CPSC 320 2023S2: Assignment 2 Solutions

3 Transitive Closure of Relations (5 marks)

This problem and the next aim to illustrate how the graph algorithms you've learned about in class and in Chapter 3 of the text are useful in solving other practical problems that at first may seem unrelated.

Let $S = \{s_1, s_2, \ldots s_k\}$ and let R be a relation on S. Assume that R is reflexive, that is, sRs for all $s \in S$, and also that R is symmetric, that is, if sRs' then $s'Rs$. Let matrix $M_R[1..k][1..k]$ represent R, that is, $M_R[i][j] = 1$ if $s_i R s_j$ and $M_R[i][j] = 0$ otherwise.

Give an algorithm that takes M_R as input and outputs the matrix M_{R^*} representing the transitive closure R^* of R. The transitive closure R^* of R is the smallest relation on S that that contains R (that is, if sRs' then sR^*s') and also is transitive (that is, if sR^*s' and $s'R^*s''$, then sR^*s''). Full marks if your algorithm runs in $O(k^2)$ time; a slower algorithm gets partial marks. (Hint: an algorithm for computing connected components is useful.)

SOLUTION: We associate an undirected graph G_R with relation R as follows. The set of nodes is the set S, and there is an edge between two nodes s and s' if and only if sRS' (and also $s'Rs$). Then it is the case that sR^*s' if and only if s and s' are in the same connected component of G_R. To see why, note that sR^*s' if and only if for some $l \geq 0$, there is a sequence $s_0 = s, s_1, \ldots, s_l = s'$ of distinct nodes such that $s_0 R s_1$, $s_1 R s_2$, and so on up to $s_{l-1} R s_l$, in which case $s = s_0$ and $s' = s_l$ are connected in G_R.

As shown in the Kleinberg and Tardos text (page 94), we can find all connected components of an undirected graph in $O(n + m)$ time, given an adjacency list representation of graph G, where n and m are the number of nodes and edges of G, respectively. We can construct an adjacency list representation of G_R from matrix M_R in $O(k^2)$ time; also, G_R has k nodes and $O(k^2)$ edges. The nodes in each connected component can be stored in a list, and the set of connected components can be represented as a list of lists.

From the lists of connected components, we can construct the matrix M_{R^*} as follows. First initialize all entries of M_{R^*} to 0. Then for each list of connected components, for each pair (i, j) of nodes in the list, set $M_{R^*}[i][j] = 1$. This takes $O(k^2)$ time in total, since every pair $(i, j), 1 \leq i, j \leq k$ is examined at most once and there are k^2 such pairs.

4 Hamming Labelings of Graphs

Let $G = (V, E)$ be an undirected, unweighted, connected graph with n nodes and m edges. Assume that G has no self-loops or multiple edges. We want to label the nodes in V with equi-length binary strings so that the length (i.e., number of edges) of the shortest path between any pair of nodes equals the Hamming distance between the corresponding node labels. We call such a labeling a *Hamming labeling*.

Here, the Hamming distance between two length-k binary strings $x_1 x_2 \ldots x_k$ and $y_1 y_2 \ldots y_k$ is the number of positions $i, 1 \leq i \leq k$ where bits x_i and y_i differ. Given a Hamming labeling of a graph, we denote the label of node v by $l(v)$. Also, the length of a given path between two nodes is the number of edges along the path.

In this problem you'll develop an efficient algorithm that, given G, determines whether or not G has a Hamming labeling.

Although not relevant for the rest of the problem, Hamming labelings are useful in many contexts. For example, suppose that nodes represent servers in a network, and edges represent links between nodes. Suppose furthermore that each node "knows" its label and that of its neighbours, but does not necessarily know anything more about the complete structure of the network. Then to route a message from node s to a node $t \neq s$, node s simply sends the message to a neighbour, say s', for which

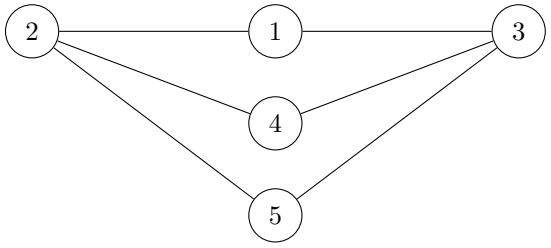
$$H(l(s), l(t)) = 1 + H(l(s'), l(t)).$$

The message is routed in a similar manner from s' and so on, until it reaches t. In this way, the message reaches its destination along a shortest path with the minimum number of "hops".

1. (2 marks) Give a Hamming labeling for the 6-cycle, namely the graph with node set $V = \{1, 2, 3, 4, 5, 6\}$ and edge set $E = \{(1, 2), (2, 3), (3, 4), (4, 5), (5, 6), (6, 1)\}$. Your labeling should use three bits. The label $l(1)$ is provided.

   SOLUTION:
   $l(1)$: 000
   $l(2)$: 001
   $l(3)$: 011
   $l(4)$: 111
   $l(5)$: 110
   $l(6)$: 100

2. (3 marks) A problem from Tutorial 4 shows that all graphs with Hamming labelings are bipartite. However, not all bipartite graphs have Hamming labelings. Explain why the following bipartite graph has no Hamming labeling.



   SOLUTION: Without loss of generality, suppose that the first two bits of $l(2)$ are 00 and the first two bits of $l(3)$ are 11 (and $l(2)$ and $l(3)$ agree on all other bits since they must have Hamming distance 2 from each other). Then it must be that one of the labels of $l(1)$ and $l(4)$ starts with 01 and the other label starts with 10, and these labels agree with those of $l(2)$ and $l(3)$ on the remaining bits; any other choice would violate the condition that Hamming distance between each of the label pairs $(l(1), l(2))$, $(l(2), l(4))$, $(l(1), l(3))$, $(l(3), l(4))$, equals 1, while $H(l(1), l(4)) = 2$.

   Consider all of the possible choices for the first two bits of $l(5)$:

   - $l(5) = 00\ldots$. Then $H(l(3), l(5)) \geq 2$, contradicting the fact that $d(3, 5) = 1$.
   - $l(5) = 11\ldots$. Then $H(l(2), l(5)) \geq 2$, contradicting the fact that $d(2, 5) = 1$.
   - $l(5) = 01\ldots$. Then $l(5)$ must agree with $l(2)$ and $l(3)$ on all other bits, so that the Hamming distance between $l(5)$ and $l(2)$ agrees with the length of the shortest path from node 2 to node 5, both being 1. Note that also, the Hamming distance between $l(5)$ and $l(3)$ agrees with the distance from node 3 to node 5, both being 1, which is good. However, the problem is that the label of node 5 must now equal the label of either nodes 1 or 4, and again we get a contradiction since all nodes must have distinct labels.

- $l(5) = 10\ldots$. Now, $l(5)$ must agree with $l(2)$ and $l(3)$ on all other bits. This leads to a contradiction in a manner similar to the previous case.

3. (5 marks) Suppose that $G = (V, E)$ is bipartite (as well as being undirected, unweighted, and connected), and let $d(i, j)$ denote the distance, i.e., length of the shortest path, from node i to node j in G. Let $(x, y) \in E$. Let $N_{xy} \subset V$ be the set nodes that are closer to x than y, that is, node i is in N_{xy} if and only $d(i, x) < d(i, y)$. Similarly, let N_{yx} be the nodes of V that are closer to y than x. No node can be equidistant to x and y because otherwise the graph would have an odd-length cycle, contradicting the fact that it is bipartite. So the pair N_{xy} and N_{yx} form a partition of the nodes of the graph.

   Describe an algorithm that takes as input G and an edge (x, y), and finds the sets N_{xy} and N_{yx}. For full marks, your algorithm should run in $O(n + m)$ time. You can represent the set N_{xy} as an array $N_{xy}[1..n]$ where $N_{xy}[i] = 1$ if node i is closer to x than y, and is $N_{xy}[i] = 0$ otherwise, and use a similar representation for N_{yx}. Explain what is the running time of your algorithm, and why your algorithm is correct.

   SOLUTION: Here's an algorithm to compute the arrays.

   (a) Set all entries of $N_{xy}[1..n]$ to 0, with the exception of entry $N_{xy}[x]$ which is initialized to 1. Similarly set all entries of $N_{yx}[1..n]$ to 0, with the exception of entry $N_{yx}[y]$ which is initialized to 1.

   (b) Run BFS from x to obtain a BFS tree T_x rooted at x, and also run BFS from y to obtain BFS tree T_y rooted at y.

   (c) For $i = 1, 2, \ldots$ up to the minimum of the depth of T_x and T_y:
   - Consider each node j at level i of T_x. If $N_{yx}[j] = 0$, set $N_{xy}[j] = 1$.
   - Consider each node j at level i of T_y. If $N_{xy}[j] = 0$, set $N_{xy}[j] = 1$.

   Step (a) of the algorithm runs in $O(n)$ time, step (b) in $O(n+m)$ time (the time for BFS), and step (c) can be implemented to run in $O(n)$ time since each node of each BFS tree is considered once, and at most two accesses to the arrays are made per considered node.

   To show correctness, we claim that at the end of each iteration i of the For loop, $N_{xy}[j] = 1$ if and only if $d(j, x) \leq i$ and $d(j, x) < d(j, y)$. The base case is that the claim is true at the end of iteration 0, i.e., before the For loop is executed, because the only 1-valued entry initially in $N_{xy}[1..n]$ is $N_{xy}[x]$ and $d(x, x) = 0 < d(x, y)$, and similarly the only 1-valued entry initially in $N_{yx}[1..n]$ is $N_{yx}[y]$ and $d(y, y) = 0 < d(y, x)$.

   Now consider iteration $i \geq 1$, and assume inductively that the claim holds at the end of iteration $i - 1$. We will show that $N_{xy}[j]$ is set to 1 on iteration i if and only if $i = d(j, x) < d(j, y)$, and this, together with the inductive hypothesis, shows the claim holds at the end of iteration i.

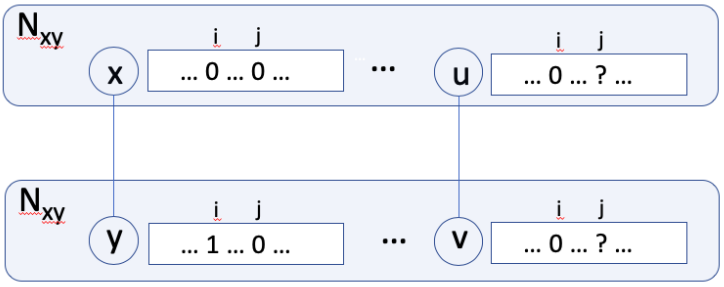
   On iteration i, $N_{xy}[j]$ is set to 1 only if node j is at level i of BFS tree T_x (and thus the distance from j to x is i) and moreover, $N_{yx}[j] = 0$, meaning that j cannot be within distance $i - 1$ of y. Also j cannot be of distance exactly i from y, since then it would be equidistant from both x and y, contradicting the fact that G is bipartite. So $N_{xy}[j]$ is set to 1 on iteration i if and only if $i = d(j, x) < d(j, y)$, and we are done.

   Similarly, at the end of iteration $i, i = 0, 1, \ldots$ of the For loop, $N_{yx}[j] = 1$ if and only if $d(j, y) \leq i$ and $d(j, y) < d(j, x)$. When the algorithm ends, nodes at all levels of the trees T_x and T_y have been examined, and so the arrays N_{xy} and N_{yx} are properly computed.

4. (4 marks) Prove the following claim.

   Claim: Suppose that graph G has a Hamming labeling $l()$. Let (x, y) be an edge of G and suppose that the labels $l(x)$ and $l(y)$ differ in the ith coordinate (note that these labels can only differ in one coordinate). Suppose furthermore that edge (u, v) crosses the (N_{xy}, N_{yx}) partition, so that v is closer to y than to x. Then $l(u)$ and $l(v)$ must also differ in the ith coordinate.

SOLUTION: Suppose to the contrary that $l(u)$ and $l(v)$ differ at a coordinate $j \neq i$. Also without loss of generality, suppose that $l(x)$ is 0 at both bits i and j, that $l(y)$ is 1 at bit i and 0 at bit j, and that $l(u)$ and $l(v)$ are both 0 at bit i, as illustrated in the following figure (labels are shown in white boxes beside each node):



Consider the two possibilities for the values of bit j of $l(u)$ and $l(v)$.

- If $l(u) = 0$ then $l(v) = 1$ and so $l(v)$ differs from $l(y)$ at both bits i and j. So
$$H(l(y), l(v)) = H(l(x), l(v)) + 1,$$
contradicting the fact that v is closer to y than to x.
- Otherwise, $l(u) = 1$. Then $l(v) = 0$ and so agrees with $l(x)$ at both bits i and j. Again we have that
$$H(l(y), l(v)) = H(l(x), l(y)) + 1$$
contradicting the fact that v is closer to y than to x.

5. (3 marks) Again, let $G = (V, E)$ be a bipartite, connected graph with n nodes and m edges. For edges $e = (x, y)$ and $e' = (u.v)$ of E, we define the relation $(x, y)R(u, v)$ if (u, v) crosses the (N_{xy}, N_{yx}) partition. (It is the case that if eRe', then $e'Re$. You don't need to prove this, but it doesn't hurt to make sure you understand why.)

   Let matrix $M_R[1..m][1..m]$ be defined as $M_R[e, e'] = 1$ if eRe', and $M_R[e, e'] = 0$ otherwise. Show how to build the matrix M_R in $O(m^2)$ time.

   SOLUTION: First, for each edge (x, y), build the partition (N_{xy}, N_{yx}). This takes $O(n + m)$ time per edge, and $O(m^2)$ time in total over all edges. Then iterate through all pairs of edges e and e'. Use the partitions to determine in $O(1)$ time whether eRe', and fill in the matrix accordingly.

6. (3 marks) We're ready for the punchline! Here's a neat result that you don't need to prove.

   Fact: An unweighted, connected, undirected graph G has a Hamming labeling if and only if G is bipartite and furthermore the relation R is transitive, that is, $R = R^*$.

   Using this fact, give an algorithm that takes as input an unweighted, connected, undirected graph G and determines if G has a Hamming labeling. (As always, briefly explain the running time of your algorithm, and why it is correct.)

   Your algorithm should run in $O(m^2)$ time, where m is the number of edges of G. In addition to using any algorithms from the textbook or class, you can assume that you have an algorithm with $O(m^2)$ running time that takes as input the matrix M_R defined in part 6.5, and computes M_{R^*}.

   SOLUTION: To test whether undirected, connected graph G has a Hamming labeling, first, check that G is bipartite. Section 3.4 (page 94) of the text shows that this can be done in $O(n + m)$ time. If G is not bipartite, output "No". Otherwise, use the algorithm from part 6.5

to construct the matrix M_R, and the transitive closure algorithm from problem 5 to compute M_{R^*}. If these two matrices are identical, output "Yes", and otherwise output "No".

Note: If the graph G has 0 or 1 nodes then relation R is empty and so trivially $R = R^*$.

Correctness of this algorithm follows directly from the given fact. Assume that the graph G has n nodes and m edges. The running time to check if G is bipartite is $O(n + m) = O(m^2)$. The previous part of this problem shows how to construct M_R in $O(m^2)$ time, and the previous problem shows how to construct M_{R^*} in $O(m^2)$ time. These two matrices can be compared in $O(m^2)$ time since they are $m \times m$ matrices.

5 Colouring within the lines

Suppose you are at the dentists' office waiting for your appointment, and to your dismay your phone has run out of battery! With nothing to keep your hands and mind busy, you notice that there is a children's table with a pile of colouring sheets and pencil crayons, so you decide to pass your time by doing some colouring. You find a blank colouring sheet with a map on it, and like any coloured map, you want a "proper" colouring, such that no two adjacent countries have the same colouring.

There are not enough pencil crayons to colour each country a different colour, so as a CPSC 320 student, your first instinct is to approach this as a graph problem.

1. (3 marks) Let degree d be the maximum number neighbouring countries of any country on the map. For any arbitrary value of d, describe a type of graph (or map) with maximum degree d that can be properly coloured with only two colours.

   SOLUTION: Let each country be a vertex, and an edge exists between all adjacent countries.

   A "star" graph. This type of graph has one central vertex connected by an edge to every other vertex, and there no other edges between the other vertices. (Imagine that there is ocean separating countries that are not the central vertex.) Therefore, a star graph with n vertices has one vertex with degree $n - 1$ (the central vertex) and $n - 1$ vertices with degree 1. The maximum degree in this graph is $d = n - 1$. We can colour the central vertex one colour, and then another colour for all the other vertices.

   Any bipartite graph with maximum degree d is another solution. By definition, a bipartite graph can be coloured using two colours such that no adjacent nodes have the same colour. We can colour the vertices on the right one colour, and then another colour for the vertices on the right.

2. (4 marks) Design a greedy algorithm that takes as input a map where each country has at most d adjacent countries, and colours the countries with at most $d + 1$ distinct colours.

   SOLUTION: Let the countries be vertices on a graph, with edges between all pairs of adjacent countries. Label the colours $1, 2, ..., d + 1$. Order the vertices arbitrarily. Assign colour 1 to the first vertex. Then, choose any next vertex v and assign it the lowest-numbered colour that has not already been assigned to any of its neighbours. If all previously-used colours appear on vertices adjacent to v, then assign v new colour that has not been used before. Repeat until all vertices are coloured. If the order of the vertices is given by array Order[1..n], then the algorithm is as follows:

```
procedure GREEDY-COLOUR(G, Order[1..n])
    ▷ colour the nodes in the order given by Order[1..n] using a greedy approach
    colour node Order[1] with colour 1.
    C ← 1
    while not all nodes are coloured do
        move on to the next node, say v
        if not all colours between 1 and C are used on the nodes adjacent to v then
            choose the lowest such colour for v
        else
            use colour C + 1 for v, and add 1 to C
```

3. (2 marks) Briefly explain why your algorithm will use at most $d + 1$ colours.

   SOLUTION: If we have $d + 1$ colours, we are guaranteed to always have a colour available for every new vertex v, since v has at most d neighbours.

4. (2 marks) Give an example of a graph where your algorithm will use all $d+1$ colours, *no matter what ordering* of the vertices is used. (Don't concern yourself with the issue of whether the graph could be laid out as a map.)

   SOLUTION: Our example graph has $d + 1$ nodes. $d - 1$ of these nodes form a *clique*, that is, there is an edge between every pair of these nodes. So they all must receive different colours. The remaining two nodes, say a and b, each have an edge to all $d - 1$ nodes in the clique, and to each other. So each node has degree d. Also a and b must get a colour that is not used for nodes in the clique, and must get different colours. So the total number of colours must be $d + 1$.

5. (4 marks) Now, assume that there is at least one country with degree less than d. Assume also that the countries are all in the same land mass, i.e., it's possible to travel from any one country to any other by land (via zero or more intermediate countries). Design an algorithm that will colour the countries with at most d colours. (Hint: the order in which nodes are coloured is important!)

   SOLUTION: We want to order vertices in such a way that every vertex except the last has at most $d - 1$ neighbours among the preceding, already-coloured vertices, and at least one "forward" edge to a vertex later in the ordering. We'll put the vertex, say v, that has degree less than d at the end of the ordering. To find such an ordering, we can build a tree, say a BFS tree, rooted at v and then order the vertices by level, starting at the leaves and working up to the root.

   Then colour the vertices using the greedy algorithm from part 2. Since each vertex has at most $d - 1$ vertices earlier in the ordering, the algorithm uses at most d colours.