path** Given a directed, weighted graph \( G \) with non-negative weights (weights can be zero), vertices \( s, t \) and an edge \((u, v)\) in \( G \):

(a) (10 marks) Describe in concise English how to determine whether \((u, v)\) occurs on every path of minimal cost from \( s \) to \( t \).

**Answer:** First use Dijkstra’s algorithm to compute the cost \( d \) of the shortest path from \( s \) to \( t \). If \( d = \infty \), then there is no path of minimal cost from \( s \) to \( t \), and it is true that \((u, v)\) occurs on every min cost path from \( s \) to \( t \).

Now consider the graph \( G' \) obtained from \( G \) by removing the edge \((u, v)\). Use Dijkstra’s algorithm again to compute the cost \( d' \) of the shortest path from \( s \) to \( t \) in \( G' \). If \( d \neq d' \) return true, otherwise return false.

To see that the algorithm works, first assume that the edge \((u, v)\) does appear on every min-cost path from \( s \) to \( t \) in \( G \). That is, \((u, v)\) appears on every path of cost \( d \) from \( s \) to \( t \). Hence the min-cost path found on \( G' \) cannot be of cost \( d \), because it does not contain \((u, v)\). Hence \( d' \neq d \), and the algorithm returns true in this case.

If there is a min-cost path from \( s \) to \( t \) not containing \((u, v)\), then the same path would be a shortest path in \( G' \), so \( d' = d \), and the algorithm returns false in this case.

The algorithm calls Dijkstra’s algorithm twice, and can be implemented in \( O(E \log V) \) time.

(b) (10 marks) Describe in concise English how to determine whether \((u, v)\) occurs on some path of minimal cost from \( s \) to \( t \).

**Answer:** Once again, we run Dijkstra’s algorithm from \( s \) to compute the weights of the shortest paths \( d(s, t) \) and \( d(s, u) \). If \( d(s, t) = \infty \), then there is no min cost path from \( s \) to \( t \), and it is false that there is such a path through \((u, v)\).

If \( d(s, t) < \infty \), we run Dijkstra’s algorithm from \( v \) to compute \( d(v, t) \). We return true if \( d(s, t) = d(s, u) + w(u, v) + d(v, t) \), and false otherwise.

To show that the algorithm works, suppose that there is a min-cost path \( p = s \overset{p_1}{\longrightarrow} \ldots \overset{u}{\longrightarrow} v \overset{p_2}{\longrightarrow} t \) through \((u, v)\). We know that a subpath of a shortest path must be a shortest path, hence \( p_1 \) is a shortest path from \( s \) to \( u \), and \( p_2 \) is a shortest path from \( v \) to \( t \). We have \( w(p) = d(s, t) \), \( w(p_1) = d(s, u) \) and \( w(p_2) = d(v, t) \). So \( d(s, t) = w(p) = w(p_1) + w(u, v) + w(p_2) = d(s, u) + w(u, v) + d(v, t) \), and the algorithm returns true in this case.

In the opposite direction, suppose that the algorithm returns true. Then we have \( d(s, t) = d(s, u) + w(u, v) + d(v, t) \). Denote by \( p_1 \) a shortest path from \( s \) to \( u \), and \( p_2 \) a shortest path from \( v \) to \( t \). Then \( w(p_1) = d(s, u) \), and \( w(p_2) = d(v, t) \). Take the path \( p = s \overset{p_1}{\longrightarrow} \ldots \overset{u}{\longrightarrow} v \overset{p_2}{\longrightarrow} t \). \( p \) obviously contains the edge \((u, v)\), and \( w(p) = w(p_1) + w(u, v) + w(p_2) = d(s, u) + w(u, v) + d(v, t) = d(s, t) \), so \( p \) is a min-cost path containing \((u, v)\), which proves that there exists such a path.

The algorithm calls Dijkstra’s algorithm twice, and can be implemented in \( O(E \log V) \) time.
5. (25 marks) **Greenhouses and plants** A botanical garden is planning to build \( n \) greenhouses, each of which is intended to represent a different climate zone. They have a list of \( m \) kinds of plants that they would like to grow there. Not every kind of plant can grow in every greenhouse, though usually one kind can grow in more than one climate; so associated with every greenhouse \( i \in [1 \ldots n] \) is a list \( C_i \) of plant varieties it can grow. In addition to that, every greenhouse can host no more than 20 varieties of plants.

(a) (15 marks) Given the \( n, m \) and \( C_i \) for \( 1 \leq i \leq n \), describe in concise English an algorithm to determine the maximal number of different varieties of plants that can be grown in these greenhouses (total over all greenhouses).

**Answer:** This problem is similar to a bipartite matching problem. We need to construct a flow network representing a “matching” of plants to greenhouses. The flow network \( F = (G, s, t, c) \) for this problem will consist of the following:

- The vertices of \( G \) will be \( s, t \) for source and sink, vertices \( g_1, \ldots, g_n \) representing greenhouses and vertices \( v_1, \ldots, v_m \) representing varieties of plants.
- Edges \((s, g_i)\) of capacity 20 for \( 1 \leq i \leq n \). This will constrain every greenhouse to have at most 20 varieties of plants.
- Edges \((g_i, v_j)\) of capacity 1, \( 1 \leq i \leq n, 1 \leq j \leq m \) for pairs \((i, j)\) such that \( j \in C_i \). These edges will match varieties of plants with greenhouses where they can grow. Since we wish to maximize the number of varieties of plants, we will not allow more than one plant of the same variety to grow in a greenhouse.
- Edges \((v_j, t)\) of capacity 1, where \( 1 \leq j \leq m \). Restricting the capacity to 1 ensures that every variety will be grown in at most one greenhouse.

![Flow network diagram](https://via.placeholder.com/150)

To find the maximal number of varieties of plants that can be grown in these greenhouses, we run the Ford-Fulkerson algorithm on this network. The value of the resulting flow is the maximal number of varieties that can be grown. Every greenhouse hosts no more than 20 varieties of plants.
due to the capacity constraint on \((s, g_i)\) edges, and these varieties can be grown there because of the choice of \((g_i, v_j)\) edges; therefore, this set of varieties can feasibly be grown. To show that it is maximal we appeal to the fact that Ford-Fulkerson always produces the maximal flow. Suppose that there is a better assignment of varieties to plants, allowing for more varieties to be grown. We argue that every pair \((i, j)\) where variety \(j\) is assigned to be grown in the greenhouse \(i\) corresponds to an augmenting path in the network; the path \((s, g_i, v_j, t)\). Since no variety is chosen more than once, and no greenhouse hosts more than 20 varieties, each pair \((i, j)\) gives rise to an augmenting path satisfying the constraints. Therefore, there would be a flow on the network with value greater than the value of the flow produced by Ford-Fulkerson; a contradiction. Thus, the value of the flow on this network is equal to the maximum number of varieties of plants that can be grown.

(b) (10 marks) How can you produce a list of chosen varieties for each greenhouse? Is this list uniquely determined? Justify your answer with an example or a proof.

**Answer:** Take the resulting flow on the network, and consider a cut between \((s, g_1, \ldots, g_n)\) and \((v_1, \ldots, v_m, t)\). Since every cut has the same flow through it, and by the integrality theorem if the capacities are integers then there is an integer-valued flow, there are exactly \(|f|\) edges in that cut that have flow of 1, and the rest have flow of 0. Now the list of chosen varieties for a greenhouse \(i\) is the list of all \(j\) such there is an edge \((g_i, v_j)\) and it has flow of 1.

The flow is not uniquely determined by the network, although its value is; the flow depends on the order in which augmenting paths are chosen. For example, suppose there is one greenhouse and 21 varieties of plants that can all grow in it. Then the resulting network will have flow of 20, and 20 out of these 21 varieties will be chosen, but it is up to the implementation of Ford-Fulkerson to determine which subset of 20 varieties will be chosen.

6. (20 marks) **Expedition** A group of people is planning an expedition. For safety, they decide to split into two groups; it does not matter how many people are in each group. Not all of them are good friends, and one reason for splitting is to separate all pairs that are not friendly to each other. Your goal is to design an algorithm that, given the map of friendship relations between people in the group, would find a partition of people into two groups of friends, or say that it is impossible.

Assume that the input is an \(n \times n\) matrix \(M\), where \(n\) is the number of people in the group and \(M(i, j) = 1\) if \(i\) and \(j\) are friends, and \(0\) otherwise. Also, if \(i\) is friendly to \(j\), then \(j\) is friendly to \(i\).
Your output should be the sequence of numbers corresponding to people assigned to the first group.

(a) (10 marks) Describe in concise English an algorithm that solves this problem.

**Answer:**
This problem is the same as determining whether a graph is bipartite, or, equivalently, two-colourable. Each of the \(n\) people is identified with a node in an undirected graph, and \(M\) identifies the edges: an edge exists between two nodes \(i\) and \(j\) if \(M(i, j) = 0\). In other words, the graph makes explicit which people are not friends.
Let the groups be labelled 1 and 2, and let $A(1\ldots n)$ hold the group assignments. $A$ will be initialized to 0, to indicate that no assignments have been made.

The problem can then be solved by a depth- or breadth-first search over the entire graph. The source node is assigned to Group 1, and the search is initiated. When a node $j$ is encountered while exploring the adjacency list for node $i$, a check is first made to see if $A(i) = A(j)$. If so, the search terminates and the algorithm reports that the partition is impossible. Otherwise, the algorithm assigns $A(j) = 3 - A(i)$. If the graph is not connected, multiple search passes are initiated from unassigned nodes until all nodes have been assigned, or the algorithm has reported that the partition is impossible.

(b) (5 marks) Provide a brief argument for why the algorithm finds a correct solution.

**Answer:**

We maintain the following loop-invariant for each search pass: $A$ will contain no conflicts and will hold part of a valid partition, if one exists.

At the start of each search pass, only one node (the source node) of the current connected component of the graph has been assigned. Since this node is not connected to any other assigned nodes, this assignment cannot create any conflicts. Moreover, since for every valid partition there is a complementary partition in which all assignments for the current connected component are flipped, there must be a valid partition in which the source node is assigned to Group 1, if any valid partition exists. Thus the loop invariant is established.

Now suppose that the loop invariant holds, and we encounter node $j$ while exploring the adjacency list for node $i$. If $j$ has been assigned and $A(j) \neq A(i)$, then the algorithm leaves $A$ unchanged and the loop invariant is maintained. If node $j$ is unassigned, this is our first encounter with node $j$, and therefore node $i$ is the only assigned node adjacent to $j$. Thus we introduce no new conflicts by assigning node $j$ to the group complementary to node $i$.

Now suppose that there is a valid partition $A^*$ consistent with $A$. Since $i$ is an assigned node in this valid solution, $j$ must belong to the complementary group in $A^*$, and thus the assignment $A(j) = 3 - A(i)$ maintains the loop invariant.

Suppose, however, that $j$ is found to already be assigned, and $A(j) = A(i)$. By the loop invariant, $A(i)$ and $A(j)$ must be part of a valid partition, if one exists. But $A(i)$ conflicts with $A(j)$. Thus no valid partition can exist, and the algorithm correctly reports this.

On every iteration, one more edge is processed, thus the algorithm will halt. Further, unless a conflict is detected, every node will be assigned. Thus if the loop terminates without a conflict being detected, $A$ will contain a valid partition, and the people in Group 1 can be recovered in a linear scan through $A$.

(c) (5 marks) Provide tight bounds on best- and worst-case running times.

**Answer:**

The best case occurs when the first source vertex is involved in a 3-clique (a cycle of length 3). For DFS, this could be detected in a constant number of operations, thus the best-case running
time is $\Theta(1)$. For BFS, this will require exploring the entire adjacency list for the source node, plus possibly only the first entry in another adjacency list, for a best-case running time of $\Theta(n)$. The worst case occurs when a compatible partition exists. Then all $n$ adjacency rows are examined, for a running time of $\Theta(n^2)$. 