Improved Dynamic Programming for the Shortes Path Problem with Resource Constraints in DAG

Lishun Zeng*
IT Department
China Southern Airlines
Guangzhou, China
e-mail: zenglsh@csair.com

Mingyu Zhao
IT Department
China Southern Airlines
Guangzhou, China
e-mail: zhaomingyu@csair.com

***Abstract*—In this work, we present the design and implementation of an efficient yet flexible solution framework for the shortest path problem with resource constraints, which is a critical subproblem in a bunch of optimization problems in the airline industry. We especially focus on directed acyclic graph since graphs in our application are acyclic by definition. Several improving strategies have been devised to improve the performance of the classical labeling algorithm based on dynamic programming. We apply the skyline algorithms to speed up the computation of pareto-optimal labels. We improve data locality and computational efficiency of label dominance with a specialized memory management strategy. Bi-directional dynamic programming techniques are also applied to further improve the performance. Numerical experiments on internal benchmark problems show that our implementation is in general an order of magnitude faster than that of the Boost Graph Library.**

Keywords-dynamic programming; optimization; shortest path; graph; skyline

I. Introduction

The *shortest path problem with resource constraints* (SPPRC) [1] searches for a shortest path in a directed graph subject to some additional resource constraints defined along the path, e.g. the total time should not be greater than some bound. Besides its direct application, SPPRC is also a key ingredient in the successful solution framework of *column generation* [2] for a wide variety of real-life optimization problems, e.g. vehicle routing [3] or crew scheduling in the airline industry [4]. In this framework, problems are usually modeled as *set partitioning* or *set covering* problems with a huge number of variables (i.e. columns), and variables of interest are generated on the fly by sequentially solving one or several subproblems modeled as SPPRC.

For ease of explanation we focus on the classical SPPRC in this paper though the methodology can be easily generalized to other variants. To be specific, SPPRC is formulated on a graph with a set of nodes and a set of arcs. Each arc is associated with a cost and one or more nonnegative resources. The cost and resource consumption of a path are simply those of all arcs along the path combined. SPPRC then seeks a path from a given source to a given destination such that the cost is minimized and the consumption of each resource is not greater than its given upper bound. Fig. 1 illustrates a typical small example considered in [5], where a pair of values $(cost, time)$ is associated with each arc. The objective of this example is to minimize the cost from A to F such that the total time is not greater than 14. The optimal path is A-B-C-D-F with cost 13 and time 13.

II. Motivation

A common solution technique for SPPRC is a labeling algorithm based on dynamic programming proposed in [6], which is a generalization of the classical Bellman-Ford algorithm for the shortest path problem without resource constraints; other approached e.g. network reduction applied for SPPRC are discussed in [7]. A label on a node represents a path from the source to this node, as well as the cost and resource consumptions along the path. From the initial label representing the trivial path with a single node, i.e. the source, the dynamic programming approach extends paths into all directions in case they are resource feasible. The algorithm is also referred as the multi-label algorithm because in contrast to the shortest path problem without resource constraints, multiple labels at the same node could be *incomparable*. One path may cost less but consume more in (at least) one resource than a second path. In this case, the former path may become infeasible after extension while the latter one remains feasible. On the other hand, one path may be better off in all criteria. In this case, the other path is *dominated* and should be removed by the idea of optimal substructure in dynamic programming. The efficiency of the multi-label algorithm heavily relies on the label dominance tests to filter *pareto-optimal* labels and remove dominated labels which cannot be extended to an optimal solution.

A generic and open-source implementation of the standard labeling algorithm for SPPRC could be found in the Boost

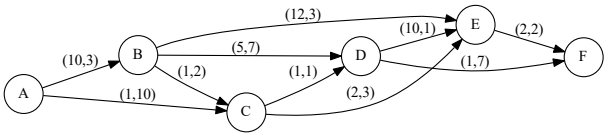

Figure 1. A small SPPRC example.

978-1-5386-8339-2/18/$31.00 ©2018 IEEE

```
1: Label first_label, cur_label, new_label
2: Set of Labels unprocessed_labels
3: initialize first_label with rc
4: INSERT(unprocessed_labels, first_label)
5: while unprocessed_labels != ∅ do
6:    cur_label := EXTRACTMIN(unprocessed_labels)
7:    if cur_label is not dominated then
8:       node i = ResidentNode(cur_label)
9:       perform pairwise dominance check over labels resident at i
10:      mark all dominated labels as dominated
11:      DELETE all labels which are dominated AND processed
12:   end if
13:   if cur_label is not dominated then
14:      mark cur_label as processed
15:      for each arc (i, j) of node i do
16:         new_label := ref(cur_label)
17:         if new_label is not feasible then
18:            DELETE(new_label)
19:         else
20:            INSERT(unprocessed_labels, new_label)
21:            INSERT(set of labels resident at j, new_label)
22:         end if
23:      end for
24:   else
25:      DELETE(cur_label)
26:   end if
27: end while
```

Figure 2. Labeling algorithm in BGL.

Graph Library (BGL), which is part of the well-known Boost C++ libraries. The pseudo-code of this implementation is illustrated in Fig.2. Users are required to define the following concepts in the algorithm:

- *rc*: the resource container of the initial label (usually zeroes for the cost and all resources).
- *ref*: the resource extension function defined on each arc in the graph, which controls how the cost and resource consumptions are aggregated along the path and the resource feasibility.
- *dominance*: the condition when one label is dominated by another one at the same node.

One may refer to the online documentation of BGL for further detail.

The algorithm in BGL is designed for a general directed graph. In many applications, the underlying graph is a *directed acyclic graph* (DAG) by definition, e.g. a time-space network where each arc represents an activity in a certain period of time; apparently, no arc goes from the future to the past. This is the case in our common applications. Thus in this work, we concentrate the discussion for DAG. The special structure of DAG can usually reduce the complexity of shortest path problems. It is very promising for the dynamic programming approach for SPPRC to benefit from the DAG structure as well.

```
1: construct a topological ordering of nodes
2: Label first_label, cur_label, new_label
3: initialize first_label with rc
4: INSERT(set of labels resident at source, first_label)
5: for each node i in order do
6:    SKYLINE(set of labels resident at i)
7:    for each cur_label resident at i do
8:       for each arc (i, j) of node i do
9:          new_label := ref(cur_label)
10:         if new_label is not feasible then
11:            DELETE(new_label)
12:         else
13:            INSERT(set of labels resident at j, new_label)
14:         end if
15:      end for
16:   end for
17: end for
```

Figure 3. Labeling algorithm in DAG.

Dominance tests among labels at the same node play a significant role in the algorithm. In pursuit of better performance, we may expect a "smarter" way to do the test than the naive implementation of two nested loops of labels. In fact, such problems have been considered in the database community known as the *maximum vector problem* or the *skyline operator* [8]. We may reduce the number of label comparison following the idea of skyline algorithms.

Moreover, notice that dominance tests are only performed between labels at the same node. Therefore, it is very attractive to organize labels in the memory in a way that labels resident at the same node are close to each other to improve memory data locality. We could design a specialized memory management strategy for this purpose.

Lastly, we consider bi-directional dynamic programming to further reduce the number of labels. Bi-directional dynamic programming has been considered to accelerate algorithms for the classical shortest path problems and the elementary SPPRC; refer to [9] for more detail. We could devise a bi-directional search specific for DAG.

III. Improvement

For DAG, there always exists a linear ordering of nodes called *a topological ordering* such that for every directed arc (i, j) from node i to node j, i comes before j. It is well-known that a topological ordering of any DAG can be found in linear time by e.g. depth-first search. It is very straightforward to modify the standard dynamic programming algorithm for DAG based on its topological ordering of nodes and process nodes one-by-one. For each node, we first perform the dominance test and obtain all pareto-optimal labels at this node, then extend these labels to successive nodes for new labels.

Obviously, any label in this DAG version is extended from a pareto-optimal label at its resident node. Notice that this is

not the case in the BGL algorithm for a general graph. In the BGL version, a label may be not dominated at first, extended, and finally found dominated by another label generated later. Therefore, the number of labels and dominance comparison could be greatly reduced for DAG.

With similar notations, the pseudo-code of the DAG variant is illustrated in Fig.3. This makes a baseline for further improvements discussed in the following context of this section. The numerical experiment results of the impact of these improvements are presented in Section IV.

A. Skyline Algorithm

The dominance test of Fig.3 is hidden in the skyline operator. Proposed in [8], the skyline operator filters out the set of pareto-optimal points from a potentially large set of points. A naive implementation of the skyline operator is composed of two nested loops of labels. For a pair of labels l_1 and l_2, we perform the dominance test in both directions: we check whether l_1 is dominated by l_2, and whether l_2 is dominated by l_1.

Several skyline algorithms are proposed to accelerate the process. We choose to apply the skyline algorithm called sort-filter-skyline proposed in [10] for both its simplicity and efficiency. The idea of the algorithm is that, if we presort the labels such that the former l_1 can never be dominated by the latter l_2 in the ordering, then we can perform the dominance test in just one direction. Thus we replace a certain amount of label comparison in dominance test by label comparison in sorting. Then we can benefit from the well-studied sorting algorithms. For the classical SPPRC we discuss, a lexicographical ordering of resources (including cost) can do the trick. If l_1 comes before l_2 in such an ordering, at least one resource of l_1 is less than l_2 unless they are exactly the same, thus l_1 is never dominated by l_2.

B. Memory Management

The significance of memory management might be underestimated in the literature. In case of large instances, the number of labels and dominance tests can be vast. If the two labels in the dominance test are located close to each other in the memory, we can accelerate memory access to take advantage of CPU caching.

As discussed in Section II, the dominance tests are only performed between two labels resident at the same node. We design a memory management strategy based on this observation. The memory for labels are organized in blocks, each of which can hold a certain amount of labels. At the beginning of the algorithm, every single node is associated with its own memory block. When a new label is generated, it is contained in the associated memory block of its resident node. Once the block of a node is full, a new memory block is allocated to this node. Blocks are occasionally re-organized if necessary such that blocks associated with the same node are located close to each other. In this way, the data locality of two labels at the same node are significantly improved. We will see the remarkable performance improvement by such a simple strategy in Section IV.

C. Bi-directional Search

Bi-directional dynamic programming for the elementary SPPRC is proposed in [9] to reduce the number of labels generated in the algorithm. Generally speaking, the bi-directional dynamic programming search is composed of three stages. First is a forward pass from the source to nodes "somewhere in the middle" to generate forward labels, just as the one-directional labeling algorithm we discussed; then a symmetric backward pass from the destination generates backward labels; and finally at each node with both forward and backward labels, it joins any pair of labels to form a feasible complete path. Intuitively, the number of labels grows exponentially with the number of arcs in the path. The overall number of labels are then expected to be smaller with shorter paths.

For any graph in general, the difficulty of applying the idea of bi-directional search is that it is hard to know whether we have reached the appropriate "somewhere in the middle". If the searches do not stop appropriately, redundant labels will be generated and duplicate paths will be obtained many times at multiple nodes. For DAG with a topological ordering, however, we can pick any node (usually right in the middle) as a *milestone* to stop the path extension. Specifically, a forward label should stop extension when it reaches a node after the milestone in the topological ordering for the first time; symmetrically, a backward label stops when it reaches a node before the milestone for the first time. Lastly, we join labels resident at only one half of the nodes, say after the milestone. The other half of the nodes should be ignored since they only provide duplicate paths.

The pseudo-code of the bi-directional labeling algorithm is presented in Fig. 4. We denote a node $i <$ *milestone* if i comes before *milestone* in the topological ordering and vice versa. Notice that compared to Fig. 3, the forward and backward versions of *rc*, *ref* and *dominance* should be used accordingly, and an additional function *join* is required to construct a complete path from a pair of forward and backward labels.

IV. Experiment Result

In this section, we evaluate the impact of all the improvements we proposed in this paper. The benchmark problems for this analysis are four SPPRC instances in DAG based on our real-life applications for crew scheduling. We order the instances in difficulty from 1 to 4 (in terms of a reasonable criterion, e.g. the number of labels or computation time). We code our algorithm in C++ on a 3.5 GHz machine with 32 GB of RAM with common compiler optimization options.

We first compare the algorithm of BGL and that we proposed in this paper, from an algorithmic point of view. Table I shows the number of generated labels of all nodes for each instance and approach. "BGL" indicates the implementation of the Boost Graph Library. "DAG" indicates the algorithm given in Fig. 3. Notice that improvements on skyline algorithm and memory management have no impact on the number of labels.

```
construct a topological ordering of nodes
Label forward_label, backward_label, new_label
initialize forward_label with forward_rc, backward_label with backward_rc
INSERT(set of forward labels at source, forward_label)
INSERT(set of backward labels at destination, backward_label)
for each node i < milestone in order do
   SKYLINE(set of forward labels at i)
   for each forward_label at i do
      for each arc (i, j) from node i do
         new_label := forward_ref(forward_label)
         if new_label is not feasible then
            DELETE(new_label)
         else
            INSERT(set of forward labels at j, new_label)
         end if
      end for
   end for
end for
for each node i > milestone in reversed order do
   SKYLINE(set of backward labels at i)
   for each backward_label at i do
      for each arc (j, i) to node i do
         new_label := backward_ref(backward_label)
         if new_label is not feasible then
            DELETE(new_label)
         else
            INSERT(set of backward labels at j, new_label)
         end if
      end for
   end for
end for
for each node i ≥ milestone do
   for each forward_label at i do
      for each backward_label at i do
         new_label := join(forward_label, backward_label)
         if new_label is not feasible then
            DELETE(new_label)
         else
            INSERT(set of complete labels at j, new_label)
         end if
      end for
   end for
   SKYLINE(set of complete labels at i)
end for
```

Figure 4. Bi-directional labeling algorithm in DAG.

"Bi-dir" indicates the number of labels of the bi-directional search, including forward, backward, and joined labels. To approximately show the problem size of each instance, we also present the number of pareto-optimal paths in the column "Opt". Fig. 5 illustrates the factors of label reduction based on the results of BGL for each approach, i.e. the number in each column of the table divided by that of BGL. We can see that the number of labels has been significantly reduced by about a factor of 3 in the DAG version. The bi-directional search, as expected, can further reduce the number of labels, especially for large instance, e.g. about 27% in Instance 4.

TABLE I
COMPARISON OF LABEL NUMBERS

Instance	Opt	BGL	DAG	Bi-dir
1	13039	2472632	660869	607749
2	30852	7886048	2292989	1924221
3	51324	21869754	7356918	5654961
4	81501	55272855	20650470	15026218

TABLE II
COMPARISON OF COMPUTATION TIMES (SEC.)

Instance	BGL	Naive	Memory	Skyline	M&S	Bi-dir
1	8.8	2.5	2.1	1.5	1.1	1.1
2	99.9	24.8	15.7	11.1	6.4	6.1
3	859.4	245.5	105.9	76.3	29.8	25.2
4	6189.9	1640.1	667.5	519.1	146.2	114.3

Table II summarizes the overall performance in single thread of the approaches we discussed in previous sections, in terms of computation time in seconds. To analyze the impact of each improvement technique, we evaluate the following solution approaches: an approach using naive nested loops skyline without or with specialized memory management ("Naive" and "Memory"), an approach using sort-filter-skyline without or with the memory management ("Skyline" and "M&S"), and the bi-directional approach with both sort-filter-skyline and the memory management ("Bi-dir"). Fig. 6 illustrates the factors of acceleration compared to the performance of BGL for each approach, i.e. divide the time of BGL by the time in each column of the table.

From Fig. 6 it is obvious that the impact of the improvements grows with the difficulty of the problem. Let's focus on Instance 4, the largest instance in our experiment to quantify the contribution of each improvement technique. We observe that the naive DAG version of the labeling algorithm, without any refinement technique, is faster than that of BGL by a factor of 3.8, which can apparently be explained by the label reduction discussed in Table I. Compared to the naive DAG, using the specialized memory management strategy alone accelerates the performance by 2.5 times; using the sort-filter-skyline algorithm alone accelerates by 3.2 times. The two refinements work even better together by a factor of 11.2. Based on the two techniques, the strategy of Bi-directional search can further improve the performance by a factor of 1.3. With the contributions of all proposed improvements, we speed up the SPPRC algorithm by 54.2 times compared to the original implementation of BGL.

V. CONCLUSION

This paper focus on the shortest path problem with resource constraints in directed acyclic graphs, which plays a significant

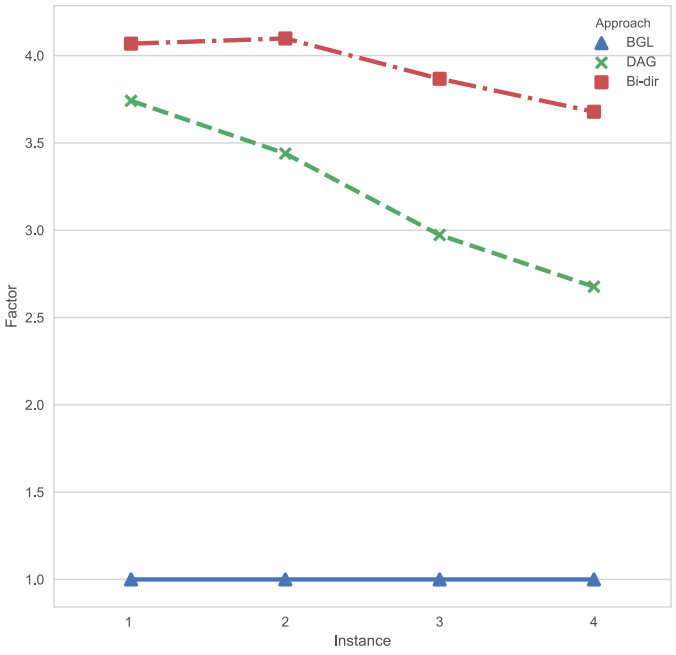

Figure 5. Label reduction.

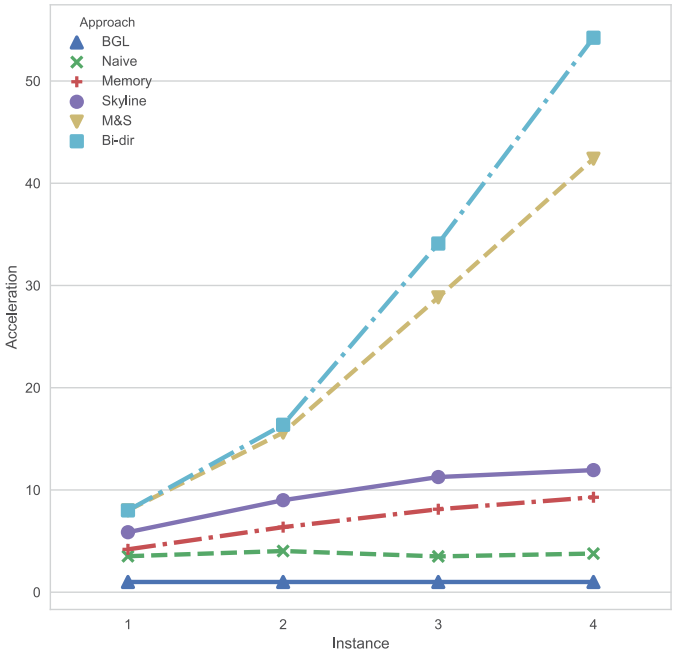

Figure 6. Performance acceleration.

role in many real-life optimization problems. Without any loss of optimal solutions, we propose several improvement techniques on the classical dynamic programming approach and obtain a performance gain by an order of magnitude in our experiments. For further research, some recent work to speed up the skyline operation, e.g. the GPU-based acceleration [11], may be a promising direction to explore.

ACKNOWLEDGMENT

This research is supported in part by Pearl River S&T Nova Program of Guangzhou (201610010188).

REFERENCES

[1] S. Irnich and G. Desaulniers, "Shortest Path Problems with Resource Constraints," in Column Generation, New York: Springer-Verlag, 2005, pp. 3365.
[2] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance, "Branch-and-price: Column generation for solving huge integer programs," Oper. Res., vol. 46, no. 3, pp. 316329, 1998.
[3] D. Feillet, "A tutorial on column generation and branch-and-price for vehicle routing problems," 4or, vol. 8, no. 4, pp. 407424, 2010.
[4] B. Gopalakrishnan and E. L. Johnson, "Airline Crew Scheduling: State-of-the-Art," Ann. Oper. Res., vol. 140, no. 1, pp. 305337, Nov. 2005.
[5] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin, Network Flows: Theory, Algorithms and Applications. Englewood Cliffs, New Jersey: Prentice-Hall, 1993.
[6] M. Desrochers and F. Suomis, "A generalized permanent labelling algorithm for the shortest path problems with time windows," INFOR, vol. 26, pp. 193-214, 1988.
[7] L. Di Puglia Pugliese and F. Guerriero, "A survey of resource constrained shortest path problems: Exact solution approaches," Networks, vol. 47, no. 1, 2013.
[8] S. Borzsony, D. Kossmann, and K. Stocker, "The Skyline operator," in Proceedings 17th International Conference on Data Engineering, 2001, pp. 421430.
[9] G. Righini and M. Salani, "Symmetry helps: Bounded bi-directional dynamic programming for the elementary shortest path problem with resource constraints," Discret. Optim., vol. 3, no. 3, pp. 255273, 2006.
[10] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with presorting," in Proceedings 19th International Conference on Data Engineering (Cat. No.03CH37405), 2002, no. October, pp. 717719.
[11] K. S. Bøgh, I. Assent, and M. Magnani, "Efficient GPU-based skyline computation," Proc. Ninth Int. Work. Data Manag. New Hardw. - DaMoN 13, p. 1, 2013.