CS 6150: Homework 4

Christopher Mertin

Due Date: November 20, 2015

This assignment has 5 questions, for a total of 100 points. Unless otherwise specified, complete and reasoned arguments will be expected for all answers.

Question	Points	Score
Cycle Cover	20	
Attend all lectures	20	
Monotone path	20	
Vertex-disjoint paths	20	
Uniqueness of mincut	20	
Total:	100	

Question 1: Cycle Cover ..... **[20]**

A cycle cover C of a directed graph $G(V, E)$ is a collection of vertex-disjoint directed cycles so that each vertex in V belongs to some cycle in C.

Give a polynomial time algorithm to compute a cycle cover of a given directed graph, or correctly report that one does not exist.

Solution: There exists a cycle cover if there exists a perfect matching for the graph. If a perfect matching does not exist, then there is no disjoint cycle cover. Therefore, finding the cycle cover can be reduced into the perfect matching problem.

From the original graph G, we can create a bipartite graph G'. To do this, take the original set of vertices and have them be in their own set $\mathcal{L} = \{u_1, u_2, \ldots, u_n\}$ to be the left side of the bipartite graph, and create a *copy* of each vertex and have it be in the set $\mathcal{R} = \{u'_1, u'_2, \ldots, u'_n\}$ to be the right side of the bipartite graph. There is a source vertex s that has an edge $(s, u_\mu)\ \forall\ \mu = \{1, 2, \ldots, n\}$ and a target vertex t that has an edge $\left(u'_\gamma, t\right)\ \forall\ \gamma = \{1, 2, \ldots, n\}$. This would take $\mathcal{O}(2VE)$ to build.

From here, we can draw an edge from $\mathcal{L}$ to $\mathcal{R}$ reprsenting each edge that existed in the original graph. For example, if the edge (u_1, u_5) was in the original graph G, then the edge (u_1, u'_5) is in the graph G'. This will create a bipartite graph that holds the same information as was in G.

EdmondsKarp(G)**:**

1. $f \leftarrow 0$; $\quad G_f \leftarrow G$
2. **while** G_f contains an s-t path P **do**
3. Let P be an s-t path in G_f with the minimum number of edges
4. Augment f using P
5. Update G_f
6. **end while**
7. **return** f

Finally, we can use a Bipartite Matching Algorithm to get the mathching from G'. This will return the set of vertices that are connected in the directed graph which, if it is a perfect matching, will represent the vertex-disjoint directed cycle cover of G. A polynomial algorithm such as the Edmonds-Karp Algorithm can be used to solve this in $\mathcal{O}\left(VE^2\right)$, which was proven in class, resulting in an overall computation time of $\mathcal{O}\left(VE^2 + 2VE\right)$.

Question 2: Attend all lectures ..... **[20]**

There are n lectures with start time s_i, end time e_i and t_{ij} being the time spent in going from lecture i to j. Now a group of students wants to attend all of the lectures by sending no more than one student to each lecture. Assume s_i, e_i, t_{ij} are all positive integers.

Find an algorithm that minimizes number of students covering all the lectures.

Solution: The question asks if there is a way to minimize the number of students that attend classes, so if a student is able to attend class i and then immediately leave for class j, then a new student doesn't have to be "dispatched." Therefore, a bipartite graph G can be created such that a *maximum matching* can be used to divide up the courses.

For each of the classes $i = \{1, 2, \ldots, n\}$, there are two nodes $\mathcal{L} = \{\ell_1, \ell_2, \ldots, \ell_n\}$ and $\mathcal{R} = \{r_1, r_2, \ldots, r_n\}$, where $\mathcal{L}$ is the left group of the bipartite graph which represents all the courses, and $\mathcal{R}$ the right which represents all courses that a student can attend right after attending another one. For each lecture i and j, for $i < j$, if $s_j \geq e_i + t_{ij}$, then we can draw a directed edge from (ℓ_i, r_j).

Bipartite Matching(G, M)**:**

1. Start a Depth First Search at a vertex in $\mathcal{L}$

2. if current vertex $\leftarrow \mathcal{L}$
3. follow an edge $e \in M$
4. else
5. follow an edge $e \notin M$

If at any point we find an unmatched vertex in $\mathcal{R}$, then the augmenting path is found.

We can define a variable M as the maximum matching of the bipartite graph, which the alogrithm is defined above. This minimum number of students who can attend the lecture can be defined as $n - |M|$, as $|M|$ would represent the number of students who attend one or more lectures. This is the optimal solution for minimizing the number of students who are attending the lecture.

This statement can be proven via *proof by contradiction*. Assume that this can be done with less students, ξ, such that $n - \xi < n - |M|$. We can define two *alternative sets* $\mathcal{I} = \{i_1, i_2, \ldots, i_\xi\}$, which represents the set of lectures where students attend more than one, and $\mathcal{J} = \{j_1, j_2, \ldots, j_\xi\}$ which represents all of the classes that were attended after lectures in $\mathcal{I}$. From here, we can create a matching, $M' = \{(\ell_{i_1}, r_{j_1}), (\ell_{i_2}, r_{j_2}), \ldots, (\ell_{i_\xi}, r_{j_\xi})\}$, which would minimize the set as it's a matching in the graph. Note that $\xi = |M'|$. Since M' is a matching in G, this implies that $|M'| = |M|$ as they should be the same values since all lectures need to be attended. However, the original assumption was that $\xi > |M|$, which was a contradiction. Therefore, $n - |M|$ is the minimum limit on the number of students to attend all the lectures.

Question 3: Monotone path ..... **[20]**

Solve all parts of question 2 in `http://web.engr.illinois.edu/~jeffe/teaching/algorithms/notes/25-maxflowext.pdf`

Breakdown is 5 + 5 + 10.

Suppose we are given an $n \times n$ grid, some of whose cells are marked; the grid is represented by an array $M[1 \ldots n, 1 \ldots, n]$ of booleans, where $M[i, j] =$ TRUE if and only if cell (i, j) is marked. A *monotone* path through the grid starts at the top-left cell, moves only right or down at each step, and ends at the bottom-right cell. Our goal is to cover the marked cells with as few monotone paths as possible.

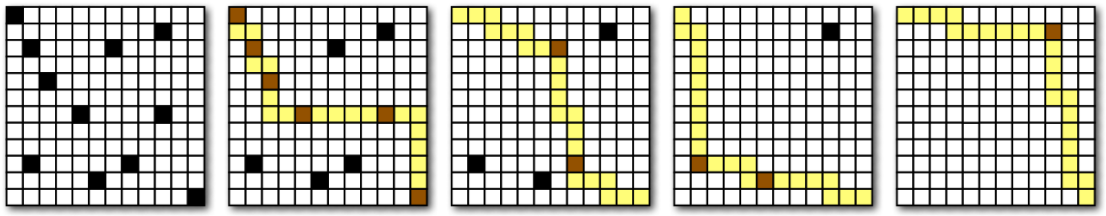


Figure 1: Greedily covering the marked cells in a grid with four monotone paths.

(a) **[5]** Describe an algorithm to find a monotone path that covers the largest number of marked cells.

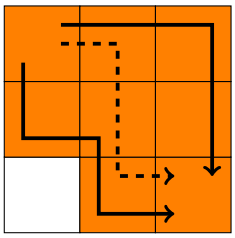
Solution: As the question didn't mention efficiency, this can be done via a brute-force method. First construct the graph such that each cell is an individual node from $\{u_{1,1}, u_{1,2}, \ldots, u_{n,n}\}$. Draw directed edges, if they exist, for all nodes from $(u_{i,j}, u_{i+1,j})$ and $(u_{i,j}, u_{i,j+1})$ to restrict movements in the graph to those defined in the problem. Finally, if a node represents a cell that is marked, give it a value of 1, else the node has a value of 0.

To find the path that coveres the *largest* number of marked cells, an algorithm such as *Depth First Search* or *Breadth First Search* can be used to determine all of the paths that are in the graph. The values of nodes traversed can be recorded such that after figuring out all of the paths, the path which traversed the largest number of marked cells would have the largest

value. This solution does not need a proof as it is looking at all possible paths and choosing the largest of the set, and is trivially true.

(b) [**5**] There is a natural greedy heuristic to find a small cover by monotone paths: If there are any marked cells, find a monotone path π that covers the largest number of marked cells, unmark any cells covered by π those marked cells, and recurse. Show that this algorithm does *not* always compute an optimal solution.

Solution: This can be shown via proof by contradiction. We can assume that if there are multiple paths with the same *largest* number of marked cells in it, then one will be chosen at random. This image can be seen below.



Each of the paths in the above figure represent possible paths with the *largest* number of marked cells. The optimal *first path* to be chosen would be either solid line, as if either is chosen then all of the marked nodes can be visited with only 2 monotone paths. However, the dashed line would take a total of 3 monotone paths to visit all of the marked nodes if it is chosen first. Therefore, simply going by the number of marked nodes visited is not sufficient to produce the minimum number of monotone paths with each iteration.

(c) [**10**] Describe and analyze an efficient algorithm to compute the smallest set of monotone paths that covers every marked cell.

Solution: This can be done by constructing a directed graph G as follows:

1. For each index in the array $M[i,j]$, create two vertices $u_{i,j}$ and $v_{i,j}$, the source vertex s and the target vertex t. This will create $\mathcal{O}\left(2n^2+2\right)$ vertices.
2. Each $u_{i,j}$ has a directed egde $(u_{i,j}, v_{i,j})$, where the edge will have a demand 1 if $M[i,j]$ is TRUE and 0 if it is FALSE. Each of these edges have no capacity, as each monotone path is independent of the other as they can overlap in cells. This creates $\mathcal{O}\left(n^2\right)$ edges.
3. Give each vertex in $v_{i,j}$ a directed edge from $(v_{i,j}, u_{i+1,j})$ and $(v_{i,j}, u_{i,j+1})$ with no demand and no capacity, for the same reasons stated above. This creates $\mathcal{O}\left(2n^2\right)$ edges as well, resulting in $\mathcal{O}\left(3n^2\right)$ total edges.
4. Create an edge $(s, u_{1,1})$ from the source to the first cell with no demand and no restrictions on capacity. We choose to not restrict the capacity of this vertex as we don't know what the capacity of the mincut is going to be yet. We do the same for the target, by creating the edge $(v_{n,n}, t)$ with again no demand and no restrictions on capacity. We can use a max flow algorithm on this new graph.

This above method creates $\mathcal{O}\left(n^2\right)$ vertices and $\mathcal{O}\left(n^2\right)$ edges and restrict the movement to those contraints in the question. From here, a max flow algorithm such as Orlin's Max Flow Algorithm can be used to calculate the maximum flow in $\mathcal{O}(VE)$ time, which is $\mathcal{O}\left(n^4\right)$ which will produce the minimum number of edges that would coincide with all the marked cells. This would produce the k-monotone paths since the max flow algoirthms minimize the number of paths required to receive the maximum flow, *i.e.* contain all of the marked cells.

Question 4: Vertex-disjoint paths ..... **[20]**

Given a square grid ($n \times n$) like the following figure, and a set of $m \leq n^2$ distinct points marked in black, devise an algorithm to determine whether there exists m vertex-disjoint paths starting at those marked points and ending at points at the boundary of the grid.

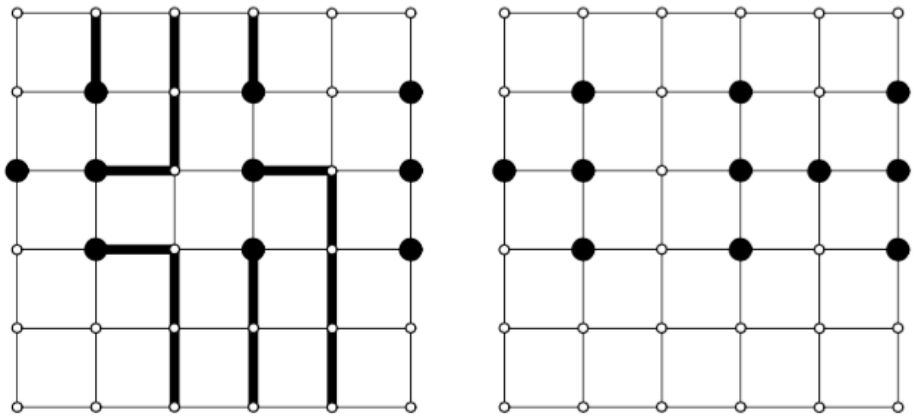


Figure 2: In the left grid there exists such a path, but in the right grid there isn't

Solution:

The solution to this can be a *Multi-source multi-sink maximum flow problem* [1] problem by building a graph G to the following specifications:

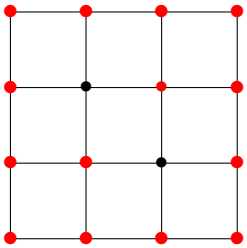
1. Create a vertex for each node in the graph, resulting in n^2 vertices.
2. For each of the m starting points, treat them as a "source point" by labeling them as s_μ, where superceeding these is an "official" source s that has a directed edge $(s, s_\mu)\ \forall\ \mu = \{1, 2, \ldots, m\}$.
3. Assign each of the $4n - 4$ boundary nodes as a target vertex, t_γ, with an "official" target t that has a directed edge $(t_\gamma, t)\ \forall\ \gamma = \{1, 2, \ldots, 4n - 4\}$.
4. Build the connections as from the images above. For example, the first node, $u_{1,1}$ has neighbors $u_{2,1}$ and $u_{1,2}$ so an edge needs to be drawn to both. However, instead of having an undirected edge, make it two directional edges, one for each direction. In other words, create edges $(u_{1,1}, u_{1,2})$ and $(u_{1,2}, u_{1,1})$ for each of the edges so that *either* direction is possible on the grid.
5. Finally, give all edges and vertices in the graph a capacity of 1.

Step 4 allows for flow in any of the directions, while step 5 restricts any overlap. Finally, we can use a max flow algorithm such as Edmonds-Karp to tell the number of vertex-disjoint paths in G. We can define the calculated maximum flow as ϑ, such that the value of ϑ represents the number of vertex-disjoint paths. Therefore if $\vartheta = m$, then m vertex-disjoint paths exist.

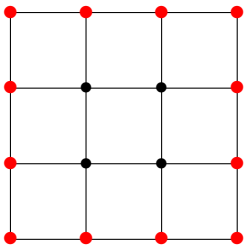
Dr. Venkatasubramanian stated that my work needed to prove that the max flow of a graph is the same as the number of vertex-disjoint paths, so this is what this does via *proof by contradiction.* First, we assume that if the max flow of G is ϑ, then there should be some other parameter ξ which represents the number of vertex-disjoint paths, such that $\xi \neq \vartheta$. If this is true, then there should never be any instance which $\xi = \vartheta$.

Suppose that there are m "source points," with λ of those nodes being on the boundary, in other words at t_γ, and $\lambda \leq m$. If λ is large enough to cover all the boundary vertices, then one of two scenarios occurs:

1. **If $\lambda < m$:** The figure below represents the type of graph that is being discussed here, with the red points representing the source points and the black points representing those which aren't sources. Recall that the above statement assumes that there are λ-vertices along the boundary of the grid. Therefore there are $(m - \lambda)$ source vertices that are *inside* the grid. However, looking at the given algorithm to build the graph, each of the vertices is given a capacity of one in step 5, so no vertices inside the grid can utilize any of the targets t_γ for $\gamma = \{1, 2, \ldots, \lambda\}$. The maximum flow and number of vertex-disjoint paths of the graph is $\xi = \vartheta = \lambda < m$.



2. **If $\lambda = m$:** The figure below shows the type of graph that is being discussed, with the red points representing the source points. Then it can trivially be seen that the max flow is λ, as λ "source points" flow directly to all the target vertices. In other words, $\xi = \vartheta = \lambda = m$.



Therefore, it was shown in the extreme case that $\xi = \vartheta$ on all orientations of the graph. However, the original assumption stated that $\xi \neq \vartheta$, which is a contradiction. Thus, the value of the *maximum flow* is the number of vertex-disjoint paths in the set.

Finally, due to the constraints imposed by the algorithm, such as in step 4 to allow for movement in any direction and in step 5 to restrict overlapping edges and vertices, the max flow would be equal to the capacity of the number of sources that made it to the boundary. This is because the flow of each source point would be 1 and the resulting flow at t would be the number of sources that made it to the target. In other words, if $\vartheta = m$ then there are m vertex-disjoint paths.

Question 5: Uniqueness of mincut ..... **[20]**

Given an s-t flow network, find a polynomial time algorithm to determine whether the mincut is unique.

Solution: This is assuming that the mincut has already been computed with an algorithm in polynomial time.

This computed mincut contains k-edges, where the cutting of these edges would interrupt the flow from s to t. These edges from the cut belong to the set $\mathcal{E} = \{e_1, e_2, \ldots, e_k\}$, which belong to the mincut C. The *value* of this mincut is represented by $|C|$, which represents the sum of all the values of the edges in $\mathcal{E}$.

For each of the edges in $\mathcal{E} \in C$, we can individually increase the capacity of each edge *one at a time*. As we iterate through the edges, each time we increase the capacity of an edge in $\mathcal{E}$, we calculate a new mincut, C_i, after which we set the capacity of that edge back to the original value before moving on to the next edge.

From this, we can conclude if the mincut is unique. If $|C| = |C_i|$ for any i, then we can say that the original mincut is *not unique*. This is due to the fact that we increased our capacity on an edge by 1, so if it was unique for that value then we shouldn't get the same value of $|C|$. On the other hand, if $|C| = |C_i| - 1$ for all $i = \{1, 2, \ldots, k\}$, then we can say that the mincut is *unique*.

This would produce a polynomial time algorithm depending on the algorithm used to calculate the mincut. For example, using the Edmonds-Karp algorithm caluclates a single mincut in $\mathcal{O}\left(VE^2\right)$, so using it in conjunction with the above algorithm to calculate the "new mincuts" produces a $\mathcal{O}\left(k \cdot VE^2\right)$ algorithm to check to see if a given mincut is unique.

References

[1] `https://en.wikipedia.org/wiki/Maximum_flow_problem#Multi-source_multi-sink_maximum_flow_problem`