Report of project 1

尚尔淦 (PB19010390)

Contents

Contents 1
1 introduction 1
2 Input and output 2
2.1 Lazy Cut 2
2.2 User Cut 3
3 Comparison between User Cut and Lazy Cut 4

1 introduction

Traveling salesman problem(TSP) asks the following question: Given a list of cities and distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

We can reformulate the problem as follows: In essence, this is to find a circuit with the smallest weight in a weighted undirected graph. And we can translate it into a constrained optimization problem via branch and bound algorithm using both lazy cut and user cut.

We define the decision variable x_{ij} as

$$x_{ij} = \begin{cases} 1 & \text{if goes from i to j} \\ 0 & \text{otherwise} \end{cases} (i,j) \in A$$

where A is the set of edges.

The objective function can be written as

$$\min \sum_{(i,j)\in A} c_{ij} x_{ij}$$

.Considering the balance between input and output of each node, we have the first two constraints:

- $\sum_{j\in V,(i,j)\in A} x_{ij} = 1 \quad \forall i \in V$
- $\sum_{i\in V,(i,j)\in A} x_{ij} = 1 \quad \forall j \in V$

where V is the sets of node.

However, if we only impose the two constraints above, the solution can be suffered from subtours, which destroy the optimality.

In order to eliminate subtours, we impose the following constraint:

$$\sum_{j\notin S, i\in S,(i,j)\in A} x_{ij} \geq 1 \qquad \sum_{i,j\in S,(i,j)\in A} x_{ij} \leq |S|-1 \quad \forall S \subset V, 2 \leq |S| \leq n-1$$

which means any node subset has edges less than its cardinality and must have edges connecting its complement set. We need to eliminate subtours through **lazy cut** and **user cut** attempting to add this constraint dynamically since numbers of this kind of constraints are $O(n^2)$.

It is easy to make a callback via lazy cut, however, we need to have a deeper insight into user cut solving a minimum cut problem:

If we denote edge ij to be chosen or not(we do not discriminate ij and ji) by $x_{[ij]}$, leading to $\frac{n(n-1)}{2}$ variables. We have the following problem

$$\begin{aligned} &\min \sum_{i,j\in V} c_{[ij]} x_{ij} \\ &\text{such that} \sum_{j\in V, j\neq i} x_{[ij]} = 2 \quad \forall i \in V \\ &x_{[ij]} \in \{0,1\} \end{aligned}$$

So that we need to find a set S_1 such that $\sum_{i\in S_i, j\notin S_1} x_{[ij]} \geq 2$. It is intuiitive to consider partitioning the nodes into two disconnected sets with weight of edges between two set attaing the minimum. As a result it will be easier to find the subtour violating the third constraint.

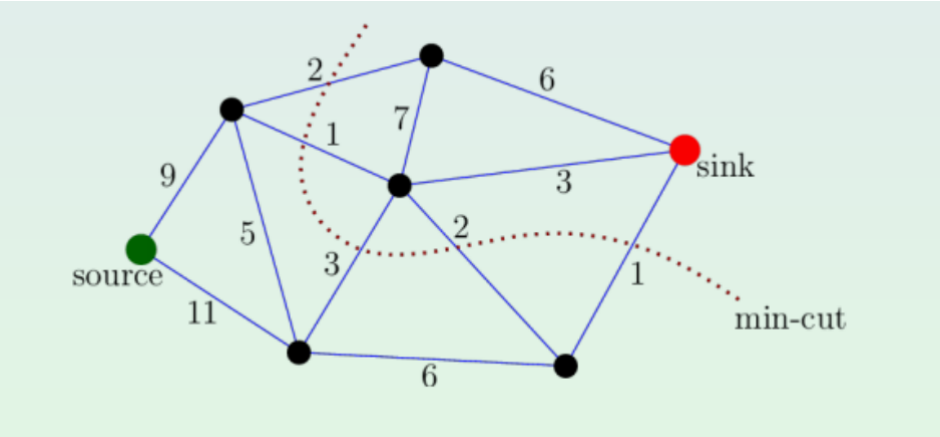


Figure 1. minimum cut

2 Input and output

2.1 Lazy Cut

Lazy cut is realized in the file "Project 1-lazy.ipynb" and please compile the function **distance, subtour, subtoureli** first.

Considering the data set uploaded by TA, we just write the following codes(using example of 20 points):

```
load C:/Users/mac/Desktop/Optimization/TSP data/20_0.py
```

As for the output, we refer the following codes:

```
city=list(range(len(points)))
coordinates={} ##dictionary
for i in city:
    coordinates[i]=(float(points[i][0]),float(points[i][1]))
dist={(c1,c2):distance(c1,c2) for c1,c2 in combinations(city,2)}
m=gp.Model()
vars=m.addVars(dist.keys(),obj=dist,vtype=GRB.BINARY,name='x')
for i,j in vars.keys():
    vars[j,i]=vars[i,j]
cons=m.addConstrs(vars.sum(c,'*')==2 for c in city)
m._vars=vars# vars see establish model
m.Params.lazyConstraints=1
%time m.optimize(subtoureli)
vals=m.getAttr('x',vars)
selected=gp.tuplelist((i,j) for i,j in vals.keys() if vals[i,j]>0.5)
tour=subtour(selected)
assert len(tour)==len(city)
tour
```

where **function distance** calculates Euclidean distance between points. **Function subtoureli** is the function(main) to optimize. Finally, we use **function subtour** to get the optimal route and **assert len(tour)==len(city)** is to test whether the optimal route is feasible and the tour is (using the example of 100_2): 0,27,23,49,9,72,13,35,82,84,25,41,43,69,34,4,17,74,76,1,87,48,30,66,3,79,18,97, 77,32,38,52,11,99,36,83,7,29,53,40,55,45,75,94,5,33,80,19,90,2,63,6,64,15,89,51,91,96,92,78,26,56,57, 54,65,37,85,39,10,81,95,24,20,86,71,16,73,68,93,58,21,59,44,70,61,22,46,31,8,12,47,50,42,60,67,14,98,62,28,88. (This result can be seen in the codes I upload).

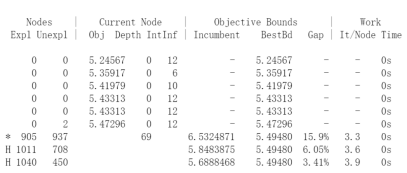
2.2 User Cut

User cut is realized in the file "project-user.ipynb" in the file "project 1-user cut" and please compile the function **ipTSP**(the main function) first.

As for the input, we use the following codes to change the data structure(changing list into dictionary):

```
from tspData.tsp20_2 import points
a = list(range(len(points)))
points_dict=dict(zip(a,points))
```

Then for output, we singly use the main function **ipTSP**:



Nodes		Current Node			Objective Bounds			Work	
Expl	Unexpl	Obj	Depth	IntInf	Incumbent	BestBd	Gap	It/Node	Time
0	0	5.24567	0	12	-	5.24567	-	-	0s
0	0	5.35917	0	6	-	5.35917	-	-	0s
0	0	5.41979	0	10	-	5.41979	-	-	0s
0	0	5.43313	0	12	-	5.43313	-	-	0s
0	0	5.43313	0	12	-	5.43313	-	-	0s
0	2	5.47296	0	12	-	5.47296	-	-	0s
* 905	937		69		6.5324871	5.49480	15.9%	3.3	0s
H 1011	708				5.8483875	5.49480	6.05%	3.6	0s
H 1040	450				5.6888468	5.49480	3.41%	3.9	0s

Fig. 2. 60_2 Lazy 0s

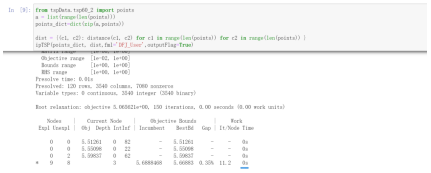
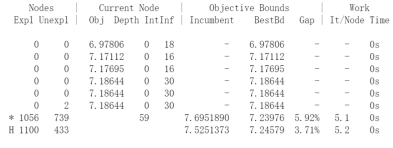


Fig. 3. 60_2 User 0s



Nodes		Current Node			Objective Bounds			Work	
Expl	Unexpl	Obj	Depth	IntInf	Incumbent	BestBd	Gap	It/Node	Time
0	0	6.97806	0	18	-	6.97806	-	-	0s
0	0	7.17112	0	16	-	7.17112	-	-	0s
0	0	7.17695	0	16	-	7.17695	-	-	0s
0	0	7.18644	0	30	-	7.18644	-	-	0s
0	0	7.18644	0	30	-	7.18644	-	-	0s
0	2	7.18644	0	30	-	7.18644	-	-	0s
* 1056	739		59		7.6951890	7.23976	5.92%	5.1	0s
H 1100	433				7.5251373	7.24579	3.71%	5.2	0s

Fig. 4. 100_1 Lazy 0s

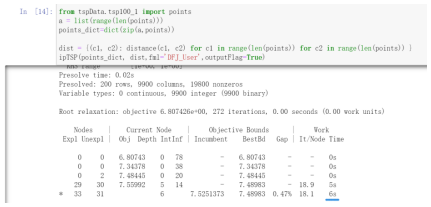


Fig. 5. 100_1 User 6s

```
dist = {(c1, c2): distance(c1, c2) for c1 in range(len(points)) for c2 in
range(len(points)) }
ipTSP(points_dict, dist,fml='DFJ_User',outputFlag=True)
```

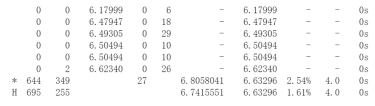
where **function distance** calculates the distance between points. And we also display the example of 100_2:0,88,28,62,98,14,
67,60,42,50,47,12,8,31,46,22,61,70,44,59,21,58,93,68,73,16,71,86,20,24,95,81,10,39,85,37,65,54,57,56,26,78,92,96,
91,51,89,15,64,6,63,2,90,19,80,33,5,94,75,45,55,40,53,29,7,83,36,99,11,52,38,32,77,97,18,79,3,66,30,48,87,1,76,74,
17,4,34,69,43,41,25,84,82,35,13,72,9,49,23,27,0.(This result can also be seen in the codes I upload.)

In fact this result is the same as the result in Lazy cut where the only difference is the ergodic order.

3 Comparison between User Cut and Lazy Cut

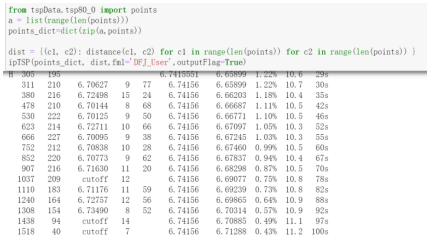
Although intuitively, the user cut should be faster than lazy cut, the fact is that lazy cut always achieves the solving time nearly zero while user cut has a volatile solving time. Now I demonstrate those facts by displaying part of outputs.(See the figures at the top and bottom of this page)

From all these comparisons, we realize that user cut does not necessarily help solve the problem faster. Instead, solving time of user cut can change volatilely, from which we can see that in 80



0	0	6.17999	0	6	-	6.17999	-	-	0s
0	0	6.47947	0	18	-	6.47947	-	-	0s
0	0	6.49305	0	29	-	6.49305	-	-	0s
0	0	6.50494	0	10	-	6.50494	-	-	0s
0	0	6.50494	0	10	-	6.50494	-	-	0s
0	2	6.62340	0	26	-	6.62340	-	-	0s
* 644	349		27		6.8058041	6.63296	2.54%	4.0	0s
H 695	255				6.7415551	6.63296	1.61%	4.0	0s

Fig. 6. 80_1 Lazy 0s



H 905	196				6.7415551	6.63899	1.22%	10.6	25s
311	210	6.70627	9	77	6.74156	6.68298	0.87%	10.7	30s
380	216	6.72498	15	24	6.74156	6.66203	1.18%	10.4	35s
478	210	6.70144	8	68	6.74156	6.66687	1.11%	10.5	42s
530	222	6.70125	9	50	6.74156	6.66771	1.10%	10.5	46s
623	214	6.72711	10	66	6.74156	6.67097	1.05%	10.3	52s
666	227	6.70095	9	38	6.74156	6.67245	1.03%	10.3	55s
752	212	6.70838	10	28	6.74156	6.67460	0.99%	10.5	60s
852	220	6.70773	9	62	6.74156	6.67837	0.94%	10.4	67s
907	216	6.71630	11	20	6.74156	6.68298	0.87%	10.5	70s
1037	209	cutoff	12		6.74156	6.69077	0.75%	10.8	78s
1110	183	6.71176	11	59	6.74156	6.69239	0.73%	10.8	82s
1240	164	6.72757	12	56	6.74156	6.69865	0.64%	10.9	88s
1308	154	6.73490	8	52	6.74156	6.70314	0.57%	10.9	92s
1438	94	cutoff	14		6.74156	6.70885	0.49%	11.1	97s
1518	40	cutoff	7		6.74156	6.71288	0.43%	11.2	100s

Fig. 7. 80_0 User 100s

points the time is 100 seconds while in 100 points the time is 6 seconds. However, for all the problems we display (and all problems in data sets TA provided), solving time of lazy cut is always nearly 0 seconds.

In summary:

- Solving time of user cut is more volatile than lazy cut and user cut is not necessarily faster than lazy cut.
- Solving time of lazy cut is more stable and is nearly 0 second for problems we met.