Branch and Bound: Assignment Problem

Assignment 505
CS 310

Cameron Moberg

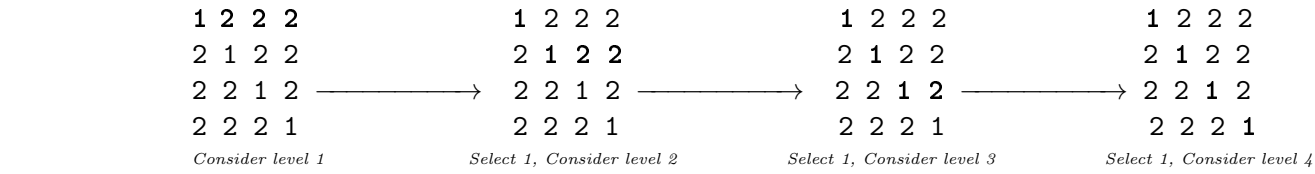
May 4, 2017

Branch and bound is a algorithm design for optimization, in this case an optimization for combinatorial optimization. In this

1 Unmodified Performance

1.1 Minimum Considers

In order to find the input that causes the least amount of considering lines, we must examine the code and work backwards. The code calculates the lower bound for the first level of input, finds the node with lowest bound, and repeats that process for the next level with the added constraint of the inability to use the column that was just used. An example input is shown below, where bold means that the job assignment has been chosen or is being considered.



Now, because of the way the code is implemented, once the solution is found, it will not end until the priority queue is empty, causing unneeded considers in this case.

1.2 Maximum Discards

Much like previously, we must examine the code and work backwards to find a least optimal input that discards the greatest amount of nodes. In order to achieve the greatest number of discards, the algorithm must generate every possible permutation ($n!$ of them) before finding a solution. This occurs when every single value is the same, e.g.

```
1 1 1 1
1 1 1 1
1 1 1 1
1 1 1 1
```

This is because when it considers the very first level, it adds them all to the priority queue, and because this priority queue's comparator is implemented to keep order, the nodes keep their order. Since all of the nodes have the same lower bound, the program cannot rule any out, and will generate every permutation until it gets to the bottom level, where it will find a solution.

1.3 Double Best Solution

The goal was to find an input that found "two best solutions". However, upon closer inspection it does not appear possible to have an input that finds two best solutions. When we take a look at the code, we see that the program only finds a best solution on the bottom level (line 127). At every step the program chooses the most promising node, the one with the smallest lower bound, and it uses strictly less than when determining the best solution (lines 129, 149), so when it finds the first solution, it will be the only solution.

2 Increasing Efficiency

There are ways to increase the efficiency of this program (no offense) by taking advantage of what we saw when finding maximum discards, minimum considers, and multiple solutions. To decrease basic operations we must first determine what the basic operations is, and we chose the if statement in the `calculate_lower_bound()` method on line 62 since it's computationally expensive and occurs most frequently.

Overall, there were three things changed to improve efficiency.

1. Quickly calculate a baseline lower bound
2. Stop after finding a solution
3. Make calculating lower bound more efficient

2.1 Calculating a Baseline Lower Bound

In the unmodified code on line 118 there is the line

```
118 uint best_solution = UINT_MAX;
```

To increase efficiency we replaced that line with the below code in the modified file.

```
120     //set base solution to potentially rule out.
121     uint best_solution;
122     for ( uint i = 1; i <= n; i++ )
123     {
124         best_solution += cost.at( i, i );
125     }
```

This sets a baseline `best_solution` which could potentially cut down on the number of lower bounds needed to calculate depending on the input and arrangement of numbers. When using this method on an input that are all the same number, as seen above, the program cuts down from 256 basic operations to only 64 in a 4 by 4 input.

2.2 Stop After Solution Found

As previously mentioned, when the program finds a solution it is the most optimal, meaning there is no need to continue checking the priority queue. On line 138, putting a simple `break;` statement in the loop allows the algorithm to potentially end much earlier than it was in the unmodified version, leading to less basic operations.

2.3 Reinventing the Wheel

To see the efficiency we ran four different programsL: