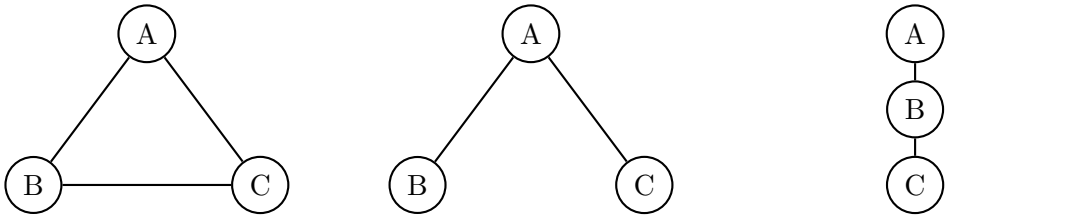
Q1

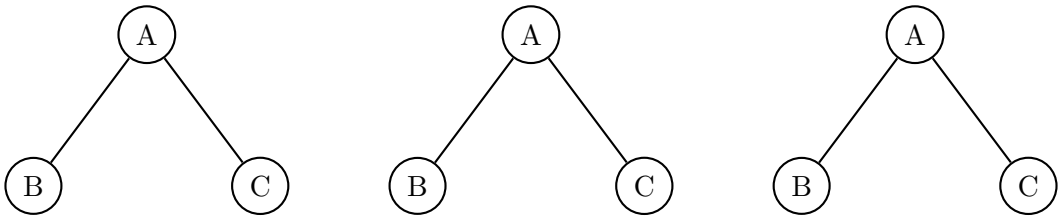
A tree by definition has $|V| - 1$ edges. A cycle cannot exist in G if both DFS and BFS produce a tree T, because the presence of a cycle would violate the above $|V| - 1$ edges rule.

Because G has no cycles, G is a tree. Thus, G=T. See Figures 1 and 2 for an example.



(a) A graph G. (b) The tree given by BFS. (c) The tree given by DFS.

Figure 1: Example of the trees resulting from BFS and DFS. Notice that for a cyclic graph, the tree are different.



(a) A graph G. (b) The tree given by BFS. (c) The tree given by DFS.

Figure 2: Example of the trees resulting from BFS and DFS. Notice that for an acyclic graph, the tree are identical. This example holds for more complex graphs - the only difference is the order in which the tree is produced. In any case, the resulting tree T is the same and is equivalent to G.

Q2

(a) This question is an example of a case where we want to identify max bipartite matching. We set up the graph in the manner shown in Figure 3.

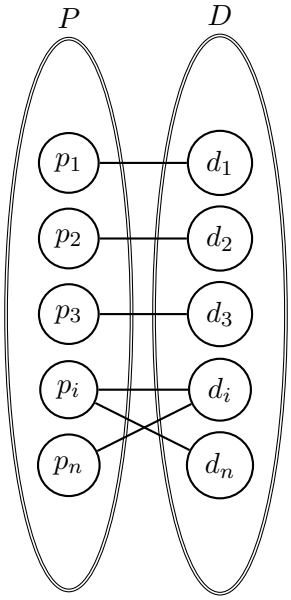


Figure 3: An example of a bipartite graph for this problem.

Now, we can solve this as a maximum flow problem using the Ford-Fulkerson method. See Figure 4.

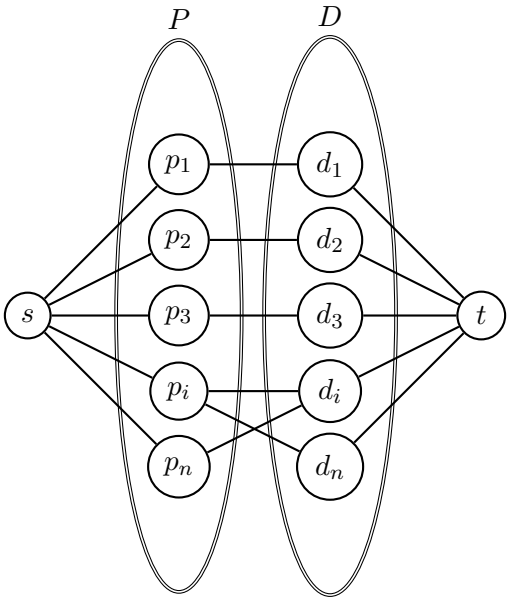


Figure 4: The graph G modified to be solved via the Ford-Fulkerson method. Each edge has a capacity of one.

If there is a perfect dinner schedule, then there will be perfect and maximum matching. That is, the max flow will be of magnitude n. See Figure 5.

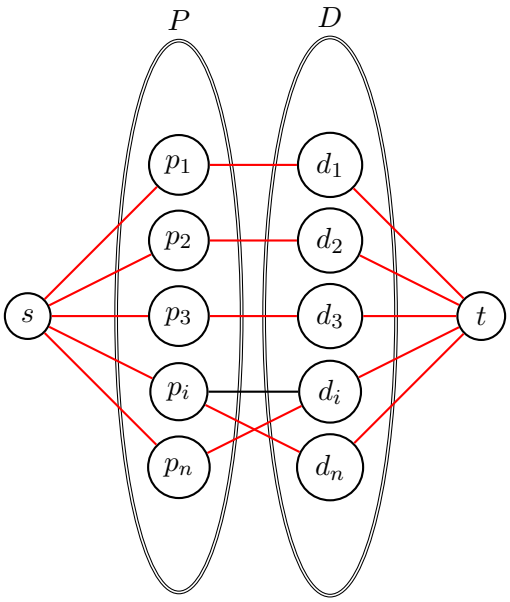
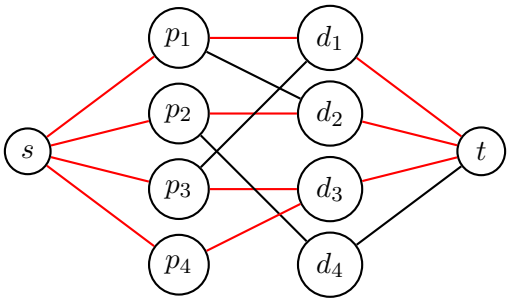
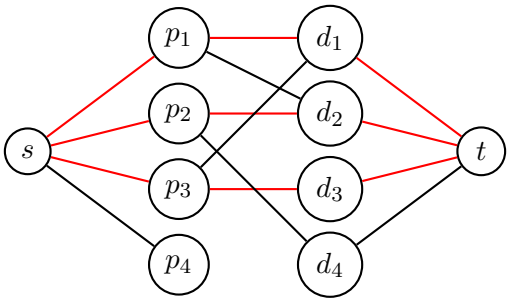


Figure 5: Perfect matching of the bipartite graph, and a perfect dinner schedule!

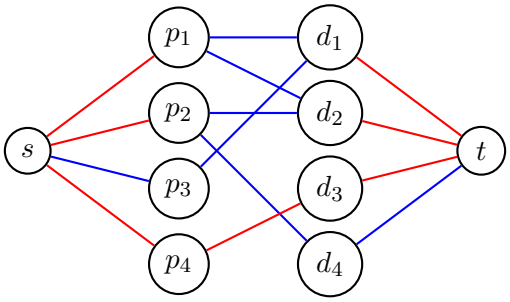
(b) We want to see if a perfect matching is possible. With Alanis's schedule, if a perfect matching is possible, then at least one of the edges (p_i, d_k) and (p_j, d_k) that are being used are not needed. Thus, we can do the following. We can evaluate two cases - one with (p_i, d_k) removed and one with (p_j, d_k) removed - and see if an augmenting path can be found in either case. If an augmenting path can be found from either case, then a perfect matching is possible and is produced in $O(n^2)$ time. The complexity is $O(n^2)$ because the number of people and days are equal, and there can be at most n edges for each of n people, resulting in n^2 possible edges to evaluate. See Figure 6 for an example of this.



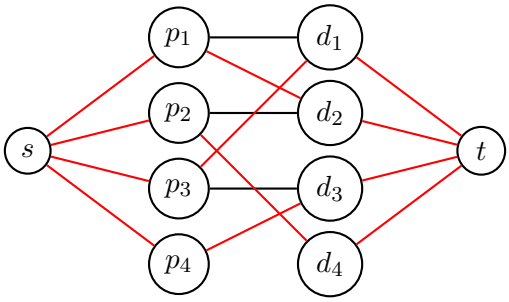
(a) The schedule proposed by Alanis.



(b) Case 1 - Remove edge (p_4, d_3). No augmenting path exists.



(c) Case 2 - Remove edge (p_3, d_3). An augmenting path does exist (shown in blue).



(d) The resulting perfect schedule!

Figure 6: Example of correcting an "almost perfect" schedule.

Q3

Here I present a modified BFS algorithm to determine whether or not a virus introduced to computer C_a at time x could have infected computer C_b by time y.

```
ModifiedBFS(G, Ca, x, Cb, y){
   // Set all nodes beside root to default settings
   for (u in G.V - {s}){
      u.color = white;
      u.parent = null;
      u.time = null;
   }

   // Set root node settings
   Ca.color = red;
   Ca.parent = new ArrayList();
   Ca.time = x;

   // Create queue and add Ca
   Q = 0;
   Enqueue(Q,Ca);

   // Primary loop
   while (Q != 0){
      u = Dequeue(Q);
      for (v in G.Adj[u] - {u.parent}){
         if (v.color == white){
            if (w(u,v) >= u.time){
               v.color = red;
               v.parent = append(u);
               v.time = w(u,v);
               Enqueue(Q,v);
            }
         }
         if (v.color == red){
            if (w(u,v) >= u.time && w(u,v) < v.time){
               v.parent = append(u);
               v.time = w(u,v);
               Enqueue(Q,v);
            }
         }

   // Check if $C_b$ is infected at time $y$
   if (Cb.color == red && Cb.time <= y){
      return true;
   }
   else{
      return false;
   }
}
```

This is a modified BFS algorithm, in which infected nodes are set to color=red and noninfected

nodes are color=white. Each node has an associated time of infection and an ArrayList of "parents", which keeps track of nodes which it could have been infected from.

If a node is white, and if the edge weight is $\geq$ the infection time of the parent, then it will be set as follows: color=red, parent=append(u), time=w(u,v). Then it will be added to the queue. Only infected nodes are added to the queue.

If a node is red, and if the edge weight is $\geq$ the infection time of the parent *and* the edge weight is $<$ its current infection time, then it will be updated as follows: parent=append(u), time=w(u,v). Then it will be added to the queue. See Figure 7 for an example.

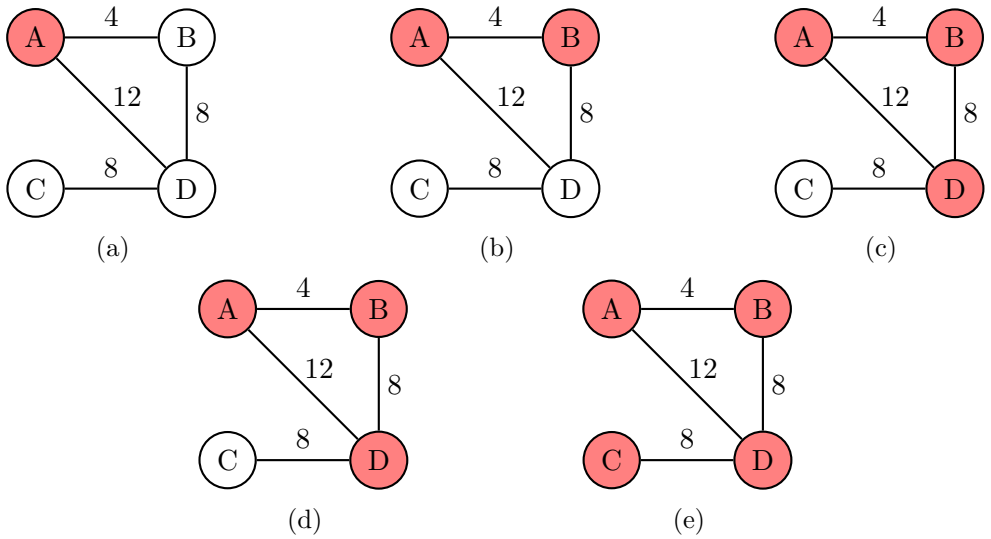


Figure 7: An example of the Modified BFS algorithm. In (a), the initial conditions are presented. Time of infection of Node A is 2. In (b), Node B is examined. Because the edge weight is greater than 2, Node B is infected at time 4. In (c), Node D is examined. Because the edge weight is greater than 2, Node D is infected at time 12. In (d), we move on to Node B as the "root" node. Because Node A is a parent of Node B, it is not examined, and only Node D is examined. Because the weight of the edge is greater than 4, but less than 12, the time of infection for Node D is updated to the weight of the edge BD, or 8. In (e), we move on to Node D as the "root" node. Because both Node A and Node B are retained as parents, neither is examined. Only Node C is examined. Because the edge weight is greater or equal to 8, Node C is infected at time 8. At this point, all edges have been examined once (m times), and the function returns true or false based on input.

We can show that this algorithm runs in $O(m)$ time because each triple represents an edge, and there are m triples. We are examining only edges that have not been examined before, so at most m edges. Figure 7d highlights this point. The cost of the queue operations and the setting of initial conditions operates only on the nodes, and there will be at most $|E| + 1$ nodes.

Q4

(a) This is similar to Problem 2 of this assignment. The best approach here is two set up two disjoint sets X and S, a source node s and a sink node t. There can be other nodes outside of these presented, but there can only be one output edge from each node in X. All edges except those from nodes in S to t must have a capacity of one. Edges from nodes in S to t have a capacity of $|X|$. Construct a graph G as shown in Figure 8.

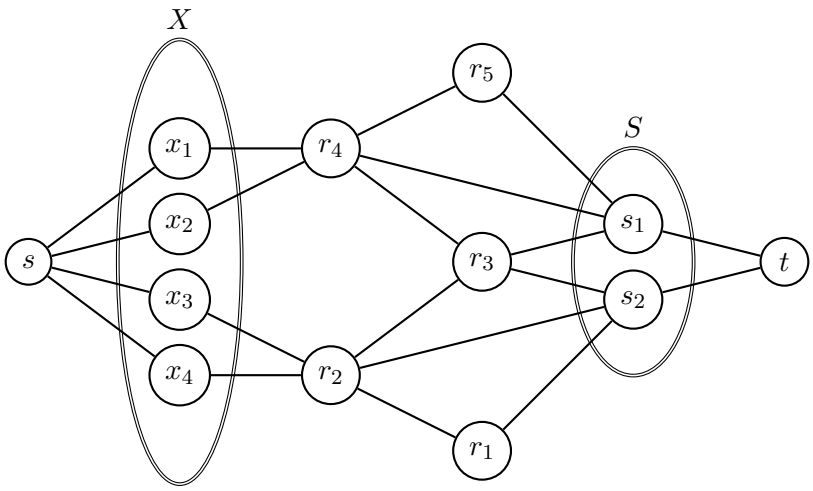


Figure 8: An example of a graph for this problem.

With the graph constructed, solve the Max Flow problem using the Ford-Fulkerson method. If the maximum flow is equal to $|X|$, then a set of routes exists.

(b) While Part A required enforcement of "edge-disjoint" rules, this part requires enforcement of "node-disjoint" rules. In Figure 8, a set of routes exists when following edge-disjoint rules, but not when following node-disjoint rules. This is because the four nodes of X converge two only two nodes r_2 and r_4. Thus only two cities would be able to escape if using node-disjoint rules.

To enforce this rule, we must modify our graph G. We could create a graph G' in which each of the "in-between" nodes r_i are duplicated. One of these duplicate nodes serves as an input, one serves as an output, and an edge of capacity one connects them. It is important that the connecting edge be of capacity one, as this enforces the node-disjoint rule by prohibiting flow from more than one source through the node. This modified graph G' can be solved via the same Ford-Fulkerson method. If the max flow is equal to $|X|$, then a set of routes exists.