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

../onetree.h

```
#ifndef ONETREE_H
#define ONETREE_H

#include <vector>
#include <set>

class branch_node {
```

```cpp
public:
    std::set<int> X, Y;
    int * pi;
    int w;
    int ncount;

    branch_node(std::set<int> X, std::set<int> Y, int ncount, int *pi, int w);
    branch_node(const branch_node& other);
    branch_node & operator=(const branch_node& other);
    ~branch_node();
};

bool operator<(const branch_node & l, const branch_node & r);

int one_tree_tsp(int ncount, int ecount, int *elist, int *elen, int upper_bound);
int w(int ncount, int ecount, int *elist, int *elen, std::set<int> X, std::set<int> Y, int * pi,
    bool update_pi, std::vector<int> * deg_not_2, std::vector<int> * tree_edges);
int w_candidate(int ncount, int ecount, int *elist, int *elen, int * elen_new, int ignore, int * pi, int
        std::set<int> X, std::set<int> Y, std::vector<int> * tree_edges);

#endif
```

../onetree.cpp

```cpp
#include "onetree.h"
#include "edge.h"
#include "graph.h"
#include <vector>
#include <algorithm>
#include <iostream>
#include <cstring>
#include <set>
#include <queue>
#include <cassert>

const int t_bar = 1;
const int p = 25;

bool do_print_tour = false;

branch_node::branch_node(std::set<int> X, std::set<int> Y, int ncount, int *pi, int w) {
    this->X = X;
    this->Y = Y;
    this->pi = new int[ncount];
    assert( pi != 0 );
    memcpy(this->pi, pi, ncount*sizeof(int));
    this->w = w;
    this->ncount = ncount;
}

branch_node::branch_node(const branch_node& other) {
    this->X = other.X;
    this->Y = other.Y;
    this->pi = new int[other.ncount];
    assert( other.pi != 0 );
    memcpy(this->pi, other.pi, other.ncount*sizeof(int));
    this->w = other.w;
    this->ncount = other.ncount;
}

branch_node & branch_node::operator=(const branch_node& other) {
        this->X = other.X;
        this->Y = other.Y;
        assert( other.pi != 0 );
        memcpy(this->pi, other.pi, other.ncount*sizeof(int));
        this->w = other.w;
        this->ncount = other.ncount;

        return (*this);
}

branch_node::~branch_node() {
    if( pi ) {
        delete [] pi; pi = 0; // Not sure why this causes crashes !?
    }
}

bool operator<(const branch_node & l, const branch_node & r) {
    return l.w > r.w; // Because we want a min heap we flip the sign
}

int one_tree_tsp(int ncount, int ecount, int *elist, int *elen, int upper_bound) {
    int branches = 0;
    int *pi = new int[ncount];
    int bound, max_w = -100000, max_w_p_ago = -100000;
    int *pi_prime = new int[ncount];
    bool do_branch;
    std::priority_queue<branch_node> Q;
    std::vector<int> edges;

    for( int i = 0; i < ncount; i++ ) pi[i] = 0;


    std::vector<int> w_cache;
    w_cache.reserve(p);

    std::set<int> X, Y;
    std::vector<int> deg_not_2, deg_not_2_prime;

    bound = w(ncount, ecount, elist, elen, X, Y, pi, false, &deg_not_2, &edges);

    branch_node start = branch_node(X,Y,ncount,pi,bound);
    Q.push(start);

    while( Q.size() != 0 ) {
        branch_node current = Q.top();
        bound = current.w;
        // std::cout << bound << std::endl;

        if( bound > upper_bound ) {
            // std::cout << "bad upper bound" << std::endl;
            // std::cout << "pi: "; for( int i = 0; i < ncount; i++ ) std::cout << current.pi[i] << " ";
            Q.pop(); branches++;
            continue;
        }

        memcpy(pi, current.pi, ncount*sizeof(int));
        X = current.X;
        Y = current.Y;

        Q.pop(); branches++;

        std::set<int>::iterator it;
        // std::cout << "Popped (" << current.w << "), branches: " << branches << ", Q size: " << Q.size(
        // std::cout << "pi: "; for( int i = 0; i < ncount; i++ ) std::cout << pi[i] << " "; std::cout <<
        // std::cout << "X: "; for( it = X.begin(); it != X.end(); it++ ) std::cout << (*it) << " "; std:
        // std::cout << "Y: "; for( it = Y.begin(); it != Y.end(); it++ ) std::cout << (*it) << " "; std:

```

```
        do_branch = true;
        max_w = -100000;
        max_w_p_ago = -100000;
        deg_not_2.erase(deg_not_2.begin(),deg_not_2.end());
        bound = w(ncount, ecount, elist, elen, X, Y, pi, false, &deg_not_2, &edges);
        if( deg_not_2.size() == 0 ) {
            int last = elist[2*edges[0]];
            std::cout << "One␣Tree␣Tour:␣" << last;
            edges.erase(edges.begin(),edges.begin()+1);
            for( int j = 0; j < ncount-1; j++ ) {
                unsigned i;
                for( i = 0; i < edges.size(); i++ ) {
                    int edge = edges[i];
                    if( elist[2*edge] == last ) {
                        last = elist[2*edge+1];
                        std::cout << "␣" << last;
                        break;
                    } else if( elist[2*edge+1] == last ) {
                        last = elist[2*edge];
                        std::cout << "␣" << last;
                        break;
                    }
                }
                edges.erase(edges.begin()+i,edges.begin()+i+1);
            }
            std::cout << std::endl << "Length:␣" << bound << std::endl;

            delete [] pi; pi = 0;
            delete [] pi_prime; pi_prime = 0;
            return bound;
        }

        for(int i = 0; ; i++) {
            deg_not_2.erase(deg_not_2.begin(),deg_not_2.end());
            bound = w(ncount, ecount, elist, elen, X, Y, pi, true, &deg_not_2, &edges);
            // std::cout << "pi: "; for( int j = 0; j < ncount; j++ ) std::cout << pi[j] << " "; std::cou

            if( i >= p ) {
                if( w_cache[i%p] > max_w_p_ago ) {
                    max_w_p_ago = w_cache[i%p];
                }
            }
            w_cache[i%p] = bound;
            if( bound > max_w ) {
                max_w = bound;
                memcpy(pi_prime, pi, ncount*sizeof(int));
                deg_not_2_prime = deg_not_2;
            }
            if( bound > upper_bound ) {
                do_branch = false;
                break;
            }
            if( max_w_p_ago == max_w ) {
                break;
            }
        }
        // std::cout << "iterated pi" << std::endl;
        if( deg_not_2_prime.size() == 0 ) {
            // std::cout << "FOUND TOUR 2" << std::endl;
            // std::cout << "branches: " << branches << std::endl;
            bound = w(ncount, ecount, elist, elen, X, Y, pi_prime, false, &deg_not_2, &edges);
            Q.push(branch_node(X,Y,ncount,pi_prime,bound));
        }
        if( do_branch ) {
            bool done = false;
            for( unsigned i = 0; i < deg_not_2_prime.size(); i++ ) {
                for( int j = 0; j < ecount; j++ ) {
                    if( (elist[2*j] == deg_not_2_prime[i] || elist[2*j+1] == deg_not_2_prime[i]) && X.fi
                        std::set<int> X_new = X;
                        std::set<int> Y_new = Y;
                        X_new.insert(j);
                        Y_new.insert(j);
                        bound = w(ncount, ecount, elist, elen, X_new, Y, pi_prime, false, &deg_not_2, &e
                        Q.push(branch_node(X_new,Y,ncount,pi_prime,bound));
                        bound = w(ncount, ecount, elist, elen, X, Y_new, pi_prime, false, &deg_not_2, &e
                        Q.push(branch_node(X,Y_new,ncount,pi_prime,bound));
                        done = true;
                        break;
                    }
                }
                if( done ) {
                    break;
                }
            }
        }

    }

    delete [] pi; pi = 0;
    delete [] pi_prime; pi_prime = 0;
    return upper_bound;
}

int w(int ncount, int ecount, int *elist, int *elen, std::set<int> X, std::set<int> Y, int * pi,
    bool update_pi, std::vector<int> * deg_not_2, std::vector<int> * tree_edges) {
    int min_w = 1000000, bound, i, ignore;
    int * v = new int[ncount], *min_v = new int[ncount], * elen_new = new int[ecount];
    std::vector<int> edges;

    for( i = 0; i < ecount; i++ ) elen_new[i] = elen[i] + pi[elist[2*i]] + pi[elist[2*i+1]];


    for( ignore = 0; ignore < ncount; ignore++ ) {
        edges.erase(edges.begin(), edges.end());
        bound = w_candidate(ncount, ecount, elist, elen, elen_new, ignore, pi, v, X, Y, &edges); //chang
        if( bound <= min_w ) {
            min_w = bound;
            memcpy(min_v, v, ncount*sizeof(int));
            (*tree_edges) = edges;
        }
    }

    // Update pi
    for( i = 0; i < ncount; i++ ) {
        if( min_v[i] != 0 ) {
            deg_not_2->push_back(i);
        }
        if( update_pi ) {
            pi[i] += min_v[i]*t_bar;
        }
    }


    delete [] v;
    delete [] min_v;
    delete [] elen_new;

    return min_w;
}

int w_candidate(int ncount, int ecount, int *elist, int * elen, int *elen_new, int ignore, int * pi,
        int * v, std::set<int> X, std::set<int> Y, std::vector<int> * tree_edges) {
```

```
    std::vector<edge> edges;
    int SMALL_LEN = -1000000;

    std::set<int>::iterator it;

    // sort only edges incident to node ignoring
    for( int i = 0; i < ecount; i++ ) {
        if(elist[2*i] == ignore || elist[2*i+1] == ignore) {
            if( X.find(i) != X.end() ) {
                edges.push_back(edge(elist[2*i], elist[2*i+1], SMALL_LEN, i));
            } else {
                edges.push_back(edge(elist[2*i], elist[2*i+1], elen_new[i], i));
            }
        }
    }
    std::sort(edges.begin(), edges.end());

    int ecount_sub = ecount - edges.size();

    int ncount_sub = ncount - 1;
    int * elist_sub = new int[ecount_sub*2];
    int * elen_sub = new int[ecount_sub];
    std::vector<int> must_include;
    int * orig_ind = new int[ecount];

    int j = 0;
    for( int i = 0; i < ecount; i++ ) {
        if(elist[2*i] != ignore  && elist[2*i+1] != ignore ) {
            if( Y.find(i) != Y.end() ) {
                ecount_sub--;
                continue;
            }
            if( elist[2*i] > ignore ) {
                elist_sub[2*j] = elist[2*i]-1;
            } else {
                elist_sub[2*j] = elist[2*i];
            }
            if( elist[2*i+1] > ignore ) {
                elist_sub[2*j+1] = elist[2*i+1]-1;
            } else {
                elist_sub[2*j+1] = elist[2*i+1];
            }
            if( X.find(i) != X.end() ) {
                must_include.push_back(j);
            }

            elen_sub[j] = elen_new[i];
            orig_ind[j] = i;
            j++;
        }
    }

    graph G;
    G.init(ncount_sub, ecount_sub, elist_sub, elen_sub);

    std::vector<int> mst = G.min_spanning_tree(must_include);

    int tot = 0;
    for( unsigned i = 0; i < mst.size(); i++ ) {
        tot += elen[orig_ind[mst[i]]];
        tree_edges->push_back(orig_ind[mst[i]]);
    }
    if ( (int)mst.size() != ncount_sub - 1 ) {
        delete [] elist_sub;
        delete [] elen_sub;
        delete [] orig_ind;

        return 100000000;
    }

    for( int i = 0; i < ncount; i++ ) v[i] = 0;

    tot += elen[edges[0].ind];
    tot += elen[edges[1].ind];

    tree_edges->push_back(edges[0].ind);
    tree_edges->push_back(edges[1].ind);

    if( edges[0].end1 == ignore ) {
        v[edges[0].end2] += 1;
    } else {
        v[edges[0].end1] += 1;
    }
    if( edges[1].end1 == ignore ) {
        v[edges[1].end2] += 1;
    } else {
        v[edges[1].end1] += 1;
    }
    int end1, end2;
    v[ignore] = 2;



    for( int i = 0; i < ncount_sub-1; i++ ) {
        end1 = elist[2*orig_ind[mst[i]]];
        end2 = elist[2*orig_ind[mst[i]]+1];
        v[end1]++;
        v[end2]++;
    }

    for( int i = 0; i < ncount; i++ ) v[i] -= 2;

    for( int i = 0; i < ncount; i++ ) {
        tot += pi[i] * v[i];
    }

    if( do_print_tour ) {
        std::cout << edges[0].ind << "␣" << edges[1].ind;
        for( unsigned i = 0; i < mst.size(); i++ ) {
            std::cout << "␣" << orig_ind[mst[i]];
        }
        std::cout << "␣--␣";
        for( int i = 0; i < ncount; i++ ) std::cout << v[i] << "␣";
        std::cout << std::endl;
    }

    delete [] elist_sub;
    delete [] elen_sub;
    delete [] orig_ind;

    return tot;
}
```