Q1 Efficiency Explanation

Advanced Algorithms Assignment 2

Anthony Medico

To help with efficiency a number of techniques were used. These range from choice of data structure, to precomputing things ahead of running the algorithm, to pruning branches of the search tree as soon as possible. The two main CPU intensive tasks for this program are the backtracking algorithm, and the permuting done for equivalence testing. The complete explanations for both of these aspects are described below.

Backtracking Efficiencies

- Precalculated valid rows.
    - Instead of building a solution one cell of the adjacency matrix or one edge at a time, all complete valid rows of degree 3 were precalculated and stored as integer arrays.
    - Now a candidate to an overall solution is an entire row, resulting in n levels rather than n^2 levels.
    - The rows were calculated in lexicographic order and then stored in that order, so their corresponding index is also their rank. In this manor, getting a row by rank is an instant lookup, rather than needing an unrank function.
- Storing the rows as integer arrays.
    - Rows could be represented by a k-subset, for example [1, 2, 4] would correspond to [1, 1, 0, 1, 0, …,0], however with many operations needing to convert the k-subset into an adjacency matrix row, I chose to store the rows as an actual adjacency matrix row. This significantly increased program execution time.
    - I chose to use primitive integer arrays where I could, so operations on them would be quicker and with less overhead than ArrayList.
- Candidates represented as a single integer – the precalculated row's rank.
    - In this manor, a solution is just a list of candidates (integers).
    - A graph can then be built quickly by getting the rows at the given rank (its index).
- Pruning branches of the search tree.

    - A valid solution will always have 0's on the diagonal of the adjacency matrix. If a candidate does not have a 0 where it's diagonal would be (depends on which row the candidate is being placed), that branch would be pruned.
    - A valid solution has degree 3 for every row and column. Anytime a new candidate was added, the corresponding adjacency matrix of the solution so far was checked to ensure no columns had degree greater than 3, or else that branch was pruned.
    - A valid solution's adjacency matrix is symmetrical about the diagonal, each new candidate was tested to ensure this property was maintained, or else that branch was pruned.
    - Choices for further efficiency for pruning:
        - Above tests only done on the newly added row (where possible) since the previous rows have already been tested and verified to meet these requirements.
        - The order chosen for the above validation checks was done in a logical way. No need to calculate the degree of each column if the diagonal had a 1, and no need to ensure symmetry if the degree of a column was greater than 3.
- Check for equivalence once a complete valid solution is constructed.
    - Equivalence checking is time consuming (many permutations).
    - Only check when a valid solution is found.
    - See next section for equivalence checking efficiencies.

Equivalence Checking Efficiencies

- Each time a unique graph is found, generate and store all its permutations.
    - In this manor, all permutations are calculated once and can then be looked up in subsequent equivalence checks.
- Store graph efficiently.
    - With so many graphs to store, it can take up a lot of memory, so for this I needed to be efficient with memory.
    - This was achieved by encoding the graph to a unique string.
    - Used the rows' ranks separated by "-" character.
    - Strings of about 2n length rather than n by n matrices of integers.
    - E.g. "3-2-1-0" would be a 4x4 adjacency matrix with rows of rank 3, 2, 1 and 0

```
[[0111],
 [1011],
 [1101],
 [1110]]
```

- All permutations (valid graphs and all its possible permutations of equivalent graphs) stored in a HashSet for fast look up.
    - HashSets provide fast look up, however they can degrade if there are many collisions. My first implementation used linear search, and then I switched to using a HashSet, which made an extreme difference in program execution speed.
    - Don't need to permute every newly generated valid solution, instead only when a new unique solution is found, and then add the new unique solution and its equivalent graphs to the HashSet.

Notes

- For n=12, I instantly ran out of memory using this approach of precomputing and storing as much information as possible to increase runtime efficiency.
- Without precomputing, n=12 takes too long.
- Conclude that with the average home computer, n=12 may be impossible to run in under 24 hours.