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

Proceedings of the 1st International Conference on Flight Scheduling, Williamstown, USA, 2014.

[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.

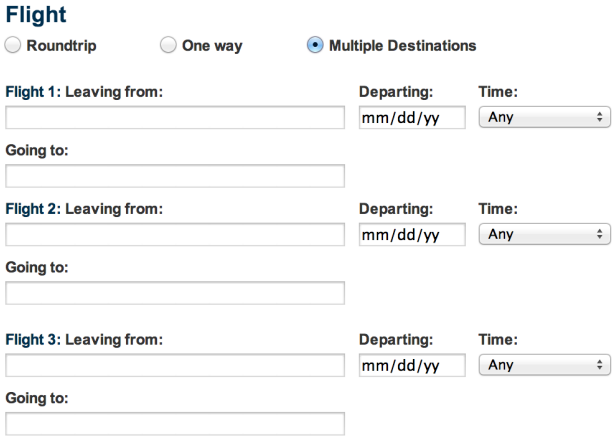

Figure 1. Travelocity lets users select multiple destinations.

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 $i, j \in \{1, 2, \ldots, n\}$.
- We have t consecutive days when we can travel: $t \in \{1, 2, \ldots, m\}$.
- 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 $f_1, \ldots, f_k$ 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:

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

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

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

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.

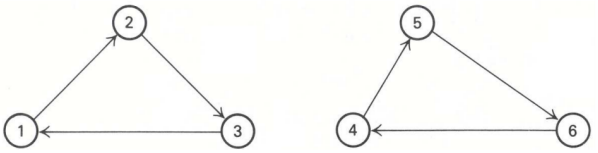

Figure 2. This represents two sets of disjoint cycles.

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

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

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 $n \geq 4$ 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 Appendix 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. Specifically, for all pairs of days t and cities c such that $t \in \{1, 2, \ldots, m-1\}$ and $c \in \{1, 2, \ldots, n\}$ we add in the following constraints:

$$\underbrace{\left(\sum_{i \neq c} x_{ict}\right)}_{\text{Term 1}} + \underbrace{\left(\sum_{j \neq c} \sum_{k \neq j} x_{jkt^+}\right)}_{\text{Term 2}} \leq 1, \qquad (6)$$

where t^+ is shorthand for $t+1$ (which explains why we don't set t to be m). Term 1 represents entering city c, and Term 2 represents leaving any city other than c. Notice that by our previous constraints, both Term 1 and Term 2 are already bounded by one, so if both are one, then this indicates that two flights on back-to-back days are not logically consistent. We therefore have the following lemma.

Lemma 3.5. *Assuming that we have n cities and n days as input, solving the binary integer programming problem using the previous constraints will return a valid, logically consistent solution.*

Of course, this doesn't prevent the problem of having more days than flights. We resolve that issue by not using constraints at all. We just run the problem and, each time a candidate solution is proposed, check its complete vector for logical consistency. Appendix B describes our handling of flight logic in more detail.

3.3. Using Balas' Additive Algorithm

To actually solve the binary integer programming problem, we use Balas' Additive Algorithm (Balas, 1965). This is a special case of the larger class of branch-and-bound algorithms, which are used to solve integer programming problems. Branch-and-bound algorithms consist of a systematic enumeration of all candidate solutions, where large subsets of fruitless candidates are discarded by using upper and lower estimated bounds of the quantity being optimized (Clausen, 1997).

Balas' algorithm makes use of the special properties of binary integer programming problems. Assuming that the costs are all non-negative, and that variables are indexed according to cost, we want to (1) set all variables to zero in order to minimize total cost, and (2) if we cannot set all variables to zero due to constraints, then we wish to set the variable with smallest index to one. What results is a rooted tree where each path of n nodes from the root indicates an assignment to the first n variables. One can use a standard depth-first search to implement the algorithm. For brevity we elide additional technical details of the algorithm and refer the reader to Appendix C.

4. Systems Component

We now switch topics and discuss the implementation side. Our code that solves the Distributed Retired Traveling Salesman Problem is made up of several components that 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 We need to fix this up ... and once I get what we're doing here, we can change the slave sever section.

[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).

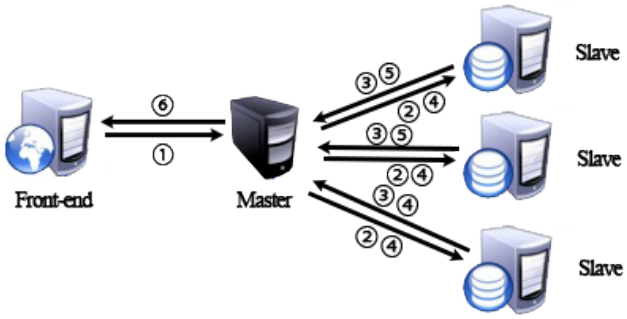

Figure 3. **TODO. We need to change this.**

```
************************** 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.

Optional extensions to add at the end of the request:
        (1) Put a number to indicate the minimum days between flights

To exit, type in "q".

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

>>> 06/11/2014 06/14/2014 CHI BOS SEA
Input is in the correct format. Now solving ...

['SEA->CHI on 06/11/2014, $161', 'CHI->BOS on 06/12/2014, $96', 'BOS->SEA on 06/13/2014, $165']
```

Figure 4. This shows our front-end server.

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), 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 frequently change its design). Here, the hypothetical user made a request to find the cheapest flight route that touches Chicago, Boston, and Seattle in the four day period of June 11, 2014 to June 14, 2014. Our code ran and the master server returned the (valid) flight ordering of Seattle to Chicago, Chicago to Boston, then Boston to Seattle.

Notice that the front-end server has an optional argument where a user can insert a number at the end of their request to indicate the minimum number of days between any two flights. (The example from Figure 4 just happened not to use it.) It is straightforward to add more of these optional arguments. If time permits, we also plan to transform the command-line interface to a website, which will expand our audience of potential clients. The front-end server does only minimal checks of the input and defers the parsing to the master server (see Section 4.2).

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

4.2. Master Server

The master server is the workhorse of our system. It takes in the user request sent in from the front-end server from Section 4.1. When a request comes in, the master sever runs a variety of checks and parses the input to make sure that the request is sound. For instance, given the optional argument where the user can specify the minimum number of days between flights, the server checks that the full date range is large enough to allow that case to happen. For the full list of checks we implement, see Appendix D. We do not catch all possible cases of possible pathological inputs, in part because it is challenging to do so in a command line argument, and also because we trust our users not to wreck the system. (There has been substantial research in the literature about trust management in systems, such as (Blaze et al., 1999); for now we eschew these details.)

After parsing the input, the master sever has two important tasks. The first is that it needs to obtain the cheapest flight that goes from cities i to j on day t for all possible combinations of i, j, and t as specified by the user. It accomplishes this by calling the crawler server, also known as the "slave server," which we discuss in Section 4.3. Once it has all the flights, it proceeds to its next major task, which is to transform the flight information and user request into inputs suitable for binary integer programming. Computing the cost vector is easy; the challenging part is making sure that the constraints are well-defined. Once the master sever has computed the necessary constraint matrix, it will call the binary integer programming solver. As discussed in Section 3.3, this runs Balas' Additive Algorithm to create the final vector of variable assignments where $x_{ijt} = 1$ indicates that the cheapest valid route involves that corresponding flight. It is possible to distribute Balas' Additive Algorithm, but for ease of implementation, the algorithm runs on only one machine and is essentially part of the master server.

Upon completion, the master server feeds the result back to the client server.

4.3. Slave Server

The main purpose of the slave server is to use a web crawler to obtain prices of real airlines. This information is then

relayed back to the master, which uses it for the binary integer programming formulation. To look up the prices, we implemented a web crawler that extracts prices from the Matrix Airfare Search[6] database, powered by ITA Software and the best flight database currently available. We currently make use of only one slave server, and the consequence is that for "small" inputs, the flight checking typically is the most time-consuming task of the full system.

5. Experiments and Results

TODO

5.1. Simple Examples

TODO

5.2. Runtime Analysis

TODO

6. Limitations

This section addresses a number of issues briefly touched upon in Section 3.2 and other areas of this paper. We avoid discussing anything that can be easily added as an extension (e.g., forcing us to start at one particular city, or making the flight route to make a stop at a city three times).

6.1. Practical Usage

One important question to consider is whether our system is practical for real-life use. Our experiments in Section 5 show that it is doable for a small enough input. Ideally, the number of cities would be limited by four, maybe five (which seems reasonable for a trip), and the number of days bounded based on the number of cities. One of the key limiting factors in our algorithm is the length of the variable vector, or the number of flights we must consider. In particular, if the user requests a flight route involving n cities over t days, then the total number of days and flight combinations possible is $n(n-1)t$, which comes from picking one of n cities to start from, one of the $n-1$ remaining cities to land to, and one of the t days for this flight to occur. This is why the bound on the days should be considered in context with the number of cities; the fewer cities we have, the more days we can allow.

Our experience shows that if $n(n-1)t$ is bounded by 100, then the problem is generally doable (i.e., completes within a few hours). A bound of 100 is still prohibitive for an extensive travel trip. If someone wants to travel for a month's worth of days, say $t = 30$, then the minimum number of flights possible for our system would be $3 \cdot 2 \cdot 30 = 180$, already quite prohibitive. The situation gets massively worse as n increases. To mitigate the effect of increasing the variable vector, we can easily distribute the flight lookup process. Specifically, if we have a period of 30 days in a request, it makes sense to utilize 30 machines and distribute all possible combinations of flights on day t to be the responsibility of the t^{th} machine.

This will work well to lower the cost of flight lookup. Unfortunately, as that process runs quicker, Balas' Additive Algorithm will become the major time bottleneck. With a large variable vector, the depth-first search process can take prohibitively long since the number of paths needed to explore is exponential. One workaround for this would be to add in more constraints, so that we can prune more often, but unless the constraints are overly restrictive — which defeats our objective of having the user supply minimal information — the number of nodes to search will still grow quickly based on the vector length (i.e., $n(n-1)t$). Balas' Additive Algorithm can be distributed by assuming that the vector starts with a certain value and then assigning the algorithm to determine the rest of them. For instance, with four machines, we can have four computers each solve Balas' algorithm under the assumption that their vector start with $[0,0]$, $[0,1]$, $[1,0]$, and $[1,1]$, respectively.

As this problem takes exponential time but can be sped up with enough machines, whether or not it can really be used in practice is an open question. Since our implementation is extremely raw but does work for small inputs, this gives us hope that future researchers can improve our system.

6.2. One Flight a Day

As stated earlier, one of our key assumptions is that every flight lasts one day and each day can only have one flight. In real life, however, it is common for people to take two, or even three, flights in a day, especially if their departure or destination cities are not one of the major airline hubs (e.g., Chicago O'Hare or Hartsfield-Jackson Atlanta).

This limitation is, in fact, often not a problem. The key is that when looking up flights between two cities, Matrix ITA defaults to searching not only for direct flights, but also for flight routes that make one or two stops. So when making a request to go from Albany to Los Angeles, our crawler will pick the cheapest flight that appears on the list, which almost always makes one stop at airports such as Chicago or Charlotte. So even though our code might output `[ALB->LAX, 07/07/2014, $400]`, there will be another flight "hidden" in our route.

A more problematic situation with our "one flight" a day limitation is that it may be cheaper to take two separate flights that touch three major cities. For example, if we wanted to travel to Chicago, Atlanta, and Denver in some

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

order over the span of six days, the cheapest route may involve going from Chicago to Atlanta on the third day, and then going from Atlanta to Denver, *also* on the third day.

Another limitation is that we only view time in terms of absolute days. If we look up a flight on day t, an overnight flights that starts on day t may be selected, and we the code may also assign a flight to start on day $t + 1$, raising some logistical issues. Even if we ignore overnight flights, we could have a flight landing at the very end of day t and the next flight departing at the beginning of day $t + 1$ with insufficient layover time. We do not have ways of facing this limitation in our system, but one possible idea is to treat time in one-hour units, rather than 24-hour units, giving us a finer grain of control. It is unclear how much this would complicate the rest of our constraints.

6.3. One Ticket, Multiple Flights

One issue related to optimality is that even if we search for all possible direct flight combinations and find the cheapest collective route, it may still not be the best possible route. Consider a hypothetical situation where a user wants to travel to Boston, Chicago, London, and Beijing. Analyzing all pairwise combinations might result in the following route: Chicago to Boston, Boston to London, London to Beijing, then Beijing to Chicago. Assume for simplicity that each flight costs $500. But there might exist a flight from Chicago to London that, according to the Matrix Airfare Search, costs $750, *but makes a stop at Boston*, despite how the cheapest direct flight from Chicago to Boston costs $500 (and similarly for Boston and London). In other words, a better route would be Chicago to London (making a stop at Boston), then London to Beijing, then Beijing to Chicago, which would cost $1750, compared to the $2000 route that our solution would provide. To resolve this, we are looking to expand our crawler so that it can detect the stopgap cities that are part of a single flight.

6.4. Web Crawler Issues

In general, our crawler is safe enough that we are confident in running it and only periodically checking back on its results. One problem, though, is that it sometimes gets stuck on a Matrix Airfare Search page after searching the price for a flight. Typically, this happens when we have it running "in the background"; bringing it back up to the foreground successfully makes it continue. There are also some problems with foreign currency units. For now, we assume that the person using this service will originate from the United States, which forces money to be in dollars. Finally, in rare cases there may be no flight found between two cities, though each time we ran into this problem, it was due to poorly formatted input.

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.

- In terms of the mathematical component, there may be other valid ways of solving binary integer programming problems, and it would be of interest to compare the feasibility of different algorithms.
- We could address the limitations we discussed in Section 6. The biggest one might be to extend the "time" constraint so that we can always ensure that our flight route has sufficient layover time at airports.
- 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.

Related to the first item described above, improving the runtime of this algorithm will hopefully be a huge part of future work. This makes sense, because our algorithm should return "correct" flight routes, but the runtime can be prohibitive for even a moderate amount of cities and days. Striking the right balance between piling on constraints/limitations (decreases runtime, increases restrictiveness) and brute-force searching (increases runtime, decreases restrictiveness) seems to be the way to go to provide a system that allows the user to plan out a cheap trip without checking all possible flight routes.

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, whose independent study provided the inspiration for this project.

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.

Blaze, Matt, Feigenbaum, Joan, Ioannidis, John, and Keromytis, Angelos D. Secure internet programming.

chapter The Role of Trust Management in Distributed Systems Security, pp. 185–210. Springer-Verlag, London, UK, UK, 1999. ISBN 3-540-66130-1.

Bradley, S.P., Hax, A.C., and Magnanti, T.L. *Applied mathematical programming*. Addison-Wesley Pub. Co., 1977. ISBN 9780201004649.

Chinnek, John. Practical optimization: A gentle introduction. Online Textbook, 2014.

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.

Clausen, Jens. Branch and bound algorithms-principles and examples. *Parallel Computing in Optimization*, pp. 239–267, 1997.

Dantzig, George B. *Linear programming and extensions*. Rand Corporation Research Study. Princeton Univ. Press, Princeton, NJ, 1963.

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.

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.

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.

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.

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.

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:

- why solutions to linear and integer programming can differ enormously
- theoretical constraints we would need to prevent flight logic errors
- a detailed discussion of our implementation of Balas' Additive Algorithm
- how we check and parse the user input in our code
- additional experiments and examples that could not fit in the text

We deemed the first eight pages of this document sufficient for presenting our results. The supplementary material is for anyone who really wants to *get* what we did, as we glossed over a few details for brevity.

A. Integer Programming versus Linear Programming

The Simplex method can usually solve linear programming problems quickly. Unfortunately, there is no efficient analogue of the Simplex method (or similar algorithms) to the case when all variables must be integers. To make matters worse, a problem can have optimal integral and linear solutions, but they may be extraordinarily different, so it is not sufficient to simply round, truncate, or investigate the "relative area" surrounding a linear programming solution. The following problem provides a concrete example and is based on the presentation of (Miller, 2012).

Problem A.1. *[The Knapsack Problem] Imagine we have a knapsack that can hold at most 100 kilograms. There are three items we can pack, each of which is worth some monetary amount per unit, and the goal is to carry as much value in the knapsack as possible. The first item weighs 51 kilograms and is worth \$150 per unit. The second item weights 50 kilograms and is worth \$100 per unit. Finally, the third item also weighs 50 kilograms, but is worth \$99 per unit. If we let x_1, x_2 and x_3 represent the amount of the first, second, and third items, respectively, and we assume that we can't take on any negative quantity, the constraint for our problem is*

$$51x_1 + 50x_2 + 50x_3 \leq 100, \tag{7}$$

and we want to maximize

$$150x_1 + 100x_2 + 99x_3. \tag{8}$$

Before presenting the solution for both the integral and linear case, think about the relative "bang of the buck" for the items. The second and third items both weigh the same, but the second is worth a dollar more, so clearly, we prefer it to the third. The first weighs just a fraction more than the second (51 versus 50 kilograms), but is worth 50 percent more in dollars (\$150 to \$100), so clearly it's the best value item. If we allow all x_j to be real numbers, the optimal answer is $x_1 = 100/51$ and $x_2 = x_3 = 0$; the value of the knapsack is about \$294.12. This makes sense, given our previous discussion. If we require all x_j to be integers, however, the optimal solution is $x_2 = 2$ and $x_1 = x_3 = 0$. The value of the knapsack is \$200, significantly lower than the linear case since the 51 kilogram weight of the first item just makes it ineligible for us to have two of it.

Incidentally, Problem A.1 demonstrates how, assuming the objective is maximization, the linear programming solution is an upper bound to the integer programming solution (and lower bound if the goal is to minimize something).

Another enlightening example of the difference between integer and linear programming is in Chapter 9 of (Bradley et al., 1977), which also demonstrates how the optimal linear and integral solutions can be arbitrarily far away from each other, in terms of distance in the plane or total cost.

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

We briefly introduced Balas' Additive Algorithm in Section 3.3 for solving binary integer programming problems, and in this section we present a deeper treatment. Our presentation is heavily influenced by (Chinnek, 2014).

C.1. Canonical Form

Balas' algorithm first requires the input to be converted to canonical form, if it is not done so already. Letting the variables and their corresponding costs be $x_1, \ldots, x_n$ and $c_1, \ldots, c_n$, respectively, canonical form is when the following are satisfied:

- The objective function is to minimize $\sum_{j=1}^{n} c_j x_j$.
- All objective function coefficients (i.e., variable costs) are non-negative.
- The m constraints are all inequalities of the form $\sum_{j=1}^{n} a_{ij} x_j \geq b_i$ for $i \in \{1, 2, \ldots, m\}$.
- All variables are ordered according to their costs, so for $x_1, x_2, \ldots, x_n$, we have $0 \leq c_1 \leq c_2 \leq \cdots \leq c_n$.

Incidentally, our code to solve Balas' algorithm *does* include checking and conversion capability, which is necessary for some constraints, such as the one requiring at most one flight per day. We also order variables by cost, so any final vector of, say, $[1, 0, 0, 1, 0, 0, 1]$ means that we pick the three flights that correspond to the cheapest, fourth cheapest, and seventh cheapest (i.e., the most expensive one).

While it may seem like requiring the input to be in canonical form is a heavy restriction, in fact it is relatively simple to do the conversion. Ordering the constraints should not be challenging in an implementation, given enough debugging to ensure that indices and other aspects line up correctly. With a less than or equal bound in one of the constraints, we multiply through by -1. With negative cost coefficients, a change of variables is required, with transforming x_i into $1 - x_i'$. Finally, to handle equality constraints, we can transform the equality into two inequalities, similar to how we can convert $x = 2$ to the equivalent condition of $x \leq 2$ and $x \geq 2$ (and then we would subsequently change the first one to be $-x \geq 2$).

As we mention in the text, the idea of Balas' algorithm is that we want to assign as many variables to zero as possible since we know that coefficients are non-negative. In almost all cases, if we tried to do this at the start, we would violate one of the constraints. So the next step is to try and set the *cheapest* variables to be one, because if we can satisfy the constraints, we prefer to do it with the cheapest variables possible.

Our implementation follows a depth-first search algorithm to enumerate all possible solutions. In other words, we form a rooted tree where a path corresponds to some assignment of variables, and each node has two successors in the tree, corresponding to a selection of 0 or a selection of 1 for the current variable, where the "current" variable is determined by the level of the tree (i.e., the root node corresponds to an empty path $[\ldots]$[7], its two successors correspond to the paths of $[1, \ldots]$ and $[0, \ldots]$, and so on).

But a word of caution — if there are four cities and twenty days to travel, that means we have $4 \cdot 3 \cdot 10 = 120$ total variables, which means listing all solutions would mean we have 2^{120} leaf nodes. This is clearly impractical, and the hope is that Balas' algorithm can avoid searching the full space of solutions by using look-ahead and pruning techniques, which we now describe.

C.2. Cost Look-Ahead

Suppose we are at a given node of the DFS tree with path $[x_1, x_2, \ldots, x_j, \ldots]$, so the first j variables are assigned and the remaining $n - j$ variables are unknown. We can consider two cases for the value of x_j:

- Suppose $x_j = 1$. Then we need to check if the current path so far can lead to a best-cost solution. If we check the constraints, and find that the assignment $\mathcal{X} = [x_1, x_2, \ldots, x_j = 1, 0, 0, \ldots, 0]$ works (i.e., at least the last $n - j$ variables are zero), then we compute the cost of $\mathcal{X}$ and compare it to the current best cost known. If $\mathcal{X}$ is better than the current best cost, then we save its path and cost. Furthermore, *there is no need to continue in this DFS branch*, because any further path must start with the first j assignments in $\mathcal{X}$, and any extra one is going to force a subsequent solution $\mathcal{X}'$ to be costlier than $\mathcal{X}$. This is the power of knowing that the costs are ordered, and that each variable is binary. Without either of those assumptions, Balas' algorithm would not work!

[7] We use the ellipses as shorthand for "the rest of the variables are unknown."

- Now suppose $x_j = 0$. Here, it is important to assume that the assignment $\mathcal{X} = [x_1, x_2, \ldots, x_j = 0, 0, 0, \ldots, 0]$ is *infeasible*. The reasoning is that if it were feasible, then we would not be in this node at all because then the previous node would have already been feasible. Since we are assuming best-case costs each time, this means that the DFS would not have continued after the previous node was expanded. Thus, we must assume $\mathcal{X}$ is infeasible. Then we set the *next* variable $x_{j+1} = 1$, and then after that, check to see if the solution $\mathcal{X}' = [x_1, x_2, \ldots, x_j = 0, x_{j+1} = 1, 0, 0, \ldots, 0]$ is feasible. If it is, there is no need to expand this DFS path any further.

As we will soon show, it is also important to be able to prune away impossible paths so that we do not need to do this "cost look-ahead" every time.

C.3. Pruning

We might have feasible solutions, but it is also important to be able to stop checking a DFS path if we can tell that its best-case flight route price is always going to be more expensive than the current best cost. Given a current path assignment $\mathcal{X} = [x_1, \ldots, x_j, \ldots]$, the way we do this is by analyzing each constraint one by one. The constraints are all of the form $\sum_{j=1}^{n} a_{ij} x_j \geq b_i$ for $i \in \{1, 2, \ldots, m\}$. So the key is to determine the *largest possible* value of the left hand side. If that value is still less than b_i, we can avoid expanding that path any further.

To make the left hand side as large as possible, set each *unknown* variable x_i to be zero if its corresponding coefficient is negative, and one if otherwise. For example, if we had the assignment $[x_1 = 1, x_2 = 0, x_3, x_4]$ (where the last two are unknown) and a constraint $x_1 - 5x_2 + 2x_3 - 3x_4 \geq 4$, then knowing the first two values mean this constraint is immediately $1 + 2x_3 - 3x_4$. Set $x_3 = 1$ and $x_4 = 0$ and we get $1 + 2 \ngeq 4$, so this can be pruned.

D. Checking User Input

We should describe the full list of checks we make, and see what might go wrong...

E. Additional Experiments and Examples

TODO