Discrete Optimization

TSP Project: Tools, Pointers and Reminders

November 20, 2011

1 Mathematical model

We are given a complete graph $G(V, E)$ with vertices V. We denote the edge set $E = E(V)$. We formulate the TSP with edge costs c using subtour elimination constraints:

$$\begin{array}{ll} \min & \sum_{e \in E} c_e x_e \\ \text{s.t.} & x \in P_{\text{sub}} \end{array} \tag{TSP}$$

with

$$P_{\text{sub}} := \Big\{ x \in \{0,1\}^{|E|} \;:\; \sum_{e \in \delta(i)} x_e = 2 \quad \forall i \in V \\ \sum_{e \in E(S)} x_e \le |S| - 1 \quad \forall S \subseteq V : 1 \le |S| \le |V| - 1 \Big\}.$$

Alternatively, one can use the equivalent P_{cut} as a feasible region, with

$$P_{\text{cut}} := \Big\{ x \in \{0,1\}^{|E|} \;:\; \sum_{e \in \delta(i)} x_e = 2 \quad \forall i \in V \\ \sum_{e \in \delta(S)} x_e \ge 2 \quad \forall S \subseteq V : 1 \le |S| \le |V| - 1 \Big\}.$$

If $n = |V|$ is the number of vertices of the considered graph, we have $|E| = \frac{1}{2}n(n-1)$ edges, n degree constraints and $2^n - 2(n-1)$ subtour elimination (or cutset) constraints.

The only necessary data are n and the cost vector c. We will work with standard TSPLIB instances (`http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/`).

2 Solver components

2.1 LP

We need to be able to optimize over the linear relaxation of P_{sub}. This could be done by standard linear programming techniques. However, since the number of subtour elimination constraints involved can quickly grow huge with increasing n, we should not include them all. Instead, we should iteratively add some of them (violated ones), until we obtain a solution that provably satisfies all.

2.2 Branch and Bound

Given a solver that can compute the optimal solution of the linear relaxation of (TSP), we now need to perform *branch and bound* in order to obtain an integer solution.

2.3 Primal Heuristics

Primal heuristics will help us obtain good integer feasible solutions for (TSP), thus providing good primal bounds to the branch and bound.

2.4 Cuts

Specific valid inequalities can be generated for (TSP) in order to strengthen the linear relaxation of P_{sub}.

3 Implementation

We provide you with a reference interface to ease interaction between the different components. You can find its source code on `http://www.montefiore.ulg.ac.be/~poirrier/discrete.php`.

3.1 Classes

TSPModel contains the cost vector c of a TSP instance. This vector has one entry for each edge of the complete graph, and entries are kept symmetric, i.e. if you assign a value to c_{ij} it will be assigned to c_{ji} too. Useful methods:

Construction/destruction/assignment	
`TSPModel()`	Create a zero-sized c
`TSPModel(int n)`	Create an uninitialized vector c for a n-vertices graph; vector size is the number of edges, i.e. $\frac{1}{2}n(n-1)$)
`TSPModel(TSPModel &)`	Copy constructor
`operator=(TSPModel &)`	Assignment operator

Inherited from `EdgeData<double>`	
`clear()`	Reset size to zero
`resize(int n)`	Resize c for a n-vertices graph (size $\frac{1}{2}n(n-1)$) elements are left uninitialized
`int vertices()`	Return n, the number of vertices of the graph
`int edges()`	Return the number of edges of the complete graph
`set(double v)`	Set all elements of c to value v
`set(int i, int j, double v)`	Set element c_{ij} to value v
`double get(int i, int j)`	Get value of element c_{ij}
`double &operator() (int i, int j)`	Lets you access element c_{ij} by writing `c(i, j)`
`double operator*(TSPSolution &)`	Lets you compute $c^T x$ by writing `c * x`

TSPSolution contains a solution vector x for a TSP instance. This vector has one entry for each edge of the complete graph, and entries are kept symmetric, i.e. if you assign a value to x_{ij} it will be assigned to x_{ji} too. Useful methods:

Construction/destruction/assignment

`TSPSolution()`	Create a zero-sized x
`TSPSolution(int n)`	Create an uninitialized vector x for a n-vertices graph; vector size is the number of edges, i.e. $\frac{1}{2}n(n-1)$)
`TSPSolution(TSPSolution &)`	Copy constructor
`operator=(TSPSolution &)`	Assignment operator

Inherited from `EdgeData<double>`

`clear()`	Reset size to zero
`resize(int n)`	Resize x for a n-vertices graph (size $\frac{1}{2}n(n-1)$) elements are left uninitialized
`int vertices()`	Return n, the number of vertices of the graph
`int edges()`	Return the number of edges of the complete graph
`set(double v)`	Set all elements of x to value v
`set(int i, int j, double v)`	Set element x_{ij} to value v
`double get(int i, int j)`	Get value of element x_{ij}
`double &operator() (int i, int j)`	Lets you access element x_{ij} by writing `x(i, j)`
`double operator*(TSPModel &)`	Lets you compute $c^T x$ by writing `x * c`

Solution properties

`double getCost(TSPModel &)`	Return $c^T x$
`bool isIntegral()`	Inspects the solution and returns `true` if it is integral
`bool isIntegral(int i, int j)`	Return `true` if x_{ij} is integral
`setFeasible(bool feas)`	Sets the feasibility flag used to mark a problem as (in)feasible
`isFeasible()`	Fetches the feasibility flag

TSPFixing describes a fixing, i.e. the fact that some variables x_e should be fixed to a constant value $\bar{x}_e \in \{0, 1\}$. Note that the constant value can *not* be fractional. A fixing has as many components as there are edges of the complete graph, and entries are kept symmetric, i.e. fixing the value of x_{ij} is equivalent to fixing the value of x_{ji}. Empty (i.e. zero-sized) fixings have a special meaning: they correspond to no variable being fixed, whatever the graph size. Useful methods:

Construction/destruction/assignment

`TSPFixing()`	Create a zero-sized fixing
`TSPFixing(int n)`	Create an uninitialized fixing for a n-vertices graph; fixing size is the number of edges, i.e. $\frac{1}{2}n(n-1)$)
`TSPFixing(TSPFixing &)`	Copy constructor
`operator=(TSPFixing &)`	Assignment operator

Inherited from `EdgeData<char>`

`clear()`	Reset size to zero
`resize(int n)`	Resize the fixing for a n-vertices graph (size $\frac{1}{2}n(n-1)$) elements are left uninitialized
`int vertices()`	Return n, the number of vertices of the graph
`int edges()`	Return the number of edges of the complete graph

Fixing attributes

`unfix()`	Unfix all variables
`unfix(int i, int j)`	Unfix x_{ij}
`fix(int i, int j, double v)`	Fix x_{ij} to $v \in \{0, 1\}$
`bool empty()`	Return `true` if the fixing is zero-sized
`bool isFixed(int i, int j)`	Return `true` if x_{ij} should be fixed
`double fixedValue(int i, int j)`	Return $\bar{x}_{ij} \in \{0, 1\}$

TSPCut describes an inequality of the form $\alpha^T x \geq \beta$ for a TSP formulation. More precisely, it contains the coefficients vector α, which has one entry per edge of the complete graph. Whether you consider x_{ij} with $i < j$ or $i > j$ does not matter: the values of α_{ij} and α_{ji} are internally mapped to the same element. Useful methods:

Construction/destruction/assignment	
`TSPCut()`	Create a zero-sized x
`TSPCut(int n)`	Create an uninitialized vector α for a n-vertices graph; vector size is the number of edges, i.e. $\frac{1}{2}n(n-1)$)
`TSPCut(TSPCut &)`	Copy constructor
`operator=(TSPCut &)`	Assignment operator

Inherited from `EdgeData<double>`	
`clear()`	Reset size to zero
`resize(int n)`	Resize α for a n-vertices graph (size $\frac{1}{2}n(n-1)$) elements are left uninitialized
`int vertices()`	Return n, the number of vertices of the graph
`int edges()`	Return the number of edges of the complete graph
`set(double v)`	Set all elements of α to value v
`set(int i, int j, double v)`	Set element α_{ij} to value v
`double get(int i, int j)`	Get value of element α_{ij}
`double &operator() (int i, int j)`	Lets you access element α_{ij} by writing `alpha(i, j)`
`double operator*(TSPSolution &)`	Lets you compute $\alpha^T x$ by writing `alpha * x`

Other cut fields	
`setSense(char sense)`	Set cut sense: `'E'` for $=$, `'G'` for $\geq$, `'L'` for $\leq$
`char getSense(char sense)`	Get cut sense (`'E'` for $=$, `'G'` for $\geq$, `'L'` for $\leq$)
`setRHS(double beta)`	Set the right-hand-side β
`double getRHS()`	Get the right-hand-side β
`double slack(TSPSolution &x)`	Compute the slack of the inequality evaluated at x: ≥ 0 for a verified inequality, < 0 for a violated one

TSPCutPool is an STL list of `TSPCut`s. It is defined as follows: `typedef std::list<TSPCut> TSPCutPool;` See the documentation of the Standard Template Library on `std::list` for more details.

3.2 Classes internal details

The subclasses `TSPModel`, `TSPSolution`, `TSPFixing` and `TSPCut` all inherit from a specialization of the template class `EdgeData`, which is a container type for edge data as addressed by adjacent vertices. The data are symmetric: they can be addressed equivalently as (i, j) or (j, i). The diagonal elements (i, i) shall not be addressed: the methods will abort (an assertion will fail) on any attempt to addressing out-of-range or diagonal elements. The template specialization specifies the data type. All subclasses also provide an assignment operator from the corresponding `EdgeData` specialization, for ease of type conversion. Specializations provided include `EdgeData<double>`, `EdgeData<char>` and `EdgeData<int>`. Method definition should be transferred to the header file if more are needed.

3.3 TSPFile parser

A parser is provided for reading data from the TSPLIB. It can both read instance description (i.e. `.tsp` files containing the c vector) or tour description (i.e. `.tour` files containing a solution vector x).

`TSPFile` methods

`TSPFile()`	Create an empty `TSPFile` structure
`TSPFile(std::string &path)`	Create a `TSPFile` structure and load a file
`clear()`	Clear all the public fields
`int load(std::string &path)`	Load a file, returning zero in case of success

`TSPFile` public fields

`int n`	Number of vertices
`std::string name`	Instance name
`EdgeData<double> edge`	Instance (c) or tour (x) description
`std::vector<TSPCoord> vertex`	For displaying purposes, this field contains coordinates for the vertices, if available
`TSPFixing forced`	A fixing can be provided in the description, forcing (non-)inclusion of specific edges (only one file in the TspLib: `linhp318.tsp`)

The `load()` method actually takes a second argument of type `int` specifying the maximal accepted number of nodes, which defaults to 1000. This is to safeguard against accidentally loading huge instances into memory, which could lead to heavy swapping and temporarily freeze the machine.

As noted before, appropriate assignment operators are provided for subclasses of `EdgeData`, allowing for the following:

```
TSPFile parseTSP("problem.tsp");
TSPFile parseTour("problem.opt.tour");

TSPModel c = parseTSP.edge;
TSPSolution x = parseTour.edge;
```

3.4 CPLEX

We will make use of an external solver for the linear programming problems arising in our different components. The Cplex solver is installed on the machine made available to you. A copy of its documentation is available at `http://thales02.montefiore.ulg.ac.be/cplex/`. The use of the default C API "Cplex Callable Library" is advised (though not mandated) over its C++ counterpart.

3.5 Interface for the LP component

The LP component should implement two interface classes, `LPSolver` and `LPNode`. The `LPSolver` class holds all data that should be global to the solving process of the instance. It has the following public methods:

```
class LPSolver {
public:
        LPSolver(const EdgeData<double> &c);
        ~LPSolver();

};
```

The `LPNode` class holds local data for one LP resolution. Its constructor shall be provided with a reference to the global `LPSolver` data. In addition, a reference to the `LPNode` object corresponding to the parent

node should be passed, except if no such parent exists (e.g. at root node). The class features the following public methods:

```
class LPNode {
public:
        LPNode(LPSolver &solver);
        LPNode(LPSolver &solver, const LPNode &parent);
        LPNode(const LPNode &node);

        bool solve(const TSPFixing &fix, const TSPCutPool &pool,
                TSPSolution &x, int maxIter = 0);
        bool solve(const EdgeData<double> &c,
                const TSPFixing &fix, const TSPCutPool &pool,
                TSPSolution &x, int maxIter = 0);
};
```

The bulk of the work is performed by the `solve()` method which computes an LP solution x. The five-arguments form of `solve()` lets us override the default cost vector c provided to the `LPSolver` object. The return value is `true` if the a result was found before the iteration limit. If `maxIter == 0`, then there is no iteration limit and the return value is always `true`. If the problem is infeasible, the corresponding flag is set accordingly in the `TSPSolution` (and the return value is still `true`).

3.6 Interface for the branch and bound component

The branch and bound component should implement the main function:

```
int main(int argc, char **argv, char **envp);
```

3.7 Interface for the heuristics component

The heuristics component implements a `TSPHeuristics` class having the following public methods:

```
class TSPHeuristics {
public:
        TSPHeuristics(const EdgeData<double> &c);
        ~TSPHeuristics();

        bool get(const TSPFixing &fix, TSPSolution &x);

        void addIP(const TSPSolution &incumbent);
        void addLP(const TSPSolution &x);
};
```

The most important method is `get()` providing a heuristic integer feasible solution. Its return value is `true` in case of success. The `TSPFixing` argument is only indicative and provides information about the current node.

The methods `addIP()` and `addLP()` lets us provide the heuristics with more information. When the branch and bound finds an integer feasible solution, it can notice the heuristics using `addIP()`. It can also provide the heuristics with the current fractional solution using `addLP()`. This information may be discarded.

3.8 Interface for the cut generator component

The cut generator implements the `TSPCutGenerator` class with the following public methods:

```
class TSPCutGenerator {
public:
        TSPCutGenerator(const EdgeData<double> &c);
        ~TSPCutGenerator();

        int get(const TSPFixing &fix, const TSPSolution &x, TSPCutPool &cuts);
};
```

The `get()` method adds valid inequalities to the `TSPCutPool` object. The return value is the number of such added cuts. The inequalities need only be valid when the variables are fixed as indicated by the `TSPFixing` object. To obtain globally-valid inequalities, one should call the `get()` method with an empty fixing argument. The `TSPSolution` x indicates the current LP fractional solution.