

**Assignment 4 Cover Page**

Due: Monday, April 5 at midnight

Please read “How to Prepare Assignment Reports” from the course web page.

Please use this page as the cover page for your assignment.

Due to the time constraints, you are required to submit solutions for only 2 of the 4 problems. However, since all 4 problems relate to material that may be on the final exam, you are strongly encouraged to attempt the other 2 problems as well (but please do not submit your solutions). A solution key will be posted on the website shortly after the due date.

Family Name:

Given Name:

Student #:

Email:

Section (M, N or P):

Family Name:

Given Name:

Student #:

Email:

Section (M, N or P):

Please indicate below the two problems you would like marked:

Problem	Marking Scheme	Score
1	20 marks	
2	20 = 10 + 10 marks	
3	20 = 10 + 10 marks	
4	20 = 10 + 10 marks	
Total	40 marks	

**Assignment 4 solutions**

Due: Monday, April 5 at 12:00 noon

Please read “How to Prepare Assignment Reports” from the course web page. You are required to do only 2 of the 4 problems assigned. However, since all 4 problems relate to material that may be on the final exam, you are strongly encouraged to attempt the other 2 problems as well (but please do not submit your solutions). A solution key will be posted on the website shortly after the due date.

[20] 1. **BFS: partitioning**

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 relation 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. **Hint:** Consider a graph in which edges represent non-friendly relations.

- [10] (a) Provide pseudo-code for an algorithm that solves this problem.

**Solution:**

The following algorithm is a modification of BFS. This problem is the same as determining whether a graph is bipartite, or, equivalently, two-colourable.

Expedition(M)  $\triangleright$  *input is a  $n \times n$  matrix  $M$*

$\triangleright$  *start with all groups unassigned*

**for**  $i = 1$  **to**  $n$  **do**

$group(i) \leftarrow 0$

$Q \leftarrow \emptyset$   $\triangleright$  *initialize the queue*

$\triangleright$  *run BFS on each connected component of the graph*

**for**  $i = 1$  **to**  $n$  **do**

$\triangleright$  *start with an arbitrary vertex (e.g., smallest-indexed vertex) in each component*

**if** ( $group(i) = 0$ ) **then**

$group(i) \leftarrow 1$   $\triangleright$  *put this vertex in the first group*

        ENQUEUE(Q,i)

**while**  $Q \neq \emptyset$  **do**

$\triangleright$  *Loop invariant*

$u \leftarrow DEQUEUE(Q)$

**for**  $v = 1$  **to**  $n$  **do**  $\triangleright$  *find all neighbours (enemies) of  $u$*

**if**  $M(u, v) = 0$  **then**  $\triangleright$   *$v$  is a enemy of  $u$*

**if** ( $group(u) = group(v)$ ) **then**  $\triangleright$  *enemies in the same group*

**print** "No partition possible"

**exit**

**else if** ( $group(v) = 0$ ) **then**  $\triangleright$   *$v$  is not assigned yet*

$group(v) = 3 - group(u)$   $\triangleright$  *put  $v$  in the group opposite of  $u$*

                        ENQUEUE(Q, v)

$\triangleright$  *output the answer*

**for**  $i = 1$  **to**  $n$  **do**

**if**  $group(i) = 1$  **then print**  $i$

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

The algorithm operates on a graph in which vertices represent individuals and edges represent enmities. The algorithm conducts an iterative breadth-first-search on all connected components of the graph. For convenience, we will refer to vertices in a component that have been fully processed as *black* vertices, vertices that have not yet been encountered as *white* vertices, and vertices that have been encountered, but whose adjacency rows have not been examined, as *grey* vertices.

In processing a component in the inner while loop, a loop invariant is maintained:

**LI:**

- i.  $Q$  contains exactly those vertices that are grey.
- ii.  $\text{group}(k)=0$  if vertex  $k$  is white, otherwise 1 if it is an even distance from the source vertex  $i$ , 2 if an odd distance.
- iii. All black vertices in Group 1 are compatible and all black vertices in Group 2 are compatible.

When first entering the loop,  $Q$  is initialized to the source vertex  $i$ . Since  $i$  was previously unassigned, it has not previously been encountered, and its adjacency row has not been examined. Thus vertex  $i$  is grey and the first component of the loop invariant is satisfied upon entry. The distance of vertex  $i$  from the source is 0 (even), and it has correctly been assigned to Group 1. All other vertices  $k$  in the component have not been encountered and thus  $\text{group}(k)=0$ . Thus the second component of the loop invariant is also satisfied on entry. Since Group 1 contains only vertex  $i$  and Group 2 is empty, the third component is satisfied trivially.

In one iteration of the inner while loop, the first element  $j$  from the non-empty queue is dequeued and adjacent vertices are extracted. The program halts if any of these adjacent vertices have been assigned to the same group as  $j$ . Otherwise, the subset that had not been encountered (the white subset) are assigned to the other group, so that all vertices adjacent to  $j$  have been assigned to the other group. If  $j$  is an even distance from the source vertex, it will have been assigned to Group 1. The adjacent vertices were either already assigned to Group 2 and therefore must be an odd distance from the source vertex, or were not previously encountered, and thus must be an odd distance from the source vertex. A symmetric argument applies to the case where  $j$  is an odd distance from the source vertex. Thus the second component of the loop invariant is maintained. The queue is augmented with the subset of adjacent vertices not previously encountered,

and whose adjacency rows have thus not been examined (the new grey vertices). Thus the first part of the loop invariant is maintained.

Suppose individuals  $k$  and  $l$  are enemies. Then the graph contains the edge  $(k, l)$ . If both vertices are black, both have been dequeued. Without loss of generality, suppose that vertex  $k$  is the first to be dequeued. If vertex  $l$  had previously been assigned to the same group as vertex  $k$ , the program halts, contradicting the assumption that both vertices are black. Thus vertex  $l$  was either previously assigned to the other group, or is unassigned, in which case it is now assigned to the other group. Thus the third part of the loop invariant is maintained.

On termination (without warning), the queue is empty, which means that all vertices in the component are black, and have been assigned to one of two compatible groups.

The outer loop simply repeats this process over all connected components, augmenting the two groups. Note that since these components are not connected, they do not constrain each other (there are no enmities between them).

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

The best case occurs when a conflict is detected after dequeuing the second vertex (a graph contains a triangle). At this stage, one entire adjacency row has been examined, and as little as one element of a second row has been examined, at a total cost of  $n + 1$ . Other costs are  $O(n)$ . Thus the best-case running time is  $\Theta(n)$ .

The time complexity analysis in the worst case is similar to that of BFS, except that the graph is given in adjacency-matrix representation rather than adjacency-list, so time needed to check all neighbours of a vertex is  $\Theta(n)$ . The worst case occurs when a compatible partition exists. Then all  $n$  adjacency rows are examined, at a cost of  $n^2$ . Other costs are  $O(n^2)$ . Thus the best-case running time is  $\Theta(n^2)$ .

[20] 2. **Fixed edge on a shortest path**

Given a directed, weighted graph  $G$  with non-negative weights, vertices  $s, t$  and an edge  $(u, v)$  in  $G$ :

- [10] (a) Describe (in English or pseudo-code) how to determine whether  $(u, v)$  occurs on *every* path of minimal cost from  $s$  to  $t$ .

**Solution:** 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  $t$ .

No 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 the 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  $O(E \log V)$ .

- [10] (b) Describe (in English or pseudo-code) how to determine whether  $(u, v)$  occurs on *some* path of minimal cost from  $s$  to  $t$ .

**Solution:** Once again, we run Dijkstra's algorithm from  $s$  to compute the weights of the shortest path  $d(s, t)$  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  $d(s, t) < \infty$ , run Dijkstra's algorithm from  $v$  to compute  $d(v, t)$ . 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  $p = s \xrightarrow{\underbrace{\dots}_{p_1}} u \rightarrow v \xrightarrow{\underbrace{\dots}_{p_2}} t$  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  $p = s \xrightarrow{\underbrace{\dots}_{p_1}} u \rightarrow v \xrightarrow{\underbrace{\dots}_{p_2}} t$ .  $p$  obviously contains the edge  $(u, v)$ , and  $w(p) = w(p_1) + w(p_1) + w(u, v) + w(p_2) = d(s, u) + w(u, v) + d(v, t) = d(s, t)$ , 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  $O(E \log V)$ .

[20] 3. **Descending Kruskal's algorithm**

Another approach to the minimum spanning tree algorithms is to remove "heavy" edges from the graph until only the tree is left, rather than adding "light" edges to an originally empty graph.

- [10] (a) Design a version of Kruskal's algorithm that would exploit this idea. Your algorithm should work by removing heavy edges from the original graph until only an MST is left. Provide pseudo-code for your algorithm.

**Solution:**

Sort the edges  $e_1, \dots, e_m$  in *non-increasing* order of weights

Set  $S \leftarrow G$

**for**  $i = 1$  **to**  $m$  **do**

▷ *To check connectivity: let*  $e_i = (u, v)$ ,

▷ *run BFS starting from*  $u$  *and see if*  $v$  *is reachable from*  $v$

**if**  $S - \{e_i\}$  **is connected** **then**

$S \leftarrow S - \{e_i\}$

**end if**

**end for**

**return**  $S$

- [10] (b) Prove that your algorithm always returns an MST of  $G$ . The steps of your proof will be similar to that of Kruskal's algorithm.

**Solution:**

Let  $S_i$  be the value of  $S$  after  $i$  iteration of the "for loop"; that is after examining edges  $e_1, \dots, e_i$ . So  $S_0$  is the initial value of  $S$ , which is  $G$ . We prove that the following

predicate  $LI(i)$  holds for all values of  $i$ ,  $0 \leq i \leq m$ :

**Loop invariant**  $LI(i)$ : There exists a minimum spanning tree  $S_{opt}$  of  $G$  such that  $S_i \cap \{e_1, \dots, e_i\} \subseteq S_{opt} \subseteq S_i$ . That is, there exists an optimal solution  $S_{opt}$  that can be obtained by removing from  $S_i$  some subset of edges from among  $\{e_{i+1}, \dots, e_m\}$ .

It is easy to see that once we prove this predicate for all values of  $i$  we are done, because:  $LI(m)$  implies that for some minimum spanning tree  $T_{opt}$ :  $S_m \cap \{e_1, \dots, e_m\} \subseteq T_{opt} \subseteq S_m$ . Since  $\{e_1, \dots, e_m\}$  is the set of all the edges, therefore,  $S_m \cap \{e_1, \dots, e_m\} = S_m$ . This, shows that  $S_m \subseteq T_{opt} \subseteq S_m$ , i.e.  $S_m = T_{opt}$ .

Now we prove the loop invariant by induction on  $i$ .

**Base case.**  $LI(0)$  holds since  $S_0 = G$  and since there exists an optimal (minimum spanning) tree  $T_{opt}$  for any connected undirected graph. Moreover, any such  $T_{opt}$  is a subset of  $G$  and  $S_0 \cap \{\} = \emptyset \subseteq T_{opt}$ .

**Induction step.** Let  $0 \leq i < m$  and assume that  $LI(i)$  holds. We want to prove that  $LI(i+1)$  holds. Since  $LI(i)$  holds there is an optimal tree  $T_{opt}$ , s.t.  $S_i \cap \{e_1, \dots, e_i\} \subseteq T_{opt} \subseteq S_i$ .

**Case 1.** If  $e_{i+1}$  is not removed, i.e.  $S_{i+1} = S_i$ , then  $S_i - e_{i+1}$  would be disconnected. Since  $T_{opt} \subseteq S_i$  and  $T_{opt}$  is connected,  $e_{i+1} \in T_{opt}$ . So  $T_{opt} \subseteq S_i = S_{i+1}$ , and since  $e_{i+1} \in T_{opt}$ :  $S_{i+1} \cap \{e_1, \dots, e_{i+1}\} = S_i \cap \{e_1, \dots, e_{i+1}\} \subseteq T_{opt}$ .

**Case 2.** Now consider the case that  $e_{i+1}$  is removed from  $S_i$ , i.e.  $S_{i+1} = S_i - e_{i+1}$ . Note that  $S_{i+1} \cap \{e_1, \dots, e_{i+1}\} = S_i \cap \{e_1, \dots, e_i\} \subseteq T_{opt}$ .

**Case 2a.** If  $e_{i+1} \notin T_{opt}$  then  $T_{opt} \subseteq S_{i+1}$  and we are done.

**Case 2b.** Now assume that  $e_{i+1} \in T_{opt}$ . Consider a cut between two connected components of  $T_{opt} - e_{i+1}$ . Since  $S_{i+1}$  is connected, it contains some edges crossing that cut. Take the edge  $e_j$  such that  $j$  is the largest index of all edges in  $S_{i+1}$  crossing that cut. We claim that  $T''_{opt} = T_{opt} - e_{i+1} + e_j$  is a minimal spanning tree. First,  $T''_{opt}$  is connected because the two connected components of  $T_{opt} - e_{i+1}$  are joined by the edge  $e_j$  (by the choice of  $e_j$ ). Since  $T''_{opt}$  is a connected graph on all vertices of  $G$ , and  $T''_{opt}$  has  $n - 1$  edges, it is a tree. So  $T''_{opt}$  is a spanning tree of  $G$ . It remains to show that it has minimal weight. Since  $e_j$  comes after  $e_{i+1}$  in the sorted by weight order of edges,  $c(e_j) \leq c(e_{i+1})$ . So  $c(T_{opt}) \leq c(T''_{opt})$ . But since  $T_{opt}$  had minimal weight, this is an equality: the weight of  $T''_{opt}$  must be equal to the weight of  $T_{opt}$ . Therefore, both  $e_{i+1}$  and  $e_j$  are light edges

for the cut described above, and both are safe for growing a minimal spanning tree. So  $T'_{opt} \subseteq S_{i+1}$  since  $T'_{opt}$  does not have  $e_{i+1}$  in it, and  $S_{i+1} \cap \{e_1, \dots, e_{i+1}\} \subseteq T'_{opt}$ . Thus,  $LI(i+1)$  holds.

[20] 4. **Greenhouses and plants**

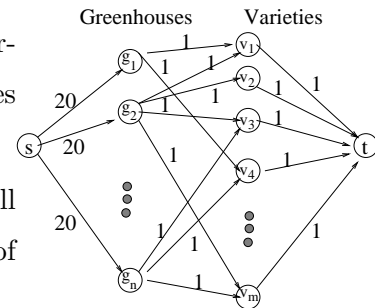
A botanical garden is planning to build  $n$  greenhouses, each of which is intended to represent a different climate zone. They have a list of  $m$  kinds of plants that they would like to grow there. Not every kind of plant can grow in every greenhouse, though usually one kind can grow in more than one climate; so associated with every kind of plants  $i$  is a list  $C_i$  of greenhouses where it can grow. In addition to that, every greenhouse can host no more than 20 varieties of plants.

- [10] (a) Given the  $n, m$  and  $C_i$  for  $1 \leq i \leq n$ , specify an algorithm to determine the maximal number of different varieties of plants that can be grown in these greenhouses (total over all greenhouses). Hint: design a flow network that represents this problem.

**Solutions**

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

- The vertices of  $G$  will be  $s, t$  for source and sink, vertices  $g_1, \dots, g_n$  representing greenhouses and vertices  $v_1, \dots, v_m$  representing varieties of plants.
- Edges  $(s, g_i)$  of capacity 20 for  $1 \leq i \leq n$ . This will force 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 are interested in the maximal number of varieties of plants in each greenhouse, 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  $1 \leq j \leq m$ . Because we want to know the maximal number of varieties over all greenhouses, we are counting each variety once; this is



why the capacity on this edge is 1.

To find the maximal number of varieties of plants that can be grown in these greenhouses, we run Ford-Fulkerson algorithm on this network. The value of the resulting flow is the maximal number of varieties that can be grown. Indeed, 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 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.

- [10] (b) How can you produce a list of chosen varieties for each greenhouse given the maximum flow on your flow network? Is it uniquely determined by the original network?

**Solution:** Take the resulting flow on the network, and consider a cut between  $(s, g_1, \dots, g_n)$  and  $(v_1, \dots, v_m, t)$ . Since every cut has the same flow through it, and by 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.