

Solving Small TSP Instances

You'll Clean That Up Before You Leave

Shawn Brunstig, Ian Dimock, Yuguang Zhang
University of Waterloo

March 19, 2015

1 Introduction

2 Methods

2.1 Enumeration

2.2 Held-Karp 1-Trees

2.3 Bellman-Held-Karp Dynamic Programming

2.4 Subtour Branch and Cut

3 Testing Environment

4 Test Data

5 Discussion

6 Conclusions

Appendix

Result Tables

Code

```
1  ../onetree.h  
2  #ifndef ONETREE_H  
3  #define ONETREE_H  
4  #include <vector>  
5  #include <set>  
6  
7  class branch_node {
```

```

8 public:
9     std::set<int> X, Y;
10    int * pi;
11    int w;
12    int ncount;
13
14    branch_node(std::set<int> X, std::set<int> Y, int ncount, int *pi, int w);
15    branch_node(const branch_node& other);
16    branch_node & operator=(const branch_node& other);
17    ~branch_node();
18 };
19
20 bool operator<(const branch_node & l, const branch_node & r);
21
22 int one_tree_tsp(int ncount, int ecount, int *elist, int *elen, int upper_bound);
23 int w(int ncount, int ecount, int *elist, int *elen, std::set<int> X, std::set<int> Y, int * pi,
24 bool update_pi, std::vector<int> * deg_not_2, std::vector<int> * tree_edges);
25 int w_candidate(int ncount, int ecount, int *elist, int *elen, int * elen_new, int ignore, int * pi, int
26 std::set<int> X, std::set<int> Y, std::vector<int> * tree_edges);
27
28 #endif

```

../onetree.cpp

```

1  #include "onetree.h"
2  #include "edge.h"
3  #include "graph.h"
4  #include <vector>
5  #include <algorithm>
6  #include <iostream>
7  #include <cstring>
8  #include <set>
9  #include <queue>
10 #include <cassert>
11
12 const int t_bar = 1;
13 const int p = 25;
14
15 bool do_print_tour = false;
16
17 branch_node::branch_node(std::set<int> X, std::set<int> Y, int ncount, int *pi, int w) {
18     this->X = X;
19     this->Y = Y;
20     this->pi = new int[ncount];
21     assert( pi != 0 );
22     memcpy(this->pi, pi, ncount*sizeof(int));
23     this->w = w;
24     this->ncount = ncount;
25 }
26
27 branch_node::branch_node(const branch_node& other) {
28     this->X = other.X;
29     this->Y = other.Y;
30     this->pi = new int[other.ncount];
31     assert( other.pi != 0 );
32     memcpy(this->pi, other.pi, other.ncount*sizeof(int));
33     this->w = other.w;
34     this->ncount = other.ncount;
35 }
36
37 branch_node & branch_node::operator=(const branch_node& other) {
38     this->X = other.X;

```

```

39     this->Y = other.Y;
40     assert( other.pi != 0 );
41     memcpy(this->pi, other.pi, other.ncount*sizeof(int));
42     this->w = other.w;
43     this->ncount = other.ncount;
44
45     return (*this);
46 }
47
48 branch_node::~branch_node() {
49     if( pi ) {
50         delete [] pi; pi = 0; // Not sure why this causes crashes !?
51     }
52 }
53
54 bool operator<(const branch_node & l, const branch_node & r) {
55     return l.w > r.w; // Because we want a min heap we flip the sign
56 }
57
58 int one_tree_tsp(int ncount, int ecount, int *elist, int *elen, int upper_bound) {
59     int branches = 0;
60     int *pi = new int[ncount];
61     int bound, max_w = -100000, max_w_p_ago = -100000;
62     int *pi_prime = new int[ncount];
63     bool do_branch;
64     std::priority_queue<branch_node> Q;
65     std::vector<int> edges;
66
67     for( int i = 0; i < ncount; i++ ) pi[i] = 0;
68
69
70     std::vector<int> w_cache;
71     w_cache.reserve(p);
72
73     std::set<int> X, Y;
74     std::vector<int> deg_not_2, deg_not_2_prime;
75
76     bound = w(ncount, ecount, elist, elen, X, Y, pi, false, &deg_not_2, &edges);
77
78     branch_node start = branch_node(X,Y,ncount,pi,bound);
79     Q.push(start);
80
81     while( Q.size() != 0 ) {
82         branch_node current = Q.top();
83         bound = current.w;
84         // std::cout << bound << std::endl;
85
86         if( bound > upper_bound ) {
87             // std::cout << "bad upper bound" << std::endl;
88             // std::cout << "pi: "; for( int i = 0; i < ncount; i++ ) std::cout << current.pi[i] << " ";
89             Q.pop(); branches++;
90             continue;
91         }
92
93         memcpy(pi, current.pi, ncount*sizeof(int));
94         X = current.X;
95         Y = current.Y;
96
97         Q.pop(); branches++;
98
99         std::set<int>::iterator it;
100         // std::cout << "Popped (" << current.w << "), branches: " << branches << ", Q size: " << Q.size();
101         // std::cout << "pi: "; for( int i = 0; i < ncount; i++ ) std::cout << pi[i] << " "; std::cout << "\n";
102         // std::cout << "X: "; for( it = X.begin(); it != X.end(); it++ ) std::cout << (*it) << " "; std::cout << "\n";
103         // std::cout << "Y: "; for( it = Y.begin(); it != Y.end(); it++ ) std::cout << (*it) << " "; std::cout << "\n";
104

```

```

105     do_branch = true;
106     max_w = -100000;
107     max_w_p_ago = -100000;
108     deg_not_2.erase(deg_not_2.begin(),deg_not_2.end());
109     bound = w(ncount, ecount, elist, elen, X, Y, pi, false, &deg_not_2, &edges);
110     if( deg_not_2.size() == 0 ) {
111         int last = elist[2*edges[0]];
112         std::cout << "One_Tree_Tour:" << last;
113         edges.erase(edges.begin(),edges.begin()+1);
114         for( int j = 0; j < ncount-1; j++ ) {
115             unsigned i;
116             for( i = 0; i < edges.size(); i++ ) {
117                 int edge = edges[i];
118                 if( elist[2*edge] == last ) {
119                     last = elist[2*edge+1];
120                     std::cout << " " << last;
121                     break;
122                 } else if( elist[2*edge+1] == last ) {
123                     last = elist[2*edge];
124                     std::cout << " " << last;
125                     break;
126                 }
127             }
128             edges.erase(edges.begin()+i,edges.begin()+i+1);
129         }
130         std::cout << std::endl << "Length:" << bound << std::endl;
131
132         delete [] pi; pi = 0;
133         delete [] pi_prime; pi_prime = 0;
134         return bound;
135     }
136
137     for(int i = 0; ; i++) {
138         deg_not_2.erase(deg_not_2.begin(),deg_not_2.end());
139         bound = w(ncount, ecount, elist, elen, X, Y, pi, true, &deg_not_2, &edges);
140         // std::cout << "pi: "; for( int j = 0; j < ncount; j++ ) std::cout << pi[j] << " "; std::cout << std::endl;
141
142         if( i >= p ) {
143             if( w_cache[i%p] > max_w_p_ago ) {
144                 max_w_p_ago = w_cache[i%p];
145             }
146             w_cache[i%p] = bound;
147             if( bound > max_w ) {
148                 max_w = bound;
149                 memcpy(pi_prime, pi, ncount*sizeof(int));
150                 deg_not_2_prime = deg_not_2;
151             }
152             if( bound > upper_bound ) {
153                 do_branch = false;
154                 break;
155             }
156             if( max_w_p_ago == max_w ) {
157                 break;
158             }
159         }
160     }
161     // std::cout << "iterated pi" << std::endl;
162     if( deg_not_2_prime.size() == 0 ) {
163         // std::cout << "FOUND TOUR 2" << std::endl;
164         // std::cout << "branches: " << branches << std::endl;
165         bound = w(ncount, ecount, elist, elen, X, Y, pi_prime, false, &deg_not_2, &edges);
166         Q.push(branch_node(X,Y,ncount,pi_prime,bound));
167     }
168     if( do_branch ) {
169         bool done = false;
170         for( unsigned i = 0; i < deg_not_2_prime.size(); i++ ) {

```

```

171         for( int j = 0; j < ecoun; j++ ) {
172             if( (elist[2*j] == deg_not_2_prime[i] || elist[2*j+1] == deg_not_2_prime[i]) && X.find(
173                 std::set<int> X_new = X;
174                 std::set<int> Y_new = Y;
175                 X_new.insert(j);
176                 Y_new.insert(j);
177                 bound = w(ncount, ecoun, elist, elen, X_new, Y, pi_prime, false, &deg_not_2, &edges);
178                 Q.push(branch_node(X_new,Y,ncount,pi_prime,bound));
179                 bound = w(ncount, ecoun, elist, elen, X, Y_new, pi_prime, false, &deg_not_2, &edges);
180                 Q.push(branch_node(X,Y_new,ncount,pi_prime,bound));
181                 done = true;
182                 break;
183             }
184         }
185         if( done ) {
186             break;
187         }
188     }
189 }
190
191 }
192
193 delete [] pi; pi = 0;
194 delete [] pi_prime; pi_prime = 0;
195 return upper_bound;
196 }
197
198 int w(int ncount, int ecoun, int *elist, int *elen, std::set<int> X, std::set<int> Y, int * pi,
199 bool update_pi, std::vector<int> * deg_not_2, std::vector<int> * tree_edges) {
200     int min_w = 1000000, bound, i, ignore;
201     int * v = new int[ncount], *min_v = new int[ncount], * elen_new = new int[ecoun];
202     std::vector<int> edges;
203
204     for( i = 0; i < ecoun; i++ ) elen_new[i] = elen[i] + pi[elist[2*i]] + pi[elist[2*i+1]];
205
206     for( ignore = 0; ignore < ncount; ignore++ ) {
207         edges.erase(edges.begin(), edges.end());
208         bound = w_candidate(ncount, ecoun, elist, elen, elen_new, ignore, pi, v, X, Y, &edges); //change
209         if( bound <= min_w ) {
210             min_w = bound;
211             memcpy(min_v, v, ncount*sizeof(int));
212             (*tree_edges) = edges;
213         }
214     }
215
216     // Update pi
217     for( i = 0; i < ncount; i++ ) {
218         if( min_v[i] != 0 ) {
219             deg_not_2->push_back(i);
220         }
221         if( update_pi ) {
222             pi[i] += min_v[i]*t_bar;
223         }
224     }
225
226     delete [] v;
227     delete [] min_v;
228     delete [] elen_new;
229
230     return min_w;
231 }
232
233 }
234
235 int w_candidate(int ncount, int ecoun, int *elist, int * elen, int *elen_new, int ignore, int * pi,
236 int * v, std::set<int> X, std::set<int> Y, std::vector<int> * tree_edges) {

```

```

237     std::vector<edge> edges;
238     int SMALL_LEN = -1000000;
239
240     std::set<int>::iterator it;
241
242     // sort only edges incident to node ignoring
243     for( int i = 0; i < ecount; i++ ) {
244         if(elist[2*i] == ignore || elist[2*i+1] == ignore) {
245             if( X.find(i) != X.end() ) {
246                 edges.push_back(edge(elist[2*i], elist[2*i+1], SMALL_LEN, i));
247             } else {
248                 edges.push_back(edge(elist[2*i], elist[2*i+1], elen_new[i], i));
249             }
250         }
251     }
252     std::sort(edges.begin(), edges.end());
253
254     int ecount_sub = ecount - edges.size();
255
256     int ncount_sub = ncount - 1;
257     int * elist_sub = new int[ecount_sub*2];
258     int * elen_sub = new int[ecount_sub];
259     std::vector<int> must_include;
260     int * orig_ind = new int[ecount];
261
262     int j = 0;
263     for( int i = 0; i < ecount; i++ ) {
264         if(elist[2*i] != ignore && elist[2*i+1] != ignore ) {
265             if( Y.find(i) != Y.end() ) {
266                 ecount_sub--;
267                 continue;
268             }
269             if( elist[2*i] > ignore ) {
270                 elist_sub[2*j] = elist[2*i]-1;
271             } else {
272                 elist_sub[2*j] = elist[2*i];
273             }
274             if( elist[2*i+1] > ignore ) {
275                 elist_sub[2*j+1] = elist[2*i+1]-1;
276             } else {
277                 elist_sub[2*j+1] = elist[2*i+1];
278             }
279             if( X.find(i) != X.end() ) {
280                 must_include.push_back(j);
281             }
282
283             elen_sub[j] = elen_new[i];
284             orig_ind[j] = i;
285             j++;
286         }
287     }
288
289     graph G;
290     G.init(ncount_sub, ecount_sub, elist_sub, elen_sub);
291
292     std::vector<int> mst = G.min_spanning_tree(must_include);
293
294     int tot = 0;
295     for( unsigned i = 0; i < mst.size(); i++ ) {
296         tot += elen[orig_ind[mst[i]]];
297         tree_edges->push_back(orig_ind[mst[i]]);
298     }
299     if ( (int)mst.size() != ncount_sub - 1 ) {
300         delete [] elist_sub;
301         delete [] elen_sub;
302         delete [] orig_ind;

```

```

303
304         return 1000000000;
305     }
306
307     for( int i = 0; i < ncount; i++ ) v[i] = 0;
308
309     tot += elen[edges[0].ind];
310     tot += elen[edges[1].ind];
311
312     tree_edges->push_back(edges[0].ind);
313     tree_edges->push_back(edges[1].ind);
314
315     if( edges[0].end1 == ignore ) {
316         v[edges[0].end2] += 1;
317     } else {
318         v[edges[0].end1] += 1;
319     }
320     if( edges[1].end1 == ignore ) {
321         v[edges[1].end2] += 1;
322     } else {
323         v[edges[1].end1] += 1;
324     }
325     int end1, end2;
326     v[ignore] = 2;
327
328
329
330     for( int i = 0; i < ncount_sub-1; i++ ) {
331         end1 = elist[2*orig_ind[mst[i]]];
332         end2 = elist[2*orig_ind[mst[i]]+1];
333         v[end1]++;
334         v[end2]++;
335     }
336
337     for( int i = 0; i < ncount; i++ ) v[i] -= 2;
338
339     for( int i = 0; i < ncount; i++ ) {
340         tot += pi[i] * v[i];
341     }
342
343     if( do_print_tour ) {
344         std::cout << edges[0].ind << "␣" << edges[1].ind;
345         for( unsigned i = 0; i < mst.size(); i++ ) {
346             std::cout << "␣" << orig_ind[mst[i]];
347         }
348         std::cout << "␣--␣";
349         for( int i = 0; i < ncount; i++ ) std::cout << v[i] << "␣";
350         std::cout << std::endl;
351     }
352
353     delete [] elist_sub;
354     delete [] elen_sub;
355     delete [] orig_ind;
356
357     return tot;
358 }

```