Single Source Multi-Commodity Network Flow

Ashwin Panchapakesan
University of Ottawa
apanc006@uottawa.ca

Revised February 27, 2012

Contents

1 Motivation

The motivation of this paper is based on the setting of a popular conference, which has a high attendance. Suppose that multiple attendees attend this conference, which takes place in a large city, which has multiple large roads and multiple smaller, "back-roads" between neighborhoods. Suppose that an earthquake during the conference structurally damages the large roads so that each of these large roads now has a limit on how many cars it can support before it crumbles and becomes unusable. This has the following two effects:

1. In-car GPS systems can no longer be used reliably as they would not be aware of such recent changes.
2. While some of the attendees may still take the shortest path to their hotels (their destinations), it is very likely that a larger subset of the set of all attendees will need to take longer, more time expensive routes to their destinations.

Note that the hotels in this city offer singular cottage-style accommodation, where each cottage has two ways to access the roads:

1. A private access, exclusive to that cottage.
2. A shared access, shared by multiple (not necessarily all) cottages in a specific neighborhood.

Like many conferences, this one is held in an exotic location, whose government has deployed the nation's military forces to ensure that all the attendees reach their respective homes safely. Engaging the military as such is resource-expensive and as a result, it is important to ensure that all attendees reach their destinations safely and in minimal time.

Now, the important question asks which attendees should be re-routed and through which major roads in order to reach their respective destinations in minimal time.[1] Note that this does not ask to minimize the total time it takes to route all the attendees to their destinations. Rather it asks to minimize the time it takes for the last attendee to reach their destination[2]. This is because the military must remain engaged until the last attendee reaches their destination, regardless of how many attendees reach their destinations last.

2 Formalization

1. Let G = (V,E) be a directed graph (where V is the set of all neighborhoods and E is the set of all roads, large and small)
2. Let s $\in$ V be the source - the conference venue
3. Let $A = \bigcup_{0<i\leq k} a_i$ be the k attendees of the conference
4. $\forall a_i \in A$, let $t_i \in V$ be the destination/terminal of a_i
5. Let $T = \bigcup_{0<i\leq k} t_i$ be the set of all terminals. Note that T $\subseteq$ V
6. $\forall e \in E$, w(e) is the weight/cost of the edge. This captures the notion of the time it takes to travel that edge (road)
7. $\forall e \in E$, c(e) is the capacity of the edge
8. $\forall e \in E$, f(e) is the total flow through that edge
9. Let $N = \{n_1, n_2, ..., n_j\}$ be j disjoint subsets of V ($\forall i \neq j, n_i \cap n_j = \emptyset$) such that $\bigcup_{i=1}^{j} n_i = V$, where each $n_i \in N$ is a neighborhood. Thus, N is a partitioning of V. Clearly, each a_i's destination is exactly one neighborhood. Since it is possible for one neighborhood to be the destination of multiple attendees, $\forall v \in V, |t \in v \cap T| \geq 0$

[1] Note that there is a notion of isomorphism here in that routing two different attendees that share a terminal through the same neighboring terminal can be considered as isomorphisms of the same solution. See section 9 on page 10

[2] Note that if there are multiple such last attendees, optimizing the path of only one will not affect the overall quality of the solution

10. $\forall a_i \in A, path(a_i) = (s, v_1), (v_1, v_2), ..., (v_k, t_i)$

11. $\forall e = (n_i, n_j) \in E$, c(e) $= \infty$

It is now possible to model this problem as a single-source multi-commodity flow problem where $C = \{c_1, c_2, ...c_j\}$ (`j` commodities) and each a_i is exactly one commodity c_k. Note that C is a partitioning of A.
Thus, if d_i is the demand for commodity c_i at its terminal n_i , and
$d_i = |\{a \in A | a \text{ is commodity } c_i\}$

3 Simplifications

1. Since graph traversal is not a problem that is being solved in this paper, it is assumed that the input includes a path for each attendee - a path from the source to their destination/terminal. This can be modeled after the path that they would have travelled by, to get to their destinations if there was no earthquake.

2. It is also assumed that the capacity of each edge has a magnitude that is at least as large as the largest demand. This allows for simplified routings of unsplittable flows without having to worry about congestion-relief factors[1].

3. It is further assumed that the set of all attendees includes at least one attendee from each neighborhood. This assumption captures the notion that in order to reroute any attendee a_i, through terminal t_j to its terminal t_i ($t_i \neq t_j$), that attendee (a_i) need only travel by the path travelled by any attendee whose terminal is t_j (as long as that path has residual flow capacity, i.e. $\forall e \in path, c(e) - f(e) > 0$) and then travel by the shortest small road between t_i and t_j.

4. The edges in the graph are not directional. This captures the notion that all roads allow traffic in both directions. This simplification can be easily undone by replacing every edge (`e=(u,v)`) with two edges `(u,v)` and `(v,u)`. Since graph traversal is not within the scope of this paper, this simplification makes no difference to the end result. The only foreseeable problem with this simplification is that this model does not currently capture cities with one way streets. However, as has been mentioned, this is an issue only for graph traversal and therefore does not affect the algorithm presented here.

It is important to note that each commodity is routed unsplittably from the source to its terminal. This is because every edge (road) had infinity capacity before the earthquake. As a direct result of the earthquake, the capacitation of the large roads may now require that the routing of these commodities be done in a manner that is not unsplittable. Finding a minimal maximum-cost routing for such a non-unsplittable flow is therefore the question that this algorithm attempts to solve.

4 Solution Design

A Tabu Search is used in order to solve this problem.

List of Algorithms

4.1 Inputs

1. a set of all vertices in the graph, including meta data on what terminals they contain.
2. a set of all edges in the graph, including meta data on their flows, capacities and weights.
3. a mapping of all attendees/agents to the paths that they would have taken to their destinations if not for the earthquake.

4.2 Neighborhood Structure

A neighbor to the current graph state (G=(V,E)) is defined as a modified graph state (G=(V',E'))) where:

An agent $a_i \in A$ has been identified as a_{last} in G and has been rerouted through terminal $t_i \neq t_{last}$ thereby altering the flows in certain edges of E (see step 1(b)vi in section 5 for details). This modified set of edges forms E'. Also note that in step1(b)v in section 5, t_{last} is reassigned to a different node $v \in V \cap T$. Thus V is altered to produce V'.

4.3 Initial Solution

The algorithm begins with the point in the search space corresponding to the inputs. This point consists of a mapping of all agents to the paths that they would have traveled by from the source to their respective terminals if the earthquake had not occurred.

4.4 Objective Function

The objective function computes the sum of the weights of the edges in the path of every attendee from the source to the respective destinations. Any attendee that travels the heaviest path can be identified as a_{last}. The sum of the weights of the edges on the path of a_{last} is the value returned by the objective function for any point in the search space.

4.5 Search Strategy

Identify an attendee a_i as a_{last} if a_i takes the most time to arrive at its terminal. Note that it is possible for multiple attendees to fit this description (i.e. multiple distinct attendees may be identified as a_{last} at any point in time).

Attempt to re-route a_i through a different terminal such that the total time taken for a_i to reach its terminal is reduced (if such a terminal exists). Thus a_i can no longer be identified a_{last} if multiple attendees were initially identified as a_{last}. Note however, if a_i was the only $a \in A$ to be identified as a_{last}, then it is still possible (though, not certain) that a_i is identified as a_{last}, notwithstanding its reduction in time. It may then need to be routed through yet another neighborhood in order to reduce the total time of its path (if such a neighborhood exists).

If such a neighborhood does not exist for any a_i that can be identified as a_{last}, pick a random attendee ($a_j \in A$) and attempt to re-route a_j through a different neighborhood such that the total time taken for a_j to reach its terminal t_j is reduced (as long as such a terminal exists).

5 Detailed Algorithm Description

5.1 Inputs

`PATHS` A mapping (hash map) of agents to paths - each agent is mapped to the path that they would have taken to their destination, if not for the earthquake.

`G=(V,E)` The current state of the graph - all the vertices [3], the edges [4]

[3] see 7.1 on page 9 for a description of the Node class
[4] see 7.2 on page 9 for a description of the Edge class

`N` The number of samples of the neighborhood to be considered by the Tabu Search [5]

`Dist` A distance parameter explained in 1a

`T` The size of the Tabu list to be used

`CMAX` The maximum number of iterations to run the Tabu Search for. This is required as there are no other known stopping criteria such as the optimum value (see section 9).

5.2 Functionality

1. Sample the neighborhood by computing neighbors. This is done in the getNeighbor function (see pseudo code in 6 on the next page)[6].

   (a) The getNeighbor function takes four input parameters:

   `G=(V,E)` the current state of the graph. All the nodes and edges, with associated flows, capacities, terminals, etc

   `Paths` the current mapping of agents to the paths they take to their respective destinations

   `Dist` an extra parameter that defines the maximum distance that an agent is willing to "deviate". [7] This has been included for two reasons:

   - to reduce the search space
   - to allow for flexibility of this model so that it can account for other parameters (discussed in Future Work).

   `rerouts` A list of 2-tuples, each of which contains:

   Index 0 the terminal node t_i of agent a_i that has been rerouted

   Index 1 the node through which a_i has been rerouted

   (b) The getNeighbor function is functions as follows:

   i. With these three parameters, this function computes a list of nodes that have the following properties:
      - The node must be a terminal node, i.e. it must be contained in T.
      - The node must be connected to another terminal node by an edge whose weight is no larger than Dist.
      - There must exist an agent whose destination is this node (this property will be satisfied by the node being a terminal node), and for every edge, e, on whose path, $c(e) - f(e) > 0$. The latter condition is computed by the routable function (see pseudo code)
      - There must exist no agent a_i in the neighborhood such that this node is $t_i$and (t_i,`v`) (for some `v` $\in$`V`) exists in `rerouted`.

   ii. If a_{last} has one of these nodes as its terminal node, then let the variable `u` be t_{last}. Else, let `u` be any randomly chosen node from this list of nodes.

   iii. Let n be a randomly chosen node such that there exists an infinitely capacitated edge between it and `u` , whose weight is no greater than Dist.

   iv. Let r be either a_{last} (if `u` was not randomly chosen in step 1(b)ii) or a randomly chosen agent whose terminal node is `u` (if `u` was randomly chosen in step 1(b)ii).

   v. Change the terminal node of `r` from its original terminal node, to `n` . In doing so, make sure to reduce the flow along the edges of `r`'s original path by one.

[5] see 7.4 on page 9 for a description of the implementation of the neighborhood

[6] In order to not compute isomorphic neighbors, a `rerouted` list is maintained. See item (1a) for a description of this list. Also see 9 on page 10 for an explanation of isomorphic neighbors and how calculating them has been avoided here.

[7] This captures the notion that any terminal that an agent will be rerouted through will be a distance of at most `Dist` away from the agent's destination.

vi. Change the path of `r` to be a lightest path (with residual capacity larger than 0) of the agents whose terminals are `n` , with the addition of the lightest, infinitely capacitated edge from `n` to `r`'s original terminal. In doing so, make sure to increase the flow along the edges of `r`'s new path by one. [8]

2. Of the neighbors not in the tabu list, select one that in which the last agents to reach their respective destinations do so in the least amount of time/distance when compared to those of other such neighbors.

   (a) The time taken (or distance traveled) by the last agents to reach their respective destinations is computed by the `evaluate` function (see step 2b), which takes as input, a hash map of agents to the paths they travel from the source to their respective terminals.

   (b) The `evaluate` function functions as follows:

      i. for each agent in the set of all agents, calculate the sum of the weights of each edge in the path it travels from the source to its terminal.

      ii. Return the maximum such sum

3. Let this neighbor become the (new) current state of the graph and the mapping of the agents to their paths. Also, add this (new) current state of the graph to the tabu list.

4. Update the variable containing the best graph state seen so far if the current state of the graph has a more optimal result than the best graph state seen so far.

5. Repeat steps 1 through 4, C_{MAX} many times.

6 Pseudo-code

Algorithm 1 The routable Function

```
function routable(node, paths)
    // for every attendee for whom node is a terminal node
    ∀r ∈ {a_i ∈ A|t_i ∈ node} do
        // if every edge, e, on that attendees path has
        // c(e) − f(e) > 0
        if (∀e ∈ paths[r], c(e) > f(e)) then
            return True
    return False
```

Algorithm 2 The evaluate Function

```
function evaluate(paths)
    heaviestPath := NULL
    cost := 0
    foreach path in paths:
        if (∑_{e∈path} w(e) > cost) then
            heaviestPath := path
            cost ∑_{e∈path} w(e)
    return ∑_{e∈path} w(e)
```

[8] Thus, `r` still reaches its terminal and does so by adopting and extending the lightest possible path of agents whose terminals are as close as possible to `r`'s terminal.

Algorithm 3 The Tabu Search

```
function search(V, E, PATHS, N, dist, T, CMAX)
    tabu := TabuList of size T
    answer := NULL

    // the number of iterations of the TabuSearch so far
    c := 0

    // time taken by the last agent to reach its terminal
    time := ∞
    n := (V, E)
    while (c < CMAX) do
        c := c+1
```

step 1

```
        neighborhood := {}
        rerouted := {}
        while (|neighborhood| < N) do
            v, e, paths, u, uu :=
                    getNeighbor(n, PATHS, dist, rerouted)
            if (paths ∉ tabu) then
                neighborhood := neighborhood ∪ (v,e,paths)
                rerouted := rerouted ∪ (u,uu)
```

step 2

```
        v, e, paths := min_{evaluate(nn.values)} neighborhood
```

step 3

```
        n := (v,e)
        nn := paths
        t := evaluate(nn.values)
        tabu.add(n)
```

step 4

```
        if t < time then
            answer := t
            time := answer
    return answer
```

Algorithm 4 The getNeighbor Function

```
function getNeighbor(V,E, paths, dist, rerouts)
    extern A
    u := NULL
    n := NULL
    while (u = NULL) DO
```

step 1(b)i

```
        us := {v ∈ V | ∃t ∈ T ∩ v,
                  ∄(v, v′) ∈ rerouts for any v′ ∈ V
                  (∃e ∈ E|v ∈ e, c(e) = ∞),
                  routable(u, paths) if ∃u ∈ V|(u, v) ∈ E
                  else routable(u, paths)
                       where (u,v) ∈ E, w(e) ≤ dist}
```

step 1(b)ii

```
        U := max_{∑_{e∈paths.values} w(e)} { paths } // a_last
        if (|v ∈ V|t_i ∈ v where U = t_i| > 0) then
             u := randomChoice(v ∈ V|t_i ∈ v where U = t_i)
        else
             u := randomChoice(us)
```

step 1(b)iii

```
    while (n = NULL) do
        ns := {v ∈ V|u ≠ v,
             (∃e ∈ E|e = (u, v) or e = (v, u), c(e) = ∞),
             min_{w(e)} e ∈ E|e = (u, v) or e = (v, u) ≤ dist}
    n := randomChoice(ns)
```

step 1(b)iv

```
    ress := {a_i ∈ A|t_i ∈ u }
    r := randomChoice(ress)
```

step 1(b)v

```
    t_i := n
    ∀e ∈ paths[r] do
        f(e) := f(e) − 1
```

step 1(b)vi

```
    p = min_{∑_path w(e)} { path | path = paths[r],
                                 ∃a_i ∈ A|r = a_i, t_i ∈ n,
                                 ∀e ∈ paths[r]c(e) − f(e) > 0 }
    ∀e ∈ p do
             f(e) := f(e) + 1
    paths[r] := paths[r] + min_{w(e)} {e ∈ E|e = (u, n)ore = (n, u) }
```

```
    return (V,E), paths, u, n
```

7 Classes and Data Structures

In order to properly model the data, the following classes data structures were used

7.1 Class Node

All vertices are instances of the `Node` class. The Node class has the following attributes:

ID The unique identifier for the node. Two nodes that have the same ID are considered to be identical.

Residents A list of identifiers of all agents for whom this node is a terminal node.

7.2 Class Edge

All edges are instances of the `Edge` class. The Edge class has the following attributes:

ID The unique identifier for the edge. Two edges that have the same ID are considered to be identical.

Source The source node of this edge[9].

Destination The destination node of this edge[9].

Weight The weight value associated with this edge. This captures the notion of how long it takes to traverse the edge.

Capacity The maximum flow that the edge can support.

Flow The total current flow through the edge.

7.3 Class TabuList

A Tabu list is implemented as a list with a size limitation, which is passed to the constructor. It maintains an internal list (which is empty at initialization) into which items are added. It contains a specialized `add` function, which is used to add items into the tabu list.

Add This is a method that facilitates the addition of new items into the internal list. The purpose of this function is to add a check to make sure that the size constraint on the tabu list is not violated. As such, this functions as follows:

1. Given a new element, e, insert it into index 0 of the internal list.
2. Compute the length of the list after insertion.
3. If the list now violates the size constraint, remove the item in the last index of the list. This will restore the list to a "legal" size.

7.4 Neighborhood

At every iteration of the Tabu search algorithm, the neighborhood is calculated and is maintained in a `set`. A `set` is a data structure used to emulate a mathematical set in which elements may be present at most once. This is used so as to consider only unique neighbors, in order to not unduly restrict the search space and to not recompute identical graph states. This, along with avoiding the computation of isomorphic graph states, helps the algorithm explore only meaningful neighbors.

[9]Note that since this is an undirected graph, Source and Destination are simply placeholder variables to identify the nodes at either end of the edge.

8 Experiment Setup

8.1 Data set

The data set for this problem consists of a manually created graph derived from a portion of an Ottawa city map. This graph is included as a separate document. Also included are spreadsheets containing:

1. the listing of all nodes, along with any terminals they contain
2. the listing of all edges, along with their source and destination nodes, capacities and weights
3. the listing of all agents and the original paths they travel from the source to their respective terminals

8.2 Experimentation with Parameter Settings

It has been found from the literature [2, 3], that under 500 iterations of a tabu search suffices for similar problems. As a result of this and just to be cautious, a C_{MAX} of 600 iterations has been imposed on the tabu search algorithm described in section 5.

The tabu search was run on the following parameters:

Size of Neighborhood Sample 15

Size of Tabu List 10

C_{MAX} 600

The size of the tabu list was determined experimentally from initial experiments which showed the above to be good values for the tabu search for this problem.

The size of the neighborhood sample was also determined experimentally from initial experiments. However, it is important to note that this value intuitively makes sense from the way the neighborhood is sampled to avoid calculating isomorphic neighbors (see section 9).

With these parameters, the tabu search was run on various values of `dist` ranging from 5 to 11. These values for `dist` were determined to be good values for experimentation based on analysis of the structure of the graph (see 8.1).

8.3 Running the Experiment

20 runs of the tabu search were run on each combination of parameters. Each was run on a single 2.0GHz core processor of a quad core computer running Linux Fedora 12 with 8GB of memory.

9 Discussion

It is clear from just the algorithm that there are a few variables that are of importance, whose values need to be chosen carefully:

1. the size of the neighborhood
2. the size of the tabu list
3. the evaluation function
4. the maximum distance of a terminal through which an attendee may be re-routed to its own terminal

1 and 2 are standard variables to any tabu search. They determine (among other properties of the algorithm) the run time and the amount of choice the algorithm has in terms of making a better choice. Clearly, the larger the neighborhood, the more likely it is that every sample of the neighborhood will contain a move in the positive direction.

On the other hand, a tabu list that is too large will reduce the likelihood of generating the same solution multiple times, regardless of how good it is. This has the effect of reducing redundancy in terms of sampling

the same neighborhoods multiple times. However, due to the definition of, and the way in which isomorphic graph states are avoided (explained below), it is possible that some isomorphs which could lead to better solutions were not computed.

This notion of isomorphism can be addressed as follows. Suppose that two agents who share a terminal are each re-routed through a specific neighboring terminal. With no other parameters on this algorithm, these two solutions are isomorphisms of each other. In order to avoid generating isomorphisms, the search function keeps track of all reroutings in the current iteration, which it then passes to the `getNeighbor` function (see step 1(b)i on page 5 in section 6 on page 6). The `getNeighbor` function then generates neighboring graph states in such a way that no two generated neighbors are a result of rerouting agents whose terminals are in the same node.

By itself, this does not ensure that isomorphic solutions will not be computed across different iterations of the tabu search. In order to accomplish this, the terminal t_i of an agent a_i, that is rerouted through a neighboring terminal n is now set to n (see steps 1(b)v on page 5 and 1(b)vi on page 6). Since this changes the terminal node of an agent, it ensures that isomorphic graph states will not be computed by the tabu search algorithm.

However, this introduces the concern that agent a_i that has been rerouted through neighboring terminal node n will not reach it's terminal node t_i as a result of the change of it's terminal node. This is concern is addressed in step 1(b)vi on page 6 with the addition of the lightest possible edge from n to t_i.

While this method ensures that only unique graph states in the neighborhood are computed, it does not ensure that all such graph states could be computed. This is one explanation for the unsatisfactory results (see section 10.2 on the following page).

Like many other well known problems, this problem does not have a known optimal solution. This means that there exists no obvious stopping criterion that can be implemented to meaningfully stop the algorithm before it reaches C_{MAX} many iterations. If the assumption that the set of paths provided as input is a set of shortest paths is a valid assumption, then one possible evaluation function could be to find the difference between the time taken by the last attendee in the input paths and the last attendee in the computed paths. This heuristic was not implemented in the experiments as this is highly dependent on the application. For example, it has a high likelihood of failure if there exists an attendee in the input data who chooses to travel to their destination in a less optimal path (for example, chooses to travel along a "scenic route"). Since there was such an attendee in the input data, it was deemed an unusable heuristic.

The `dist` parameter in the search is one of special interest. It provides a means of pruning the search space a priori. It dictates the maximum distance that an attendee may be re-routed through. Essentially, this means that in order to re-route an attendee a_i, a_i may not be re-routed through a neighborhood `n` that is a distance greater than `dist` away from t_i. Equivalently, a_i may not be re-routed through `n` to t_i if $\nexists e \in E | e = (t_i, n)$ or $e = (n, t_i), w(e) > dist$

10 Results

10.1 Data

The averages of the twenty runs of the tabu search on each combination of the parameters are presented below.

Table 1: evaluation of a_{last} at the first and last iterations of the tabu search

`dist`	`evaluate`(a_{last}) at iteration 1	`evaluate`(a_{last}) at iteration 600
5	74	74
7	74	74
9	74	74
11	74	74

This seems odd, and could be indicative of buggy code. To investigate further, auxiliary information from the data collection process was considered, specifically, the sum of the weights of all edges of the paths of all agents:

Table 2: sum of all weights of the edges of all paths in the first and last iterations of the tabu search

`dist`	$\sum_{a \in A}$ evaluate(a) at iteration 1	$\sum_{a \in A}$ evaluate(a) at iteration 600
5	10388	10388
7	10388	10390
9	10388	10391
11	10388	10388

10.2 Data Analysis

It is very interesting to note that the data shows that the time taken by the last attendee to reach their terminal is constant and does not change, even over 600 iterations of the tabu search. However, it is interesting to note that the sum of the times taken by all attendees does change over the iterations.

This could be interpreted in one of two ways:

1. The input is a feasible solution that is already optimized (a solution shall be deemed feasible if all edges have a non-negative residual capacity, i.e. $\forall e \in E, f(e) \leq c(e)$
2. The search space is far too large to effect a change in the time taken for the last attendee to reach their destination
3. The method of ensuring that isomorphic graphs are not computed in the `getNeighbor` function is too restrictive and unduly narrows the search space

(1) is a ludicrous interpretation as the data shows that it is not the case. The input data does contain at least one attendee who takes an inefficient path to their terminal, for whom there is a more efficient path in the capacitated graph.

(2) is also unlikely to be the correct explanation for the observed behavior. This is because [2, 3] have found that fewer iterations of a tabu search work well for even more complex models (see section 9 on page 10).

Only (3) remains as a viable explanation as to why this behavior is observed. This remains an improvement that could be implemented in any future work.

References

[1] Dinitz, Yefim, Naveen Garg, and Michel Goemans. "On the Single- Source Unsplittable Flow Problem ." (1998): n. page. Web. 29 Oct. 2011. <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.34.5780&rep=rep1&type=pdf>.

[2] Crainic, Teodor, Michael Gendreau, Patrick Soriano, and Michel Toulouse. "A tabu search procedure for multicommodity location/allocation with balancing requirements." Annals of Operations Research . 41. (1993): 353-383. Web. 20 Feb. 2012. <http://www.springerlink.com/content/l86ktr37747m07h2/>.

[3] Ghamlouch, Ilfat, Teodor Crainic, Michael Gendreau, and . "Cycle-Based Neighbourhoods for Fixed-Charge Capacitated Multicommodity Network Design." Institute for Operations Research and the Management Studies. 51.4 (2003): 655-667. Web. 20 Feb. 2012. <http://www.jstor.org/stable/4132426>.