CS598 Intermediate Project Report:Solving Integer Programming(TSP) with Branch-and-Bound

Cheng Li, Zhangxiaowen Gong, Xuan Wang, Yiyi Wang

October 31, 2016

Introduction

The intermediate report includes the algorithm solving travelling salesman problem, using Gurobi doing the linear optimize work and the parallel branch and bound algorithm.

Traveling Salesman Problem

We target at Traveling Salesman Problem as start point. The traveling salesman problem is a problem in graph theory requiring the most efficient (i.e., least total distance) Hamiltonian cycle a salesman can take through each of n cities. The problem then consists of finding the shortest tour which visits every city on the itinerary.

The notations of TSP are as follows:
$G = \langle V, E \rangle$ is a positive weight complete graph, where $V = \{v_i\}(1 \leq i \leq n)$ is a vertex set.
$E = \{(v_i, v_j)|v_i, v_j \in V$ and $i \neq j\}$ is a set of undirected edges.
$d_{ij} = (v_i, v_j)$ is positive weight of edges.

$A = \begin{bmatrix} d_{11} & \dots & d_{1n} \\ \vdots & \ddots & \vdots \\ d_{n1} & \dots & d_{nn} \end{bmatrix}$ is an adjacency matrix of graph G. Among them,

$$d_{ij} = \begin{cases} > 0 \text{ and } d_{ij} = d_{ji} & i \neq j \\ +\infty & i = j \end{cases}$$

$$A_{ij} = \begin{cases} d_{ji} & (v_i, v_j) \in E(G) \\ +\infty & \text{others} \end{cases}$$

We also define x_{ij} as logic variable that identifies whether the edge (v_i, v_j) belongs to search path or not.

$$x_{ij} = \begin{cases} 1 & (v_i, v_j) \text{ belongs to search path} \\ 0 & \text{others} \end{cases}$$

Then our solution to calculate the minimum cost cyclic route ($MinCR$) in graph G would become:

$$MinCR = min \sum_{i=1}^{n} \sum_{j=1}^{n} d_{ij} x_{ij} \tag{1}$$

In order not to generate sub loop by using the formula (1), it needs to add the following constraints:

$$\begin{cases} \sum_{j \neq i} x_{ij} = 2 & i, j \in \{1, 2, ..., n\} \\ \sum_{i,j \in S} x_{ij} \leq |S| - 1 & 2 \leq S \leq n-2, S \subset \{1, 2, ...n\} \end{cases} \tag{2}$$

The first condition ensures that each vertex in search path just has two and only two edges; the second condition makes sure the graph G has no sub loop. If the graph G has some sub loops, then the number of edges in all sub loops equals to the number of vertices.

In our project, we will use the dataset from the University of Waterloo website. The dataset includes a set of coordinates of cities. So far we have built a Gurobi version of integer programming solver. The Gurobi by default is multi-threaded and will use all possible cores of the machine. We have tested our Gurobi version on a local machine with a testing dataset that consists of 200 cities, and it took around 20s to find optimized solutions. However, when we incremented the city num to 300, the execution time increased tremendously. Therefore, we believe that the parallel version that we will build will highly increase the performance of solving TSP compared to Gurobi version. In our version, we will only use Gurobi to solve the linear programming part. The branch-and-bound part will be implemented in parallelism using `Charm++`.

Gurobi Usage

We use gurobi to find the linear programming optimal solution. After download and installation, gurobi can solve the optimize problem. We need to transform the problem to appropriate format and send it to gurobi. Then we will get the solution.

Here is a simple example:
Suppose we want to maximize $x + y + 2z$
Subject to

$$\begin{cases} x + 2y + 3z \leq 4 \\ x + y \geq 1 \\ x, y, z \in [0, 1] \end{cases}$$

We need to

1. Create Gurobi model:

   ```
   GRBEnv env = GRBEnv();
   GRBModel model = GRBModel(env);
   ```

2. Create variables:

   ```
   GRBVar x = model.addVar(0.0, 1.0, 0.0, GRB_CONTINUOUS, "x");
   GRBVar y = model.addVar(0.0, 1.0, 0.0, GRB_CONTINUOUS, "y");
   GRBVar z = model.addVar(0.0, 1.0, 0.0, GRB_CONTINUOUS, "z");
   model.update();
   ```

3. Set objective function:

   ```
   model.setObjective(x + y + 2 * z, GRB_MAXIMIZE);
   ```

4. Add constraints:

   ```
   model.addConstr(x + 2 * y + 3 * z <= 4, "c0");
   model.addConstr(x + y >= 1, "c1");
   ```

5. Optimize model:

   ```
   model.optimize();
   ```

6. Finally we can access the solution:

   ```
   cout << x.get(GRB_StringAttr_VarName) << " "  << x.get(GRB_DoubleAttr_X) << endl;
   ```

Now we can install and use Gurobi successfully on local machine. However, Gurobi requires license on installation and usage in each process. We may meet difficulty on running it parallelly on Taub. If we really can not solve it, we may think about to use other library or write our own code to do the linear optimization work. However, the procedure should be similar and the experience on Gurobi can be helpful.

Branch-and-Bound

Branch and bound algorithm fully searches the whole solution space and obtain the global optimal solution. It cuts the nodes which do not meet the requirements according to pre-defined Objective Function Bound(OFB) to reduce the search range. Also it can select the best which meets the requirements of OFB as an expanded node from the all nodes and it makes the search move towards the optimal branch. In detail, branch and bound algorithm takes the growing search path as objects, takes the pre-estimated OFB as constraint condition . By calculating the cost path of the current search path, it will make judgment as follow: if cost of partial path(CPP) is greater than Objective Function Up Bound(OFUB), then it should discard this partial path; if cost of total path (CTP) is lower than OFUB, then it should update the OFUB to CTP, and continue to search for another total path whose CTP is much less. Objective Function Low Bound(OFLB) is used to constrain the search space. When CTP of some total path is closest to OFLB, it is the best search path. By continuously amending the OFUB, it can gradually narrow the range[OFLB, OFUB], and find the global optimal solution which is limited between the OFUB and OFLB bounds as soon as possible. Briefly, branch and bound algorithm has two features: estimating the OFB in advance, and searching the path while cutting branch and amending OFUB value. It is suitable for distributed and parallel computing as it can be divided into independent sub-problems.

Design and Implementation on `Charm++`
We take a similar approach to the "Master-slave" mode designed for multi-core environment[]. Instead of having "master core" and "slave cores", we introduce "master object" and "slave objects" to take advantage of `Charm++`. A master object is responsible for task allocation. It creates some slave objects responsible for the parallel execution of the branch and bound algorithm. The process is summarized as follows.

1. Master object searches all the children nodes of root node and calculate initial OFB; then it creates slave objects that have private data structures and variable OFB respectively.
2. Master object triggers the slave objects to work in parallel by task allocation. There's no partial task distribution issue as in [] because the number of slave objects are not limited by the number of physical cores. (In [], for a partial task distribution where the number of children nodes is larger than the number of slave objects, the master object has to check for available slave objects intermittently until all of the tasks are distributed).
3. Slave objects execute the tasks assigned by master in parallel. A slave object searches an optimal path as follows: if the path cost is greater than OFB, it will stop execution immediately, empty the private queue and notify the master object; if the slave object finds a total search path and this path cost is less than OFUB, then it will update the OFUB to this path cost and update all the other path costs in other slave objects by

message broadcasting.