CAAM 471/571: Traveling Salesman Project

Kevin Burleigh and Julio Ledesma

2017-04-19

1 Problem Description

We were tasked to solve the Traveling Salesman Problem using the Branch and Cut method (TSPBC), utilizing Gurobi.

Given a graph $G = (N, E)$ with nodes N and edges E, and an associated cost c_e for each edge, the TSP can be formulated as:

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} c_e x_e \\ \text{subject to} \quad & \sum_{e \in \delta(\{n\})} x_e = 2, \quad \forall n \in N \\ & x_e \in \{0, 1\}, \quad \forall e \in E \end{aligned}$$

where x_e is a decision variable indicating whether or not the associated edge is part of the tour, and $\delta(S)$ is the set of edges in the cut of node set S. Each constraint in this formulation ensures that each node is entered exactly once and then exited exactly once along a different edge.

The LP relaxation of

$$\begin{aligned} \text{minimize} \quad & \sum_{e \in E} c_e x_e \\ \text{subject to} \quad & \sum_{e \in \delta(\{n\})} x_e = 2, \quad \forall n \in N \\ & 0 \le x_e \le 1, \quad \forall e \in E \end{aligned}$$

2 Algorithm

The algorithmic approach to solving TSPBC is as follows: 1. Initialize a Gurobi model with input data and add to a priority queue. 2. While the priority queue is not empty, pop a model off and set as current model 3. Process the current model i.e. adding constraints, and create new branches 4. For each new branch, add to priority queue Each step in the algorithm will be explored in further detail.

2.1 Main Loop

The main control loop:

1. creates an initial model
2. adds the initial model to the model pool
3. until the model pool is empty:
    (a) removes a model from the pool
    (b) processes the model
    (c) adds any resulting new models (and their associated objective lower bounds) to the pool

Processing ends when the model pool is empty.

```python
## tsp_solver.py
class TspBranchAndCut(object):
    def solve(self):
        initial_model = self.create_initial_model()
        self.add_model_to_pool(model=initial_model, obj_lb=-float('inf'))

        while not self.model_pool_is_empty():
            model = self.remove_next_model_from_pool()

            for obj_lb,new_model in self.process_model(model):
                self.add_model_to_pool(model=new_model, obj_lb=obj_lb)
```

2.2 The Initial Model

```python
## tsp_solver.py
class TspBranchAndCut(object):
```

```
def create_initial_model(self):
    model = grb.Model('tsp')
    xx = model.addVars(self.edges,
        lb    = 0.0,
        ub    = 1.0,
        vtype = grb.GRB.CONTINUOUS,
        name  = 'xx',
        obj   = self.cost_by_edge
    )
    degree_constrs = model.addConstrs(
      (xx.sum(node,'*') + xx.sum('*',node) == 2.0 for node in
          self.nodes),
      'degree'
    )
    model.update()
    return model
```

Once we developed our algorithm for solving TSPBC, we looked to Python and the gurobipy module. Additionally, we implemented a Graph object to facilitate our computation of TSPBC. Graph is initialized by helper functions that assign its node, edge and edge weight attributes. Graph also contains methods to find the minimum cut, to compute connected components, and to determine if the instance forms a tour. Instances of graph are copied before more branches are produced. Additionally, there are auxiliary functions that easily convert a Graph instance into a Gurobi model.

3 Initialize model

Given an input fie, the data is translated as a series of nodes, edge and their assigned weights, with each node having an edge to every other node. This data is transformed into a Gurobi model, initialized with degree constraints and bounds on the edge decision variables. This creates our initial linear program (LP) for the TSPBC. Our initial LP is then added to our priority queue.

4 Process

While the priority queue is not empty, we pop a model and set as our current model. We then update the model and solve for the optimal solution. Before we start to branch, we first check to see whether the solution is feasible. If infeasible, we stop the branch and move on to the next item in the priority queue. Next, we check if the model can be cut down by adding constraints that lower the objective function. We then check if our solution forms a tour, else we stop the branch. If possible, we add new constraints that help our model reduce its objective value. Once global and local constraints are added to the models, we create two new branches and continue once again for the next model in the priority queue.

5 Constraints

These specific cuts are comb and blossom inequalities, integral and non-integral sub-tour constraints, and mixed-integer Gomory cuts. For the comb and blossom inequalities, we searched for cuts based of this equation in our candidate solution: [INSERT BLOSSOM INEQUALITY] For both integral and non-integral sub-tour constraints, we check to see if the current model formed a sub-tour, and if so we added a constraint. Lastly, for mixed integer Gomory cuts, we searched for cuts based of [INSERT MIXED INTEGER GOMORY CUTS]. For each cut, constraints are added to each Gurobi model and the model is then solved and branched out again.

6 Min Cut

To compute the min cut, we use the Stoer-Wagner min-cut algorithm in the provided paper. Our implementation of this algorithm returned the nodes that made up the min cut as well as the size of the min cut. The value of the minimum cut was less than two if and only if there is a constraint that has been violated. This function is contained as a method in Graph.

7 Computing Tour

Given a candidate branch, we must check if it forms a potential solution. In our python implementation, our solution is a vector of values in [0.0, 1.0]. These values represent if an edge is included in our solution. We can eliminate infeasible branches if their vector contains a value that isn't either 0.0 or 1.0. Additionally, we need to check if our potential solution satisfies the degree constraint [degree]. Finally, we must check that our candidate solution produces at most one connected component. We can compute the connected components of our candidate solution by running breadth-first search on every edge in our candidate solution. If our candidate solution passes that each edge is either 0.0 or 1.0, then we have each node in our tour meeting the degree constraint [degree], and that our candidate solution forms at most one connected components [degree], then this is sufficient to declare our candidate solution a tour. This function is contained as a method in Graph.

8 Implementation

8.1 The Graph Class

A *Graph* class was created to encapsulate purely graph-related functionality, such as identifying:

1. nodes in connected components
2. edges in a min-cut
3. edges in the cut for a set of nodes
4. whether or not a set of edges forms a tour

9 Results

att48

berlin52

gr21

hk48

ulysses22

pr76

st70

10 Improvements

11 Closing Remarks

Our implementation of TSPBC was successful in that we could correctly solve each of the data sets provided. However, as stated in our improvements section, we could have added more types of cuts to reduce the runtime of pr76.

12 Section Title Here

12.1 Subsection Title Here

Some text here.

$$p(a \leq Z \leq b) = \int_a^b \phi(x)\,dx \tag{1}$$

Reference equation Equation 1 here.

$$\begin{aligned} p(a \leq Z \leq b) &= p(Z \leq b) - p(Z \leq a) \\ &= \int_{-\infty}^{b} \phi(x)\,dx - \int_{-\infty}^{a} \phi(x)\,dx \\ &= \Phi(b) - \Phi(a) \end{aligned}$$

Inline $\Phi(\alpha)$ here.

$$\Phi(\alpha) = \int_{-\infty}^{\alpha} \phi(x)\,dx$$