

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:

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

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 possibility of subtours, which are eliminated by the second set of constraints.

2 Branch-and-Cut Control Code

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:

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

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'))
```

```

model11.addConstr(m1var == 0.0)
model11.update()

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

return (model11, model2)

```

3 Constraint (Cut) Algorithms

3.1 Subtour

The subtour constraints in the original TSP IP formulation (1) cannot be enumerated in practice for large TSP problems, which is why they are dropped in the initial LP relaxation (2).

Given a solution \mathbf{x}^* to an LP relaxation, it is possible to find and re-apply a violated subtour constraint by examining min-cuts on the resulting graph: if the min-cut is less than two, then some subtour constraint has been violated. By dividing the nodes into two subsets (S and $N \setminus S$) as determined by the edges in the min-cut, the following constraint can be constructed:

$$\sum_{e \in S} x_e \geq 2 \tag{3}$$

Two algorithms were developed to find cuts based on this property: One works for binary \mathbf{x}^* and the other works for any \mathbf{x}^* . Both utilize the same underlying graph utilities.

Subtour constraints are universal: they apply to any TSP model regardless of what other decisions (e.g., variable fixing) might have been made. As a result, when a subtour constraint is found, it is added to *all* models (not just the current model). This avoids searching for the identical constraint multiple times.

3.1.1 Integral

If \mathbf{x}^* is binary ($x_e \in \{0,1\} \forall e \in E$) then finding a min-cut is the same as finding a cut for some connected component (assuming there is more than one). This cut will obviously have value zero (since \mathbf{x}^* is binary), which is below the subtour constraint threshold of two.

```
## tsp_solver.py
class TspBranchAndCut(object):
    def add_integral_subtour_constraints(self, model):
        graph, xx = self.convert_model(model)
        connected_component_nodes, connected_component_edges =
            graph.binary_connected_components(solution=xx)

        if len(connected_component_nodes) == 1:
            return False

        for cur_model in all_models:
            mvars = cur_model.getVars()

            for cc in connected_component_nodes:
                ## translate the edge indices into model variables
                cut_edges = graph.get_cut_edges(nodes=cc)
                var_idxxs = sorted([self.idx_by_edge[edge] for edge in
                                    cut_edges])
                cvars = [mvars[idx] for idx in var_idxxs]
                coeffs = [1.0 for idx in var_idxxs]

                cur_model.addConstr(grb.quicksum(cvars) >= 2.0,
                                    'subtour-integral')

            constraints_were_added = True

        return constraints_were_added
```

3.1.2 Non-Integral

Instead of using connected components like the integral approach, this algorithm looks for any min-cut whose value is less than two. In fact, it accumulates *all* of the violated subtour constraints it encounters for the given solution.

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

```

def add_nonintegral_subtour_constraints(self, model):
    graph, xx = self.convert_model(model)

    modified_graph, xx = self.convert_model(model)
    all_cuts = []
    while still_adding_new_cuts:
        cur_cuts = modified_graph.find_min_cut()
        all_cuts.extend(cur_cuts)

        if cur_cuts[0][0] > 0.0:
            break

    new_nodes = set(modified_graph.nodes) - set(cur_cuts[0][1])
    new_edges = set()
    new_weights = {}
    for node in new_nodes:
        for edge in modified_graph.edges_by_node[node]:
            if (edge[0] in new_nodes) and (edge[1] in new_nodes):
                new_edges.add(edge)
                new_weights[edge] =
                    modified_graph.weight_by_edge[edge]

    modified_graph = Graph(nodes=new_nodes, edges=new_edges,
                           weight_by_edge=new_weights)

    all_cuts = sorted(all_cuts, key=lambda x: x[0])
    if len(all_cuts) == 0:
        return False
    if all_cuts[0][0] >= 2.0:
        return False

    for cur_model in all_models:
        for var_idx in non_dup_cuts:
            add_constr_to_model(model=cur_model, var_idx=var_idx)

    return constraints_were_added

```

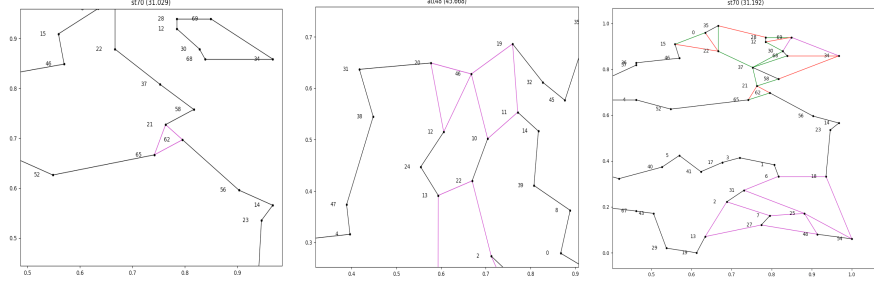
3.2 Comb (Blossom) Inequalities

Given an optimal tour on $G = (N, E)$, it can be shown that, for some 'handle' node set $H \subset N$ and 'tooth' edges $\{T_1, \dots, T_k\} = \delta(H)$, that:

$$\sum_{e \in \delta(H)} x_e + \sum_{i=1}^k \sum_{e \in \delta(T_i)} x_e \geq 3k + 1 \quad (4)$$

if k is odd.

Equation (4) is called the *comb inequality* and can be used to find subtour constraint violations.



3.3 Objective

In our particular flavor of the TSP, all edge costs c_e are integral. This implies that the objective value of a valid TSP tour must also be integral because all x_e decision variables must be integral.

This makes it possible to create a new constraint whenever the LP relaxation objective value is non-integral by just rounding it up:

$$\sum_{e \in E} c_e x_e \geq \left\lceil \sum_{e \in E} c_e x_e^* \right\rceil \quad (5)$$

where x_e^* are the current LP relaxation decision variables.

It turns out that gurobi allows constraints to be modified, so only one objective constraint is ever present at any given time. Also, because this constraint relies on locally-made decisions (like variable fixing), it is applied to the current model only. Some precautions were taken to avoid applying this constraint inappropriately due to floating-point rounding issues.

```
## tsp_solver.py
class TspBranchAndCut(object):
    def add_objective_constraints(self, model):
        obj_val = model.getAttr('ObjVal')
        ceil_obj_val = math.ceil(obj_val)
        floor_obj_val = math.floor(obj_val)

        if (abs(obj_val - floor_obj_val) < 1e-6) or (abs(obj_val -
            ceil_obj_val) < 1e-6):
            return False

        target_constr_name = 'objective-roundup'
```

```

target_constr = None
for constr in model.getConstrs():
    if constr.getAttr('ConstrName') == target_constr_name:
        target_constr = constr
        break

if target_constr is not None:
    target_constr.setAttr('RHS', ceil_obj_val)
else:
    mvars = model.getVars()

    var_idx = sorted([self.idx_by_edge[edge] for edge in
                      self.edges])
    cvars = [mvars[idx].getAttr('Obj')*mvars[idx] for idx in
             var_idx]

    model.addConstr(grb.quicksum(cvars) >= ceil_obj_val,
                    'objective-roundup')

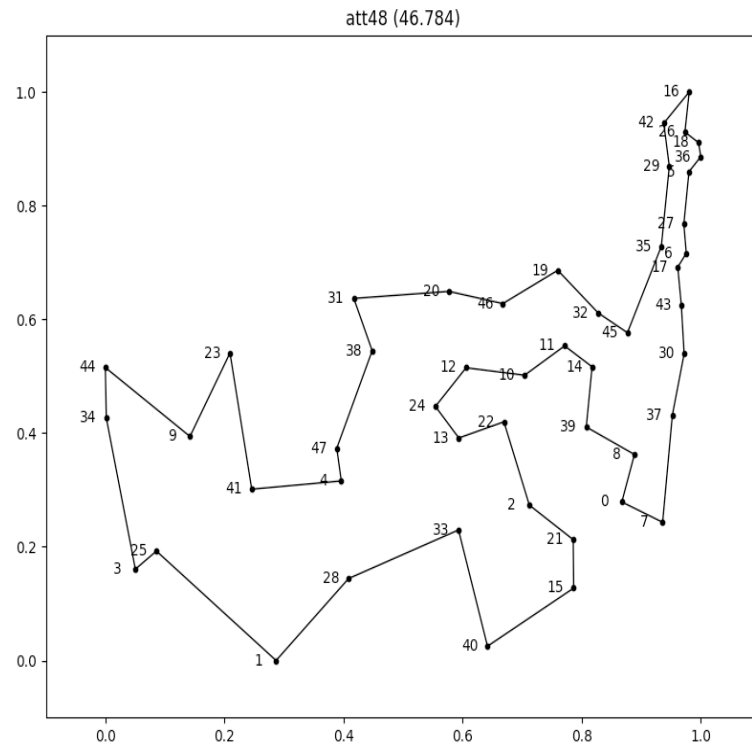
return True

```

3.4 Gomory

4 Results

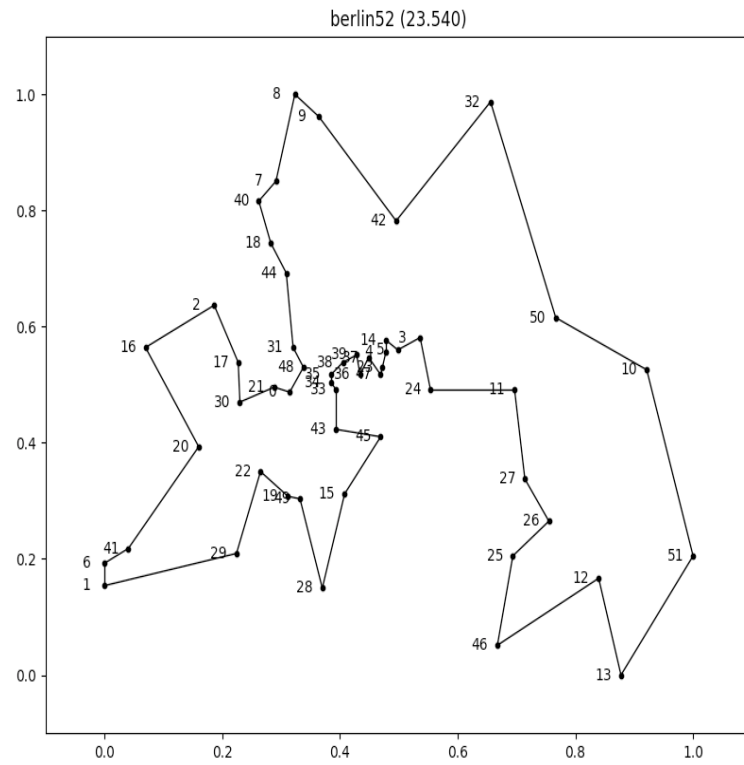
4.1 att48



0	7	178
7	37	312
37	30	183
30	43	139
43	17	112
17	6	54
6	27	85
27	5	153
5	36	66
36	18	42
18	26	64
26	16	117
16	42	139
42	29	127
29	35	233
35	45	286

```
45 32 134
32 19 207
19 46 246
46 20 225
20 31 393
31 38 169
38 47 317
47 4 97
4 41 370
41 23 403
23 9 292
9 44 401
44 34 146
34 3 451
3 25 102
25 1 585
1 28 381
28 33 474
33 40 356
40 15 393
15 21 140
21 2 207
2 22 262
22 13 192
13 24 133
24 12 169
12 10 242
10 11 186
11 14 129
14 39 175
39 8 214
8 0 147
The cost of the best tour is: 10628.0
```

4.2 berlin52



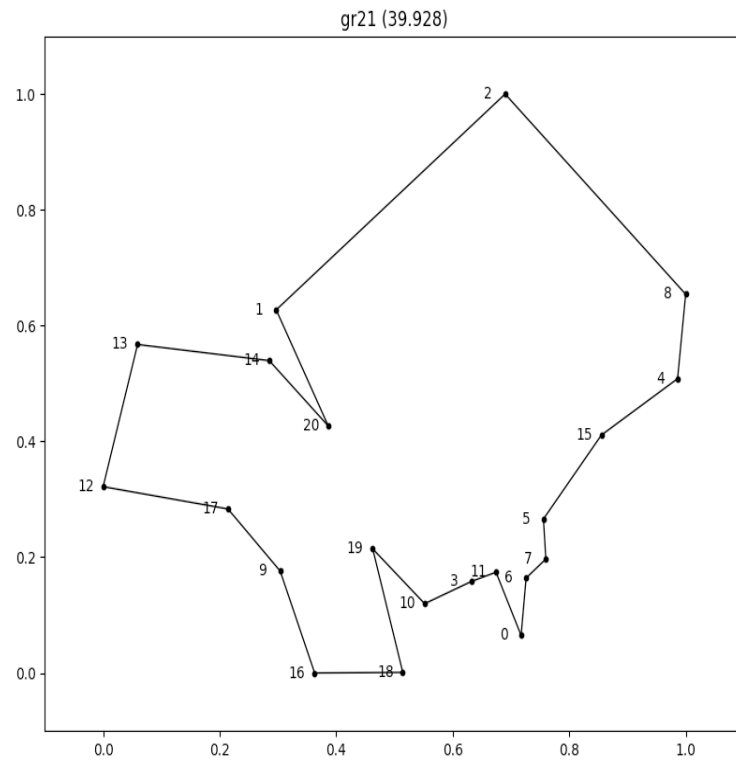
```

0  21  46
21 30 104
30 17  80
17  2 135
 2 16 217
16 20 253
20 41 290
41  6  76
 6  1  45
 1 29 390
29 22 179
22 19  94
19 49  35
49 28 191
28 15 201
15 45 156
45 43 131
43 33  80
33 34  21

```

```
34 35 15
35 38 43
38 39 43
39 36 41
36 37 43
37 47 49
47 23 16
23 4 32
4 14 25
14 5 40
5 3 70
3 24 109
24 11 245
11 27 182
27 26 110
26 25 126
25 46 186
46 12 324
12 13 206
13 51 319
51 10 399
10 50 285
50 32 475
32 42 365
42 9 308
9 8 83
8 7 183
7 40 64
40 18 92
18 44 75
44 31 151
31 48 50
48 0 64
The cost of the best tour is: 7542.0
```

4.3 gr21



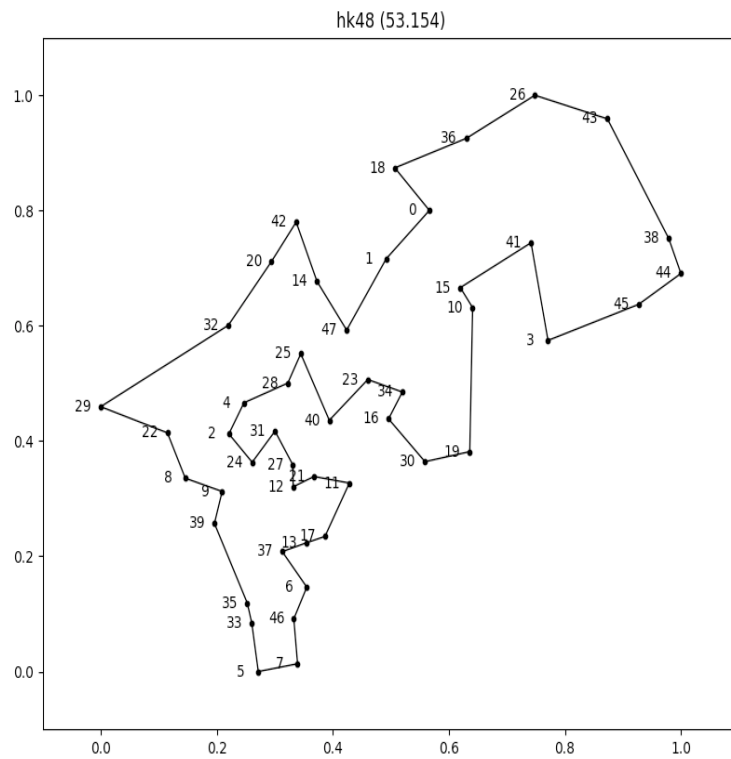
0	6	110
6	7	29
7	5	36
5	15	125
15	4	125
4	8	120
8	2	295
2	1	355
1	20	140
20	14	105
14	13	170
13	12	190
12	17	180
17	9	77
9	16	150
16	18	87
18	19	155
19	10	100
10	3	63

```

3  11 27
11 0  68
The cost of the best tour is: 2707.0

```

4.4 hk48



```

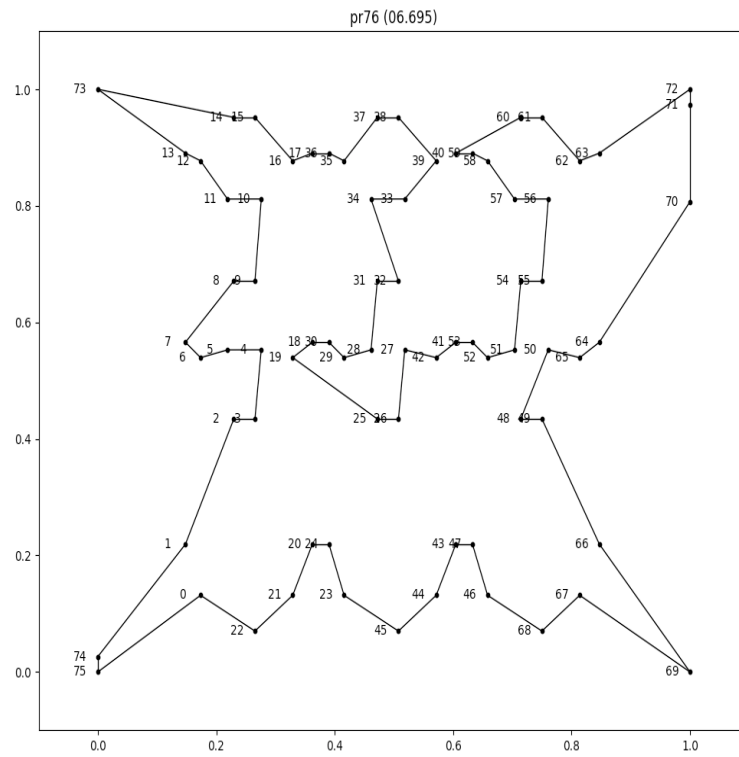
0  1  273
1  47 335
47 14 236
14 42 252
42 20 189
20 32 318
32 29 669
29 22 326
22 8  197
8  9  177
9  39 130
39 35 345

```


35	33	83
33	5	188
5	7	182
7	46	177
46	6	138
6	37	178
37	13	115
13	17	90
17	11	238
11	21	165
21	12	100
12	27	90
27	31	154
31	24	161
24	2	155
2	4	140
4	28	217
28	25	129
25	40	291
40	23	238
23	34	166
34	16	123
16	30	235
30	19	214
19	10	563
10	15	96
15	41	371
41	3	387
3	45	443
45	44	229
44	38	145
38	43	534
43	26	347
26	36	357
36	18	346
18	0	229

The cost of the best tour is: 11461.0

4.5 pr76



```

0  22 1931
22 21 1433
21 20 1193
20 24 550
24 23 1118
23 45 1931
45 44 1433
44 43 1193
43 47 550
47 46 1118
46 68 1931
68 67 1433
67 69 3946
69 66 3905
66 49 3100
49 48 700
48 50 1629
50 65 1053
65 64 716

```

64	70	4070
70	71	1900
71	72	300
72	63	3250
63	62	667
62	61	1512
61	60	700
60	40	2261
40	59	550
59	58	522
58	57	1164
57	56	1118
56	55	1617
55	54	700
54	51	1364
51	52	906
52	53	583
53	41	550
41	42	716
42	27	1053
27	26	1369
26	25	700
25	19	3046
19	18	716
18	30	550
30	29	583
29	28	906
28	31	1364
31	32	700
32	34	1841
34	33	1118
33	39	1280
39	38	1512
38	37	700
37	35	1390
35	36	522
36	17	550
17	16	667
16	15	1512
15	14	700
14	73	4533
73	13	3158
13	12	522
12	11	1164
11	10	1118
10	9	1617
9	8	700
8	7	2000
7	6	583
6	5	906

50 55 9
55 64 8
64 63 20
63 10 24
10 47 9
47 66 5
66 53 13
53 61 11
61 32 18
32 33 14
33 20 1
20 11 8
11 59 9
59 51 9
51 9 6
9 4 6
4 52 9
52 5 20
5 40 6
40 42 11
42 16 4
16 8 16
8 39 11
39 60 5
60 38 9
38 44 7
44 24 8
24 45 19
45 26 6
26 67 21
67 43 4
43 29 15
29 19 7
19 13 7
13 27 14
27 48 14
48 54 8
54 25 16
25 7 8
7 2 12
2 31 6
31 41 13
41 17 6
17 3 4
3 1 9
1 6 5
6 18 11
18 23 20
23 14 4
14 56 7

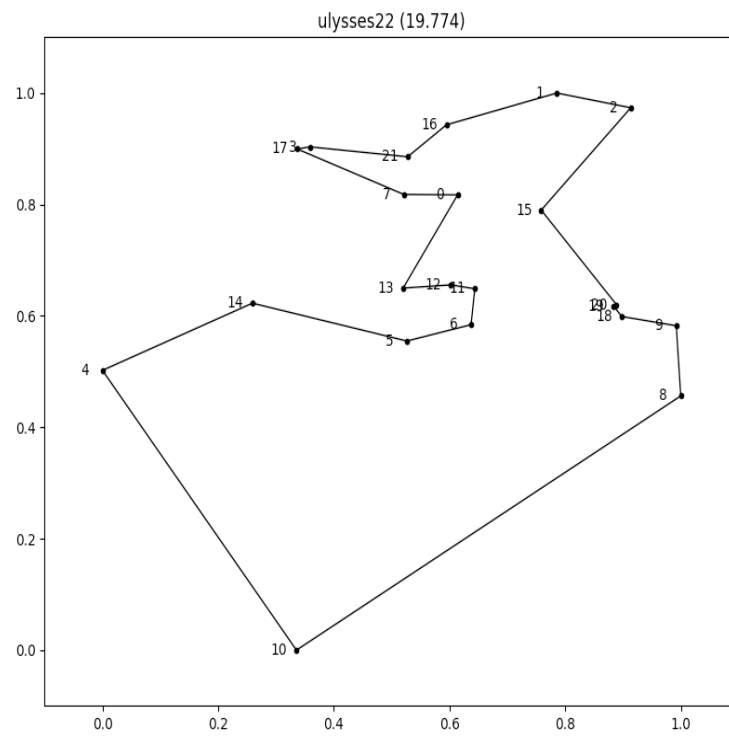
```

56 62 14
62 65 6
65 21 6
21 58 6
58 37 8
37 30 10
30 68 2
68 34 12
34 69 14
69 12 6
12 28 2
28 35 12
35 0 4

```

The cost of the best tour is: 675.0

4.7 ulysses22



0 7 60

```

7  17  278
17  3   37
3  21  171
21  16  148
16  1   246
1  2   126
2  15  499
15  20  486
20  19   14
19  18   33
18  9   96
9  8   328
8  10  1387
10  4  1504
4  14  401
14  5   308
5  6   115
6  11  177
11  12   68
12  13   52
13  0   479
The cost of the best tour is: 7013.0

```

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.

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.

$$p(a \leq Z \leq b) = \int_a^b \phi(x) dx \tag{6}$$

Reference equation Equation 6 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.

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