CSOR4231 Final Project

Team Enigma (sdn2124, nk2913, yw3472, mu2288)

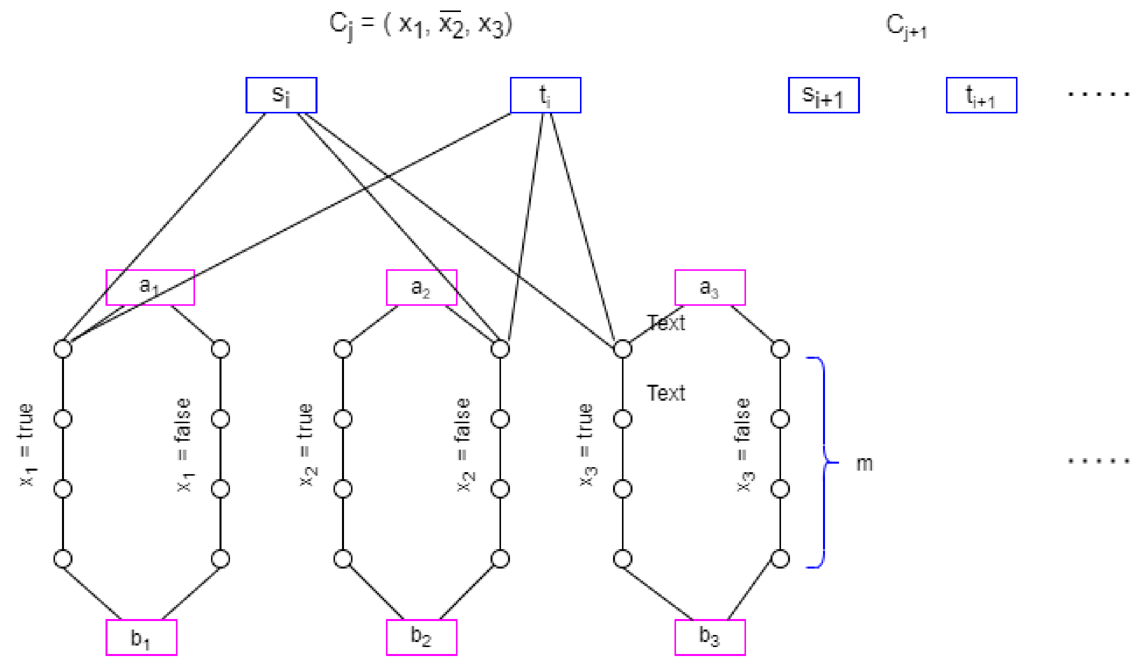
April 15, 2021

1 Proving NP Completeness

We first prove that the mutually avoiding path problem is NP by showing that we can verify the solution in polynomial time. This is trivial, since we can easily go through the k paths in the solution and check if there are any overlap between the nodes and verify connectivity, which can be done in polynomial time.

Now, we proceed to show NP Completeness by reducing 3SAT to the mutually avoiding path problem. We will consider an instance of the 3SAT problem with variables $x_i, ..., x_n$ and clauses $C_i, ..., C_m$. Here the gadget for each variable x_i will have the pair a_i and b_i as vertices, between them we will have two paths that don't share any internal nodes of length m connecting a_i and b_i; here, we notice that there are only two paths possible to reach from a_i to b_i, namely the *true* path and the $false$ path depending on the truth assignment to x_i. For each clause, say $C_j = (x_1, \overline{x_2}, x_3)$, we introduce two vertices a source and a sink s_i and t_i which will be connected in the following manner: we need to add an edge from s_i to a node in the corresponding variable gadget x_i and an edge from that same node back to t_i. Thus that would lead to 3 paths of length 2 connecting s_i and t_i. How should we choose the which node to connect to? Generally speaking, we simply take a distinct node (that no other clauses have picked from) in either x_i's *true* or $false$ path based on the truth assignment to x_i. Consider the C_j defined above, if x_1 is *true*, that means that the *true* path for a_1 and b_1 is taken, which leaves us the $false$ path to pick a node from. If x_2 is *true*, that means that the $false$ path for a_2 and b_2 are taken which leaves the $false$ path open, so we pick a node from the *true* path to connect s_i and t_i. Same goes for x_3, if x_3 is *true*, then the *true* path is taken which leaves us to pick a node from the $false$ path for s_i and t_i. See the figure below for an example construction. In general, all the a_i to b_i paths are the truth assignment to variable x_i and the s_i and t_i paths are the choice of a satisfied literal in C_j. We will know if the k pairs each have a path if every variable x_i has a truth assignment and every clause is satisfied (meaning that some literal in C_j evaluates to true). In addition, we know that there are no common nodes being used because that would only happen if a path a_i and b_i crosses with a path s_i and t_i, which by the we we are constructing is not possible. Each k pair also only have one path possible, by the way we define either the *true* or $false$ path.

By proving that the mutually avoiding path problem is NP and reducing 3SAT to it, we showed that the mutually avoiding path problem is NP Complete.



2 Easy Problem

In the easy problem, given a directed graph, G, with n number of nodes and k pairs (a_i, b_i) of sources and sinks, we want to find non overlapping paths between any a and b node without using the same nodes. Essentially, we are looking for vertex disjoint paths between pairs of nodes. We will show that ultimately we can solve this easy problem by reducing it to the max flow problem. We will create a source node S and a sink node T, then we simply connect all the a_i's to S and all the b_i's to T. Assign all the edges with an edge weight of 1 and apply the max flow algorithm (Ford-Fulkerson algorithm) from S to T. The maximum flow will be maximum number of paths (which is also the minimum cut of edges) it can find, which we can then check against k. To be more precise, this will produce edge-disjoint paths because of flow conservation having a maximum flow of k means there are k edge-disjoint paths. But we can further reduce our vertex-disjoint paths problem to use the above algorithm. Since we want to enforce that any vertex be visited only once, we can reduce the paths to edges by splitting each node n to n_1 and n_2 and adding an internal edge between them. Visually: each node $--[n]--$ will turn into $--[n_1]--[n_2]--$. Now, if two paths were to pass through node n will both have to use edge (n_1, n_2), which contradicts the "edge-disjoint" part of the algorithm above.

Thus, our final solution to the easy problem is first reduce all the path in G to edges by splitting the nodes, and assigning edge weights of 1 to each edge. Then connect all the a_i's to a source node S and connect all the b_i's to a sink node T, now run the max flow algorithm from S to T to get the maximum number of paths possible, which is some integer less than or equal to k. To recover the paths, we can do a simple modification to record the previous node and backtrack after we run the max flow algorithm. Modifying the nodes and creating the connectivity to S and T take polynomial time and the max flow algorithm is also polynomial, therefore, the easy problem can be solved in polynomial time.

Reference: http://www.cs.umd.edu/class/spring2011/cmsc651/lec25.pdf