Optimal Nadir Observation Scheduling

Name

Affiliation

Abstract

We present optimal algorithms for nadir (instrument pointing straight down) observation scheduling for spacecraft with fixed orbits. We show that this problem is NP-complete and present a mixed integer/linear program formulation and a flow network formulation, each capable of solving instances of these problems optimally. Neither technique dominates the other, and we characterize their respective advantages and disadvantages.

Introduction

Nadir is the opposite of zenith; it is basically "straight down." Orbiting spacecraft often have immobile imaging instruments. Generally, such spacecraft maintain a fixed orientation with respect to the body that they are orbiting, therefore most instruments point straight down, or nearly straight down. This is called either nadir or off-nadir observing. Figure 1 (right) shows a nadir pointing spacecraft orbiting a spherical body and the area that is in view of the instrument. (The arrow indicates the direction of travel.)

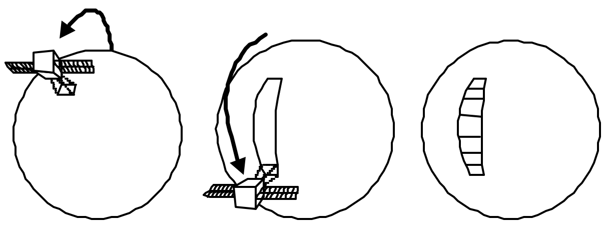

Figure 1 Swath and segments of a nadir pointing spacecraft

In these cases, the instrument gathers data along a fixed trajectory called a *swath*. Figure 1 (center) shows the swath that could be imaged by the orbiting spacecraft. In practice, the swaths are broken into smaller swaths called *segments*. A segment is the smallest area that can be individually imaged. Therefore, a single observation equates to a segment. It is important to note that segments are both areas that could be imaged and intervals of time for the observation. Figure 1 (left) shows some example segments.

The purpose of such spacecraft is to image various areas, called *targets*, according to the investigator's priorities. Figure 2 shows an example of a target.

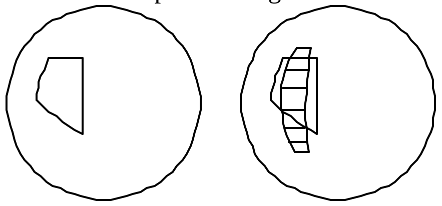

Figure 2 A target, without and with segments

Overlapping segments imply a potential for waste if the overlapping area is collected and transmitted more than once. Figure 3 shows more segments as a result of a subsequent orbit by the spacecraft. (The arrow indicates the direction of travel.) Note that the segment in the center of the target (shaded area) overlaps one of the previous segments.

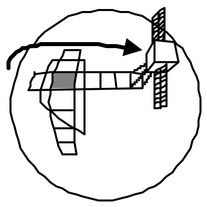

Figure 3 Another swath and its associated segments, with an overlapping segment (shaded area)

Collecting overlapping segments is a problem because there are limits to how many segments we may collect. This is usually due to limited on-board memory, and limited downlink times and capacities. Not surprisingly, segments are usually carefully chosen to reduce overlap. But choosing the best segments can be problematic, especially for large numbers of observations. This problem is called the *swath segment selection problem.*

Problem

The Swath Segment Selection Problem (SSSP) consists of selecting a subset of data collection opportunities from all that are available such that the most valuable data are collected given the limitations of bounded memory and bounded communication. In practice, the segments (candidate observation times) are usually determined beforehand.

Definitions

The swath segment selection problem (SSSP) consists of: a set of target polygons, a set of swath segments, a set of downlinks, and a memory capacity. From the segments, choose a subset that respects the memory capacity and downlink capacity that maximizes the area of the targets downlinked.

A *shard* is a sub-section of a target. We use shards to represent pieces of the target that can be gathered and downlinked. They are the natural result of combining the target and the edges of the segments. The term *shard* is taken from the basic appearance of these polygons as shards of broken glass, especially in larger problem instances. Figure 4 (left) shows a set of example shards derived from the segments and the target. We draw dotted lines around the inside of each shard for easier identification. For our formulation, we will assume that the targets are already broken into shards, and will therefore only refer to the shards.

Thus, more formally, given:

- a set of shard polygons H where each $h \in H$ is a simple (but possibly concave) polygon in the Euclidean plane and the real-valued reward function $reward(h)$ that represents the reward for collecting the shard (piece of a target), e.g. $\{\alpha, \beta, \chi, \delta, \varepsilon, \phi\}$ (see xxx, left),
- a set of swath-segments S where each $s \in S$ is a convex quadrilateral in the Euclidean plane (these are not necessarily parallel due to viewing angle warping the projection of the instrument on the surface to be imaged), e.g., {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11,} (see xxx, right), the real-valued capacity cost function $cap(s)$ that represents the memory required to store the segment, and the function $shards(s)$ that returns the set of shards that intersect with s.
- a set of downlinks D where each $d \in D$ has a real-valued capacity function $cap(d)$ that represents the maximum amount of memory that can be communicated during the downlink, e.g., $\{D_1, D_2\}$.
- a memory limit m that represents the maximum amount memory that can be stored between downlink operations.
- a route **R** that is a permutation of the segments and downlinks, e.g., 1, 2, 3, 4, 5, 6, D_1, 7, 8, 9, 10, 11, D_2.

Find a solution route **R'** that is a subset of **R** that maximizes the reward of the *collected shards* while *respecting memory capacity limits*.

The collected shards are the union of $shards(s \in \mathbf{R}')$. Respecting memory capacity limits means that the sum of all segments previous to a given $d \in D$ but not previous to any other $d \in D$ must be less than m and less than $cap(d)$.

We presume real-valued functions $area(s \in S)$ and $area(h \in H)$ that gives the area of any segment or shard.

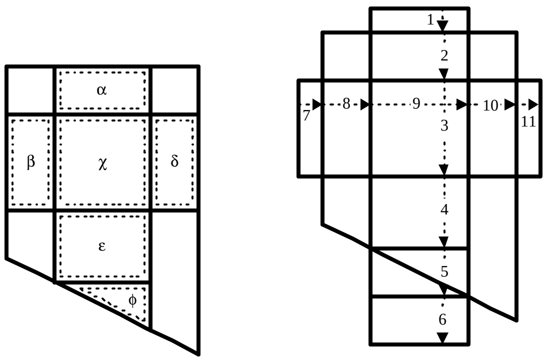

Figure 4 Example shards and segments

We continue with two solution examples. Let us consider that there is a downlink opportunity after between segments 6 and 7 that can transfer up to 32 units of memory, and a downlink opportunity after segment 11 that can transfer the same amount of memory. The total memory on board is 33 units. Table 1 shows us the area of each segment. One solution **R'** is 2, 3, 5, D_1, 8, 10, D_2 (as shown in the left of Figure 5). This respects the downlink limits as segments 2, 3, and 5 (32 total units) can be handled by the first downlink, and segments 8 and 10 (16 total units) can be handled by the second downlink. The collected shards are *a*, *b*, *c*, *d*, and *f*. The quality of this solution is the collected target area (the summed area of the collected shards) – 44 units. The optimal solution, however, is 2, 4, D_1, 5, 8, 9, 10, D_2 (as shown in the right of Figure 5). 2, 4, and 5 (28 total units) handled by the first downlink, and 8, 9, and 10 (32 total units) handled by the second. This results in all shards being collected, yet still respects the downlink limits, for a quality of 56 units.

segment	area
1	4
2	8
3	16
4	12
5	8
6	8
7	4
8	8
9	16
10	8
11	4

shard	area
α	8
β	8
χ	16
δ	8
ε	12
ϕ	4

Table 1 Example segment and shard areas

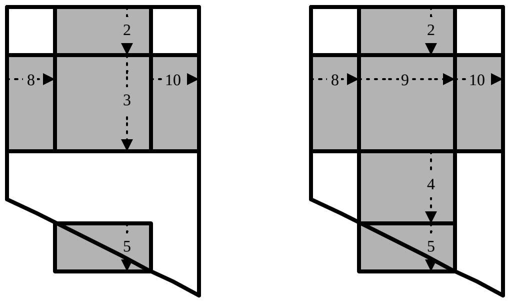

Figure 5 Example solutions

Characterization

Not surprisingly, the SSSP is NP-complete.

1. SSSP is contained in NP in that its associated decision problem can clearly be solved using a non-deterministic algorithm that guesses **R'** and then a polynomial algorithm that computes the validity and quality of the solution, and finally compares the quality of the solution with the bound given for the decision problem.

SSSP contains NP because number partitioning can be reduced to it.

2. Number partitioning is reducible to the SSSP. As a reminder, number partitioning is, given a set of positive integers V, partition V into two subsets $V1$ and $V2$ that do not intersect, yet contain all elements of V in their union. The sum of the elements of $V1$ and $V2$ must be equal.

The idea underlying our transformation is to determine $V1$ by forcing a selection that just exactly fits into $V1$. Our transformation is as follows.

Let $c = \frac{\sum_{i=1}^{|V|} V_i}{2}$, i.e. c is the target sum for each of $V1$ (and $V2$).

Let $D = \{d\}$, with $cap(d)=c$.

$\forall\ i \mid V_i \in V$,add a shard $h = \{ (0, i), (V_i, i), (V_i, i+1), (0, i+1) \}$to H, i.e., for each value in V, add a rectangle with the same area as the value to the targets.

$\forall\ i \mid H_i \in H$, add a segment s of the same dimensions as H_i to S.

Let **R** be all s in S, in order, followed by d.

Having the targets and segments, solve the SSSP. If the SSSP has a value of c, then a perfect partition exists. For example, if $V = \{1, 2, 4, 10, 15\}$, then

$D = \{d\}$,
$cap(d) = 16$,
$H = \{$
$\{ (0,1), (1,1), (1,2), (0,2) \}$,
$\{ (0,2), (2,2), (2,3), (0,3) \}$,
$\{ (0,3), (4,3), (4,4), (0,4) \}$,
$\{ (0,4), (10,4), (10,5), (0,5) \}$,
$\{ (0,5), (15,5), (15,6), (0,6) \}$
$\}$

$S = H$,
and $\mathbf{R} = S_1, S_2, S_3, S_4, S_5, d$.
One candidate solution would be $\mathbf{R'} = S_1, S_5, d$

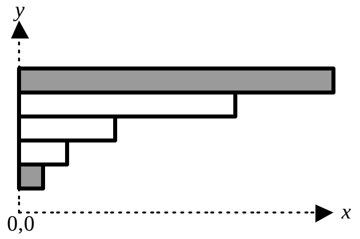

Figure 6 Number partitioning to SSSP example

Figure 6 shows the targets in our example, along with a solution that is shaded in gray. By 1 and 2, we conclude that the SSSP is NP-complete □.

Solution Techniques

The state of the art system for solving the SSSP is the ASTER scheduling algorithm [Muraoka 1998][Muraoka 1998]. This algorithm partitions the problem into a single day's worth of observations, and then solves each day's observations in order using a greedy technique (greatest reward), breaking ties by choosing the earliest segment. There may be many months in a given problem. This is not an optimal technique, but it is very fast and allows the users to try various schedules very quickly. This technique would give the solution **R'** = 2, 3, 5, D_1, 8, 10, D_2, as is illustrated by Figure 5.

Our goal, however, is to solve problems as optimally as possible. In fact, our solutions are the first optimal solutions for these problems. Here we describe the various solution approaches that we have implemented for the SSSP.

Forward Dispatch

Forward Dispatch (FD) is a greedy algorithm that collects rewards in time order until its capacity threshold is met. The approach is simple: we add sets that give us positive reward in order of **R** until adding more would oversubscribe the capacity of either m or the next downlink. Not surprisingly, this approach does not fare too well versus other approaches with respect to quality, but it is the fastest approach; thus we understand its allure. For example, if we examine sets in order of their membership for our example domain, we would arrive at the solution **R'** = 2, 3, 5, D_1, 8, 10, D_2, as is illustrated by Figure 5.

Depth first Branch and Bound

Depth first branch and bound (DFBnB) is a depth first search that prunes the search based on an estimate of how well the search could possibly progress given how well the search has progressed already. g is considered the reward of a partial solution, and h is considered to be the estimated reward of finishing the solution. h is known as the heuristic reward estimator. If h never underestimates this reward, then we say that h is admissible, and decisions made by using h will still lead to an optimal solution. In practice, we search until we find a solution, and then backtrack and try other paths, looking for the best solution. If it is ever the case that $g + h$ for our current point in the search is less than or equal to the reward of our best known solution, then we need not search further as we know that no solution lies along this path that is better.

When formulating a problem using DFBnB, one must first decide on a search space, a reward estimator, and a node ordering heuristic. The search space is the definition of the decisions that when made delivers a solution and in what order to make the decisions. The node ordering heuristic determines which of a set of possible decision values to apply first, second, and so on. As previously discussed, the reward estimator estimates the reward of finishing a partial solution.

There are two problem formulations that we examine that use this approach. The first is an Integer Programming (IP) formulation; the second is a flow network formulation. The search space for both integer programming and flow network can be thought of as a binary tree, where at each level a subset is chosen for inclusion or not. They differ primarily in their reward estimators and the node ordering heuristics. Both of these techniques if allowed to search to conclusion give optimal solutions.

Integer Programming Formulation

Integer programming expresses a problem in the form of a variable vector for which integer values need to be selected. The criterion for comparing different solutions is a linear equation on the vector. The criteria for correct solutions are a set of linear inequalities on the vector. So the search space is the vector assignment space. The reward estimator is the assignment of continuous values to the variable vector, otherwise known as the linear programming relaxation, which is a tractable computation. The node ordering heuristic binds the integer variable that is closest to being integer to its nearest value first, and then other values second. We now describe the IP formulation in detail.

A linear program consists a vector $\mathbf{c}$, a matrix A, and a vector $\mathbf{b}$. The goal is to assign values to a vector $\mathbf{x}$ such that we maximize (in our case) the linear objective function:

$$\sum_{i=1}^{|\mathbf{x}|} \mathbf{c}_i \mathbf{x}_i \ .$$

An integer program includes the extra constraint that all values for $\mathbf{x}$ be integral. A mixed integer program consists of both integer and continuous values for various members of $\mathbf{x}$. Our formulation is a mixed integer program. Therefore, we wish to identify our variables (indices of $\mathbf{x}$) (as well as which variables are integer), our objective function (values for $\mathbf{c}$), and our constraints (The set of inequalities of $\mathbf{b}$ on $\mathbf{x} \times A$).

A common reward estimator used in solving integer programs is to relax the problem and assume all variables are continuous, resulting in a polynomial time solvable relaxation. This is interesting in that it implies the existence of a solution for a more capable system—specifically, a system that can deal with arbitrarily divided members. With respect to the SSSP, if a highly capable spacecraft existed that could "cut-up" the images and register them perfectly, saving only what it needed, then this problem becomes polynomial. Unfortunately, no such spacecraft exists.

Variables

We include a variable for each member of $\mathbf{R}$ (both downlink's and segments) and each element in H. The notation for the variable corresponding to the first element of $\mathbf{R}$, or $\mathbf{R}_1$, is $x(\mathbf{R}_1)$. Therefore, $\mathbf{x} = x(\mathbf{R}_1), x(\mathbf{R}_2), \ldots x(\mathbf{R}_n), x(H_1), x(H_2), \ldots x(H_m)$ where $n = |\mathbf{R}|$ and $m = |H|$. Variables corresponding to sets are binary and a 1 indicates that the set is included in the solution. Variables corresponding to depots are continuous and indicate the amount of capacity transferred by that depot. Variables corresponding to elements are continuous from 0 to 1, but in practice will always be binary due to the objective function and constraints. These indicate which individual elements we collect a reward for.

Objective Function

$$\text{maximize} \sum_{i=1}^{|H|} reward\,(H_i)\,x(H_i)$$

Constraints

We assume all variables are greater than or equal to zero. The segment selection variables are binary (note that a set may appear more than once on a route):

$\forall\ i \,|\, \mathbf{R}_i \in S\ , x(\mathbf{R}_i)$ integer

$\forall\ i \,|\, \mathbf{R}_i \in S\ , x(\mathbf{R}_i) \leq 1$

Element selection variables are at most 1:

$\forall\ h \in H\ , x(h) \leq 1$

Downlink usage variables may not exceed their capacity:

$\forall\ i \,|\, \mathbf{R}_i \in D\ , x(\mathbf{R}_i) \leq cap(\mathbf{R}_i)$

A shard variable's value can only be non-zero if at least one of its associated segments is selected:

$$\forall h \in H, -x(h) + \sum_{\forall i, \mathbf{R}_i \in S \wedge h \in shards\,(\mathbf{R}_i)} x(\mathbf{R}_i) \geq 0$$

The sum of capacity use before each depot cannot exceed the maximum capacity: (Overflow constraints.)

$$\forall i \,|\, \mathbf{R}_i \in D, \sum_{\forall j, j<i \wedge \mathbf{R}_j \in S} cap(\mathbf{R}_j)\,x(\mathbf{R}_j) - \sum_{\forall k, k<i \wedge \mathbf{R}_k \in D} x(\mathbf{R}_k) \leq m$$

The sum of capacity use after each depot cannot be sub-zero: (Underflow constraints.)

$$\forall i \,|\, \mathbf{R}_i \in D, -x(\mathbf{R}_i) + \sum_{\forall j, j<i \wedge \mathbf{R}_j \in S} cap(\mathbf{R}_j)\,x(\mathbf{R}_j) - \sum_{\forall k, k<i \wedge \mathbf{R}_k \in D} x(\mathbf{R}_k) \geq 0$$

If it were the case that all capacity must be regained at each depot, then we use the following constraint instead of the previous:

$$\forall i \mid \mathbf{R}_i \in D, -x(\mathbf{R}_i) + \sum_{\forall j, j<i \wedge \mathbf{R}_j \in S} cap(\mathbf{R}_j)x(\mathbf{R}_j) - \sum_{\forall k, k<i \wedge \mathbf{R}_k \in D} x(\mathbf{R}_k) = 0$$

This results in a very large formulation, but most of the variables are continuous, leaving only the set selection variables as binary.

Search ensues thusly:

1. Set the best score found so far to be impossibly bad.
2. Clear the best solution found so far.
3. Solve the linear programming relaxation.
4. If the value of the integer programming relaxation is worse than the best score found so far, backtrack. If impossible, return the best solution found so far.
5. If all variables are bound, we have a new best solution. Remember it and backtrack. If impossible, return the best solution found so far.
6. Select the integer variable (subset inclusion variable) that is most closely integer in value (i.e., apply the node ordering heuristic).
7. Bind the variable to the value that it is closest to. If it is 0.5 in value, bind to 1.
8. Proceed to 3, above. If asked to backtrack, proceed to the next line.
9. Bind the variable to the other value (since this is 0-1, only one branch here).
10. Proceed to 3, above. If asked to backtrack, proceed to the next line.
11. Return the best solution found so far.

Network Flow

This approach searches the space of set-inclusions, starting with a solution that includes no sets. The cost estimator is a network flow formulation of the problem. The node ordering heuristic orders selections based on the reward to capacity cost ratio of a set. We refer to this technique as *Flow*.

Network Flow Formulation

This provides a heuristic reward estimator that is faster than the linear programming formulation while providing similar information. We use this as an admissible heuristic in conjunction with a branch and bound algorithm described later. The goal is to generate a flow network that represents the flow of capacity usage through the problem. It is important to note that this formulation does have some limitations; most importantly it assumes that the reward for an element scales with its capacity cost, which might not be the case. Under those circumstances, the IP formulation is the more accurate, and probably should be used. The rest of this subsection describes the construction of the flow network. Given our previous example, we would first add a node called *src* and a node called *snk* to our graph (see Figure 7.)

Figure 7 Source and sink nodes for the flow network

We then add a node for each *h* in *H*, and add an edge representing the reward for collecting the shard from *src* to the elements node (see Figure 8). **Note that capacity cost and reward must be equivalent for the network flow formulation to be used!**

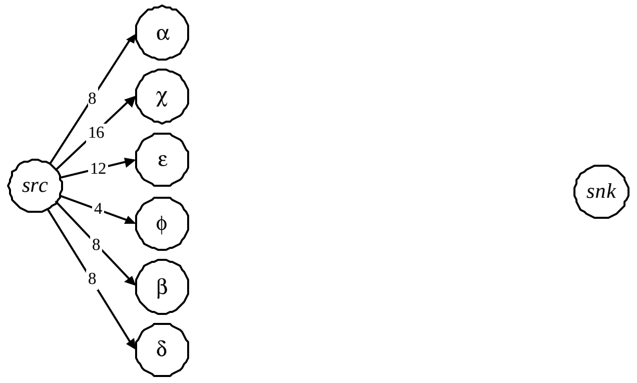

Figure 8 Shard nodes and reward edges

Then, for each segment, we add a node and add an edge from each shard that the segment contains with the same capacity as the reward for the shard (see Figure 9). Note that in our example, shard χ belongs to segment 3 and segment 9.

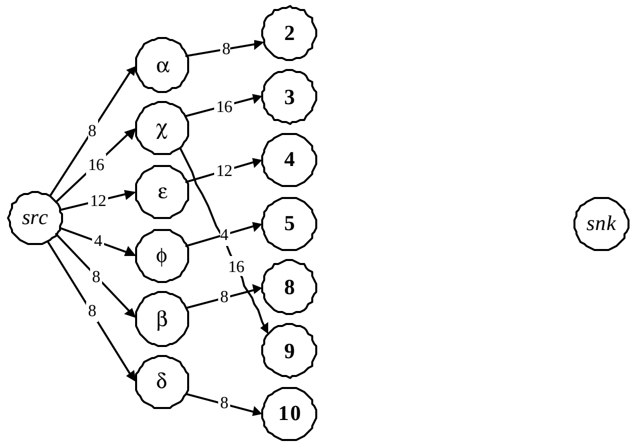

Figure 9 Segment nodes and shard-intersection edges

Then, for each downlink, we add two nodes. One node collects all of the segments, and we designate it the *in* node. The other sends the collected reward to the sink. So, for all segments of the leg previous to the downlink, we add an edge of the same capacity as the segment from each segment to the in node. We then add an edge of the same capacity as *m* between the in node and the second node. Finally, we add an edge of the downlink capacity from the second node to *snk*. Figure 10 shows the final flow network for our example problem, where the edge-labels represent the capacity of the edge. Figure 11 shows one possible solution to the network flow problem, where the edge-labels represent the flow, and Figure 12 shows what this solution would imply when translated back into a swath segment selection problem.

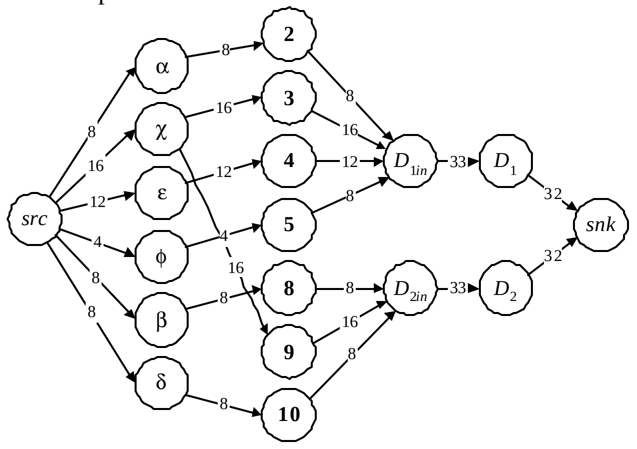

Figure 10 Flow Network example

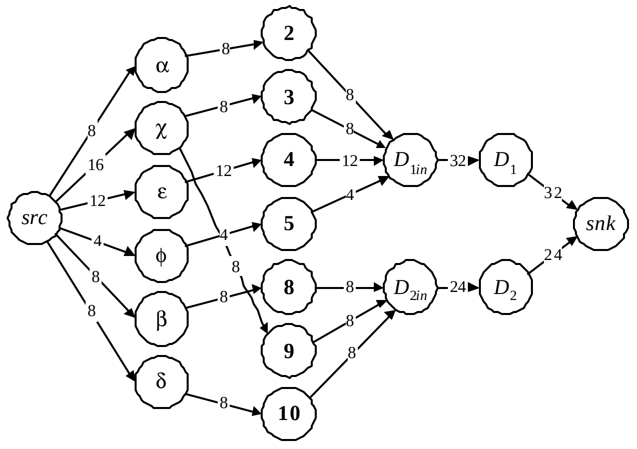

Figure 11 Example network flow solution

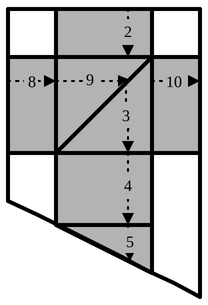

Figure 12 Implied relaxed swath segment selection problem solution

Given the network flow relaxation, we are tempted to simply use this as our heuristic reward estimator for a branch and bound search, but this would be a mistake. As soon as a segment is selected during search, the flow network needs to be adjusted to reflect the lost capacity due to the segment selection. Often, waste is associated with this allocation. We have implemented an incremental flow network capacity update that allows us to change the capacities upon segment selection and de-selection. The time complexity of the update is proportional to the total number of depot's and the total number of elements associated with the segment being updated.

Now we have a good heuristic reward estimator that we can apply to a traditional branch and bound search. We need a node ordering heuristic that takes a partial solution and the search options available and orders the options accordingly, hoping to find good solutions early. Specifically, we need to take the partial solution $\mathbf{R'}$ and consider which segments $s \in \mathbf{R}$, $s \notin \mathbf{R'}$ to include. The basic approach is to calculate the reward/cost for including each set not yet included. More formally,

$$score(i \mid \mathbf{R}_i \in S \wedge \mathbf{R}_i \notin \mathbf{R'}) = \frac{\sum_{\forall h \in shards(\mathbf{R}_i) \wedge h \notin \mathbf{R'}} reward(h)}{cap(\mathbf{R}_i)}$$

i.e., the score for selecting a segment is the sum of the reward of each of the shards covered by the segment that have not yet been collected divided by the capacity cost of the segment. Ties are broken randomly.

Search ensues thusly:

1. Set the best score found so far to be impossibly bad.
2. Clear the best solution found so far.
3. Solve the network flow relaxation.
4. If the value of the network flow relaxation is worse than the best score found so far, backtrack. If impossible, return the best solution found so far.
5. If all segments have been examined for inclusion or exclusion, we have a new best solution. Remember it and backtrack. If impossible, return the best solution found so far.
6. Select the segment that gives the best reward/cost value (i.e., apply the node ordering heuristic) and modify the network flow to force its inclusion.
7. Proceed to 3, above. If asked to backtrack, proceed to the next line.
8. Modify the flow network to force the exclusion of the segment.
9. Proceed to 3, above. If asked to backtrack, proceed to the next line.
10. Return the best solution found so far.

Results

We report "first solution" time and quality results for Forward Dispatch, Flow, and Integer Programming for many sizes of random problems. Problems are randomly generated SSSPs.

Easily computable metrics that appear to reflect on the scale of the problems and the quality of solutions are the number of shards in H for each problem and the initial network flow approximation, thus we report these for the sizes of problems here. We report results for 100 instances per size, with a time cutoff of 1 hour. It is important to note that Forward Dispatch required less than 1 second for any instance.

Forward dispatch dominates in terms of generating a fast solution. But the time cost of Flow is minimal compared to the solution quality. Integer programming returns an optimal solution, but does not outperform Flow, and for relatively small problems doesn't terminate. Thus, in terms of any-time performance, the best strategy appears to be to first use forward dispatch followed immediately by Flow. (See Figure 14 and Figure 15.) But, as mentioned earlier, in the case that the reward for a member of a set is not proportional to the capacity cost of a set, then IP is the stronger formulation.

Figure 13 compares a solution for a typical SSSP reduced to a MVCSCSP and solved using FD and Flow. Shading indicates the solution area.

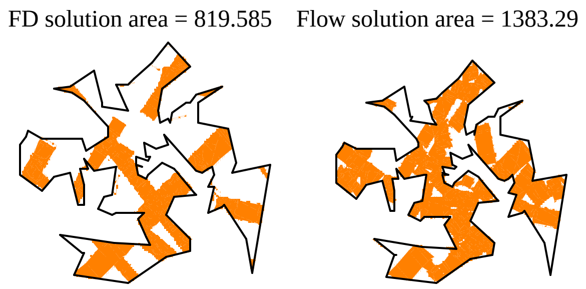

Figure 13 FD and Flow comparison

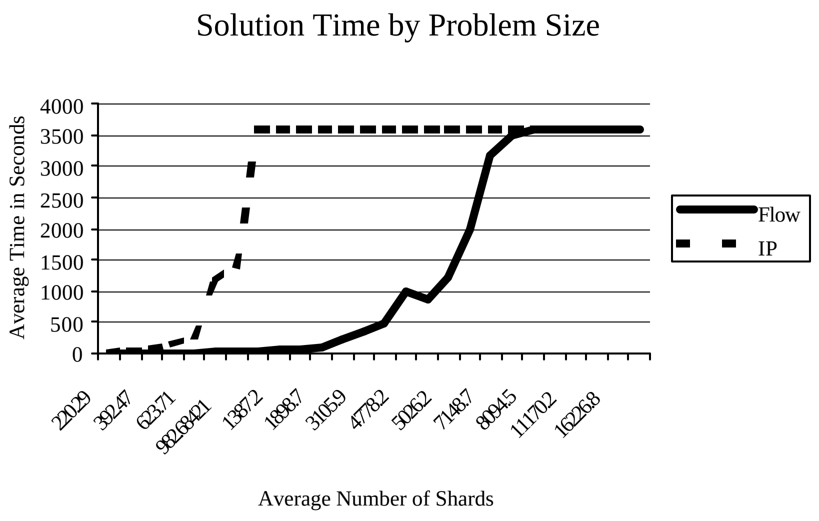

Figure 14 Comparative time performance

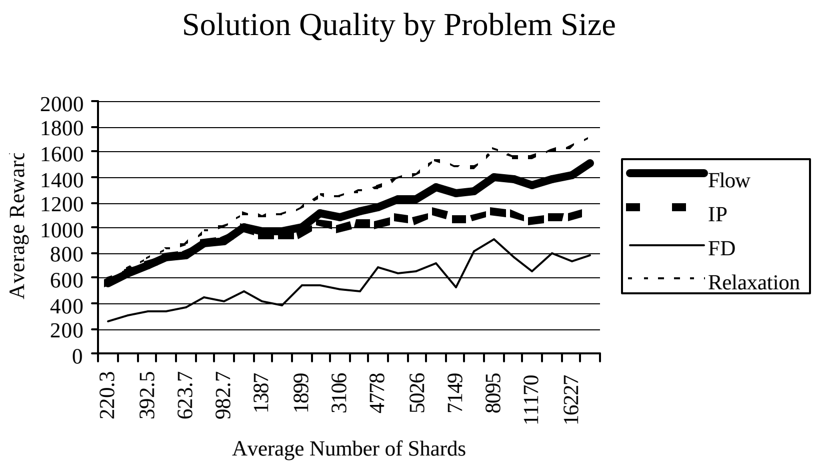

Figure 15 Comparative quality performance

Related Work

The best previous problem solver for SSSPs is the ASTER system [Muraoka 1998]. They use a combination of forward dispatching and greedy maximization to find a solution. Their algorithm is deterministic and very fast. Basically, they subdivide the problem into separate legs, and schedule each leg. As it turns out, a leg corresponds to a day of operations. For each leg, they include the segment that has the best reward/cost value, until no more segments can be accommodated. They break ties by choosing the earlier segment.

Work on a somewhat similar problem with more degrees of freedom is reported by [Frank 2000]. In this case, a route for an aircraft-borne off zenith observatory must be planned that maintains pointing at celestial targets over an interval of time. The route is flexible (as opposed to our fixed routes) and more constraints (maximum fuel usage, round trip travel, etc.) are considered, but on-board memory is not a prohibitive factor.

Work on DFBnB is common in the literature, starting with [Papadimitriou and Steiglitz 1982], with interesting any-time aspects of DFBnB in [Zhang 2000]. Our flow network implementation came from [Corman *et al]*. Work on integer programming for use in operations research is a booming field, for a good overview read [Schrijver 1986], and for a good example of a polyhedral solution to a combinatorial optimization problem see [Ruland 1986]. Any NP-completeness proof benefits from a read of [Garey and Johnson 1979].

Acknowledements

To be added after review.

References

[Corman *et al*] Thomas H. Corman, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[Frank 2000] J. Frank. "SOFIA's Choice: Automating the Scheduling of Airborne Observations" Proceedings of the 2d NASA Workshop on Planning and Scheduling for Space, March 2000.

[Garey and Johnson 1979] M.R. Garey and D.S. Johnson. Computers and Intractability. W.H. Freeman and Company, San Francisco, 1979.

[Karp 1972] R. M. Karp "Reducibility among combinatorial problems." In R. E. Miller and J. W. Thatcher (eds.) *Complexity of Computer Computations*, Plenum Press, New York, 85-103.

[Muraoka 1998] H. Muraoka, R. H. Cohen, T. Ohno, and N.Doi. ASTER Observation Scheduling Algorithm. SpaceOps 98. 1998, 1-5 June, Tokyo, Japan.

[Papadimitriou and Steiglitz 1982] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, NJ, 1982.

[Ruland 1986] K. Ruland. Polyhedral solution to the pickup and delivery problem. Washington University, Sever Institute of Systems Science and Mathematics. http://rodin.wustl.edu/~kevin/dissert/dissert.html (Dissertation). St. Louis Missouri, 1995.

[Schrijver 1986] A. Schrijver. Theory of Linear and Integer Programming, Wiley, 1986.

[Zhang 2000] W. Zhang. "Depth-First Branch-and-Bound versus Local Search: A Case Study." In Proceedings of the 17th National Conference on Artificial Intelligence (AAAI 2000), pages. 930-935, Austin, Texas, 2000.