CAAM 471/571: Traveling Salesman Project

Kevin Burleigh and Julio Ledesma

2017-04-19

1 Problem Description

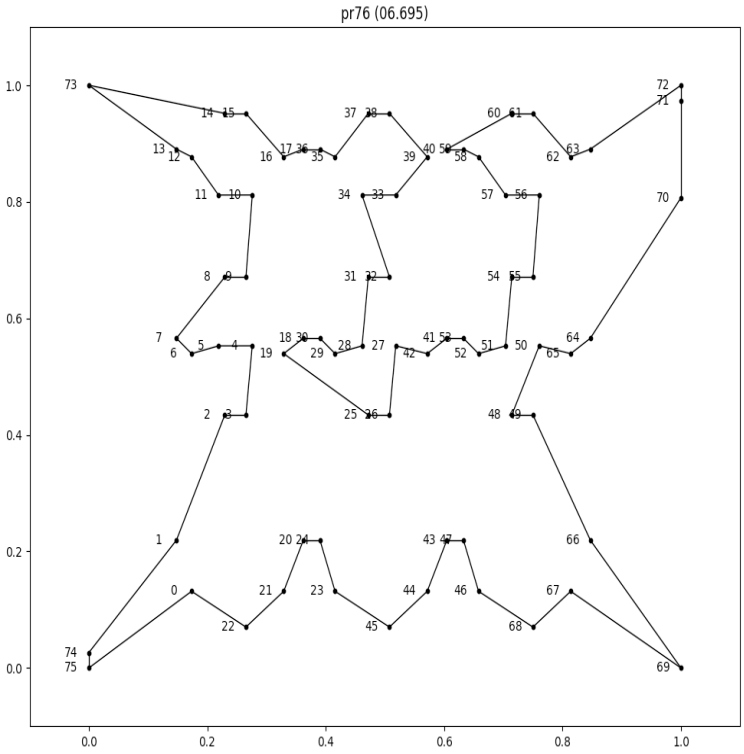
We were tasked to solve the Traveling Salesman Problem using the branch-and-cut method, utilizing Gurobi to solve only linear programming relaxations of integer programs.

Given a graph $G = (N, E)$ with nodes N and edges E, and an associated cost c_e for each edge, the goal of the TSP is to find the least costly path which visits each node exactly once and returns to the starting node (a Hamiltonian cycle).

The TSP can be formulated as the following integer program:

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

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. The first set of constraints in (1) ensures that each node is entered exactly once and then exited exactly once. This leaves open the possibilty of subtours, which are eliminated by the second set of constraints.



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.

```
## 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

The initial LP relaxation of (1) is:

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

where the decision variables x_e are now allowed to take any value between zero and one, and the subtour constraints have been removed (they will gradually be re-introduced as the algorithm progresses).

```
## 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
```

2.3 Model Processing

Models are processed using a variation of the branch-and-bound technique called branch-and-cut.

Once a model has been pulled from the model pool, it is optimized using gurobi.

If the model is infeasible, or if it cannot possibly yield a new best tour because its LP relaxation lower bound is greater than the current best tour cost, it is discarded (the 'bound' part of branch-and-cut) and processing stops.

If the current solution is a valid tour (and therefore integral) with a cost less than that of the current best tour, the current solution becomes the new best tour and processing stops.

Otherwise the current solution either non-integral or not a tour, so, if possible, new constraints are added to the model (the 'cut' part of branch-and-cut) and it is re-optimized. If no cuts could be added, new variable fixing models (where some non-integral x_e is forced to take on a value of 0 or 1) are created and returned to the caller (the 'branch' part of branch-and-cut).

```
## tsp_solver.py
class TspBranchAndCut(object):
    def process_model(self, model):
        new_model_info = []
        while True:
            model.update()
            model.optimize()

            if self.solution_is_infeasible(model):
                break

            if not self.solution_can_become_new_best(model):
```

```
                break

            if self.solution_is_tour(model):
                if self.solution_is_new_best(model):
                    self.update_best(model)
                break

            if self.add_cuts_to_model(model):
                continue

            branch_models = self.create_branch_models(model)
            for branch_model in branch_models:
                new_model_info.append( (model.getAttr('ObjVal'),
                    branch_model) )

            break

        return new_model_info
```

2.4 Adding Cuts (Constraints)

Several algorithms were developed for finding violated constraints and adding them to models. These algorithms are called sequentially, and once a new constraint is added, the model updated and re-optimized.

Details of the constraint-generating algorithms can be found in section 3.

```
## tsp_solver.py
class TspBranchAndCut(object):
    def add_cuts_to_model(self, model):
        constraints_were_added = False

        if self.add_comb_constraints(model):
            constraints_were_added = True
        elif self.add_integral_subtour_constraints(model):
            constraints_were_added = True
        elif self.add_nonintegral_subtour_constraints(model):
            constraints_were_added = True
        elif self.add_objective_constraints(model):
            constraints_were_added = True
        # elif self.add_gomory_constraints(model):
        #     constraints_were_added = True

        return constraints_were_added
```

2.5 Variable Fixing (Branching)

Variable fixing (branching) is implemented by finding the first non-integral solution variable x_e and creating two new models: one forcing x_e to take on exactly 0, and one forcing x_e to take on exactly 1. The newly-created models are then returned to the caller, where they will be added to the model pool for further processing.

```
## tsp_solver.py
class TspBranchAndCut(object):
    def create_branch_models(self, model):
        best_idx = best_var = best_val = None

        for idx,mvar in enumerate(model.getVars()):
            val = mvar.getAttr('X')
            if abs(val - int(val)) != 0.0:
                if (best_val is None) or (abs(val - 0.5) < best_val):
                    best_val = abs(val - 0.5)
                    best_var = mvar
                    best_idx = idx

        model1 = grb.Model.copy(model)
        m1var  = model1.getVarByName(best_var.getAttr('VarName'))
        model1.addConstr(m1var == 0.0)
        model1.update()

        model2 = grb.Model.copy(model)
        m2var  = model2.getVarByName(best_var.getAttr('VarName'))
        model2.addConstr(m2var == 1.0)
        model2.update()

        return (model1, model2)
```

3 Constraint Algoritms

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.

4 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.

5 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.

6 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.

7 Implementation

7.1 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

8 Results

att48

berlin52

gr21

hk48

ulysses22

pr76

st70

9 Improvements

10 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.

11 Section Title Here

11.1 Subsection Title Here

Some text here.

$$p(a \leq Z \leq b) = \int_a^b \phi(x)\,dx \tag{3}$$

Reference equation Equation 3 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.

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