Project Reflection

Minan Wu, Xinyu Fu, Amy Kwon

Overall

The program is built on a search tree with various strategies and heuristic.

For the problem we face, the state space is too large to search thoroughly, so either vanilla BFS or DFS will not work well on this problem. Hence we propose a greedy algorithm with multiple heuristics to approximate the optimal solutions.

Search Strategies

We built the search tree based on a priority queue, with two priority values, the current score as the first value and the score divided by the number of operations as the second value. We found that different transition functions will return the same next state, so we prune the priority queue each time we search for new states.

We also use different levels of greed in the search. The first is always moving to the next state from the best state in the queue, trying to achieve a higher score with limited computation. The second is always moving to the next state from the worst state in the queue. This strategy is to combat against local optima and also the adversarial inputs. The third is a mix of the last two.

We use all of the three strategies. We found for the small graph, the least greedy algo works the best. And for the medium and large, the most greedy and the mix algos work better for being cheap. At last, we collect the best solutions from all the passes and submit them.

Like BFS, we try to enforce the same depth, which is the number of operations, for each state, but we found this works poorly.

We also try different queue sizes. For less greedy strategies, a large queue size generally works better but also expensive. Greedy strategies are less sensitive to the queue size. To bound the total run time, we use a queue size scheduling. When the number of searches arrives at some limits, we shrink the queue and force the algo to produce a feasible solution.

Transition Functions

We use both randomness and heuristics for the transition function. The random function will select a random edge in the shortest path. We also choose deterministically the first and the last edge in the shortest path, which we regard as random. We try to remove the random functions, but this generally works worse.

For heuristic functions, we first try to remove the edge at the s-t cut with minimum cardinality. Because the s-t path must cross some s-t cuts, and if removing an edge at the minimum cardinality cut, it may need more edges to construct a new shortest path. We try to delete two edges at the minimum cardinality cut, but it works poorly. The second is to remove the lightest edge on the path, and the reason is very intuitive. At last, we remove the edge that can gain the most distance. This needs to find disjoint paths for each pair of adjacent nodes on the shortest path, and see which pair has the longest minimum disjoint path, then remove the edge between the pair. This heuristic works the best and is much more expensive than the last two. We don’t have a chance to use it to run multiple times for the large input.

We built transition functions for nodes based on the ideas for edges. We haven’t developed functions specifically for nodes.

Due to the randomness of the algo, each time we run it can found better solutions for some inputs.

Computation

We run the program only on laptop computers with Intel Core I7 processors, which we believe are not as good as the AMD’s. The limit on computation restricts the use of better but more expensive functions and the number of passes we can run the program.