IE 716 Project
120050030 (Aman Gour) and 143050045 (Prakhar Gupta)
Spring 2016
IIT Bombay

Problem Statement:

Implemented a branch-and-bound solver for MILP problems in C++, C or Python. Used existing open-source libraries to read the problem and solve the LP relaxation. In particular, Implemented routines to do a depth-first search, breadth-first search and a best-first search.

Code: The complete code can be downloaded from the git repository: **https://github.com/amangour30/MILPSolver2016**

Approaches Implemented:

Depth First Implementation: In the implementation of the depth first approach in branch and bound we used a stack to make the program memory efficient and added the constraints for problems on the fly. This approach helped us to develop a really fast implementation of the DFS approach.

Breadth First Implementation: Similar to the DFS approach here we employed a queue data structure to minimize the redundant data copy while calling the child functions.

Best First Approach: Best first tries to find a solution as fast as possible by looking at the most interesting part of the search space first (most interesting = estimate). For the implementation of the best first approach we created a priority queue of elements which keeps the nodes sorted on the basis of the value of the LP problem. At every iteration we pick the node which has the smallest LP value (min case) and then add the child nodes (if any) to the same queue. In this way the heuristic of *best optimum value* is used to do the node selection.

Augmented Depth First Approach: In all the above approaches, the first fractional variable in the solution of a node is chosen for branching but it may lead to lots of nodes being explored. The variable selection is another important paradigm in the branch and bound algorithms. In augmented DFS we implemented the heuristic which solves a node with various constraints added and only adds the best one to the stack first.

Structures for a Node: The structs for a node in the tree, LPNode, is shown below.

```
class Soln:
    def __init__(self,status,xSol,ySol,optVal):
        self.status = status
        self.xSol = xSol
        self.ySol = ySol
        self.optVal = optVal

class LPNode:
    def __init__(self,code):
        self.code = code
        self.status = 0
        self.AddlC = []
        self.AddlB = []

    def add_Constraint(self, AddlConstraints,AddlB):
        self.AddlC = AddlConstraints
        self.AddlB = AddlB

    def sol(self, variables, isSolved, optVal):
        self.variables = variables
        self.isSolved = isSolved
        self.optVal = optVal

    def Bounds(self, UB):
        self.UB = UB

    def updateStatus(self,status):
        self.status = status
```

Experimental Results and Comparison with PuLP solver:

Problem	**Depth First Solver**	**Breadth First Solver**	**Best First Solver**	**PuLP Solver**
Travelling Salesman Problem (5 vertices)	0.41 s	0.35 s	0.18 s	0.09 s
Facility Location Problem (3 Facilities and 30 sites)	1.84 s	1.85 s	1.46 s	0.05 s

Conclusion: We can see that our solver's best-first approach is comparable to PuLP's performance for the Travelling Salesman Problem. However, for facility location problem, our solver performs poorly. This might be attributed to the fact that PuLP solver is equipped with many other heuristics as well as cutting plane methods to reach the optimal quickly.

Future work:

(1) Threading for parallel execution of the nodes at same level
(2) Different heuristics for node and variable selection
(3) Including cutting planes approach

Reference:

(1) Depth-first branch-and-bound search, http://artint.info/html/ArtInt_63.html
(2) https://en.wikipedia.org/wiki/Branch_and_bound
(3) M. Conforti, G. Cornuejols, and G. Zambelli, Integer Programming, Grad. Texts in Math. 271, Springer, 2014.
(4) Python Documentation