The Distributed Retired Traveling Salesman Problem

Daniel Seita DTS1@WILLIAMS.EDU
Ziang Zhang ZZ2@WILLIAMS.EDU
Department of Computer Science, Williams College, Williamstown, MA 01267 USA

Abstract

The use of major online travel agencies has made scheduling long-term travel much easier by allowing users to easily identify a set of flights in just a few clicks. Current travel agencies allow users to plan out long-term trips involving flights to more than two cities, but they require specific arrival and departure dates for each city and a city ordering. We present a system that does not burden the user with these decisions. Specifically, our code takes in two required inputs: a list of cities the user wishes to travel to, and a date range over which they are willing to travel, and outputs the cheapest set of flights within that range that form a valid route. It essentially solves a harder version of the Traveling Salesman Problem since costs are not constant between two cities. Under several weak assumptions, and assuming that the number of cities and days is sufficiently limited, then our algorithm should successfully find the cheapest cost flight in a reasonable amount of time. We present the theoretical and systematic components of the project and discuss empirical results.

1. Introduction and Motivation

The use of major online travel agencies, such as Travelocity[1] and Kayak[2], has made scheduling long-term travel much easier by allowing users to easily select a set of flights to purchase tickets from. People use a combination of factors to help them make their decision, such as the total price of the flights and the days they wish to land and depart from a city. Current travel agencies allow users to plan out trips involving airlines to multiple destinations (see Figure 1 for an example using Travelocity), but they require (1) specific dates for each city and (2) an ordering.

This limitation can make searching for flight routes time-consuming for users who are not restricted to arriving at cities on particular dates. For instance, suppose one resides in New York City (NYC) and wants to schedule a summer trip to Paris, London, and Tokyo from June 1 to June 20, and it does not matter to him how long he stays in a city. Suppose that it also does not matter the ordering that he arrives at the cities. Thus (for now) we will only care about the following factors[3]:

- The full flight route occurs within the date range (in this case, between June 1 and June 20).
- The flight route must arrive and leave at least once for NYC, Paris, London, and Tokyo.
- The flight route is valid and logically consistent. For instance, if the first flight takes him from NYC to Tokyo, the second flight in the route should not depart from a place other than Tokyo.

There are a vast pool of valid flight routes. We might assume that out of all these, one would want the *cheapest* route. Of course, this glosses over how the user may want to spend some number of days at each city, but this is the most general case.

Unfortunately, finding the cheapest route possible is challenging using current agencies, because they require knowing the arrival and departure dates for each city. The reason for this is simple: the problem of finding this flight route is a hard problem to solve. In fact, this problem is very similar to the well-known Traveling Salesman Problem (Applegate et al., 2007), which asks if there exists a cycle in a graph that touches each node exactly once (i.e., a Hamiltonian cycle). Each edge has a cost associated with it, and the decision version of the problem, which asks if there exists a valid Hamiltonian cycle with cost bounded by B, is NP-complete (Karp, 1972). In our case, we do not force

[1] www.travelocity.com
[2] www.kayak.com

[3] For brevity, we list a high-level overview of the problem without getting into too many technical details; Section 6 describes some of our assumptions and limitations.

Proceedings of the 1st International Conference on Distributed Systems, Williamstown, USA, 2014. Program Chair: Jeannie Albrecht.

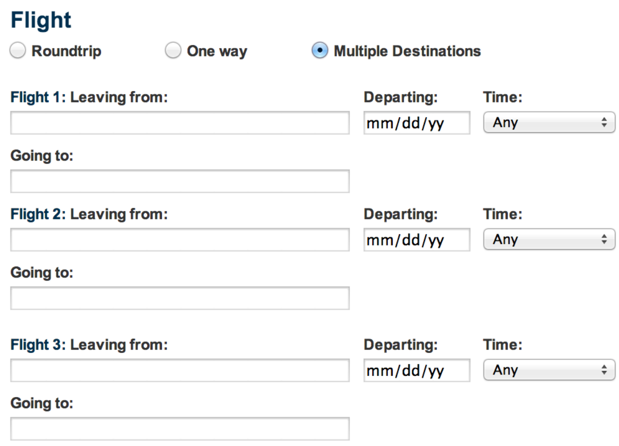

Figure 1. Travelocity lets users select multiple destinations.

the user to visit each city exactly once because the cheapest flight through a sequence of cities *may require* visiting a city more than once.

What makes our problem (likely) much harder than the Traveling Salesman Problem is that the cost of an edge between two nodes in our graph (i.e., two cities) is not constant, because flight prices frequently change. We coin our problem the "Distributed Retired Traveling Salesman Problem." The "Retired" portion comes from how we assume that whoever is planning this trip is retired, because otherwise, how would he or she have the time and money? The "Distributed" part is because we incorporate concepts from the design and practice of distributed systems to build software that can tackle this problem. We discuss the mathematical and systems portions of our solution in Sections 3 and 4, respectively.

2. Related Work

The Traveling Salesman Problem (TSP) is one of the oldest and most well-known problems in combinatorial optimization. While no exact, polynomial time solution for the decision problem is known, there is a $3/2$-approximation algorithm due to Christofides (Christofides, 1976). That algorithm was long the standard approximation algorithm until a stunning recent result in (Gharan et al., 2011), which has thus spurred a cascade of additional research relating to the TSP (e.g., (Moemke & Svensson, 2011)).

In contrast to the attention devoted to TSP, there appears to be very little research focusing on the special case of our problem, which introduces additional challenges. These can be mathematically-oriented, such as how to manage the variable costs, or systems-oriented, such as how to even obtain the actual flight costs. We have found nothing in the literature that particularly fits our problem.

3. Mathematical Component

The mathematical portion of our work uses a technique known as *binary integer linear programming*, which falls under the broader category of optimization techniques. The goal in optimization is straightforward: given a set of variables and a set of constraints, the goal is to find the best, feasible solution according to some criteria, such as cost (in which case, "best" means "minimal").

3.1. Linear and Integer Programming

One of the most commonly-used optimization techniques is linear programming, introduced by Dantzig in (Dantzig, 1963).

Definition 3.1. *A* linear programming problem *consists of three components:*

1. *a finite collection of linear inequalities or equations in a finite number of unknowns, $x_1, \ldots, x_n$;*
2. *sign constraints $x_i \geq 0$ on some (possibly empty) subset of the unknowns;*
3. *a linear function to be minimized or maximized.*

An assignment to the variables $x_1, \ldots, x_n$ satisfying the first two conditions is a feasible *solution. If it also satisfies the third, then it is an* optimal *solution (Franklin, 2002).*

Linear programming has tremendous applications, and a full list of them would be impossible to create. Some problems well-suited to linear programming include investment management, scheduling problems, and the diet problem. (The last problem concerns the rather interesting question of what is the minimum cost of a nutritionally adequate diet.) The reason for linear programming's great versatility is the ease at which constraints can be added to a model.

More specific cases of linear programming are integer and binary linear programming. (Sometimes we drop the "linear" part for brevity.)

Definition 3.2. *An* integer programming problem *is a linear programming problem with the added restriction that all variables (i.e., unknowns) $x_1, \ldots, x_n$ are integers.*

Definition 3.3. *A* binary integer programming problem *is an integer programming problem with the added restriction that all variables $x_1, \ldots, x_n$ are such that $x_i \in \{0, 1\}$.*

We see that our problem is particularly suited to binary integer programming, because all the possible flights we could take can be viewed as a set of binary random variables, each of which has value 1 if we decide to take that flight, and 0 otherwise. We now formulate this problem in more detail.

3.2. An Integer Programming Formulation

To start formulating the problem, we make it concrete what we mean by cities, days, and costs.

- We have n cities to reach; we index cities by i or j, where $i, j \in \{1, 2, \dots, n\}$.
- We have t consecutive days when we can travel: $t \in \{1, 2, \dots, m\}$.
- The minimum cost for traveling from city i to j on day t is c_{ijt}.

Using the above lets us define the following variables:

$$x_{ijt} = \begin{cases} 1 & \text{if we go from cities } i \text{ to } j \text{ on day } t, \\ 0 & \text{otherwise.} \end{cases}$$

Thus, a solution to our problem will be the full assignment to each of the above variables. The set of the binary variables that equal one will (assuming the set has cardinality k) correspond to k flights $f_1, \dots, f_k$ ordered by date.

For now, we make several simplifying assumptions, and defer a more detailed discussion about the realism of this project in Section 6. Perhaps the most important one is that we assume each flight is assigned to exactly one day. We will also force a valid solution to have flights all on different days. This means, at the moment, we do not consider issues related to (1) overnight flights (we will pretend they do not exist), (2) multiple-stop flights on the same day, and (3) possible logistic impossibilities such as flight f_i arriving at 11:59 PM and then the next flight f_{i+1} departing from the same airport two minutes later (but on the next day).

The goal is to solve this minimization problem:

$$\text{Minimize} \sum_{t=1}^{m} \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ijt} x_{ijt}, \tag{1}$$

subject to the following constraints:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} x_{ijt} \le 1 \text{ for all } t \in \{1, 2, \dots, m\}, \tag{2}$$

$$\sum_{t=1}^{m} \sum_{i=1}^{n} x_{ijt} \ge 1 \text{ for all } j \in \{1, 2, \dots, n\}, \tag{3}$$

$$\sum_{t=1}^{m} \sum_{j=1}^{n} x_{ijt} \ge 1 \text{ for all } i \in \{1, 2, \dots, n\}. \tag{4}$$

The first constraint ensures that we have at most one flight per day, which is generally too restrictive for real-life, but will suffice for now. The second constraint ensures we enter each city at least once. The third constraint ensures we leave each city at least once. These last two reinforce the notion that going through the cheapest route may mean visiting several cities more than once.

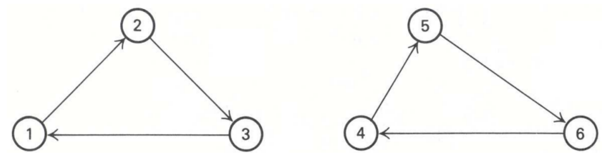

Figure 2. This represents two sets of disjoint cycles.

Sadly, these previous constraints are *not* enough to solve our problem. There are two glaring issues with the current constraints that, if we were to use them to solve this flight scheduling problem, could result in a logically inconsistent flight route.

3.2.1. Disjoint Cycles

The first problem is that the constraints do not prevent disjoint cycles. Figure 2 shows a graph with six nodes, which may represent six cities, and a possible edge assignment that is indicative of the route we take to visit all cities. Assuming the flights are all on different days, the route satisfies our constraints, because we enter and exit each city at least once, but this is not a Hamiltonian cycle.

To prevent disjoint cycles, we need to add additional constraints that correspond to all the ways we can subdivide the cities into two groups so that both of them contain at least two cities. (A subdivision where one group only has one city is already covered by our earlier constraints since we would have to leave and enter that city at least once, which requires connecting with the other group.) For instance, with the six-variable situation in Figure 2, one constraint would be

$$\begin{aligned} \sum_{t=1}^{m} (x_{14t} + x_{15t} + x_{16t} + x_{24t} + x_{25t} + \\ x_{26t} + x_{34t} + x_{35t} + x_{36t}) \ge 1, \end{aligned} \tag{5}$$

which ensures that at least one leg of the tour connects cities 1, 2, and 3 with cities 4, 5, and 6, so this corresponds to the subdivision of $\{[1, 2, 3], [4, 5, 6]\}$. We can characterize the number of constraints we need to add to prevent disjoint cycles.

Lemma 3.4. *In a problem that has $n \ge 4$ constraints, the number of ways to subdivide the cities so that each has two groups is $\binom{n}{2} + \binom{n}{3} + \cdots + \binom{n}{n-2}$.*

The reasoning is straightforward: we have two groups, and we want to pick the number of elements for one group. We can pick some number out of n elements to be in one group, and assign all the rest to the others.

One technical note is that the number of constraints from Lemma 3.4 can be halved by symmetry. Still, the number of equations needed grows rapidly with respect to n, and the need to implement these various restrictions is why integer programming is a hard task. Integer programming is, in fact, an NP-hard problem, and the binary case is NP-complete (Karp, 1972). In contrast, faster algorithms such as the Simplex method[4] are used to solve linear programming problems. Note that simply taking a solution to the linear programming problem and then rounding it to form a "solution" to the integer problem will not work, as we show in Section A.

3.2.2. Consecutive Flights in Different Cities

The second problem with the constraints posed earlier in Section 3.2, which the constraints in 3.2.1 do *not* resolve, is that we can get flight orderings that are logically inconsistent in the sense that consecutive flights may not agree in their choice of cities. It makes no sense, for instance, to have flights f_i and f_{i+1} where f_i takes the participant from NYC to Tokyo, and f_{i+1} takes him or her from Paris to London.

Unfortunately, fixing the flight ordering by adding in constraints is challenging and requires a large number of equations. Our implementation only adds in one layer of constraints for here, and defers a complete flight logic check to the code that actually generates the tree. Thus, we resolve this by just solving the problem as we would normally without the constraint, and then checking all proposed solutions for flight logic before accepting them.

For more details about this, see Section B.

TODO

3.3. Using Balas' Additive Algorithm

We will use this algorithm to solve Integer Programming. See (Balas, 1965).

TODO

4. Systems Component

Our code that solves the Distributed Retired Traveling Salesman Problem is made up of several components that

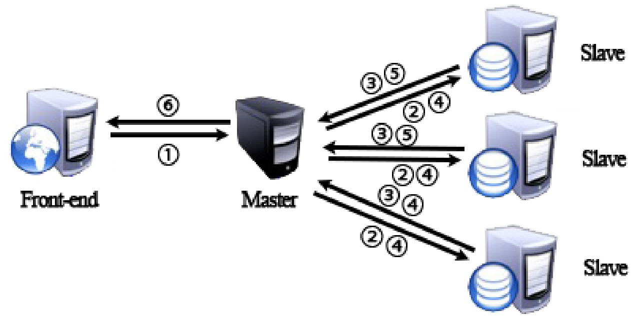

Figure 3. This represents our setup.

form a multi-tiered client-server model (for an introduction to the client-server model, see (Tanenbaum & Steen, 2006)). We have a front-end server, a master server, and a group of slave servers in the system, as shown in Figure 3. The slave servers come from machines that we use. Originally, we wanted to use PlanetLab (Peterson et al., 2006), which is a global platform for deploying and evaluating network services and allows people to use portions of other PlanetLab computers. This was too complicated to work with, so we stuck with the Williams lab computers.

TODO

4.1. Front-end Server

The front-end server acts as a point of communication between the client[5] and our master server. It is a Python script that starts on the command line and outputs introductory messages to the client so that he or she knows how to use our system. The front-end serve receives user input, forwards the request to the master server (step `1` in Figure 3), listens to the master for the result (step `6` in Figure 3), and displays the result when it is ready. The result that we print back to the user is currently the single best list of the flights ordered by departure date. If there is more than one optimal solution, it only returns one of them.

Figure 4 shows a version of our front-end server (we change its design fairly frequently). Here, the hypothetical user made a request to find the cheapest flight route that touches Denver, Atlanta, and New York City from the four day period of July 3, 2014 to July 7, 2014. Our code ran and the master server fed back the resulting flight ordering of Atlanta to Denver, then Denver to New York City, then New York City back to Atlanta.

If time permits, we may transform the command-line interface to a website, which will expand our audience of poten-

[4]The Simplex Method performs well in practice but its runtime has historically been difficult to evaluate, as in the worst case it *can* run in exponential time (Klee & Minty, 1972). The simplex algorithm has *smoothed complexity* polynomial in the input size; for details about this, see (Spielman & Teng, 2004).

[5]Somewhat impolitely, we can refer to the set of clients, minus us, as the "ignorant masses."

```
*************************** DANIEL AND LUCKY'S FLIGHT SCHEDULER ***************************

Instructions: find a series of flights by listing the start date, end date, and cities.

Specific formatting requirements:
        (1) Start and end dates must be the first and second arguments, respectively
        (2) Separate all arguments by at least one whitespace
        (3) Use MM/DD/YYYY format for dates
        (4) Spell city names (or 3-letter abbreviation) correctly
        (5) Use at least three cities

Example request: "06/11/2014 06/16/2014 CHI BOS SEA"
This searches flights from June 11 to 16 (in 2014) that touch Chicago, Boston, and Seattle.
To exit, type in "q".

******************************************************************************************

>>> 07/03/2014 07/06/2014 DEN ATL NYC
Input is in the correct format. Now solving ...

['ATL->DEN on 07/03/2014, $125', 'DEN->NYC on 07/04/2014, $137', 'NYC->ATL on 07/05/2014, $138']
```

Figure 4. This shows our front-end server.

tial clients. It is also straightforward to add more optional arguments to the command line, so the user might add in an extra flag that could indicate that he or she wishes to have a minimum of three days between flights.

4.2. Master Server

The Master server is the brain of the system. It listens to the user request sent from the front-end server (step `1` in Figure 3). When a request comes in, it first comes up with a list of flight prices on all possible dates and for all possible destinations in the problem that we need to get for the computation, distributes the flight price queries among all the slaves (step `2`), gathers all the price data back from the slaves (step `3`), and combines them. It then upload the combined data to a distributed file system to make the data available for parallel computing on all the slaves. Then it starts the parallel computing for the cheapest route using a distributed zero-one linear programming algorithm on all slaves (steps `4` and `5`). After the computation, it sends the results of the cheapest route back to the front-end server (step `6`).

TODO! This needs more information.

4.3. Slave Server

We have a group of slave servers that listens to the command of the master server and does the distributed tasks of flight price querying (using the web crawler we write) and parallel zero-one linear programming.

The slave server uses this! Be clear on that!

We will use the Matrix Airfare Search[6] database to obtain information about our flights; the information itself will be extracted with our own web crawler. We coin this "The Retired Traveler Problem," primarily because if we assume the dates are set far enough apart, it would interfere with a non-retired person's occupation. For details on the real problem, see sources such as.

TODO! This needs more information.

5. Experiments and Results

TODO

5.1. Simple Examples

TODO

5.2. Runtime Analysis

TODO

5.3. Feasibility

TODO

6. Limtiations

TODO

6.1. One Flight a Day

TODO

Explain that matrix ita will sometimes give us two or three flights for "one route" so it actually works out ok...in some ways.

6.2. One Ticket, Multiple Flights

TODO

Explain that this will give us problems.

7. Conclusion

We view this as a beginning, rather than an end, of the Distributed Retired Traveling Salesman Problem. There are several ways of proceeding from our work.

- We could improve the aesthetics of our implementation. Most prominently, our front-end server is in the form of a command-line interface, but many people do not know how to use those. Consequently, we could reach out to and would be much more comfortable with a streamlined website.

Overall, this was an extremely fun project. It was one of the most successful projects where we had only a vague plan but ended up with hopefully a working implementation that could be useful under the right circumstances.

TODO

[6] `http://matrix.itasoftware.com/`

Acknowledgments

We thank Brent Heeringa for introducing us to the world of NP-completeness, Jeannie Albrecht for teaching us how to play around with servers and design distributed systems, and Steven Miller for carving out time from his super-busy schedule to provide us with an independent study that gave us the requisite mathematical background.

References

Applegate, David L., Bixby, Robert E., Chvatal, Vasek, and Cook, William J. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, Princeton, NJ, USA, 2007. ISBN 0691129932, 9780691129938.

Balas, Egon. An additive algorithm for solving linear programs with zero-one variables. *Operations Research*, 13(4):517–546, 1965. doi: 10.1287/opre.13.4.517. URL http://pubsonline.informs.org/doi/abs/10.1287/opre.13.4.517.

Christofides, Nicos. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical Report 388, Graduate School of Industrial Administration, Carnegie Mellon University, 1976.

Dantzig, George B. *Linear programming and extensions*. Rand Corporation Research Study. Princeton Univ. Press, Princeton, NJ, 1963. URL http://gso.gbv.de/DB=2.1/CMD?ACT=SRCHA&SRT=YOP&IKT=1016&TRM=ppn+180926950&sourceid=fbw_bibsonomy.

Franklin, Joel N. *Methods of mathematical economics : linear and nonlinear programming, fixed-point theorems*. Classics in applied mathematics. SIAM, Philadelphia, 2002. ISBN 0-89871-509-1. URL http://opac.inria.fr/record=b1105716.

Gharan, Shayan Oveis, Saberi, Amin, and Singh, Mohit. A randomized rounding approach to the traveling salesman problem. In Ostrovsky, Rafail (ed.), *FOCS*, pp. 550–559. IEEE, 2011. ISBN 978-1-4577-1843-4. URL http://dblp.uni-trier.de/db/conf/focs/focs2011.html#GharanSS11.

Karp, R. Reducibility among combinatorial problems. In Miller, R. and Thatcher, J. (eds.), *Complexity of Computer Computations*, pp. 85–103. Plenum Press, 1972.

Klee, V. and Minty, G. J. How Good is the Simplex Algorithm? In Shisha, O. (ed.), *Inequalities III*, pp. 159–175. Academic Press Inc., New York, 1972.

Miller, Steven. An introduction to advanced linear algebra. Operations Research Class Notes, 2012.

Moemke, Tobias and Svensson, Ola. Approximating graphic tsp by matchings. In *Proceedings of the 2011 IEEE 52Nd Annual Symposium on Foundations of Computer Science*, FOCS '11, pp. 560–569, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4571-4. doi: 10.1109/FOCS.2011.56. URL http://dx.doi.org/10.1109/FOCS.2011.56.

Peterson, Larry L., Bavier, Andy C., Fiuczynski, Marc E., and Muir, Steve. Experiences building planetlab. In Bershad, Brian N. and Mogul, Jeffrey C. (eds.), *OSDI*, pp. 351–366. USENIX Association, 2006. ISBN 1-931971-47-1. URL http://dblp.uni-trier.de/db/conf/osdi/osdi2006.html#PetersonBFM06.

Spielman, Daniel A. and Teng, Shang-Hua. Smoothed analysis of algorithms: Why the simplex algorithm usually takes polynomial time. *J. ACM*, 51(3):385–463, May 2004. ISSN 0004-5411. doi: 10.1145/990308.990310. URL http://doi.acm.org/10.1145/990308.990310.

Tanenbaum, Andrew S. and Steen, Maarten van. *Distributed Systems: Principles and Paradigms (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. ISBN 0132392275.

Supplementary Material

Outline of Supplementary Material:

- TODO
- TODO
- TODO
- TODO

A. Integer Programming versus Linear Programming

Here, we should describe the interaction between integer and linear programming, and show why integer programming is harder and cannot be obtained by rounding from the linear solution. Our work, of course, is heavily influenced by (Miller, 2012).

TODO

B. Adding Constraints to Prevent Flight Logic Errors

Here, we should describe the constraints that we added in for flight logic. I only added in one layer of constraints.

TODO

C. Balas' Additive Algorithm

TODO

D. Additional Experiments and Examples

TODO