

The swath segment selection problem: extending AI search techniques to a novel real-world problem

Content Areas: search, scheduling, heuristics, geometric reasoning, automated reasoning

Abstract

We introduce the Swath Segment Selection problem (SSSP). The SSSP consists of a constrained geometric covering problem and a capacitated resource problem. It comes from the real-life problem of scheduling on- and off-times for air- or space-borne instruments that image a target by flying over and collecting a "swath" of information. This information needs to be stored on board and downlinked. We provide a formal description of the SSSP, an NP-completeness proof, a random problem generator, and several solvers, including a forward-dispatch greedy solver, an integer program solver, and a depth-first branch and bound solver. We compare the results of the solvers, with a mix of results.

1 Introduction

The Swath Segment Selection Problem is a real-life problem of imaging areas using aircraft or spacecraft. It 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. We will employ a hypothetical synthetic aperture radar (SAR) mission as an example.

Our SAR mission consists of an orbiting spacecraft with a SAR instrument, a series of downlink opportunities when data may be transferred to the ground from the spacecraft, a collection of targets from that the scientists wish to gather data, and a series of imaging opportunities. An example (rather short) operation would be to turn on the SAR for 10 seconds while flying over Antarctica and store the data on-board, then downlink the data when the receiving antenna on the Earth is visible.

The challenging aspect of such a mission is to choose times for gathering data and subsequently downlinking it that maximize the amount of coverage for the scientists without overrunning our available memory or communications capacities. This problem has two interesting aspects: 1) a constrained geometric relationship between the overlapping swaths and area to be imaged and 2)

memory constraints with respect to on-board storage and downlink capacity.

In the rest of this paper we describe and compare solutions for this problem, as well as the problem generators used. We also characterize the problem's complexity. We point out a few limitations of our approach, and conclude with a description of related works.

2 Definitions

Having the basic idea of the SSSP, we now informally define our terminology.

A *target* is the area of interest to be imaged or "covered." Figure 1 shows an example target.

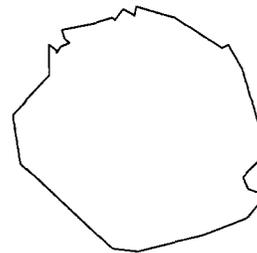


Figure 1 Example Target

A *swath* is an area that represents the coverage of the instrument during an interval. It is usually rectangular and quite "stretched." Figure 2 shows two example swaths. The delta shapes indicate the direction of travel.

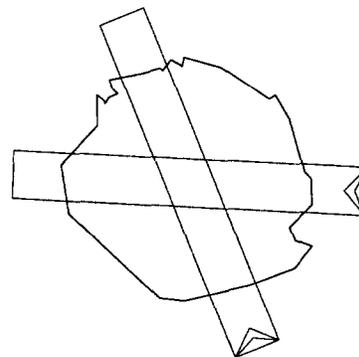


Figure 2 Example Swaths

A *segment* is a sub-section of a swath that is also in itself a swath in that it is usually rectangular and is an area that represents the coverage of the instrument during an interval. Segments are derived from the natural interactions of the lines describing the target and the lines describing the swaths. Thus, the problem is to select a collection of segments. Figure 3 shows a set of segments derived from the target and the swaths.

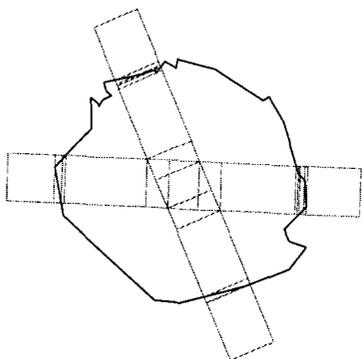


Figure 3 Example Segments

A *downlink* is an opportunity to transfer data in memory to the ground. A downlink is over an interval that is disjoint from those of the swaths. Downlinks result in restored memory on-board and “awarded” data. Of course, downlinks are of limited capacity.

A *shard* is a sub-section of the 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 shows a set of example shards derived from the segments and the target. We draw dotted lines from the edges of each shard to its center for easier identification.

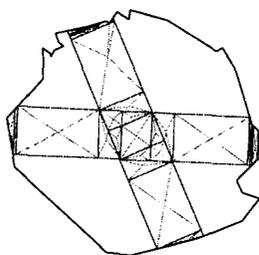


Figure 4 Example Shards

3 SSSP

We continue with a formal specification of the swath segment selection problem and a characterization of its complexity.

3.1 Formulation

The swath segment selection problem (SSSP) consists of: a set of 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.

We presume real-valued functions $area(s)$ and $area(p)$ that gives the area of the segment or polygon. We also assume that the area of the polygons or segments is proportional to the amount of memory required to store them.

Thus, more formally, given:

- a set of polygons P where each $p \in P$ is a simple (but possible concave) polygon in the Euclidean plane,
- a set of swath-segments S where each $s \in S$ is a convex quadrilateral in the Euclidean plane, including the edge-valued functions $startEdge(s)$ and $endEdge(s)$ that return non-adjacent edges that represent the start and end of the segment accordingly, and including the real-valued functions $startTime(s)$ and $endTime(s)$ that return the starting time and ending time of the segment.
- 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, and real-valued functions $startTime(d)$ and $endTime(d)$ that represent the interval of the downlink d .
- a memory limit m that represents the maximum amount memory that can be stored between downlink operations.

Note: we shall use the notation $x < y$ to mean the same as the expression $endTime(x) < startTime(y)$. We use the notation $consec(x, y)$ to mean that x and y are consecutive and are members of the same set, or, more formally,

$$\exists S \mid x \in S \wedge y \in S$$

$$\wedge x < y$$

$$\wedge \forall z \mid z \in S \wedge z \neq x \wedge z \neq y, (z < x \wedge z < y) \vee (z > x \wedge z > y)$$

We also use the notation for D_i to mean the i 'th element in D ordered according to *consec*. We assume that no intervals overlap.

A solution is a subset $S' \subseteq S$ such that:

- $\sum_{s \in S' \mid s < D_1} area(s) < m$

i.e. the sum of the areas of the selected segments that occur before the first downlink must be accommodated by the available memory. We call this sum the pre-utilization or *preUtil*. The computation of *preUtil* is linear in the number segments preceding the first downlink.

$\forall d_1 \in D \mid \exists d_2 \in D, \text{consec}(d_1, d_2)$

$$\sum_{s \in S' \mid d_1 < s < d_2} \text{area}(s) + \text{carry}(d_1) < m$$

i.e. the sum of all the areas of the selected segments that occur between two consecutive downlinks d_1 and d_2 plus the amount of memory not accommodated by previous downlinks must be accommodated by the available memory. We define the real-valued function $\text{carry}(d \in D)$ as such:

$$\text{carry}(D_i) = \min(0, \text{preUtil} - \text{cap}(D_i))$$

$$\text{carry}(D_i) = \min\left(0, \text{carry}(D_{i-1}) + \sum_{s \in S \mid D_{i-1} < s < D_i} \text{area}(s) - \text{cap}(D_i)\right)$$

i.e. $\text{carry}(d)$ returns the memory that d couldn't accommodate and therefore must keep on-board until the next downlink. The computation of all values for carry is linear in the total number of downlinks and segments.

The quality of a solution is the area of the geometric intersection of all selected segments with the target polygons.

Additionally, we add that the computation of the set of shards H is polynomial in the total number of edges in the problem. For each $h \in H$ we assume the real valued function $\text{area}(h)$, as well as the function $\text{segments}(h)$ that returns a subset of S whose members are those segments that geometrically contain h . We make use of H during our solution formulations.

Finally, we include the function $\text{shards}(s \in S)$ that returns the set of shards that geometrically intersect s .

3.2 Characterization

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 S' 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.

2. SSSP contains NP because the subset sum problem (a well-known NP-complete problem) can be reduced to it.

The subset sum problem can be stated as such: given a set of (possibly redundant) values V , compute a subset $G \subseteq V$ such that the sum of all values in G equals a constant value c . Let q be sum of all values in V .

Our transformation is as follows. Consider a collection of targets T that consists of one square for each value in S . Each side of the square is measured as the square root of its associated sub-set sum value, thus its area is the same as the value. Place each of these squares in the plane such that the linear extension of their edges never crosses another square (e.g. diagonally). Now, for each square, overlap it with a vertical segment of the exact same dimensions as the square, and impose a random

temporal ordering on the vertical segments. Add a single downlink d_v to D with $\text{cap}(d_v)=c$. Assign times to d_v such that all vertical segments are previous to it. Add a set of horizontal segments in the same manner as the vertical segments to S . Impose a random ordering on the times for the horizontal segments ensuring that each succeeds d_v . Finally, add a downlink d_h to D that succeeds all swaths with $\text{cap}(d_h)=q - c$. Note that no partial usage of a segment facilitates the solution because this imposes a necessary waste of capacity, the total capacity of the system being q . A solution to the SSSP that is of value q is also a solution to the subset sum problem. Simply include in G each value that is associated to a vertical segment that is selected. Figure 5 shows an example subset sum problem and its solution using the SSSP reduction. Note: continuously varying the timing of the segments is no help, thus even the continuous case of this problem is NP-hard.

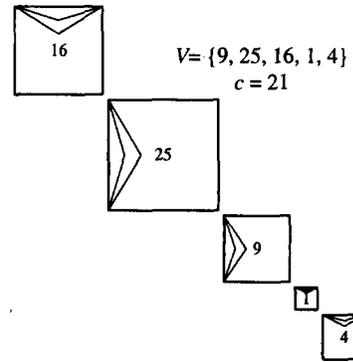


Figure 5 Subset Sum reduction example

By 1 and 2, we conclude that the SSSP is NP-complete \square .

4 Solutions

Here we describe the various solution approaches that we have implemented for the SSSP.

4.1 Forward Dispatch

Forward Dispatch (FD) is the only automated solution of those presented here that existed previously to this work to the best of our knowledge. In practice, an SSSP was formulated and solved using a forward-dispatching algorithm and then tweaked by hand until a "good enough" schedule was discovered.

The approach is simple: we add segments in temporal order until we oversubscribe the system. If adding a segment results in an over-subscription, we back up and remove the offending segment from our selection, and proceed. Not surprisingly, this approach does not fare too well vis a vis other approaches with respect to quality, but it is the fastest approach; thus we understand its allure.

4.2 Integer Programming

On the other hand, the integer programming (IP) approach gives us optimal answers, but it is the slowest

approach (even in generating interim sub-optimal answers). 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 minimize (or maximize) the objective function. 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}).

Variables

We include a variable for each segment, shard, and downlink.

Objective Function

$$\max \sum_{i=1}^{|H|} \text{area}(h_i) x(h_i)$$

Constraints

We assume all variables are greater than or equal to zero. The segment variables are binary:

$$\forall s \in S, x(s) \text{ integer}$$

$$\forall s \in S, x(s) \leq 1$$

Shard variables are at most 1:

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

Downlink variables may not exceed their capacity:

$$\forall d \in D, x(d) \leq \text{cap}(d)$$

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

$$\forall h \in H, -x(h) + \sum_{s \in \text{segments}(h)} x(s) \geq 0$$

The sum of memory before each downlink cannot exceed the total memory capacity: (Overflow constraints.)

$$\forall d \in D, \sum_{s \in S \wedge s \prec d} \text{area}(s) x(s) - \sum_{d' \in D \wedge d_p \prec d'} x(d_p) \leq m$$

The sum of memory after each downlink cannot be sub-zero: (Underflow constraints.)

$$\forall d \in D, -x(d) + \sum_{s \in S \wedge s \prec d} \text{area}(s) x(s) - \sum_{d' \in D \wedge d_p \prec d'} x(d_p) \geq 0$$

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

A common technique 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 segments. Thus, 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 polyno-

mial. Unfortunately, no such spacecraft exists. But, we can formulate our relaxation in a more efficient framework than a linear program.

4.3 Network Flow Relaxation

This provides a solution that is faster than the linear programming relaxation while providing the same 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 information through the problem. The rest of this subsection describes its construction.

Our network flow graph $G=(V,E)$ is a directed, edge labeled graph with the real valued edge label function $\text{cap}(e \in E)$ that returns the capacity of the edge, a source vertex $\text{src} \in V$, and a sink vertex $\text{snk} \in V, \text{snk} \neq \text{src}$. The solution is a function $\text{flow}(e \in E)$ that indicates the amount of flow across any edge and a real value f that represents the total flow through the network from the source to the sink. Our formulation is as follows:

Vertices

Add vertices src and snk to V .

$\forall h \in H$, add a vertex $v(h)$ to V .

$\forall d \in D$, add a vertex $v(d)$ to V .

$\forall d \in D$, add a vertex $v(d)_{\text{mem}}$ to V .

Edges

$\forall h \in H$, add an edge $e=(\text{src}, v(h))$ to $E, \text{cap}(e)=\text{area}(h)$.

$\forall d \in D$, add an edge $e=(v(d), \text{snk})$ to $E, \text{cap}(e)=\text{cap}(d)$.

$\forall d \in D$, add an edge $e=(v(d)_{\text{mem}}, v(d), \text{snk})$ to $E, \text{cap}(e)=m$.

$\forall d_1 \in D, \exists d_2 \in D, \text{consec}(d_1, d_2)$, add an edge $e=(v(d_1), v(d_2)_{\text{mem}})$ to $E, \text{cap}(e)=m$.

$\forall h \in H, \forall s \in \text{segments}(h) | \exists d \in D, d$

$\forall h \in H,$

$\forall s \in \text{segments}(h) |$

$\exists d \in D, s \prec d \wedge \neg \exists d' \in D, d' \neq d \wedge s \prec d' \wedge d' \prec d,$

add an edge $e=(v(h), v(d)_{\text{mem}})$ to $E, \text{cap}(e)=\text{area}(h)$, i.e. for every segment associated with a shard, identify the nearest subsequent downlink d and add an edge from the shard vertex to the memory vertex associated with d .

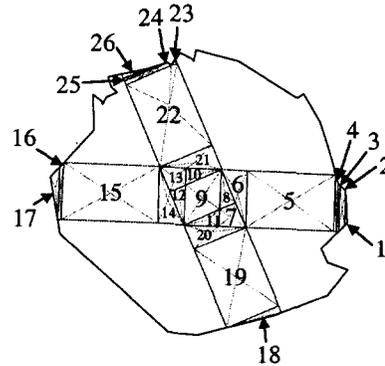


Figure 6 Labeled Shards

Figure 6 shows all 26 shards from our previous example. Figure 7 shows the equivalent flow network given 2 downlinks, one after the first swath and one after the second swath.

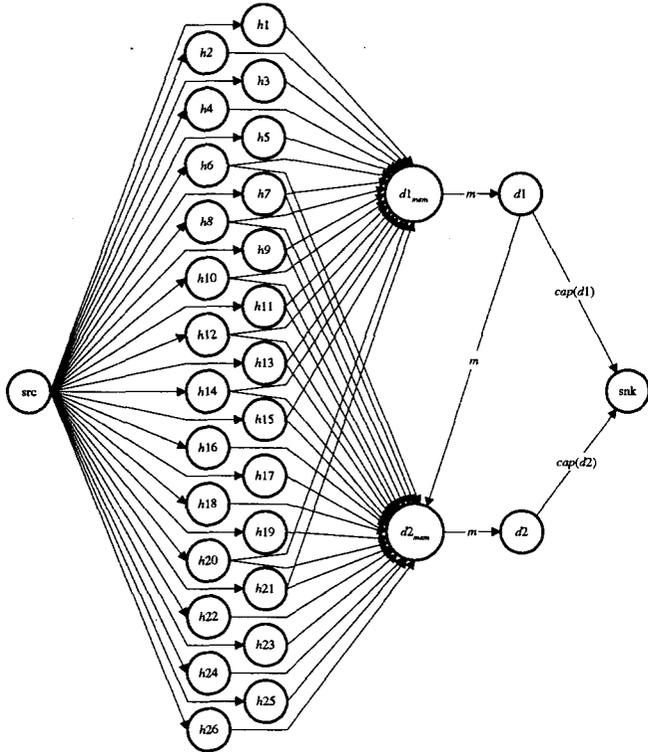


Figure 7 Flow Network example

4.4 Depth first Branch and Bound

We assume knowledge on the part of the reader of depth first branch and bound (DFBnB), the role of the g value, the h value, and the node ordering heuristic.

Given the network flow relaxation, we are tempted to simply use this as our h 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 downlinks and the total number of shards associated with the segment being updated.

Given this, we now have a good heuristic estimator, or h function, that we can apply to a traditional branch and bound search. We employ the following node ordering heuristic:

$$priority(s \in S \wedge s \notin S') = \frac{\sum_{h \in shards(s)} area(h)}{area(s)}$$

i.e. the priority value of a segment s is the area of the unclaimed shards geometrically contained by it divided by its area. Ties are broken randomly.

Performance

We report “first solution” time and quality results for Forward Dispatch, Depth First Branch and Bound (greedy, in this case), and Integer Programming for many sizes of random problems. But first, we describe our problem generator.

Problem Generation

The problem generator creates swaths, downlinks, and polygons.

We generate polygons by selecting points normally distributed around a single point, and then solve its associated Euclidean traveling salesman problem using the insert-furthest heuristic and then removing edge crossings.

Swath and downlink generation interact because we do not generate problems which have simultaneous swath and downlink intervals. Thus we first generate a set of candidate times for each, and then fold them together by postponing either the swaths or the downlinks (chosen randomly) at each temporal intersection in a forward dispatching manner.

We chose a duration, width, length, orientation, and position for each swath from normal distributions. We chose a capacity for each downlink in a like manner.

Finally, we chose the memory capacity from a normal distribution.

Results

Easily computable metrics that appear to reflect on the scale of the problems and the quality of solutions are the number of shards in each problem and the initial network flow approximation, thus we report these for the sizes of problems here. The actual instances are available on our web site, <to be filled in after review>. We report results for 100 instances per size, with a time cutoff of 1 hour.

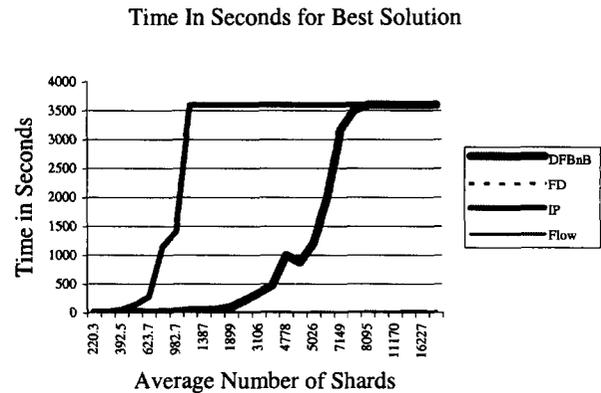


Figure 8 Comparative Time Performance

We see in general that the forward dispatch algorithm dominates in terms of generating a fast solution. But the time cost of DFBnB is minimal compared to the solution quality. Integer programming returns an optimal solution, but does not outperform DFBnB, and for relatively small problems never terminates. Thus, in terms of any-time performance, the best strategy appears to be to first use forward dispatch followed immediately by DFBnB. (See Figure 8 and Figure 9.)

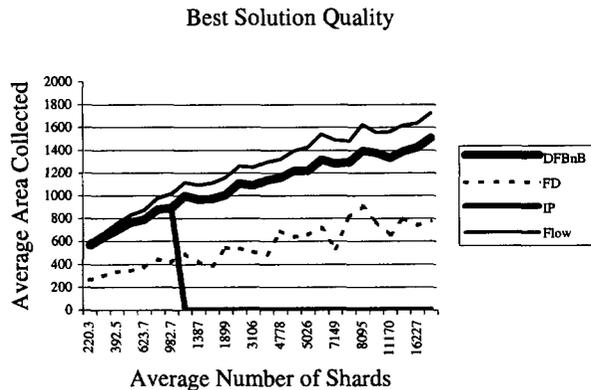


Figure 9 Comparative Quality Performance

Figure 10 compares a solution for a typical problem using FD and DFBnB. Shading indicates the solution area.

FD solution area=819.585 DFBnB solution area=1383.29

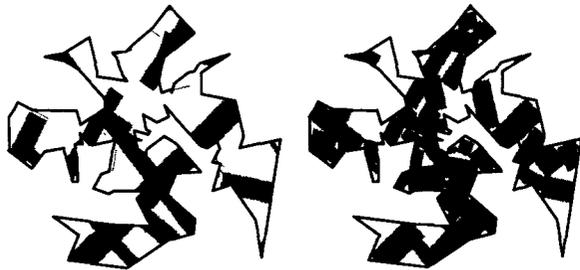


Figure 10 FD and DFBnB comparison

5 Conclusions

The Swath Segment Selection Problem provides an interesting real-world problem that can work as a test-bed for various search strategies. We see that extending these strategies to handle the SSSP has brought to light some limitations of these approaches, and we have overcome some of them. Due to the ease of creating these problems, we hope that the community will embrace them as a challenge and extend this seminal work.

6 Related Work

Work on a somewhat similar problem with more degrees of freedom is reported by [Frank 2000]. 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 polydral solution to a combinatorial optimization problem see [Ruland 1986]. Any NP-completeness proof benefits from a read of [Garey and Johnson 1979].

Acknowledgements

We would like to acknowledge the work of our coworkers at <to be filled in after review>. Also, my advisor <to be filled in after review> has provided excellent advice with respect to direction and formalism. Parts of this work are funded by <to be filled in 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.
- [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.