CS170 Project Final Writeup

Daniel Van Der Maden, Willow Watkins, Joseph Kraut

1 How the Algorithm Works

There are two distinct steps to this algorithm. First, the algorithm assigns students to buses greedily based on a series of heuristics; we try a few different heuristics for this part. Then, the greedy assignment is handed to an optimizer which climbs the optimization space to find a better solution. Finally, we write our solution to file only if it is better than the solution we already have saved. This allows us to test many methods and take the best solutions on an input by input basis.

1.1 Greedy Assignment

Our method of greedy assignment involves the following heuristic

$$H(b,i) = \frac{f(b,i)+1}{\max_{g\in L, i\in g} \Phi(r(i,b,g))+1}$$

where $f(b,i)$ is the number of vertex i's friends currently on bus b, $r(b,g)$ is the number of members of rowdy group g on bus b. The $+1$ in numerator and denominator is for smoothing and continuity purposes. The function Φ is a function we design to fit the purpose of bringing down the heuristic as we approach a full rowdy group. For this Φ function we used a normal approximation to the dirac delta function which has a peak of size ∞ at some specified value; we brought this peak down and spread out the function density to make the heuristic smooth and penalize close to full buses as well as completely full buses. To decide the order in which to process vertices (computing their heuristic values and adding them to the bus that produces the maximum heuristic) we tried a number of approaches including lowest degree, highest degree, and potential gain to score in the number of friends they could be placed with. To break ties between heuristic values, we also tried a number of methods; least full bus, most full bus, and most friends on a bus. For each input we tried every combination of tie breaking and process order to determine the best possible solution, except for large inputs where we only tried the combination that worked best for small/medium inputs.

1.2 Optimization

We tried two separate combinatorial optimization methods for this task. Each searches the optimization space; the units of which are swaps between two students or a student and an empty seat. The first optimizer, which we called the basic optimizer, simply samples a specified number of swaps on each iteration and takes the best swap possible; repeating for a set number of iterations. The more advanced optimizer we tried was inspired by the rollout policy of a monte carlo tree search. This method samples paths in the optimization space of a set length, taking the best path on each iteration and repeating for a set number of iterations. Of the two, the tree search optimizer performed better.

2 Other Methods

The methods above were all that we tried, noting that we tried every combination of method for each input.

3 Computational Resources

The computational resources we used included Daniel's high performance PC and instructional machines; which Daniel took care of to ensure only one of us used the machines at a time.