CS 4102 **Collaborator 1** : Nick Lytle nal3gc
Fall 2014 **Collaborator 2** : Stefanie Van Rafelghem smv3xn

Zachary Seid zws3mb

1 Min-cut Max-Flow Proof

Show that the following are equivalent given some flow f for a flow network G:

1. There exists a cut $C = (A, B)$ such that the $capacity(C) = value(f)$.
2. f is the maximum flow
3. There does not exist an augmenting path in the rediual graph G' calculable from G using flow f.

Proof: Show $1 \implies 2, 2 \implies 3, 3 \implies 1$
$1 \implies 2$: If there exists a cut C, such that $capacityofC = valueoff$, then f is the maximum flow
First, it is known that any flow f in G is bounded by the capacity of any cut of G.
This is obvious from the definition of a cut and the conservation of flow–a cut is a summation of the capacity of the edges spanning the disjoint sets created by the cut, and the flow in minus flow out of a cut's disjoint set must be zero; therefore, the net flow accross any cut is greater than or equal to the inflow, which is equivalent to the summation of the capacities, or the cut.
$capacity(S, T) = \Sigma_{u \in S} \Sigma_{v \in T} c(u, v)$
$|f| = f(S, T)$
$= \Sigma_{u \in S} \Sigma_{v \in T} f(u, v)_{inflow} - \Sigma_{u \in S} \Sigma_{v \in T} f(v, u)_{outflow}$
$\leq \Sigma_{u \in S} \Sigma_{v \in T} f(u, v)_{inflow}$ (Since outflow is positive)
$\leq \Sigma_{u \in S} \Sigma_{v \in T} c(u, v)$ (by the capacity constraint)
$|f| \leq c(S, T)$
$c(S, T) \equiv c(A, B)$
$|f| \leq c(A, B)$
Since $|f| = c(A, B)$ (1), it is as large as can be, and thus maximum flow.
$2 \implies 3$: If f is the maximum flow, there does not exist an augmenting path in the residual graph G' calcuable from G using flow f
For the sake of contradiction, suppose that that f is the maximum flow, but G' has an augmenting path p.
By definition, this augmenting path must contribute some non-negative flow.
Then, the flow of G_f must be the flow in G + the additional flow from p, or
$|faugmentedbyf_p| = |f| + |f_p| > |f|$
But this is a contradiction, since f is defined as a maximum flow.

$3 \implies 1$: If there does not exist an augmenting path in the residual graph G' calcluable from G using flow f, then there exists a cut $C = (A, B)$ such that the $capacity(C) = value(f)$
By definition of lacking an augmenting path, there is no path from s, t source to terminal.
Let $A = v \in V : there exists a path from s to v in G'$, or a set of some arbitrary vertices in G'.
Let B be the set of vertices created by the cut (A, B) in G' such that $B = V - A$, and $s \in A$ and $t \notin A$ because there is no path from s to t in G'
Consider a pair of vertices, one such that $u \in A$ and $v \in B$. The edge between u and v must be at max flow, otherwise it would be an augmenting path (i.e. possible for flow) and v would be in A because by definition all edges with a path in the residual graph are in A.
If instead the edge is from v to u (a backedge), its flow must be 0 otherwise the residual capacity of the forward edge would be equal to the flow of the backedge, and the path from v to u would meet the requirements of the set A. If neither direction is a valid edge, then its flow must be zero.
Thus, the flow of the cut is the inflow - outflow, and by applying the above reasoning to all forward edges and all backedges, we know forward edges must be at capacity and back edges must be zero, and the flow $f(A, B)$ is simply $c(A, B)$, which is equal to $|f|$.

2 Fixing the Dam

Given a matrix $wall[i][j]$ with 1's representing holes and 0's representing otherwise, first define a function $color(i, j)$ which partitions any input i, j into a class red or white, such that no red is adjacent to another red and no white is adjacent to another white. This is akin to a checkerboard pattern and can be accomplished in constant time via odd/even comparison of x and y (i.e. i%2==0 and j%2==0 or i%2==0 and j%2!=0 $\implies$ red, otw white).
Then, maintaining a list of holes, or nodes, in each class, red or white, linearly check through all $wall[i][j]$. On an encounter of a hole (1), add a node with it's coordinate's representation to the list of appropriate class, if it does not already exist, and check the 4 adjacent locations (or fewer, depending on boundaries). If a hole exists in the adjacent spots, similarly add if not yet added a node with the appropriate coordinate representation to the other class as determined by $color(i, j)$ and add an edge of capacity 1 between the originally found hole and the new node of the adjacent hole (if an edge does not already exist). Continue in this fashion until all i, j have been checked.
Construct a bipartite flow network by creating a source and connecting every element in red to the source with an edge of capacity 1, and similarly for every element in white to the terminus with an edge of capacity 1.
Run ford-fulkerson on this flow network to determine the maximum flow. If this flow is equal to the number of nodes in both red and white lists combined, divided by 2, every hole has been patched and the brick placements are equivalent to the edges with flow. If the flow is not equal, not all holes can be patched and the answer is false.
This algorithm works intuitively by only considering adjacent holes and limiting the consideration of brick placement with capacity 1 edges to each element in the bipartite matching.
Though not asked for, the runtime of this algorithm is bounded by the linear traversal $i * j$,

and Ford-fulkerson runs in $O(ef)$, where here the maximum flow is the number of holes/2. e is at most 4, due to adjacency constraints, so ford fulkerson is O(4i*j/2) which is still bounded by O(i*j).

3 Multi-agent pathfinding

Given a graph $G = (V, E)$, and two robots with starting nodes $s_1, s_2 \in V$ and destination nodes $d_1, d_2 \in V$, design a schedule along the edges for each robot to move from start to destination such that each step in the schedule a single robot moves, at no point in time do the two robots exist at the same or adjacent nodes, and given the fact that the start and end nodes for each robot are not the same or adjacent.
A decision tree will be generated enumerating the possible moves for each step in the schedule. The raw permutations can be generated via a breadth first search from s_1, d_1 and s_2, d_2, but for each first-step, the following check must be applied:
Given an adjacency matrix, if the prescribed edge connects to the location of the other robot, the step is not valid and should not be included in the decision tree. Likewise, the adjacencies of each considered step must not contain the other robot, whose location is known. As is usual for BFS, a node already visited is not valid.
The decision tree, constructed thus to the respective destinations for each thread of action, represents all of the feasible solutions and is essentially a union of the adjacency-filtered BFS permutations for each robot to destination. The shortest path can be found and used to signal termination by each robot reaching it's destination state.
To build the decision tree, BFS will simply visit each of the options. As an upper bound, the options of each robot can be stated as all other nodes except the node currently occupied, though that is not as tight as possible because the opposing robot and it's adjacent edges are eliminated via the worst-case linear (w.r.t nodes) condition checking. Let the number of nodes be n–the above statement says each step in the schedule considers an order n number of nodes per robot, and worst case every node must be visited to reach each robot's destination. This yields $O(2 * n * n^2) \in O(n^3)$ for the two robot case.
For the general case, each robot expands the decision tree by order n number of moves, and this adds n^r new possible choices for each step. Worst case each node must be visited per robot, making n iterations of n^r choices per robot. An upper bound on this is $O(r * n * n^r)$