An Adaptive Discretization Method for the Shortest Path Problem with Time Windows

by

Yu Tang

B.Sc., Simon Fraser University, 2017
B.Sc., Zhejiang University, 2017

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Fall 2020

Declaration of Committee

Name:	**Yu Tang**
Degree:	**Master of Science**
Thesis title:	**An Adaptive Discretization Method for the Shortest Path Problem with Time Windows**
Committee:	**Chair:** Thomas Shermer Professor, Computing Science
	Binay Bhattacharya Supervisor Professor, Computing Science
	Ramesh Krishnamurti Committee Member Professor Emeritus, Computing Science
	Abraham Punnen Examiner Professor, Mathematics

Abstract

The Shortest Path Problem with Time Windows (SPPTW) is an important generalization of the classical shortest path problem. SPPTW has been extensively studied in practical problems, such as transportation optimization, scheduling, and routing problems [31, 38, 47]. It also appears as a sub-problem in the column-generation process of the vehicle routing problem with time windows [6, 11].

In SPPTW, we consider a time-constrained graph, where each node is assigned with a time window, each edge is assigned with a cost and a travel time. The objective is to find the shortest path from a source node to a destination node while respecting the time window constraints. When the graph contains negative cycles, the problem becomes Elementary Shortest Path Problem with Time Windows (ESPPTW).

In this thesis, we adopt the time-expanded network approach, extend it by incorporating the adaptive expansion idea and propose a new approach: Adaptive Time Window Discretization (ATWD) method. We demonstrate that the ATWD method can be easily combined with label setting algorithms and label correcting algorithms for solving SPPTW. We further extend the ATWD embedded label correcting algorithm by adding k-cycle elimination to solve ESPPTW on graphs with negative cycles. We also propose an ATWD based integer programming solution for solving ESPPTW. The objective of our study is to show that optimal solutions in a time-constrained network can be found without first constructing the entire time-expanded network.

Keywords: shortest path problem with time windows, time-expanded network, negative cycles, elementary shortest path problem with time windows, adaptive time window discretization, label correcting method, integer programming

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Binay Bhattacharya, who gave me the opportunity to study and do research in Simon Fraser University. He provided me with guidance and assisted me a lot throughout this research. Also, I would like to express my deep gratefulness to my parents, who supported my pursuit of graduate studies. Finally, huge thank you to my wife for providing me with precious advises and encouragements.

Table of Contents

List of Tables

List of Figures

Chapter 1

Shortest Path Problem with Time Windows

In this chapter, we formally define the Shortest Path Problem with Time Windows (SPPTW) and briefly review the literature. This chapter is structured as follows: In Section 1.1, we discuss the classical Shortest Path Problem (SPP). SPPTW is described in Section 1.2, which includes problem definition and the Integer Programming (IP) formulation of SPPTW. In Section 1.3, we introduce the Elementary Shortest Path Problem with Time Windows (ESPPTW) and discuss the techniques for tackling networks with negative cycles.

1.1 SPP

Given a connected graph $G(V, E)$, where V represents the set of nodes and E represents the set of edges, SPP aims to find the lowest cost path from a source node s to a destination node d. SPP is a well-studied problem, and there exist exact solutions to SPP on different types of graphs: directed or undirected, weighted or unweighted, etc.

For unweighted graphs, the SPP can be solved using *Breadth-first Search*, which has an $O(|V| + |E|)$ time complexity. For weighted graphs, Dijkstra [13] proposed a fast algorithm for solving single source SPP in directed graphs with non-negative edge weights. The algorithm can be implemented in $O(|V|^2)$ using arrays. Later, Fredman & Tarjan [26] implemented Dijkstra's algorithm in $O(|E|+|V|log(|V|)$ time using a Fibonacci heap min-priority queue. A* algorithm, proposed by Hart et al. [29] is an extension to Dijkstra's algorithm, which generally achieves better performance by using heuristics to guide the search. The Bellman-Ford algorithm, which was first proposed by Shimbel in 1955 [43], but was named after Bellman [3] and Ford [24], was developed to solve single source SPP on directed graphs

with negative edge weights. The graph is assumed to have no negative cycle, and the algorithm will report any negative cycle once detected. The time complexity of Bellman-Ford algorithm is $O(|V||E|)$.

The Floyd-Warshall algorithm solves all pairs SPP in weighted graphs. It was published in its currently recognized form by Floyd in 1962 [23]. The complexity of the algorithm is $O(|V|^3)$. Another famous solution for solving all pairs SPP is Johnson's algorithm, published by Johnson in 1977 [34]. The algorithm uses Bellman-Ford to re-weight all edges in the original graph to make them all positive, then apply Dijkstra on the new graph. By using a Fibonacci heap in the implementation of Dijkstra's algorithm, Johnson's algorithm achieves $O(|V|^2 log(|V|) + |V||E|)$ time complexity, which outperforms the Floyd-Warshall algorithm when the graph is sparse.

1.2 SPPTW

The Shortest Path Problem with Time Windows (SPPTW) is a problem that is often solved in scheduling problems [10, 49]. It is also solved as a sub-problem in the column generation process when solving Vehicle Routing Problems with Time Windows (VRPTW) [11, 47]. Since graphs representing real-world transportation networks can be huge, it is crucial to develop an efficient solution for these problems.

SPPTW was first introduced in Desrosiers, Soumis, and Desrochers [11] as a sub-problem for the multiple traveling salesman problem with time windows. They also proved that SPPTW is NP-Hard in the ordinary sense, and there exist pseudo-polynomial algorithms for it. SPPTW aims to find the shortest path on a time-constrained network while ensuring each node is visited during its time window. Existing solutions for solving SPPTW involve the following strategies: building dynamic programming based labeling algorithms [4, 19, 39, 41, 42], constructing time-expanded networks [21], or solving Integer Programming (IP) formulations [14, 48]. Desrochers, Martin, and Soumis [9] proposed a generalized permanent labeling algorithm for solving SPPTW in pseudo-polynomial time. Desaulniers and Villeneuve [7] presented the IP formulation for SPPTW. Researchers further extend SPPTW by adding the elementary constraint [41, 42]. Another line of research focuses on solving a generalized version of SPPTW: Shortest Path Problems with Resource Constraints [4, 37]. These problems are briefly introduced in Section 1.3.2.

1.2.1 Problem Definition

SPPTW is defined on a connected graph $G(V, E)$ where $V = \{0, 1, 2...n\}$ represents the set of nodes and E represents the set of directed edges. Each node i is assigned with a time window $[e_i, l_i]$, and each edge (i, j) is assigned with a cost c_{ij} and a travel time t_{ij}. Note that SPPTW is only defined on graphs without negative cycle. For graphs with negative cycles, the problem will become ESPPTW which will be discussed in the next section. The objective of SPPTW is to find the shortest path from the source node s to the destination node d, while ensuring each node is visited within its time window. Early arrivals are allowed but need to wait until the time window opens, while late arrivals are prohibited.

We do not consider late arrivals in this thesis. However the proposed approaches can be easily applied on graphs with late arrivals allowed. We briefly address this in Chapter 7.

1.3 ESPPTW

1.3.1 Adding the Elementary Constraint

When the graph contains negative cycles, a common restriction is added to SPPTW: each node in the graph can only be visited at most once. This restriction prevents the algorithm from looping over the negative cycles. As a result, it also adds complexity as SPPTW is then extended to Elementary Shortest Path Problem with Time Windows (ESPPTW). SPPTW is NP-Hard in the ordinary sense, ESPPTW is proved to be NP-Hard in the strong sense and there is no pseudo-polynomial algorithm for it unless P=NP [39]. The elementary constraint even turns SPP into NP-Hard, as was proved using a simple reduction from the Hamiltonian path problem [48]. Solutions to elementary shortest path problem include exact solutions [14] and heuristic-based approximated solutions [27, 48].

There are plenty of studies focusing on finding the exact solutions of ESPPTW with the presence of negative cycles, and most of them aim to solve a generalized version of ESPPTW, which is Elementary Shortest Path Problem with Resource Constraints (ESPPRC). ESPPTW is a special case of ESPPRC as the time window can be one of the resource constraints. In the following paragraphs, we briefly introduce ESPPRC. We also review the literature of ESPPRC as most of the related works are directly applicable to ESPPTW.

1.3.2 ESPPRC

The Shortest Path Problem with Resource Constraints (SPPRC) was first introduced by Desrochers [12]. The objective of SPPRC is to find the shortest path from a source node to a destination node while satisfying all resource constraints along the path. The resource constraints are assigned to nodes as resource intervals [32], which bound the resource values upon arrival at each node after consuming the resources. Time is often used as a resource in SPPRC. SPPRC contributes to solving a wide variety of vehicle routing and crew scheduling problems [6, 10], as the column generation approach of these problems formulates SPPRC or one of its variants as sub-problems. Since SPPRC is proven to be NP-Hard by Handler et al. [28], variants of SPPRC are all computationally hard to solve.

In most Vehicle Routing Problems (VRP) solved by column generation, the subproblems often correspond to Elementary Shortest Path Problems with Resource Constraints (ESPPRC). Desrochers et al. proposed a column generation approach for solving Vehicle Routing Problem with Time Windows (VRPTW) [8], and the resulting subproblem is the shortest path problem with time window and capacity constraints (ESPPTWCC). ESPPRC contributes to the column generation by its three major advantages [32]. Firstly, since resource constraint defines a variety of rules, it is a flexible model for complex structures of a route. Secondly, as discussed in Desrochers et al. [11], the column generation gives tighter bounds than linear relaxation. Thirdly, there exist efficient algorithms for solving some variants of the ESPPRC, especially for the ESPPTW variant. Hence, efficiently solving ESPPTW contributes to solving VRPTW, which reflects the importance of our work since VRPTW is commonly solved in many practical problems.

Righini and Salani [42] presented and compared three different methods for solving ESPPRC: dynamic programming with bi-directional search with resource-based bounding [41], branch-and-bound where lower bounds are computed by dynamic programming with state-space relaxation, and decremental state space relaxation. Drexl et al. [15] described an exact and two heuristic labeling algorithms for solving the ESPPRC with the consideration of EU drivers' rules. Pugliese et al. [40] addressed solutions to elementary shortest path problem with forbidden paths. Lozano et al. [37] proposed an exact solution to ESPPRC based on implicit enumeration with a novel bounding scheme.

Finding the elementary solutions for SPPRC adds another level of complexity to the problem. Numerous researches have been focusing on solving the non-elementary SPPRC, and the result is still useful in practice (e.g., column generation). Some existing branch-and-price solutions for VRP on cyclic graphs focused on solving non-elementary SPPRC sub-problems [2, 16, 28]. Other researchers proposed different strategies to eliminate cycles in the solution in order to improve the lower bound. For example, SPPRC with k-cycle

elimination (SPPRC-k-cycle) is to find SPPRC while ensuring no cycle exists with length $\leq k$. Examples from Solomon's benchmark dataset [45, 46] show that eliminating cycles for a small value of k improves the master problem lower bounds [32]. The case $k = 2$ was first analyzed by Houck et al. [17]. The idea was applied in Vehicle Routing Problem with Time Windows (VRPTW) by Kolen et al. [36] and Desrochers et al. [8]. Irnich and Villeneuve [33] also proposed an algorithm which applied a new definition of the dominance rule for the general case of $k \geq 2$.

1.3.3 ESPPTW IP Formulation

In this section we present the integer programming formulation of the ESPPTW. There are two sets of decision vectors x and y in the model. Consider a time-constrained directed weighted graph $G(V, E)$, the decision vector x is defined for each edge $(i, j) \in E$:

$$x_{ij} = \begin{cases} 1 & \text{if edge } (i,j) \text{ is selected in the resulting path} \\ 0 & \text{otherwise} \end{cases}$$

The decision vector y is defined for each node, which denotes the departure time. For a node i with time window $[l_i, e_i]$, the decision variable y_i can be defined as:

$$y_i \in [l_i, e_i]$$

The objective is to find a path from a source node s to a destination node d which minimizes the total cost, while satisfying the following constraints:

- each node is visited exactly once.
- the time window constraints for all nodes along the resulting path are satisfied.

The IP formulation of ESPPTW is given as follows [11].

$$\min \sum_{(i,j)\in E} c_{ij} x_{ij}, \text{ s.t.} \tag{1.1}$$

$$\sum_{(s,j)\in E} x_{sj} = 1 \tag{1.2}$$

$$\sum_{(i,h)\in E} x_{ih} = \sum_{(h,j)\in E} x_{hj} \leq 1 \quad \forall h \in V \tag{1.3}$$

$$\sum_{(i,d)\in E} x_{id} = 1 \tag{1.4}$$

$$x_{ij}\left(y_i + t_{ij} - y_j\right) \leq 0 \quad \forall (i,j) \in E \tag{1.5}$$

$$l_i \leq y_i \leq e_i \quad \forall i \in V \tag{1.6}$$

$$x_{ij} \in \{0,1\} \quad \forall (i,j) \in E \tag{1.7}$$

The objective function is defined in equation 1.1. Equation 1.2 ensures that exactly one edge originated from the source node s is selected. Equation 1.3 represents the edge connectivity constraint (i.e., the number of in-edge should equal the number of out-edge on each node). It also restricts that each node can be visited at most once by bounding the number of incoming edges and outgoing edges to be less than or equal to one. Equation 1.4 ensures exactly one edge connected to the destination node d is selected. Equation 1.5 states that if a particular edge (i, j), is chosen, the departure time at j cannot be earlier than the arrival time at j. Equation 1.6 restricts that each node's departure time must be within its time window. Equation 1.7 is the integrality constraint.

1.3.4 Solving SPPTW/ESPPTW Efficiently

In the column generation approach for VRPTW, the sub-problems aim to find feasible routes in a graph with reduced costs. This problem can be defined as SPPTW. Hence, SPPTW is solved repeatedly, and it is crucial to develop an efficient algorithm for it.

The graph with reduced cost may contain negative cycles which makes ESPPTW problem strongly NP-Hard. In order to solve ESPPTW, besides exact solutions (e.g. integer programming), a wide variety of approximation algorithms have been proposed for solving ESPPTW. A well-known approximation strategy is to perform k-cycle elimination [33], which will remove cycles up to length k. In this thesis, we discuss both k-cycle elimination and IP based solutions for solving ESPPTW.

1.4 Thesis Outline

The structure of the thesis is as follows.

Chapter 2 introduces the time-expanded network strategy. We briefly review some related works, and we also introduce the general idea of the Adaptive Time Window Discretization (ATWD) approach.

Chapter 3 illustrates how the ATWD approach can be combined with label setting algorithms to solve SPPTW on graphs with positive edge weights. We use Dijkstra's algorithm and extend the algorithm to search a dynamically generated time-expanded network. We demonstrate that the shortest path can be found without constucting the entire time-expanded graph. We experimentally evaluate the algorithm performance.

Chapter 4 introduces how the ATWD approach can be combined with label correcting algorithms. As different from the label setting algorithm, the label correcting algorithm handles graphs with negative edge weights (but no negative cycles). We describe the ATWD based label correcting algorithm in detail. The proposed algorithm allows the user to specify the maximum length of the path that the user is interested in. We show that with a specified maximum length l_{max}, the algorithm is able to solve the problem by partially discretizing the graph.

In Chapter 5, we focus on solving ESPPTW on graphs with negative cycles. We extend the algorithm proposed in Chapter 4 by adding the k-cycle elimination step. We bound the number of labels per node by introducing a threshold parameter τ. We demonstrate the effectiveness of the algorithm by showing experiments on graphs with negative cycles.

Chapter 6 presents an Integer Programming based approach for solving ESPPTW. We show that IP formulations of ESPPTW can be efficiently solved on partial time-expanded graphs constructed by the ATWD method. We demonstrate that the designed algorithm is able to build the time-expanded network on-the-fly while iteratively solving the IP formulations.

In Section 7, we review our work and describe further research directions.

Chapter 2

Time-expanded Network

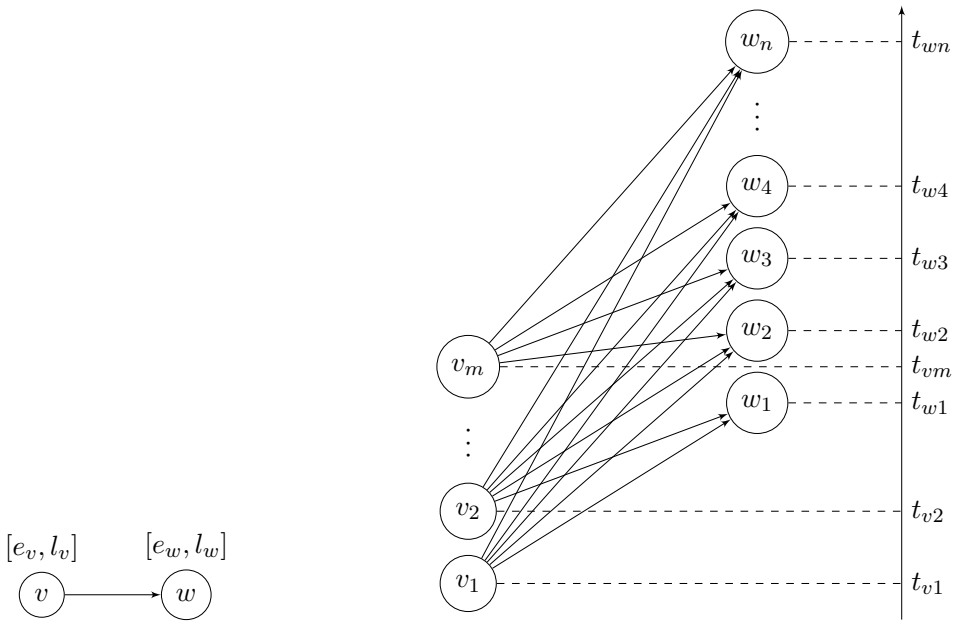
A time-expanded network is a well-known strategy to solve time-constrained problems, and it has been introduced in many research problems [5, 18, 21, 35, 44]. The idea is to duplicate the nodes and edges in the graph by adding timed node copies. Since the naive construction of a time-expanded graph may result in a huge graph, researchers have been focusing on optimizing the graph by constructing partial time-expanded graphs [5, 21]. In this chapter, we define the time-expanded network strategy, review the related works, and present a brief introduction to the Adaptive Time Window Discretization (ATWD) approach for building a partially time-expanded graph. We explain in Chapters 3-6 on how does the ATWD method work with various algorithms, and we also demonstrate that SPPTW/ESPPTW can be efficiently solved using these algorithms under different scenarios.

2.1 Definition of Time-expanded Network

The time-expanded network in the context of routing problems is defined as follows. Consider a time-constrained network $G(V, E)$ where each node i has a time window $[e_i, l_i]$, each edge (i, j) is assigned with a travel time t_{ij}. A time-expanded network $G_T(V_T, E_T)$ is constructed as follows:

- Replacing each node v in G with a set of node copies $v_0, v_1...$ by discretizing the time window $[e_v, l_v]$ into a list of timestamps $t_0, t_1...$ and create a node copy for each timestamp.
- Connecting all pairs of node copies (v_i, w_j) such that $(v, w) \in E$ and $t_v + t_{vw} \leq t_w$.

Figure 2.1 shows an example of constructing time-expanded network with respect to replicating a single edge (v, w). The time window of node v is discretized into timestamps



(a) Edge (v, w) in original network G

(b) Replication of edge (v, w) in time-expanded network G_T, assuming t_{vw} is very small

Figure 2.1: Illustration of constructing a time-expanded network

$t_{v1}...t_{vm}$ and are assigned to m node copies $v_1...v_m$. Similarly, node w is replicated into $w_1...w_n$ where the time window of w is discretized into timestamps $t_{w1}...t_{wn}$. All feasible edges between copies are connected.

The time-expanded network can be used for solving SPPTW/ESPPTW using algorithms for the static version of the same problem since the time dependency is embedded in the time-expanded network itself. However, the time dependency may not be captured in a lossless way unless the discretization is highly refined.

2.2 Reducing the Graph Complexity

As shown in Figure 2.1b, a large number of nodes and edges can be added when building a time-expanded network. Thus, the price one has to pay for simplifying the problem complexity is the dramatic increase in the graph size. The strategy of setting discrete timestamps can be optimized to reduce the size of the time-expanded graph. For example, we can set a larger discrete interval so that fewer node copies will be added. But this will also reduce the quality of the solution. The trade-off between the roughness of the discretization and the quality of the achievable solutions was evaluated by Fleischer et al. [22].

Another strategy to reduce the size of a time-expanded network is to use partial discretization. Considered with full discretization whereby setting a fixed discrete time step interval, a partial discretization can sometimes reduce the number of node copies. For example, some nodes in the graph may always be visited at around the same time, hence only one node copy will be sufficient for representing all these visits. Partial discretization allows one to add node copies whenever it is necessary. In this thesis, the proposed algorithm for constructing the time-expanded network exploits this idea. The algorithm adaptively discretize the nodes on-demand, resulting in a light growth network while still guaranteeing a high precision of the solution.

2.3 The Literature

The concept of the time-expanded network was first introduced by Ford and Fulkerson in 1962 [25]. They transformed the maximum dynamic network flow problem into a maximum static network flow problem by building the time-expanded network. Fleischer and Skutella [22] built a condensed time-expanded network by setting a rougher discretization of time. Kohler et al. [35] developed a time-expanded network with flow-dependent transit times such that the algorithms for solving static flows can be directly applied. Several researchers combined the time-expanded network idea with other modeling: Ferrati and Pallottino [20] proposed a time-expanded network based algorithm that can be applied on robotic vehicles for solving collision and energy consumption problems; Ho et al. [30] used time-expanded network to eliminate the computational load in dynamic space logistics network optimization for solving a human Mars exploration architecture design problem. Related works focused on building a partially time-expanded network for reducing the graph size includes: Boland et al. [5] proposed an approach for building partially time-expanded networks that can be used to provide upper bound and lower bound of the traveling salesman problem with time windows; Later Vu et al. [50] further extended this work to accommodate time-dependent travel times.

2.4 ATWD

In this thesis, we propose time-expanded network-based approaches for efficiently solving SPPTW/ESPPTW. The approach is named as Adaptive Time Window Discretization (ATWD). In general, the ATWD approach allows one to generate a partially time-expanded network, with a partial discretization of time. We also combine the ATWD with various algorithms to enable discretization on-the-fly. In this section, we introduce the general idea of adaptive discretization. We present the ATWD approach and illustrate how it is able

to reduce the complexity of time-expanded networks using an example. We also define the property and the optimality of the partially time-expanded graph constructed by the ATWD method.

2.4.1 Full Discretization

The full discretization algorithm for constructing a time-expanded graph works as follows. For every node v, a set of *copies* is created, and each copy represents a *timed* node:(v, t_i). The first attribute represents the node index, and the second attribute represents the discrete timestamp assigned to each node copy. For example, if we define the time step parameter (i.e. threshold parameter δ) to be 1, then a node v with time window $[e_v, l_v]$ will be replicated as $(v, e_v), (v, e_v + 1)...(v, l_v)$, assuming e_v, l_v are integral. After all node copies are created, arcs are introduced to the time-expanded graph by connecting all reachable node pairs. An arc $((v, t_v), (w, t_w))$ will be created if $t_v + t_{vw} \leq t_w$, where (v, w) is an edge in the original graph and t_{vw} represents the travel time on the edge.

The resulting time-expanded graph $G_T(V_T, E_T)$ is constructed from the original graph $G(V, E)$ by adding the following properties: each node v can only be visited at a set of discrete times $e_v, (e_v + 1), ...l_v$. This procedure is also called time window discretization. If we restrict SPPTW with integral values of time (i.e., when the travel times are integral), then this time-expanded network can be used to find the optimal solution of SPPTW by directly applying existing algorithms such as Dijkstra's algorithm or Bellman-Ford algorithm. However, a major drawback of this approach is that the expanded graph has $V_T \in O(t_{max}|V|)$ nodes, where t_{max} represents the maximum time window size of all nodes. Even if we run the most efficient shortest path algorithms, the running time is still going to be huge.

2.4.2 Adaptive Discretization

The full discretization method described above generates lots of redundant nodes and edges in practice. In many cases, only a subset of nodes and edges in a time-expanded network will be useful for finding the optimal solution. Thus, in order to generate an effective partially time-expanded graph, we present the ATWD approach in the following paragraphs.

Definitions

In the ATWD algorithm, we construct a partially time-expanded graph $G_P(V_P, E_P)$ such that $V_P \subseteq V_T, E_P \subseteq E_T$. The input consists of the original graph $G(V, E)$, the source node

s, the destination node d, and the threshold parameter δ. We define each node v in the partially time-expanded network $G_P(V_P, E_P)$ as having three attributes:

1. node index $v.i$, which represents the node index in the original graph.
2. copy index $v.c$, which is used to distinguish between node copies with the same index.
3. departure time $v.t$, which is the time assigned to the copy

Scope of Our Solution

In this thesis, we focus on solving SPPTW/ESPPTW on graphs with integral time values. Note that the proposed algorithms still work for graphs with real times, but the solutions are only guaranteed to be upper bounds to optimal solutions. We introduce a discretization threshold parameter δ, which is applied to all of the ATWD based algorithms, to control the discretization granularity. δ specifies the maximum discrete time step between timed node copies. The experimental results in this thesis show that the value of δ controls both the quality of the result and the size of the time-expanded graph. More detailed usage of δ is explained in the next paragraph.

Graph Construction Process

The graph construction always happens simultaneously with edge relaxation. Edge relaxation is a repeated step in dynamic programming algorithms for solving shortest path problems. When edge relaxation happens on edge (v, w), the algorithm will check if labels on w are needed to be updated by examining the cost of labels on v plus the cost of edge (v, w). In a time-constrained graph, when relaxing an edge, we will also have the departure time on node v and the travel time on edge (v, w), which can be used to calculate the arrival time on node w.

We use threshold parameter δ to determine whether a new timed node copy should be created when relaxing an edge. Suppose the arrival time on node w is $t_{arrival}$, then a new node copy of w will be created if there is no timed copy w_a such that $w_a.t - t_{arrival} < \delta$. If node new copies of w is created after the edge relaxation, which means w_a exists, then an extra waiting time of $w_a.t - t_{arrival}$ will be added before the next departure from w_a.

Suppose the original time-constrained graph is $G(V, E)$, and we want to construct a partially time-expanded graph $G_P(V_P, E_P)$. When the ATWD process starts, we create an initial node copy of the starting node, which is often the source node, and assign the beginning time of its time window to it. We then add the node copy to V_P and start

constructing the time-expanded graph by relaxing the outgoing edges of the starting node in E, and new node copies are added to V_P whenever needed. The edge relaxation order may vary in different algorithms. For example, when combining with label setting algorithms, a Dijkstra-like order can be used, which always relaxes the outgoing edges from the node with minimum label cost. When combining with label correcting algorithms, a breadth-first-search order from the source node can be used. Note that the graph construction process only needs to happen once per node copy: whenever the neighbors of a node copy are checked and discretized, there is no need to revisit the node copy again. In label setting algorithms, we examine each node at most once, so we can tightly combine the graph construction with the edge relaxation. However, in label correcting algorithms, each node is examined multiple times. Thus, we add another step of checking whether a node copy has already been examined in order to avoid any repeated checks (Algorithm 1, line 6).

2.4.3 ATWD: Algorithm Explanation

In this section, we present and explain the ATWD algorithm. Note that the algorithm will always be slightly modified in order to be combined with various algorithms: label setting algorithms, label correcting algorithms, or integer programming based algorithms. Here we present a breadth-first-search version of the algorithm, which is suitable for label correcting algorithms.

At the beginning of the algorithm, we define several variables that are used in the algorithm: Set $\mathcal{N}_\mathcal{R}$ is for saving remaining nodes to be examined, node copy set $\mathcal{C}(v)$ is for saving node copies of each node index v. Initially, we create a copy for the source node $s : (s, 0, e_s)$ and add it to the set V_P. We mark its copy index as 0, set its time as the beginning time of its time window, and add it to $\mathcal{N}_\mathcal{R}$. Then we repeatedly select nodes from $\mathcal{N}_\mathcal{R}$ and check if any of its **unvisited** neighbors can be further discretized. We define unvisited neighbors as the neighbors in G that is not yet reachable in G_P, and line 6 iterates through all unvisited neighbor indexes. Line 7 calculates the arrival time t' at the neighbor index w. Line 8 uses a function *GetCopyIndex* to determine the index of the new node copy, which is further explained in Algorithm 2. If the function returns that the neighbor is not reachable, then we continue to the next node in the set $\mathcal{N}_\mathcal{R}$, otherwise, we create a node copy w_b in line 10 using the arrival time t', and assign a copy index. If w_b does not exist in saved node copies of index w (line 11), then we add the newly created copy w_b to graph G_P (line 12). We connect it with the current node v_a (line 14). The algorithm terminates either when a stopping criterion has been met (line 15), or when the remaining node-set $\mathcal{N}_\mathcal{R}$ is empty. If $\mathcal{N}_\mathcal{R}$ is empty, then the discretization process terminates. The stopping criteria is for terminating the algorithm earlier, which may vary for different algorithms. We will further explain this in later chapters.

Algorithm 1: ATWD

Input: Graph $G(V, E)$, source node s, destination node d, threshold δ
Output: Partially time-expended graph $G_P(V_P, E_P)$

```
1  initialize update := False, V_P := ∅, N_R := ∅, ∀v ∈ V : C(v) := ∅
2  s_0 = (s, 0, e_s);
3  V_P := V_P ∪ {s_0}, N_R := N_R ∪ {s_0};
4  foreach v_a ∈ N_R do
5      N_R = N_R \ v_a;
6      foreach neighbor index w in G that is not yet connected with v_a in G_P do
7          t' = v_a.t + t_vw;
8          b = GetCopyIndex(w, t', C(w), δ);
9          if b < 0 then continue;
10         w_b := (w, b, t');
11         if w_b ∉ C(w) then
12             V_P := V_P ∪ {w_b}, N_R := N_R ∪ {w_b};
13             C(w) := C(w) ∪ {w_b};
14         E_P := E_P ∪ (v_a, w_b);
15     if stopping criteria has been met then break;
16 return G_P(V_P, E_P);
```

The function *GetCopyIndex* is presented in Algorithm 2. It first checks if the arrival is later than the neighbor's due time. If yes, then it returns a negative integer -1 to indicate the neighbor is unreachable. If not, we first adjust the arrival time to force all early arrivals to wait until the node becomes ready (in line 3). We then check which "slot" does t fit in. The slot is pre-allocated according to threshold parameter δ. For example, consider a node in G with time window $[0, 10]$, the discrete times allocated under $\delta = 2$ will be $0, 2, 4, 6, 8, 10$. We fit the time t into these discrete timestamps by rounding it up to the closest value (e.g., $t = 1$ will be matched to slot with time=2). Note that we only do round up because we can adjust an arrival to a later time by forcing it to wait, but it is impossible to adjust it to an earlier time. If the slot is already occupied, which means there exists another node copy j with the time of the slot, we return that node copy's index number and time (line 6), meaning that there is no need to create a new node. Otherwise, we create a new index number which increments one on top of the current maximum index number (line 7), and return it together with the time of the slot t' (line 8).

An Illustrative Example

We now use an example to explain how the ATWD algorithm works. Given the time-constrained network shown in Figure 2.2, the algorithm proceeds as follows. Initially, a copy of s, s_0 is created and added to N_R. Then we pop the node copy s_0 and loop over all its neighbors, which are nodes with index a, b, c. Note that up to this point, no copy

Algorithm 2: GetCopyIndex

Input: Node index v, arrival time t, copies set $\mathcal{C}$, threshold δ.
Output: Copy index a of the node copy, assigned time t'

```
if t > l_v then a := −1;
else
    if t ≤ e_v then t := e_v;
    determine the slot where t fits in;
    if find a node copy v_a ∈ C already in the slot then
        return v_a.c, v_a.t
    a := len(C);
    t' := time of the slot;
return a, t';
```

is saved in the copy dictionary, meaning that $\mathcal{C}(a) = \mathcal{C}(b) = \mathcal{C}(c) = \emptyset$, so we will create a new node copy with index number 0 for each of these indexes. For index a, we create a new copy a_0 with time 10; For index b, we create a new copy b_0 with time 8; For index c, we create a new copy c_0 with time 5. We connect between s_0 and these copies, which form Figure 2.3a. Then all newly created nodes a_0, b_0, c_0 are added to the set N_R. In the following iterations, we pop and loop over the neighbors of a_0, b_0, c_0, and create new node copies a_1, d_0, d_1, d_2 accordingly (Figure 2.3b - 2.3d. When we probe the neighbors of node a_0, we create a new copy d_0 with time 18, and we connect (a_0, d_0). When probing neighbors of b_0 we create a new copy a_1 because the arrival time is 14, and the slot with time 14 is not occupied yet. Then we connect (b_0, a_1). Similarly, we create a new copy d_1 and assign its time as 14, then we connect (b_0, d_1). When probing the neighbors of node c_0, we create a new copy d_2 with time 16, and we connect (c_0, d_2). Note that we do not create a new copy under index b since the arrival time t' is $c_0.t + t_{cb} = 7$, and $b_0.t - t' = 1 < \delta$, so the out-edge from c_0 points to copy b_0, meaning that we are forcing an extra waiting time of 1 for all traffic from c_0 coming into b_0. Then all newly created nodes a_1, d_0, d_1, d_2 are added to the set N_R. In the following iterations, we check the neighbors of each node a_1, d_0, d_1, d_2, and the graph remains unchanged (Figure 2.3e), then the algorithm terminates as the remaining node set $\mathcal{N}_{\mathcal{R}}$ becomes empty. In the algorithm, whenever we create a new copy or connect two copies, we add a new node or a new edge to graph G_P, respectively. Hence, when the ATWD algorithm terminates, we will have the partially time-expended network $G_P(V_P, E_P)$ constructed.

2.4.4 Notion of Adaptive

As illustrated by the ATWD algorithm, the term "Adaptive" is two-fold: 1. the node copies are generated in a partial discretization of time; 2. the node copies and arcs are generated on demand.

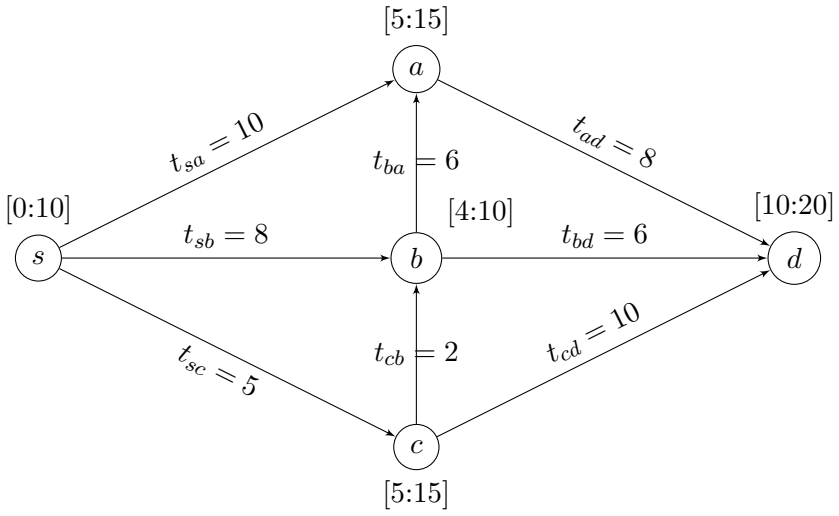


Figure 2.2: An example graph, each node's time window is shown in blue, each edge is assigned with travel time t

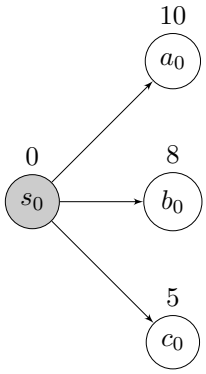
For type 1, the intuition is that we find it is common to have some time-isolated nodes that can only be reached during a small time period. For example, consider the graph in Figure 2.2, a will only be reachable from s after time $t = 10$, b will only be reachable from s after $t = 8$, d will only be reachable from s after $t = 14$. Hence, it is unnecessary to create copies before the earliest reachable time. With full discretization with $\delta = 2$, we will be creating six copies for each of nodes s, a, c, d, and four copies for node b, and we need to connect any pair of copies that are reachable, which will definitely result in a huge time-expanded graph. But with the adaptive discretization, we will create a partially time-expanded graph, which is sufficient to find the solution of SPPTW with the same quality.

For type 2, as we embed the time-expanded network generation into the edge relaxation process, the algorithm only discretizes a node whenever visiting a new neighbor. Hence, the time-expanded network will only be constructed on-demand: only a small graph will be discretized in order to find the shortest path from the source node to the destination node.

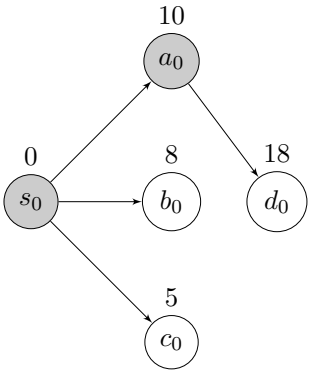
2.4.5 Properties of the Resulting Graph

The partially time-expanded network $G_P(V_P, E_P)$ constructed from time-constrained network $G(V, E)$ using Algorithm 1 has the following properties.

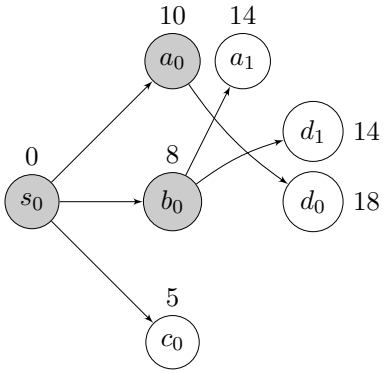
Property 1. For any edge $(v_i, w_j) \in E_P$, $v_i.t + t_{vw} \leq w_j.t$.



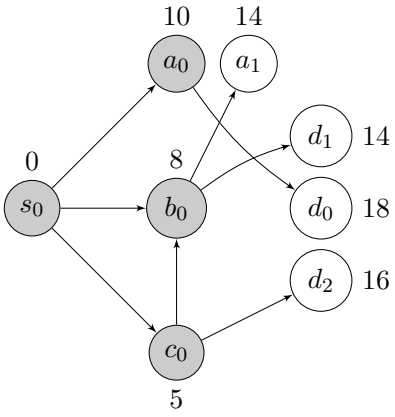
(a) After probing s_0's neighbors. a_0, b_0, c_0 are added.



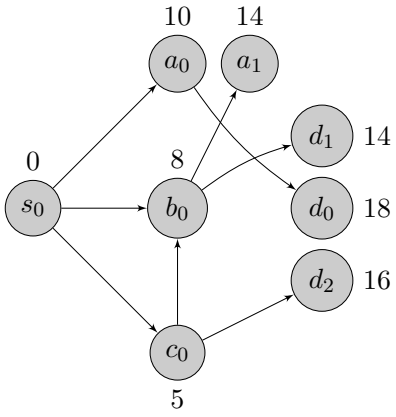
(b) After probing a_0's neighbors. d_0 is added.



(c) After probing b_0's neighbors. a_1,d_1 are added.



(d) After probing c_0's neighbors. d_2 is added.



(e) After probing a_1,d_0,d_1,d_2's neighbors. Graph is unchanged.

Figure 2.3: ATWD algorithm illustration with $\delta = 2$. Nodes in grey means their neighbor nodes have been discretized.

Property 2. Given a node copy $v_i \in V_P$, for each of its neighbor index w in G, v_i is connected with at most one node copy of w in G_P.

Property 3. A node copy v_i will only be created if there does not exist any node copy v_j in G_P such that $v_i.i = v_j.i$ and $v_j.t - v_i.t \in [0, \delta)$

We give a brief proof of the above properties. Property 1 is obviously correct according to Algorithm 2, line 3. For property 2, note that we only build out-edges of a node copy when it is popped from set $\mathcal{N}_\mathcal{R}$, and we only add each node copy to $\mathcal{N}_\mathcal{R}$ once. Since in Algorithm 2, we build at most one edge between v_i and any neighbor indexes of v, at most one copy of any neighbor index of v will be connected. Property 3 can be easily proved according to Algorithm 2, line 5. Note that this property also implies that any edge $(v_i, w_j) \in E_P$ is causing an extra wait time Δt such that $\Delta t \in [0, \delta)$.

Corollary 1. If node copy v_i is added to the graph G_P, then node v_i is reachable from the earliest node copy of the source node s_0 in G_P, node v is reachable from the source node s in G.

Suppose node copy $w_j \in V_P$, then according to Algorithm 1, line 14, there exists another node copy v_i such that $(v_i, w_j) \in E_P$. According to property 1, w_j is reachable from v_i since $v_i.t + t_{vw} \leq w_j.t$. By induction, w_j is reachable from s_0 in G_P, and w is reachable from s in G. Hence, the above corollary is correct.

Corollary 2. When $\delta = 1$, and all times in G are integral, the shortest path in G_P is identical to the shortest path in G.

According to property 3, an extra waiting time of $[0, \delta)$ will be added when visiting any edge in E_P. If all times are integral, the waiting time must also be an integer. Since $\delta = 1$, the extra waiting time can only be 0. Thus, whenever we create a node copy v_i, the time assigned to the copy represents the exact arrival time from the previous node. According to corollary 1, v_i must be reachable from s_0. By induction, it is not difficult to prove that any path from s_0 to v_i has no accumulated waiting time. Now we prove that the shortest path in G_P is identical to the shortest path in G. Suppose the shortest path in G can be represented as the following route: $s, v_1, v_2 ... d$. Since v_1 is reachable from s, if we leave s at its start time e_s, the arrival time at node v_1 cannot be greater than the due time l_v. Thus, in the first iteration of the algorithm, a node copy v_{1i} will either be created or found to already exist for representing this arrival. Since no accumulated waiting time exists in any of the paths in G_P, $v_{1i}.t$, therefore, represents the exact arrival time, which is also the *earliest* possible arrival time at v_1 while visiting the sequence of nodes: s, v_1. Thus, v_1 is reachable from s_0 in G_P. By induction, we can prove that all internal node indices in the optimal route $v_1, v_2 ...$ are reachable from s_0 in G_P since there exist node copies for each index: $v_{1i}, v_{2j} ...$ that are

all reachable from s_0. Thus, there exists a path in G_P, which represents the shortest path in G. This implies that the shortest path in G_P and the shortest path in G are identical.

Chapter 3

ATWD on Label Setting Algorithm

In this chapter, we present how to embed the Adaptive Time Window Discretization (ATWD) in label setting algorithms for solving the Shortest Path Problem with Time Windows (SPPTW). Due to the limitation of label setting algorithms, we only focus on solving SPPTW on graphs with positive edge weights in this chapter. We extend Dijkstra's algorithm by adding the ATWD process, and we show that the SPPTW can be solved without discretizing the whole graph. This chapter is organized as follows. In Section 3.1, we define the SPPTW problem. In Section 3.2, we describe the ATWD based label setting algorithm using an example. In Section 3.3, we present the complete algorithm, the analysis of correctness, and time complexity. In Section 3.4, we present our experimental results, and we briefly conclude this chapter in Section 3.5

3.1 Problem Definition

The SPPTW problem is defined as follows. Given a connected graph $G(V, E)$, where V represents the set of nodes and E represents the set of arcs. Each node v is assigned with a time window $[e_v, l_v]$. Each arc (v, w) is assigned with a non-negative cost c_{vw} and a travel time t_{vw}. The objective is to find a minimum cost path that departs from the source node s, visiting a (possibly) set of internal nodes, and arrives at the destination node d. Each node on the path (including source and destination nodes) can be visited at must once and must be visited during its time window.

As already explained in Chapter 2, Section 2.4.2, we focus on solving SPPTW using positive threshold parameter δ to control the discretization granularity. If the time constrained graph G contains real times, then the proposed algorithm provides an upper bound of the

optimal solution, the bound becomes tighter with a smaller δ. However, if G only contains integral times, the algorithm is guaranteed to find the optimal solution with $\delta = 1$.

In this thesis, we test the proposed algorithms only on graphs with integral times. We apply the algorithms on graphs generated from Solomon's benchmarking data set [45], which only contains integer times.

3.2 Label Setting Algorithm with ATWD

In this section, we show how the ATWD method can be applied on a label setting algorithm to solve the SPPTW on graphs with positive edge weights.

Algorithm 3: ATWD based Label Setting Algorithm

Input: Graph $G(V, E)$, source node s, destination node d
Output: Partially time-expended graph $G_P(V_P, E_P)$, result path P

initialize $Q := \emptyset$, $\forall v \in V : \mathcal{C}(v) := \emptyset$, $\forall v \in V_P : \mathcal{L}(v) := \emptyset$.
$s_0 = (s, 0, e_s)$;
add s_0 to Q;
cost(s_0) := 0, $\mathcal{L}(s_0)$:=NULL;
while Q is not empty **do**
    v_a := minimum cost node in Q;
    remove v_a from Q;
    if $v = d$ **then** break;
    foreach neighbor w of v in G **do**
        $t' = v_a.t + t_{vw}$;
        $b = GetCopyIndex(w, t', \mathcal{C}(w), \delta)$;
        if $b < 0$ **then continue**;
        $w_b := (w, b, t')$;
        if $w_b \notin \mathcal{C}(w)$ **then**
            $V_P := V_P \cup \{w_b\}$;$\mathcal{C}(w) := \mathcal{C}(w) \cup \{w_b\}$;
            cost(w_b) :=cost(v_a) + c_{vw}; $\mathcal{L}(w_b) := v_a$;
            add w_b to Q;
        else
            if $cost(w_b) > cost(v_a) + c_{vw}$ **then**
                cost(w_b) :=cost(v_a) + c_{vw};
                $\mathcal{L}(w_b) := v_a$;
        $E_P := E_P \cup (v_a, w_b)$;
d_m := node copy with minimum cost with destination index d;
P:= find path from s_0 to d_m via backtracking from $\mathcal{L}(d_m)$;
return P;

3.2.1 Algorithm Description

The label setting algorithm is presented in Algorithm 3. It is very similar to Djikstra's algorithm. We initialize a priority queue Q for saving the node copies, $\mathcal{C}(v)$ is for saving the node copies with index v. We recursively fetch the node v_a with the minimum cost from Q (line 5 - 7). If the node copy's index i is the destination index, then we terminate the loop (line 8). Otherwise, we will iterate over all neighbor indices j of that node index in G (line 9). Line 10 calculates the arrival time t', which will be used to determine the node copy index b using function *GetCopyIndex* (introduced in Chapter 2) in line 11. If the function returns w is unreachable, then we skip this neighbor index. Otherwise, if the returned copy index is already assigned to a node copy (line 18), which means w_b already exists in V_P, then we will relax the edge and update the label (line 19 - 21). Otherwise, b is not yet assigned to any existing node copy (line 14). Then we have to create a new node copy w_b, assign the cost, and put it into priority queue Q. We connect v_a with w_b and add it to E_P (line 22). After the while loop terminates, we find the minimum cost node copy with index d (line 23), then the shortest path from s to d will be returned via backtracking $\mathcal{L}$ (line 24).

Labels

In shortest path problems, the label is defined as a set of nodes that are visited from source node to the current node. For a given node v, there may exist different paths from the source node to v. According to the label definition, v can have multiple labels. Most of the label setting or label correcting algorithms introduce the idea of pruning labels. Shortest path algorithms (e.g., Dijkstra's, Bellman-Ford algorithm) only keep one label per node, and the label is also reduced to include only one predecessor node.

In the above label setting algorithm, the label $\mathcal{L}(v_i)$ on node copy v_i is defined as the predecessor node copy of v_i. The label definition is the same as in Dijkstra's algorithm or Bellman-Ford algorithm. We prove the correctness of such a label definition in the ATWD based label setting algorithm in Section 3.2.2.

Lemma 1. Consider a time-expanded graph G_P with no negative cycle. Suppose there are two different paths P_1, P_2 from one node copy v_a to another node copy w_b. If $cost(P_1) < cost(P_2)$, then P_1 *dominates* P_2.

Proof of Lemma 1. P_1 dominates P_2 if any path extension of P_2 can be applied to P_1 with a smaller accumulated cost. The ATWD algorithm constructs a time-expanded network by adding timed nodes to the graph, and for any two paths arriving at the same node v_a, they can be considered as arriving at the same time $v_a.t$. Hence, any path extension of P_1 is also

a valid extension for P_2. The reason is that, firstly, since the path extensions for P_1 and P_2 both start at the same time, any neighbor that is reachable from P_1 is also reachable from P_2. Secondly, since there is no negative cycle, the shortest path extension of P_1 or P_2 must not include visited nodes because if there is a cycle in the path, we can always eliminate the cycle from the path and get a new path with a smaller cost. The dominance rule indicates that the time dependencies have been embedded in the graph itself. Hence algorithms for solving the traditional shortest path problems, including label setting algorithms (e.g., Dijkstra's algorithm) and label correcting algorithms (e.g., Bellman-Ford), can be directly applied to solve SPPTW on the time-expanded graph. □

3.2.2 Proof of Correctness

Now we prove the correctness of the ATWD based label setting algorithm. According to the dominance rule in Lemma 1, in the label setting algorithm, whenever we examine an edge (v, w), we only need to update and save the shortest path from the source node to node w. Since other sub-optimal paths that have higher costs will be dominated, it is safe to prune them. Furthermore, as we only save one path per node, we do not need to save the entire path. Instead, we just need to save the predecessor node of w in the path from the source node to w. The whole path can be derived via backtracking the predecessor links. Thus, the label definition can be used to find the optimal solution in a time-expanded graph G_P.

3.2.3 Running Time Complexity

In this section, we analyze the time complexity of the ATWD based label setting algorithm. Like Dijkstra's algorithm, line 5 will loop over all the node copies, which takes $\mathcal{O}(|V_P|)$, line 9 will check all edges in E_P, which takes $\mathcal{O}(|E_P|)$. By implementing Q as a priority queue, line $6-7$ takes $\mathcal{O}(log(|V_P|))$. Line 11 takes constant time if we design $\mathcal{C}(n)$ as a hash table. The rest of the operations takes constant times. Hence, the running time complexity is $\mathcal{O}((|E_P| + |V_P|)log(|V_P|))$. According to Algorithm 2, the number of node copies for each node i in V is bounded by $\mathcal{O}((l_i - e_i)/\delta)$, and the number of nodes in V_P is $\mathcal{O}(|V|t_{max}/\delta)$, where t_{max} represents the maximum time window size of the nodes in V. Since Algorithm 2 restricts that each node copy is connected to at most one node copy from each of its neighbor index, the total number of edges in E_P will be $O(|E|t_{max}/\delta)$.

Thus, the overall worst-case time complexity of ATWD based label setting algorithm is $\mathcal{O}((|E|t_{max}/\delta + |V|t_{max}/\delta)log(|V|t_{max}/\delta))$. The experimental results will indicate that in practice, the size of the partial graph is much smaller than the size of the fully time-expanded network.

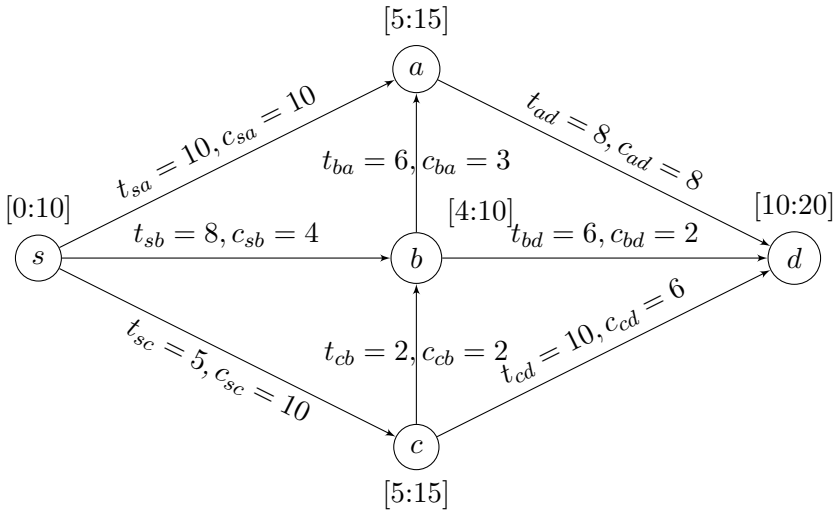
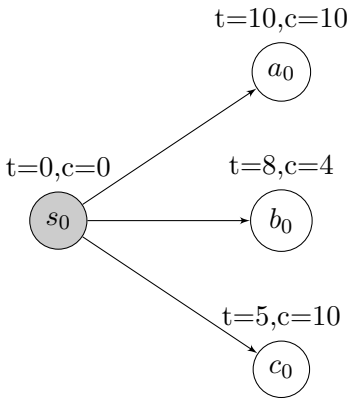


Figure 3.1: An example graph, each node is assigned with a time window, each edge is assigned with travel time t and cost c.

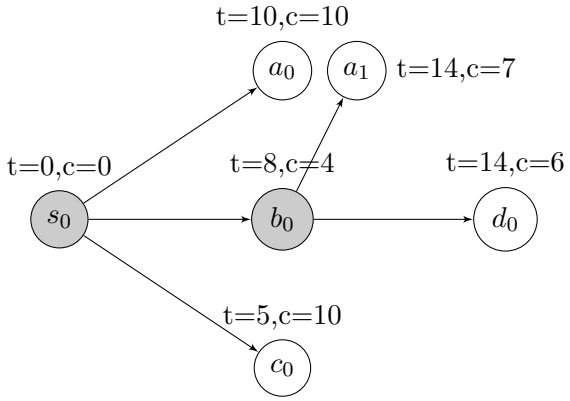
3.3 An Illustrative Example

Consider a time-constrained G presented in Figure 3.1. Note that this graph has the same structure as the one presented in 2.4.3. Initially, a node copy $s_0(s, 0, 0)$ is created, we assign cost(s_0)=0, $\mathcal{L}(s_0)$=NULL, and add it to priority queue Q. In the first iteration, we pop s_0 from Q and iterate through all its neighbors in $G : a, b, c$. We create new node copies a_0, b_0, c_0, assign cost and connect s_0 to each of them, then add these nodes to Q (Fig. 3.2a). In the second iteration, we pop the node b_0 from Q since it has the minimum cost, then we iterate through its neighbors in $G : a, d$, create node copies a_1, d_0, assign cost, and connect b_0 to each of them, then add these nodes to Q (Fig. 3.2b). In the third iteration, we pop d_0 from Q, and since d_0 has the same index as d, we terminate the algorithm. The shortest path s, b, d can be derived via backtracking from d_0.

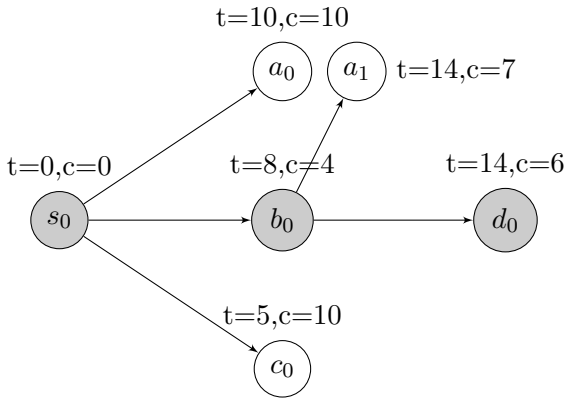
Note that although the original graph G in Figure 3.1 has the same structure as Figure 2.2, the constructed time-expanded graph G_P in Fig 3.2c using ATWD based label setting algorithm is even smaller than the time-expanded graph in Fig 2.3e. Since Dijkstra's algorithm always relaxes neighbors of the current smallest cost node in queue, if a node is never popped from the queue, then any path extension from the node will not be in the optimal shortest path solution (as all edge weights are positive). Hence we do not even need to construct the neighbors of that node in G_P. In general, ATWD based label setting algorithm guides the direction of graph construction, which efficiently solves SPPTW on time-constrained graph with positive edge weights by discretizing only a subgraph containing the optimal shortest path.



(a) Pop s_0 from Q. a_0, b_0, c_0 are added to Q, b_0 has the current minimum cost.



(b) Pop b_0 from Q, a_1, d_0 are created and added to Q, d_0 has the current minimum cost.



(c) Pop d_0 from Q, now we reach the destination, the algorithm terminates and shortest path s, b, d can be retrieved via backtracking the labels

Figure 3.2: ATWD label setting algorithm illustration with $\delta = 2$.

3.4 Experiments

We evaluate the performance of the ATWD based Label Setting Algorithm (ATWD-LSA) in this section by measuring the precision and efficiency. We define precision as follows. Denote the cost of the optimal path as $c_{optimal}$ and the cost of our solution as c, then the precision of our solution is:

$$precision = 1 - \frac{(c - c_{optimal})}{c_{optimal}}$$

The efficiency is evaluated using the size of G_P and the algorithm solving time. Note that these criteria apply to the experiment sections of the succeeding chapters as well.

3.4.1 Building the Experimental Graphs

We build the experimental graphs using Solomon's Benchmark data set [45]. The data set was originally designed for Vehicle routing Problems with Time Windows (VRPTW). Since each node in the data set represents a time-constrained customer with a time window and a service time, it is also a good resource for SPPTW. The data sets contain six sets of problems:

- R1: randomly generated geographical data with a short time window size.
- R2: randomly generated geographical data with a long time window size.
- C1: clustered geographical data with short time window size.
- C2: clustered geographical data with long time window size.
- RC1: a mix of random and clustered geographical data with a short time window size.
- RC2: a mix of random and clustered geographical data with a long time window size.

In Solomon's benchmarking problems, the cost and travel time between two nodes are defined as the Euclidean distance, and every pair of nodes is adjacent. However, if we directly map this setting to SPPTW, then the problem will be simple and easy to solve. Thus, in order to fully explore the power of the designed algorithm, we build the test graphs by setting the following guidelines. Assuming the mean and the standard deviation of euclidean distances between all pair of nodes is d_{mean} and d_{std}:

- For each edge (i, j) we assign the euclidean distance as its cost:

$$c_{ij} = \sqrt{(i.x - j.x)^2 + (i.y - j.y)^2}$$

- Break any "long" edge (i, j) if its euclidean distance is longer than $d_{mean} - d_{std}$.
- For each edge (i, j), initially assign euclidean distance of (i, j) as the travel time, then normalize all travel times to $[0, 10]$ and shift the travel time up by service time.

The intuition of breaking "long" edges is to allow more intermediate nodes to be included in the optimal shortest path solution. Otherwise, in the complete graph, the length of the shortest path will be small for any source-destination pair, and such a graph will not be a good fit for testing the algorithms. We mark an edge as a "long" edge if the euclidean distance between two endpoints is longer than $d_{mean} - d_{std}$.

Note that Solomon's data have a universal fixed service time $t_{service} = 10$, and in the problem setting proposed in this thesis, we incorporate this service time into the edge travel time. We normalize travel times for all edges into $[10, 20]$ with a euclidean distance distribution. This is because we want to solve SPPTW on VRPTW data, so we keep the travel times to be relatively small in order to allow a "vehicle" to visit more "customers". On the other hand, since we are breaking a portion of edges in the graph, it becomes more likely that randomly selected pairs are either not reachable, or reachable via only a few different paths. Hence by reducing the travel time of each edge, there will be more relevant paths between any pair of nodes in the graph, which makes the SPPTW harder to solve.

In all the experiments in this thesis, we focus only on RC2 data sets due to the following reasons:

- we ignore the clustered data set because according to our strategy of breaking long edges, the clustered data set may end up having isolated clusters, which will not be suitable for solving SPPTW.
- we ignore data set with tight time windows because we want the graphs to have greater connectivity (i.e., multiple unique paths exist between a selected pair of source and destination nodes).
- the geographical distribution in RC data set is closer to real-world maps.

3.4.2 ATWD-LSA with Different Thresholds

Since the discretization threshold δ controls the granularity of the constructed time-expanded graph G_P, with a smaller δ, the size of G_P is going to increase. We test the ATWD-LSA algorithm using different threshold values and report the statistics in Table 3.1. We generate a time-constrained directed graph G with positive edge weights using Solomon's *0100_RC201*,

which has 100 nodes, and each node has a fixed time window (TW) size 120. We randomly choose 10 pairs of source and destination nodes. For each pair, we run the ATWD-LSA algorithm under different thresholds ($\delta = 1, 2, 4, 10, 20, 30$) and collect the results. We then average the results and present them in Table 3.1.

Threshold(δ)	Precision	V_P	E_P	Running Time(s)
1	1	1130	8043	0.153
2	1	1105	7527	0.065
4	1	739	5120	0.040
10	1	433	2850	0.022
20	1	275	1706	0.013
30	0.98	217	1345	0.01

Table 3.1: ATWD based Label Setting Algorithm under different thresholds, on a graph generated from Solomon 0100_RC201, which has a fixed time window size TW=120 for all nodes.

In the original graph G generated from *0100_RC201*, there are 100 nodes and 1072 edges. If we use the full discretization approach, each node will be replicated into TW/δ copies. Then there will be over 130000 nodes in the graph when $\delta = 1$. As shown in Table 3.1, we are able to significantly reduce the complexity of the time-expanded graph by constructing a G_P. Each node is copied on an average of 11 times when δ is small. We also notice that as δ increases, the average precision does not decrease until $\delta = 30$, which means most of the optimal solutions can be found by doing only a small number of discretization work. Moreover, the number of edges in G_P also does not increase significantly comparing to the number of edges in G. Hence, the ATWD-LSA algorithm is able to efficiently solve SPPTW with high precision on positive edge weight graphs.

3.4.3 ATWD-LSA on Graphs with Different Sizes

In this section, we evaluate the performance of ATWD-LSA on different sized graphs. We selected Solomon's *RC201* type with 100, 200, 400, 800 nodes, and for each data set, we generate a time constrained directed graph G with no negative edge cost. For each graph G, we evaluate the algorithm under threshold $\delta = (1, 4, 20)$ and present the averaged statistics in Table 3.2

Note that in all cases, the ATWD-LSA can find the optimal solution. The original graph G with $100, 200, 400, 800$ nodes have $1072, 2284, 4840, 5369$ edges, respectively. When we increase the size of G from 100 nodes to 400 nodes, the size of G_P does not change much on average. And it is interesting that when G has 800 nodes, the average size of graph G_P on solving 10 random SPPTW problems even decreases. We think this is because on large graphs, our strategy of breaking "long" edges will cause too many edges to be removed (e.g.,

V	Threshold(δ)	Precision	V_P	E_P	Running Time(s)
100	1	1	1130	8043	0.153
	4	1	739	5120	0.040
	20	1	275	1706	0.013
200	1	1	1243	13655	0.122
	4	1	832	9254	0.078
	20	1	272	2638	0.0216
400	1	1	1546	18719	0.152
	4	1	791	8753	0.074
	20	1	270	2435	0.019
800	1	1	1067	10637	0.059
	4	1	655	7334	0.071
	20	1	183	1853	0.020

Table 3.2: ATWD based Label Setting Algorithm under different thresholds, on graphs with different sizes, fixed time window sizes TW=120 for all nodes.

400 nodes G has 4840 edges while 800 nodes G has 5369 edges). Hence it is hard to find a path that crosses a wide area of graph G, which will result in a smaller G_P. However, this actually demonstrates that the algorithm matches our goal of enabling "on-demand" discretization since the algorithm will construct a partially time-expanded network around the optimal path area without touching the rest of the graph G.

3.4.4 ATWD-LSA on Graphs with Different Time Window Sizes

In this section, we evaluate the performance of ATWD-LSA on graphs with variable time window sizes. We selected Solomon's 5 different types of data (*RC201, RC202, RC204, RC205, RC208*) with 100 nodes, and for each data set, we generate a time-constrained directed graph G with no negative edge weights. For each graph G, we evaluate the algorithm under threshold $\delta = (1, 4, 20)$ and present the averaged stats in Table 3.3

As shown in Table 3.3, when the average time window size increases, the size of the graph G_P, as well as the solving time increases dramatically. This is because when the time window increases, setting the same value of δ will result in more node copies. Even when the average time window size increases from 120 (*RC201*) to 717.1 (*RC204*), the ATWD-LSA algorithm is still able to find the optimal solution efficiently with a larger threshold δ. However, one observation is that the time window size on graph G has a higher impact on the complexity of our solution than the graph size, which is noticeable by comparing Table 3.3 with Table 3.2.

Data set Type	Threshold(δ)	Precision	V_P	E_P	Running Time(s)	TW(mean)	TW(std)
RC201	1	1	1130	8043	0.153	120	0
	4	1	739	5120	0.040		
	20	1	275	1706	0.013		
RC202	1	1	7700	56605	0.718	318.96	344.67
	4	1	3333	25899	0.248		
	20	1	863	6725	0.057		
RC204	1	1	16456	177615	3.073	717.1	344.93
	4	1	8681	92231	1.085		
	20	1	2181	23868	0.266		
RC205	1	1	3852	33460	0.37117	223.06	161.84
	4	1	1983	17856	0.152		
	20	1	573	4464	0.034		
RC208	1	1	18326	206609	0.686	471.93	71.31
	4	1	6864	78166	0.787		
	20	1	1527	16634	0.135		

Table 3.3: ATWD based Label Setting Algorithm under different thresholds, on 100 nodes graphs with variable time window sizes.

3.5 Conclusion

In this chapter, we present an ATWD based label setting algorithm, which can be used for solving SPPTW on graphs with positive edge weights. We define the label and the dominance rule, which are also applicable to the label correcting algorithm in the next chapter. We demonstrate that the algorithm is able to generate a small partially time-expanded network, which still can be used for finding the optimal results. We conduct our experiments on graphs with different number of nodes, different time window sizes. The results show that the algorithm is efficient in all types of graphs.

Chapter 4

ATWD on Label Correcting Algorithm

In this chapter, we present how one can embed the ATWD approach into a label correcting algorithm. With a label correcting algorithm, we can enlarge the SPPTW problem scope to handle graphs with negative edge weights. Unfortunately, a traditional label correcting algorithm cannot be directly applied for solving shortest path problems on graphs with negative cycles. We discuss the proposed solutions for handling the negative cycles in the next two chapters.

When the graph has negative edge weights, label setting algorithms may be incorrect even if there is no negative cycle. The reason is with the presence of negative edge weights, the current shortest path from source to a node may need to be corrected as we evaluate different paths arriving at the same node. For example, in Dijkstra's algorithm, the greedy approach of selecting the minimum cost label is proved to be correct only when all edge weights are positive. With negative edge weights, label correcting algorithms should be used for solving the shortest path problems.

In Section 4.1, we describe the ATWD based label correcting algorithm. The algorithm uses a breadth-first-search strategy for discretizing "unvisited" neighbors, which has already been introduced in Chapter 2 Algorithm 1. However, instead of exhaustively discretizing the whole graph, we add another input variable l_{max} to control the maximum length of the path that the current solution is allowed to find. With this variable, the algorithm is able to compute a solution for SPPTW up to path length l_{max}. In Section 4.2, we again use an illustrative example to explain the proposed algorithm. We present our experimental results and some analysis in Section 4.3, and we briefly conclude this chapter in Section 4.4.

4.1 Label Correcting Algorithm with ATWD

In this section, we show how the ATWD method can be applied on a label correcting algorithm to solve SPPTW.

Algorithm 4: ATWD based Label Correcting Algorithm

Input: Graph $G(V, E)$, source node s, destination node d, maximum length l_{max}
Output: Partially time-expended graph $G_P(V_P, E_P)$, result path P

1 **initialize** $l := 0, V_P := \emptyset, \mathcal{N}_{curr} := \emptyset, \mathcal{N}_{next} := \emptyset$, $\forall v \in V : \mathcal{C}(v) := \emptyset$, $\forall v \in V_P : \mathcal{L}(v) := \emptyset$.
2 $s_0 = (s, 0, e_s)$;
3 $V_P := V_P \cup \{s_0\}$, $\mathcal{N}_{next} := \mathcal{N}_{next} \cup \{s_0\}$;
4 cost$(s_0) := 0, \mathcal{L}(s_0) := \mathcal{L}(s_0) :=$NULL;
5 **while** $\mathcal{N}_{next} \neq \emptyset$ and $l < l_{max}$ **do**
6 $\quad \mathcal{N}_{curr} := \mathcal{N}_{next}$; $\mathcal{N}_{next} := \emptyset$;
7 $\quad$ **foreach** $v_a \in \mathcal{N}_{curr}$ **do**
8 $\quad\quad$ **foreach** neighbor index w in G that is not yet connected with v_a in G_P **do**
9 $\quad\quad\quad t' = v_a.t + t_{vw}$;
10 $\quad\quad\quad b = GetCopyIndex(w, t', \mathcal{C}(w), \delta)$;
11 $\quad\quad\quad$ **if** $b < 0$ **then continue**;
12 $\quad\quad\quad w_b := (w, b, t')$;
13 $\quad\quad\quad$ **if** $w_b \notin \mathcal{C}(w)$ **then**
14 $\quad\quad\quad\quad V_P := V_P \cup \{w_b\}, \mathcal{N}_{next} := \mathcal{N}_{next} \cup \{w_b\}$;
15 $\quad\quad\quad\quad \mathcal{C}(w) := \mathcal{C}(w) \cup \{w_b\}$; $\mathcal{L}(w_b) := v_a$;
16 $\quad\quad\quad E_P := E_P \cup (v_a, w_b)$; cost$(w_b) :=$cost$(v_a) + c_{vw}$;
17 $\quad\quad\quad update =$ True;
18 $\quad l := l + 1$;
19 **do**
20 $\quad update =$ False;
21 $\quad$ **foreach** edge $(v_a, w_b) \in E_P$ **do**
22 $\quad\quad$ **if** $cost(w_b) > cost(v_a) + c_{vw}$ **then**
23 $\quad\quad\quad$ cost$(w_b) :=$cost$(v_a) + c_{vw}$; $\mathcal{L}(w_b) := v_a$;
24 $\quad\quad\quad update =$ True;
25 **while** *update*;
26 $d_m :=$ node copy with minimum cost under destination index d;
27 $P :=$ find path from s_0 to d_m by backtracking $\mathcal{P}$;
28 **return** P;

4.1.1 Algorithm Description

The label correcting algorithm is presented in Algorithm 4. Line 5 to line 18 probes the graph and construct the partially time-expanded graph G_P using Breadth-first Search (BFS). It terminates when the graph does not change within an iteration or the path has reached the

maximum specified length. This process is almost identical to Algorithm 1 except that we bound the maximum depth in BFS. We add two sets: $\mathcal{N}_{curr}$ for saving the nodes for current depth and $\mathcal{N}_{next}$ for the next depth (current depth + 1). Line 19 to line 25 is the edge relaxation process. The logic is similar to the Bellman-Ford algorithm: we keep relaxing all edges in E_P for at most $|V| - 1$ times ($|V|$ is the number of nodes in time-constrained graph). Here we incorporate the idea of **early termination** [1] by checking the *update* value to speed up the algorithm. The variable *update* will be set to *True* if any new node copy is created or any cost is updated, and if *update* is *False*, then early termination criteria have been met. Line 6 specifies we only probe and discretize the graph when current depth is less than l_{max}. When the algorithm terminates, the solution path is the best SPPTW solution up to visiting l_{max} nodes with respect to threshold δ. The label contains only the predecessor node of the best path found so far and will be used when retrieving the optimal path via backtracking.

Labels

The definition of a label in ATWD based label correcting algorithm is the same as in the label setting algorithm (in Chapter 3, Section 3.2.1). Saving only one label with the minimum label cost is sufficient for computing the shortest path on graphs without negative cycles. The proof follows in a similar way, as presented in Chapter 3, Section 3.2.2.

4.1.2 Correctness Analysis

ATWD based label correcting algorithm is guaranteed to find the optimal shortest path in time-constrained graph G if and only if:

- the optimal shortest path in G can be represented using a path in G_T, where G_T is a time-expanded representation of G under threshold δ.
- the optimal shortest path has length $\leq l_{max}$.

If the optimal shortest path in G can be represented using a path in G_T, then the time of visit at each node along the shortest path can be represented as an integral multiple of δ. If δ is an integer, according to Corollary 2 in Section 2.4.5, we can always find a path in the partially time-expanded graph G_P, which is identical to the path in G_T.

If the l_{max} value has been reached, it implies that we have constructed G_P up to l_{max} depth using breadth-first-search probing. Thus, any shortest path that has length $\leq l_{max}$ will be contained in G_P.

If the input graph has only integral times, then setting $\delta = 1, l_{max} = |V| - 1$, the algorithm is guaranteed to find the optimal solution. Otherwise, the solution will always be an upper bound.

4.1.3 Running time Complexity

In this section, we analyze the time complexity of the ATWD based label correcting algorithm. In the worst case, the outer loop (line 5 - 24) will iterate over $min(l_{max}, |V| - 1)$ times. This is because the edge relaxation will be performed at most $|V| - 1$ times according to Bellman-Ford, where $|V|$ represents the number of vertices in the time-constrained graph G. In the time-expanded graph, since edge costs between any copy of one index to any copy of another index are the same, the edge relaxation only needs to be performed at most $|V| - 1$ times. Line 7 - 17 loops for at most $|E_P|$ times and each loop's time complexity is constant. Line 18 - 21 iterates over all edges in E_P. Note that for any edge (v, w) in graph G, the ATWD algorithm only connects each node copy with index i with at most one copy with index w. In other words, each node copy in graph G_P will only have at most one connection to all node copies with the same index. Therefore, line 18 will loop for $\mathcal{O}(|E|t_{max}/\delta)$ times (t_{max} is the maximum time window size in G). The worst time complexity for doing backtracking is $\mathcal{O}(|V|)$.

Thus, the overall time complexity of ATWD based label correcting algorithm is

$$\mathcal{O}(|V||E|(t_{max}/\delta))$$

Although the time complexity is pseudo-polynomial, in practice, the actual time complexity is much lower due to the benefit of early termination and the maximum path length l_{max}.

Further Improvements

Based on Algorithm 4, we further improve the efficiency of the algorithm by performing some of the edge relaxation tasks while doing the discretization. Specifically, we update the cost of node w_b in lines 14 - 16. And in line 18, we skip w_b if w_b is already visited in line 8. By adding these changes, the algorithm will have improved performance in practice as the edge relaxation can be done on-the-fly while discretizing the nodes.

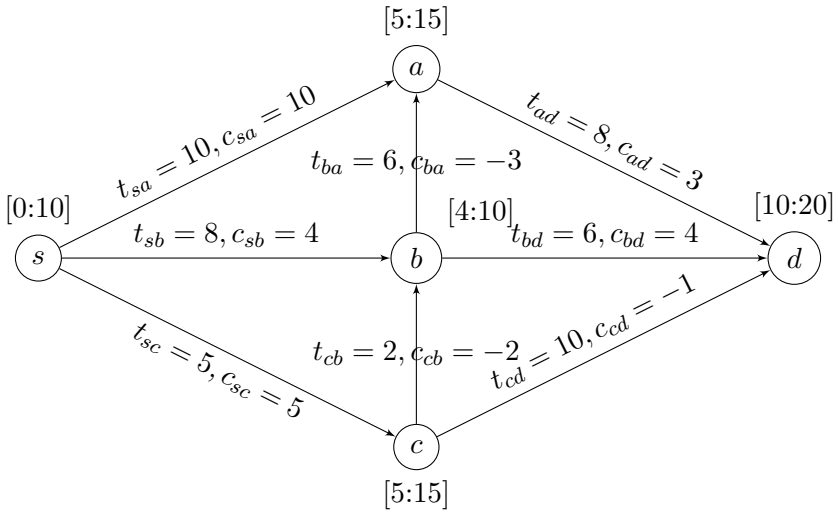


Figure 4.1: An example graph, each node is assigned with a time window, each edge is assigned with travel time t and cost c.

4.2 An Illustrative Example

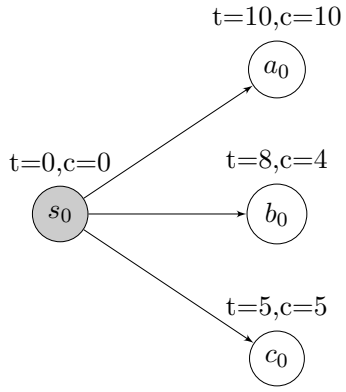
Consider the time-constrained graph G in Figure 4.1. G has the same structure as graphs in Figures 2.2 and 3.1, but now it contains negative edge weights. The graph construction process from Figure 4.2a to Figure 4.2e is identical to the process illustrated in Chapter 2, Section 2.4.3. Note that in Figure 4.2d, after we probe the neighbors of c_0, the labels on node b_0, a_1, d_1 are updated because of their already established connections c_0 and b_0. The algorithm terminates with $l = 3$. However, the optimal solution is already found in the iteration of $l = 2$, which means setting $l_{max} = 2$ is good enough for finding the optimal result in this case. In our experiments, we test the algorithm using different values of l_{max} and evaluate the results.

4.3 Experiments

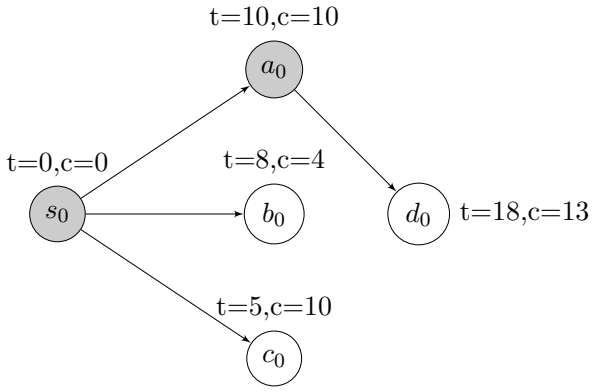
We evaluate the performance of the ATWD based Label Correcting Algorithm (ATWD-LCA) in this section.

4.3.1 Building the Experimental Graphs

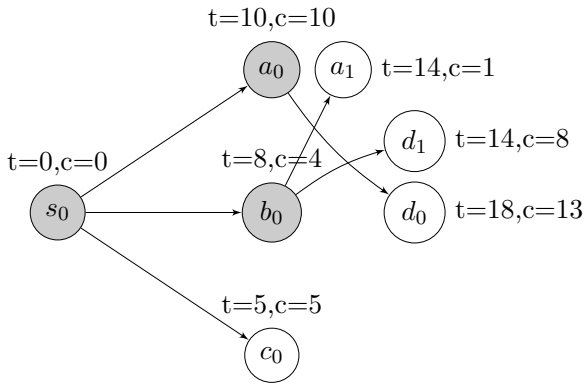
Based on the strategy introduced in Chapter 3, Section 3.4.1, we slightly modify the guidelines to generate graphs which are more suitable for testing the label correcting algorithm.



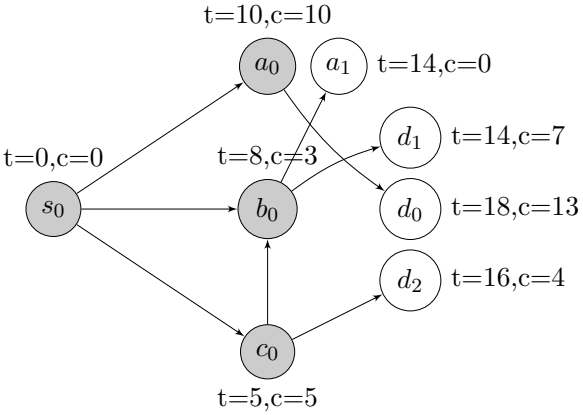
(a) After probing s_0's neighbors. a_0, b_0, c_0 are added. $l = 1$.



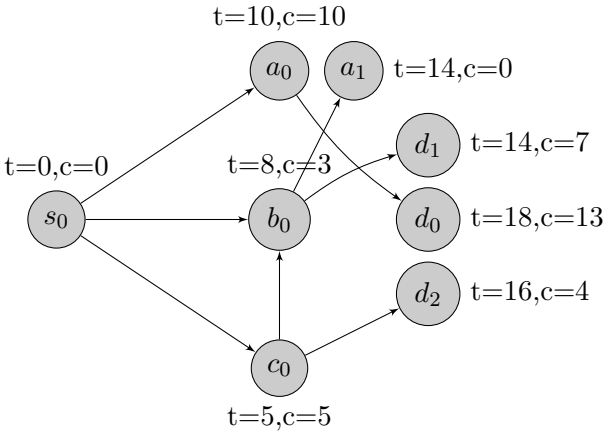
(b) After probing a_0's neighbors. d_0 is added.



(c) After probing b_0's neighbors. a_1,d_1 are added.



(d) After probing c_0's neighbors. d_2 is added. $l = 2$.



(e) After probing a_1,d_0,d_1,d_2's neighbors. $l = 3$. Graph is unchanged, algorithm terminates

Figure 4.2: ATWD based label correcting algorithm illustration with $\delta = 2, l_{max} = 5$. Nodes in grey means their neighbor nodes have been discretized.

Assuming the mean and the standard deviation of euclidean distances between all pair of nodes is d_{mean} and d_{std}:

- Break any "long" edge (i, j) if its euclidean distance $\sqrt{(i.x - j.x)^2 + (i.y - j.y)^2}$ is longer than $d_{mean} - d_{std}$.
- Define two parameters p, r. p denotes the percentage of edges whose costs are set to negative, and r denotes the distance reduction factor. We randomly choose p percentage of all edges, and for each chosen edge (i, j), we set the cost to be

  $$c_{ij} = -\sqrt{(i.x - j.x)^2 + (i.y - j.y)^2}/r$$

  For all other edges, we use euclidean distance as the cost. In this chapter, we set $p = 8\%$.
- For each edge (i, j), we initially assign euclidean distance of (i, j) as the travel time, and then normalize all travel times to $[0, 10]$ ($t_{norm} = 10 \cdot \frac{t - t_{min}}{t_{max} - t_{min}}$), and shift the travel time up by service time.

The intuition of introducing distance reduction factor r is that we do not want too many negative cycles being created. As in this chapter, we focus only on solving SPPTW (not ESPPTW). If too many negative cycles are created, it will be harder to remove them and construct a negative-cycle-free test graph.

Removing Negative Cycles

We use Algorithm 5 to remove negative cycles based on the graph generated using the strategies introduced in the above section. The algorithm works as repeatedly checking the existence of negative cycles using the Bellman-Ford algorithm. If we find a negative cycle, we trace it and find the edge (i, j) with minimum cost on the cycle. It is obvious that $c_{ij} < 0$. We then set the cost of (i, j) to positive by "reverting" it back to the euclidean distance between to endpoints. We repeat the process until no negative cycle is detected.

Algorithm 5: Negative cycle elimination

Input: Graph G
Output: Negative-cycle-free graph G

```
while G has negative cycle do
    Find the minimum cost edge (i, j) on the cycle;
    set c_ij = sqrt((i.x - j.x)^2 + (i.y - j.y)^2);
return G;
```

4.3.2 ATWD-LCA with Different Thresholds

We generate a time-constrained directed graph G with no negative cycles using Solomon's *0100_RC201*, which has 100 nodes, and each node has a fixed time window (TW) size 120. We randomly choose 10 pairs of source and destination nodes, and for each pair, we run the ATWD-LCA algorithm with $l_{max} = 99$ under different thresholds ($\delta = 1, 2, 4, 10, 20, 30$) and collect the results. We then take the average of the results and present them in Table 4.1.

$l_{max} := 99$				
Threshold(δ)	Precision	V_P	E_P	Running Time(s)
1	1	5720	49487	0.806
2	1	3472	33506	0.643
4	1	1817	17886	0.333
10	1	797	7968	0.124
20	1	436	4292	0.063
30	0.94	315	3134	0.0502

Table 4.1: ATWD Label Correcting Algorithm under different thresholds, on a graph generated from Solomon 0100_RC201, which has a fixed time window size TW=120 for all nodes.

As shown in Table 4.1, the ATWD approach still helps to significantly reduce the complexity of the time-expanded graph. However, the graph G_P is not as small as the graph constructed by the label setting algorithm, which can be observed by comparing Table 4.1 with Table 3.1. The reason is in the label setting algorithm, we can terminate whenever we "reach" the destination node, but in the label correcting algorithm, due to the presence of negative edge weights, we have to keep relaxing all the edges until no change takes place. This will result in an exhaustive discretization. But the good news is that the ATWD-LCA algorithm is still able to precisely solve SPPTW with some larger thresholds. Similar to the results in Table 3.1, ATWD-LCA is able to find an optimal solution in all cases when $\delta \leq 20$.

4.3.3 ATWD-LCA with Bounded Maximum Path Length

When we set the value of l_{max}, the algorithm will only construct the partially time-expanded G_P with a maximum of l_{max} depth using breadth-first-search. As shown in Table 4.2, if we set l_{max} to be small, we are able to make the algorithm run much faster, and the resulting graph G_P is also less complex. We experiment by setting the value of l_{max} from 3 to 10. When l_{max} increases, the precision of the solution increases, along with a penalty of an increased solving time due to the expansion of graph G_P. However, we find that setting $l_{max} = 8$ is already good enough for solving all 10 different SPPTW problems optimally. Comparing

l_{max}	Precision	V_P	E_P	Running Time(s)
3	0.53	277	964	0.012
4	0.63	631	3361	0.051
5	0.74	1086	7724	0.111
6	0.82	1572	13074	0.217
7	0.96	2039	18536	0.297
8	1	2461	23574	0.484
9	1	2821	28042	0.469
10	1	3129	31714	0.631

Table 4.2: ATWD Label Correcting Algorithm under different l_{max}, with $\delta = 1$, on a graph generated from Solomon 0100_RC201, which has a fixed time window size TW=120 for all nodes.

the graph size with respect to the number of node copies in V_P and the number of edges in E_P, we observe that with the help of l_{max}, we are able to further reduce the complexity of the ATWD-LCA algorithm by solving SPPTW on an even smaller time-expanded graph that is about half the size of the graph constructed without setting l_{max}.

4.4 Conclusion

In this chapter, we introduce the ATWD based label correcting algorithm. The algorithm differs from the label setting algorithm introduced in the previous chapter as it can be used on graphs with negative edge weights, as long as no negative cycle is present. With negative edge weights being introduced into the graph, the algorithm becomes more complex as we need to relax all edges, which means the entire time-expanded network needs to be created. Hence as shown in experiments, the constructed partially time-expanded graph is larger than that of Chapter 3. In order to further optimize the running time, we use another parameter l_{max} to restrict the maximum shortest path length we aim to find. The experiments demonstrate that it reduces the graph size by half without harming the precision in most of the cases during the experiment.

Chapter 5

Label Correcting Algorithm with k-Cycle Elimination

The label correcting algorithm introduced in the previous chapter can be used to solve SPPTW. When the graph contains negative cycles, solving Elementary Shortest Path with Time Windows (ESPPTW) is proved to be strongly NP-Hard, and, therefore, no pseudo-polynomial time algorithm can be found [39]. In this chapter, we explored the k-cycle elimination technique and extended the label correcting algorithm for solving ESPPTW approximately.

k-cycle elimination has been used in the label correcting algorithm to ensure no cycle with length $\leq k$ exists in the resulting shortest path. In the following sections, we define the dominance rule, which is used for pruning labels. We show that the k-cycle elimination strategy can be easily applied to ATWD label correcting algorithm to eliminate small cycles in the resulting path. Lastly, we generate test graphs based on Solomon's data and demonstrate that the ATWD label correcting algorithm with k-cycle elimination is able to efficiently solve the ESPPTW in most cases.

5.1 Dominance and Label Pruning

5.1.1 Labels

The concept of labels is used to store different paths represented in node sets, which has been explained in Chapter 3, Section 3.2.1. In this chapter, we apply a very similar definition, which is explained as follows. Given a path P originated from source node s, $P = (s, v_1...v_{i-1}, v_i)$, the label $L(v_i)$ associated to the path P on node v_i can be defined as a set $L(v_i) = \{s, v_1...v_{i-1}\}$.

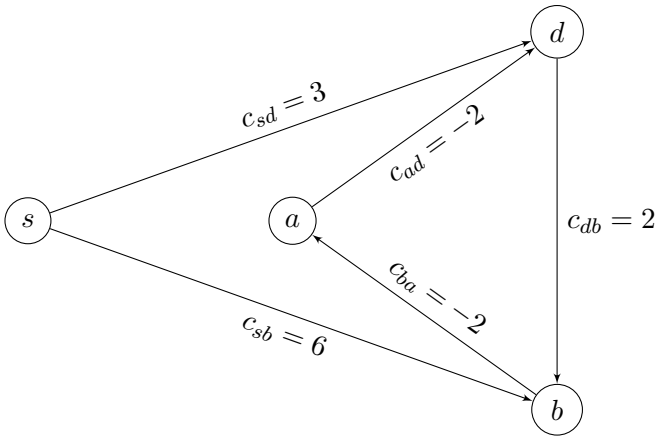


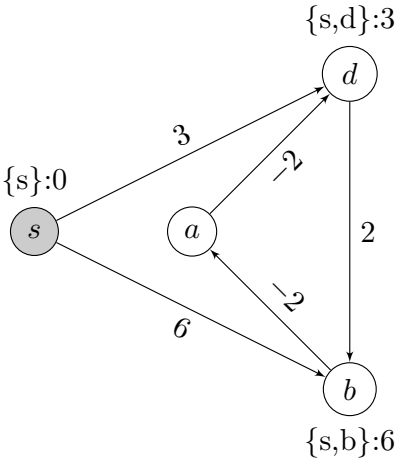
Figure 5.1: An example graph, each edge is assigned with cost, label vectors and its total cost are shown next to the nodes. ATWD label correcting algorithm will fail to find the elementary shortest path from s to d.

The above label definition is applied in ATWD based label correcting algorithm with k-cycle eliminations, which is used for solving ESPPTW. If the graph contains negative cycles, the dominance rule presented as Lemma 1 in Chapter 3, Section 3.2.1 is no longer valid. The reason is that, given two different paths P_1, P_2 from a same node a to a same node b, we cannot guarantee that the path with smaller cost can always dominate the other. Since in the presence of negative cycles, extending the longer path may result in a shorter total cost as some nodes in its extension may have been visited by the other path. This can be further explained using an example in Figure 5.1. The graph contains a negative cycle b, a, d. If we apply the dominance rule as described in Chapter 3 (assuming the node discretization steps are ignored since we are only interested in edge relaxations), the resulting shortest path from the ATWD label correcting algorithm presented in Chapter 4 is (s, d), which is incorrect as the optimal shortest path is (s, b, a, d). Details of the relaxation steps is shown in Figure 5.2.

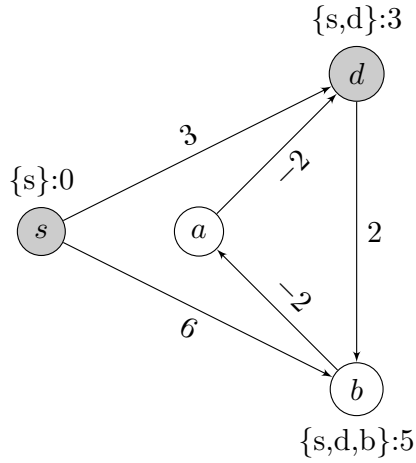
5.1.2 Dominance Rules

Lemma 2. Consider a time-expanded graph G_P with negative cycles, given two different labels $\mathcal{L}(v_i), \mathcal{L}(v_i)'$ at a same node representing the set of nodes visited on different paths P, P', if $\mathcal{L}(v_i) \subseteq \mathcal{L}(v_i)'$ and $cost(P) < cost(P')$, then P *dominates* P'.

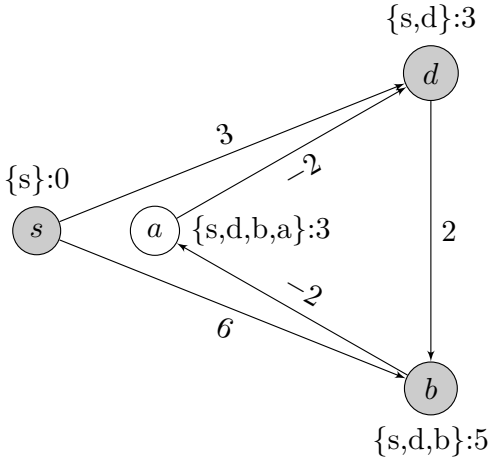
Proof of Lemma 2. If $\mathcal{L}(v_i) \subseteq \mathcal{L}(v_i)'$, then the set of visited nodes on path P is a subset of the set of visited nodes on path P'. Since both paths arrive at a same node copy, we can assume that they arrive at the same time. Hence, any path extension of P_2 must also be a valid path extension of P. Combining with the fact that $cost(P) < cost(P')$, it is clear



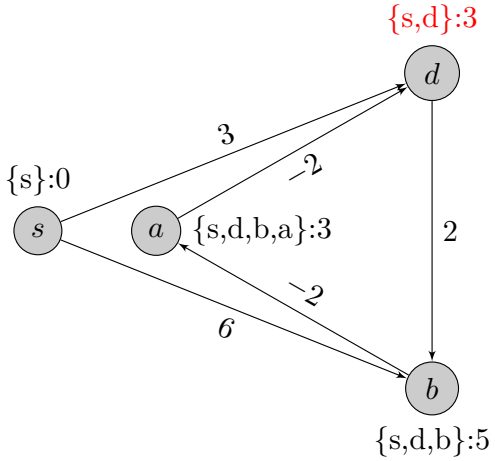
(a) After probing s's neighbors.



(b) After probing d's neighbors. Label on b is updated since $\{s, b\}$ is dominated by $\{s, d, b\}$.



(c) After probing b's neighbors.



(d) After probing a's neighbors. Label on d is not updated because label $\{s, d, b, a, d\}$ is dropped due to duplicates.

Figure 5.2: Edge relaxation steps using old dominance rules based on Figure 5.1. The labels and their path costs are presented next to the nodes. Solution path (s, d) is not optimal.

that P dominates P' because any path from source to destination extended from P' can be further optimized by replacing P' with P. □

Figure 5.3 shows how the new dominance rule can be applied to compute the correct elementary shortest path using the same example graph in Figure 5.1. Since we can only discard a label if it is dominated by at least one existing label on the same node, we need to carry all labels that are not dominated. In this case, we need to keep track of the connections between labels on adjacent nodes for backtracking purposes, and a backward link pointer pointing to the corresponding predecessor label is added for each label. When the algorithm terminates, we have two labels attached to the destination node: $\{s, d\}, \{s, b, a, d\}$, and the one with the minimum path cost is selected as the result.

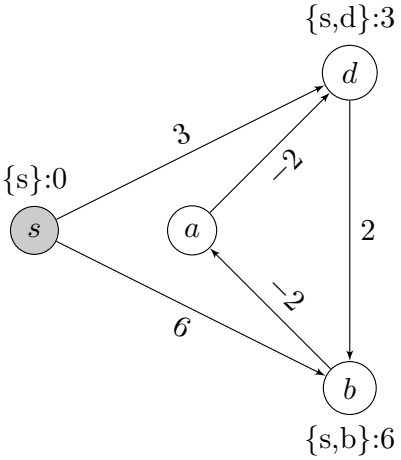
5.1.3 Label Pruning

In the label correcting algorithm introduced in Section 4.1, a label $\mathcal{L}(v_i)$ representing path P can be pruned when there exists at least another label $\mathcal{L}(v_i)'$ representing path P' such that P' dominates P. According to the dominance rule introduced in *Lemma 2* of this chapter, for each node copy v_i, we need to carry several paths from the source node to v_i. As a result, the number of labels grows very fast.

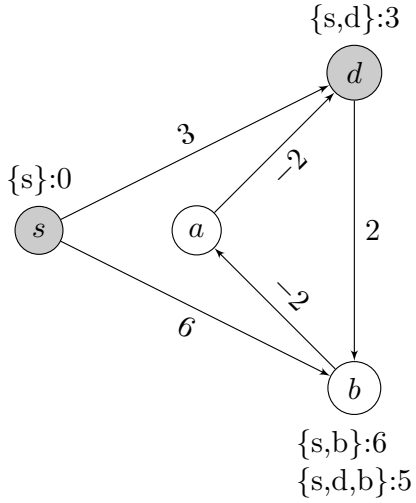
Unfortunately, even if we apply the k-cycle elimination, which will be explained in the next section, the number of labels per node still grows dramatically as k increases. Irnich and Villeneuve stated that the worst-case complexity of k-cycle elimination using different values of k grows by a factor that depends on k [33]. Moreover, on graphs with densely connected nodes, the number of labels per node can be significantly large, even with a smaller k. Initially, we designed the k-cycle elimination algorithm by keeping all non-dominated labels. In experiments, we found that with $k = 2$, there will be a maximum of hundreds of labels per node, but with $k = 3$, the number of labels grows up to thousands. The running time increases from minutes ($k = 2$) to hours ($k = 3$). Thus, based on the information collected, we introduce a strategy for bounding the number of labels for each node.

Bounding the Number of Labels

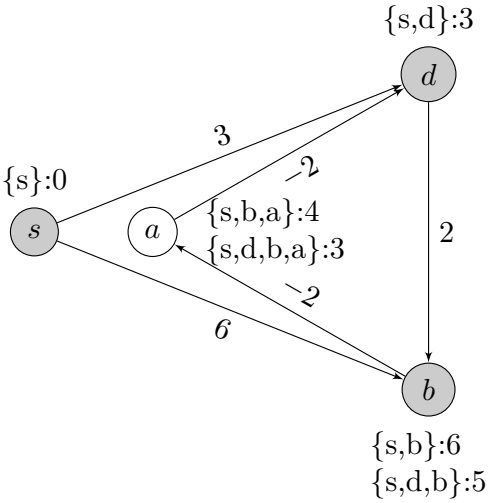
The number of labels saved per node grows exponentially with respect to k. Since we only need to keep the unique set of previous k visited nodes unordered, the exponential growth in number of labels still makes the algorithm impractical with larger k.



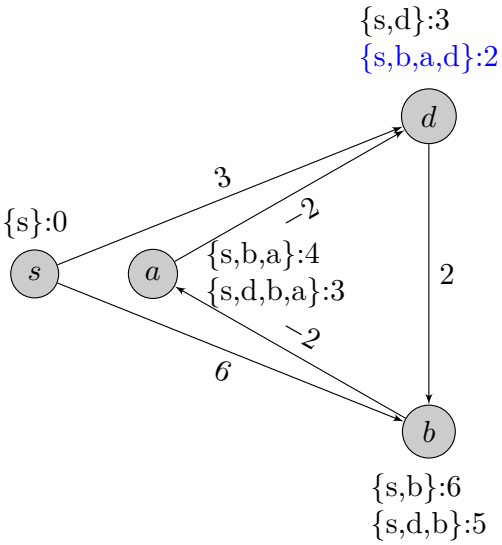
(a) After probing s's neighbors.



(b) After probing d's neighbors. A new label $\{s, d, b\}$ is added to b since it is not dominated by any existing labels.



(c) After probing b's neighbors. Node labels $\{s, b, a\}, \{s, d, b, a\}$ are added to a.



(d) After probing A's neighbors. Label $\{s, b, a, d\}$ is added to d and $\{s, d, b, a, d\}$ is dropped.

Figure 5.3: Edge relaxation steps using new dominance rules based on Figure 5.1. Solution path (s, b, a, d) is optimal.

The intuition of bounding the number of labels is developed from empirical evidence: each node often keeps lots of high-cost labels. Although a label should not be pruned if it is not dominated, we compromise the quality by eliminating labels with relatively higher costs.

We introduce a parameter τ, which denotes the maximum cost threshold. If a label has a cost greater than the threshold, then even if it is not dominated, we will still prune it. For each node copy, its maximum cost threshold is a function of the current minimum cost of all labels, standard deviation among all edge costs, and the parameter τ.

In order to present the k-cycle elimination algorithm, we define the following enhanced dominance rules:

Enhanced Dominance Rules

Consider a time-expanded graph G_P:

- A label $L(v_i)$ representing path P is *strongly dominated* if there exists another label $L(v_i)'$ representing path P' on the same node v_i such that $L(v_i)' \subseteq L(v_i)$ and $cost(P') < cost(P)$.
- A label $L(v_i)$ representing path P is *weakly dominated* if there exists another label $L(v_i)'$ representing path P' on the same node such that $cost(P) > cost(P') + \tau \cdot std$ where std is the standard deviation of cost for all edges in G_P.

The strong dominance rule is derived directly from *Lemma 2* in this chapter, and the weak dominance rule is designed using the strategy for bounding the number of labels. If the gap between the cost of path P and the minimum cost of all paths currently saved is larger than a threshold, then we consider that any path extension of P is unlikely to become the optimal solution. The threshold value is defined as $\tau \cdot std$. We include std because the gap should be relevant to the dispersion of the cost for all edges. In other words, if the standard deviation is low, which means edge costs are close, then we should set a smaller threshold since the cost of different labels on the same node will also be close. Similarly, if the standard deviation is high, then the cost of different labels can be spread out. Hence we need a higher threshold. We add τ so that the threshold can be tuned. In experiments, we notice that the result generated from $\tau = 0.2$ is the best with respect to precision and efficiency, so we fix $\tau = 0.2$ in the experiments.

5.2 ATWD based Label Correcting Algorithm with k-Cycle Elimination

The algorithm in this section mainly differs from Algorithm 4 (Chapter 4, Section 4.1) in edge relaxation. In Algorithm 4, we apply the same label update procedure as in the Bellman-Ford algorithm, which is simply updating the label whenever the new label has a lower cost. However, in the k-cycle elimination algorithm, we need to apply a more complicated label update procedure, which is presented as Algorithm 6.

Algorithm 6 is called whenever we want to relax an edge (v_a, w_b). It takes the sets of labels at v_a, w_b as input, which are denoted as $\mathcal{L}(v_a), \mathcal{L}(w_b)$ respectively. It returns an updated set of labels of w_b, and a Boolean variable *update* for indicating whether any change has been made. Line 3 checks if any cycle is going to be introduced by extending the path from v_a to w_b, and skip the current label if the checking returns true. In lines 4,6 and 10, the algorithm applies the enhanced dominance rule introduced in Section 5.1.3 to check whether the path P represented by each label L_{v_a} on node v_a is dominated. If P is dominated (either strongly or weakly), then we skip L_{v_a} (line 5). If P strongly dominates another path P' represented by label L_{w_b}, then we replace L_{w_b} with the new label generated by adding w_b to L_{v_a}, and we point L_{w_b} to L_{v_a} using a backward pointer for backtracking purposes (line 7 - 9). If P is neither dominated nor strongly dominates some existing paths, then we create a new entry for that label and add to $\mathcal{L}(w_b)$ (line 11 - 15). Note that we restrict the length of any label to be $\leq k$. In line 16 - 20, we check if the cost of the newly introduced label is the minimum in the set. If yes, then we need to perform weakly dominance scan for all existing labels in the set and remove any label that is weakly dominated.

Whenever we remove a label, all pointers pointing to that label become invalid and must be removed. Thus, in lines 21 - 24, we iterate through all backward pointers and remove any label that has its pointer pointing to a non-existent label.

We present the complete ATWD label correcting algorithm with k-cycle elimination in Algorithm 7. The overall structure is very similar to that of Algorithm 4. Note that in the algorithm we use L_{v_a} to denote both the label of v_a and the path represented by the label. Major differences are as follows:

- each label is more complicated, which contains the last k nodes of the path instead of just keeping one predecessor.
- we keep a set of labels instead of just one label for each node.
- we use enhanced dominance rules for updating labels, and the label update procedure is more complicated (as described in the above paragraph).

Algorithm 6: UpdateLabels

Input: $\mathcal{L}(v_a), \mathcal{L}(w_b)$
Output: *update*, $\mathcal{L}(w_b)$

initialize *update* := False
foreach $L_{v_a} \in \mathcal{L}(v_a)$ **do**
  if extending L_{v_a} to node w_b will introduce a cycle **then** continue;
  if L_{v_a} is strongly or weakly dominated by $L_{w_b} \in \mathcal{L}(w_b)$ **then**
    continue;
  else if L_{v_a} strongly dominates $L_{w_b} \in \mathcal{L}(w_b)$ **then**
    L_{w_b} := last k elements in $L_{v_a} \cup \{w_b\}$; $L_{w_b}.prev := L_{v_a}$;
    cost(L_{w_b}):=cost(L_{v_a})+c_{vw};
    update := True
  else
    L_{new} := last k elements in $L_{v_a} \cup \{w_b\}$;
    $L_{new}.prev := L_{v_a}$;
    cost(L_{new}):=cost(L_{v_a})+c_{vw};
    $\mathcal{L}(v_a) := \mathcal{L}(w_b) \cup L_{new}$;
    update := True
  if cost$(L_{v_a})+c_{vw}$ is the minimum in $\mathcal{L}(w_b)$ **then**
    foreach $L_{w_b} \in \mathcal{L}(w_b)$ **do**
      if L_{w_b} is weakly dominated **then**
        remove L_{w_b} from $\mathcal{L}(w_b)$;
        update := True
foreach $L_{w_b} \in \mathcal{L}(w_b)$ **do**
  if $L_{w_b}.prev$ cannot be found in L_{v_a} **then**
    remove L_{w_b} from $\mathcal{L}(w_b)$;
    update := True;
return *update*, $\mathcal{L}(w_b)$;

In lines 16 and 23, whenever we relax an edge, we call Algorithm 6 for updating labels. In line 26, we need to scan all labels from all node copies with destination index, find the label with minimum cost, then do the backtracking to print the path P.

Algorithm 7: ATWD Label Correcting Algorithm with k-Cycle Elimination

Input: Graph $G(V, E)$, source node s, destination node d, maximum length l_{max}
Output: Partially time-expended graph $G_P(V_P, E_P)$, result path P

1 **initialize** $update$:= False, $V_P := \emptyset$, $\mathcal{N}_{next} := \mathcal{N}_{next} \cup \{s_0\}$, $\forall v \in V : \mathcal{C}(v) := \emptyset$, $\forall v \in V_P : \mathcal{L}(v) := \emptyset$
2 $s_0 = (s, 0, e_s)$;
3 $V_P := V_P \cup \{s_0\}$, $\mathcal{N}_{next} := \mathcal{N}_{next} \cup \{s_0\}$;
4 L_{s_0} :=NULL,$L_{s_0}.prev$:=NULL,cost$(L_{s_0}) := 0, \mathcal{L}(s_0) := \mathcal{L}(s_0) \cup L_{s_0}$;
5 **while** $\mathcal{N}_{next} \neq \emptyset$ and $l < l_{max}$ **do**
6 $\mathcal{N}_{curr} := \mathcal{N}_{next}$; $\mathcal{N}_{next} := \emptyset$;
7 **foreach** $v_a \in \mathcal{N}_{curr}$ **do**
8 **foreach** neighbor index w in G that is not yet connected with v_a in G_P **do**
9 $t' = v_a.t + t_{vw}$;
10 $b = GetCopyIndex(w, t', \mathcal{C}(w), \delta)$;
11 **if** $b < 0$ **then continue**;
12 $w_b := (w, b, t')$;
13 **if** $w_b \notin \mathcal{C}(w)$ **then**
14 $V_P := V_P \cup \{w_b\}, \mathcal{N}_{next} := \mathcal{N}_{next} \cup \{w_b\}$;
15 $\mathcal{C}(w) := \mathcal{C}(w) \cup \{w_b\}$;
16 res, $\mathcal{L}(L_{w_b})$:= UpdateLabels$(\mathcal{L}(L_{v_a}), \mathcal{L}(L_{w_b}))$
17 $E_P := E_P \cup (v_a, w_b)$;
18 $update$:= True;
19 $l := l + 1$;
20 **do**
21 $update$ = False;
22 **foreach** edge $(v_a, w_b) \in E_P$ **do**
23 res, $\mathcal{L}(L_{w_b})$:= UpdateLabels$(\mathcal{L}(L_{v_a}), \mathcal{L}(L_{w_b}))$;
24 **if** *res* **then** $update$:= True;
25 **while** $update$;
26 find the minimum cost node copy d_{min} which has the minimum cost label L_{min} from all node copies with index d;
27 P := find path from s_0 to d_m by backtracking $\mathcal{L}(L_{min})$;
28 **return** P;

5.3 Experiments

We evaluate the performance of ATWD based Label Correcting Algorithm with k-cycle Elimination (ATWD-LCA-k-cycle). Since we have already compared different values of l_{max} in Chapter 4, in this experiment, we will fix $l_{max} = |V| - 1$, which means we always let the

algorithm do the exhaustive discretization. Our main focus is to evaluate the precision of ATWD-LCA-k-cycle and how does the algorithm perform under different values of k.

ATWD-LCA-k-cycle algorithm only eliminates cycles up to length k, and the solution path P_r we find may have a smaller cost than the optimal path (if any negative cycle is included in P_r). In other words, the solution is no longer guaranteed to be an upper bound for the optimal solution unless it is confirmed that no cycle is introduced in P_r. Hence, the precision score in experimental results of ATWD-LCA-k-cycle calculated using the definition in Chapter 3, Section 3.4 can be greater than 1. We add a notation (N) after the precision score if the solution contains negative cycles.

5.3.1 Building the Experimental Graphs

We modify the algorithm explained in Section 4.3.1 by removing the reduction factors r_1, r_2. We also remove the negative cycle elimination process so that the test graph will (possibly) have negative cycles. We select several of Solomon's data sets *0100_RC201, 0200_RC201* with 100 and 200 nodes, and we generate a time-constrained graph G for each of the data sets. We apply the ATWD-LCA-k-cycle algorithm on these graphs, and we compared the performance under different values of k. We also compared the running time of ATWD-LCA-k-cycle against the time for solving the exact ESPPTW integer programming formulation. Note that we report the statistics related to each source and destination pair instead of reporting averaged results of 10 pairs (as in earlier chapters). This is because, with the possible existence of negative cycles, the precision may be either a lower bound or an upper bound, and aggregating the precision does not make sense.

5.3.2 Tuning the Parameter k

In this section, we evaluate the performance of ATWD-LCA-k-cycle under different values of k. We fix the value of τ, δ to be $\tau = 0.2, \delta = 1$. We randomly selected three source and destination pairs and perform three sets of tests (Test # 1,2,3). We report the maximum and average number labels saved for each node copy, and we compare the running time of ATWD-LCA-k-cycle against the IP solving time (using the formulation presented in Chapter 1, Section 1.3.3. Results are presented in Table 5.1.

When solving ESPPTW on a graph with 100 nodes, the proposed algorithm beats the IP solution in one test case out of three. In test # 1, we find the optimal solution when $k \leq 6$, and the running time is faster than IP solving time. In test #2, the algorithm gives the solution faster than IP, but the precision is 0.96, which is sub-optimal. Since increasing k to 10 does not change the precision, we think the reason we cannot get the optimal solution

is because $\tau = 0.2$ is too small, and the optimal path is pruned during the execution of the algorithm. In test #3, the algorithm gets a better precision comparing to test #2, but the running time is slower, and in this test, the algorithm does not perform as well as the IP.

When solving ESPPTW on graph with 200 nodes, our solution beats the IP solution in all three tests. Since we notice that the IP solver takes a huge amount of time to solve the optimal solution, so we manually terminate the IP solver at 1000 seconds and consider the best objective as the "optimal" solution. In this case, the precision score no longer measures the gap between our solution and the optimal solution.

In test #1, the proposed algorithm finds a better solution (with no negative cycle) with $k \geq 8$ in less than a minute, which is a valid solution with a lower cost. In test #2, the algorithm finds a solution that is very close to the solution of IP solver in less time. In test #3, it is interesting that the algorithm finds the same solution as the IP solver's almost immediately, and the number of labels generated is much smaller than that of test #1 and #2. This is because the source node we use in this test has a relatively late beginning time, so it is only reachable to a few other nodes, and hence the algorithm is able to create a very small time-expanded graph and quickly find the solution.

5.4 Conclusion

In this chapter, we illustrate that the k-cycle elimination strategy can be applied to the ATWD based label correcting algorithm. We present a new label definition and an enhanced dominance rule. The proposed algorithm takes the following two parameters: k, which specifies the maximum length of a cycle we aim to eliminate; τ, which specifies how many labels we keep per node. The intuition of adding k, τ comes from empirical findings: most of the paths only contain small cycles [33], and most of the high-cost labels do not contribute to finding the shortest path (an experimental observation). By setting a proper k and τ, the proposed algorithm is able to quickly find the optimal or near-optimal solutions. We also show that the designed algorithm can be used in larger graphs where optimal solutions are extremely expensive to find.

ATWD-LCA-k-cycle with $\delta = 1, \tau = 0.2$							
V	Test(#)	k	Precision	Labels(max)	Labels(mean)	Running Time(s)	IP Time(s)
		2	1.11 (N)	8	1.60	6.216	
		4	0.98	11	1.72	6.547	
100	1	6	1	15	1.82	11.82	18.29
		8	1	11	1.9	16.39	
		10	1	15	2.1	17.04	
		2	1.21 (N)	8	1.70	9.591	
		4	1.04(N)	10	1.74	11.435	
100	2	6	1.04(N)	10	1.87	22.13	42.83
		8	0.96	12	1.9	19.88	
		10	0.96	12	1.9	20.37	
		2	1.15 (N)	12	1.74	13.277	
		4	1.03 (N)	10	1.70	17.989	
100	3	6	1.02 (N)	11	1.80	27.523	27.449
		8	0.97	15	1.9	35.94	
		10	0.97	17	2.01	57.96	
		2	1.13 (N)	16	1.98	14.45	
		4	1.13 (N)	21	2.21	22.84	
200	1	6	1.13 (N)	23	2.18	28.83	N/A
		8	1.11	17	2.17	40.433	
		10	1.11	20	2.29	56.71	
		2	1.10 (N)	14	2.04	122.80	
		4	1.04(N)	32	2.32	232.22	
200	2	6	1.02(N)	22	2.52	377.248	N/A
		8	0.99 (N)	23	2.63	442.92	
		10	0.98	20	2.29	791.29	
		2	1	4	1.24	0.029	
		4	1	6	1.25	0.032	
200	3	6	1	6	1.25	0.033	N/A
		8	1	5	1.19	0.031	
		10	1	4	1.14	0.031	

Table 5.1: ATWD Label Correcting Algorithm with k-Cycle Elimination under different k, on different sized graphs with fixed time window sizes TW=120 (RC201).

Chapter 6

ATWD with Integer Programming

In this chapter, we present the Integer Programming (IP) based solution, which incorporates the Adaptive Time Window Discretization (ATWD) idea to efficiently solve ESPPTW. We first present the IP formulation of the problem within a time-expanded network context. Then we present our approach for solving the ESPPTW. ATWD based IP solution is an iterative refinement of the graph where the discretization happens *on-the-fly* while the problem is being solved. We evaluate the approach and find that it is able to find an optimal solution without time expanding the whole graph.

6.1 Integer Programming Formulation

The ESPPTW is defined in Chapter 1, Section 1.3. Since we use IP to solve the ESPPTW on the partially time-expanded network derived by our ATWD algorithm, we need to add additional constraints to the IP formulation. Suppose partial time-expanded network is $G_P(V_P, E_P)$, where V_P specifies the set of node copies and E_P specifies the edges. For each node copy v, we use $v.i$ to denote its index and $\mathcal{C}(v.i)$ to denote the set of node copies with the same index $v.i$. We also use V to denote the set of node indices in G. s, d represent the source index and destination index respectively. The IP formulation of the ESPPTW on G_P has one decision vector x. x_{ij}, which is defined on $\forall(i,j) \in E_P$, is a binary variable which represents whether it is selected in the optimal solution path. Note that compared with the IP formulation for ESPPTW introduced in Section 1.3.3, we do not have the decision vector y because the time dependency is already incorporated in G_P.

$$\min \sum_{(i,j)\in E_P} c_{ij}x_{ij}, \text{s.t.} \tag{6.1}$$

$$\sum_{i \in \mathcal{C}(s)} \sum_{(i,j) \in E_P} x_{ij} = 1 \tag{6.2}$$

$$\sum_{j \in \mathcal{C}(s)} \sum_{(i,j) \in E_P} x_{ij} = 0 \tag{6.3}$$

$$\sum_{j \in \mathcal{C}(d)} \sum_{(i,j) \in E_P} x_{ij} = 1 \tag{6.4}$$

$$\sum_{i \in \mathcal{C}(d)} \sum_{(i,j) \in E_P} x_{ij} = 0 \tag{6.5}$$

$$\sum_{(i,v) \in E_P} x_{iv} = \sum_{(v,i) \in E_P} x_{vi} \leq 1 \quad \forall v \in V_P, \text{ s.t. } v.i \notin \{s, d\} \tag{6.6}$$

$$\sum_{(v,w) \in E_P} x_{vw} \leq 1 \quad \forall i \in V \tag{6.7}$$

$$x_{ij} \in \{0, 1\} \quad \forall (i,j) \in E_P \tag{6.8}$$

In the above formulation, the objective is defined in equation 6.1, which is to minimize the total cost of the selected path. We impose the constraints in equations 6.2 - 6.8. Equation 6.2 restricts that there must have exactly one selected outgoing edge from one of the node copies with the source index. Equation 6.3 restricts that no incoming edge should be selected that points to any node copy with the source index. Similarly, we add the incoming edge and outgoing edge restrictions to node copies with destination index in equations 6.4 - 6.5. Equation 6.6 ensures the edge connectivity: for all internal nodes, if there is one incoming edge, then there must be an outgoing edge. By combining this with equation 6.7, we add another constraint which specifies that for each index, at most one incoming edge and one outgoing edge can be selected for all node copies of that index. This ensures the elementary property of the solution, meaning that the path is not allowed to visit an index more than once. Equation 6.8 is the integrality constraint.

We denote the above IP formulation as a "discrete" version, while the ESPPTW IP formulation presented in Section 1.3.3 is considered as a "continuous" version. Solving the discrete version of IP formulation on partially time-expanded graph G_P is empirically proved to be much faster than solving the "continuous" IP formulation of ESPPTW on the original graph G, during the experiments.

6.2 ATWD based IP Approach

In this section, we present the ATWD based IP approach. We denote the IP formulation on the time-expanded network G_P as ESPPTW(G_P). Our approach works as follows:

1. create an initial time-expanded graph G_P (Algorithm 8).
2. solve ESPPTW(G_P), get the solution path p and the total cost c.
3. check if the solution implies that the graph G_P needs refinement (Algorithm 9).
4. if the answer is no, then the algorithm terminates, and the path p is returned as the optimal elementary shortest path.
5. If the answer is yes, we modify the graph by adding more node copies and edges (Algorithm 9), and refine constraints in the IP formulation to accommodate the changes to G_P.
6. We do 2 - 5 repeatedly until no need for refinement is found while inspecting the solution path. In each round a "warm start" is applied to speed up the solving.

A flowchart which illustrates this procedure is presented in Figure 6.1. Note that the solution in the previous round will be a valid solution in the next round after the graph update. Hence, we can apply warm start (i.e. IP solving starts from current solution) to significantly improve the performance.

Algorithm 8 shows how we generate an initial time-expanded graph G_P. For each node index v with time window $[e_v, l_v]$ in the original graph G, we create two node copies v_0, v_1 and assign e_v, l_v to them, respectively. For each node copy, we iterate through its neighbors and connect it with the *earliest* reachable node copy.

Algorithm 8: Graph Initialization

Input: Original graph $G(V, E)$.
Output: Time-expended graph $G_P(V_P, E_P)$.

```
foreach v ∈ V do
    add nodes v_0(v, 0, e_v), v_1(v, 1, l_v) to V_P;
    add edge (v_0, v_1) to E_P;
foreach v_i ∈ V_P do
    for neighbor index m of v in G do
        connect v_i with the earliest reachable node copy w_j with index w;
        add edge (v_i, w_j) to E_P;
return G_P(V_P, E_P);
```

Algorithm 9 updates the graph G_P via probing the "neighborhood" of the current optimal solution. In the graph initialization, each node is discretized into only two copies. Due to the sparse discretization, any pair of nodes that are reachable in G may become unreachable in G_P since the intermediate node copies are missing. In other words, the sparse discretization (with fewer copies per node) may add more waiting times when traversing

edges in G_P. Thus, the intuition of Algorithm 9 is to check for edges in G_P with high waiting times, then improve it by adding a new node copy to the graph. We present the definition of the waiting time as follows.

Definition 1. Consider a time-expanded graph $G(V_P, E_P)$, the waiting time t_{wait} on edge $(v_a, w_b) \in E_P$ is defined as follows.

$$t_{wait} = \begin{cases} 0 & \text{if } v_a.t + t_{vw} \leq e_w \\ w_b.t - (v_a.t + t_{vw}) & \text{otherwise} \end{cases}$$

Note that we only count the waiting time when the arrival time $v_a.t + t_{vw}$ is after the beginning time of w: e_w. This is because if the arrival is before the beginning time, then there is no need to create a new node copy on index w since the waiting time is unavoidable.

Algorithm 9 proceeds as follows. In the loop from line 2 - 26, we loop over all nodes along with the current solution path P, and for each node $x_a \in P$, we probe its "neighborhood" area by performing *Breath-First Search* (line 4, 6, 7). We use node set $\mathcal{N}_{curr}$ to save the nodes to examine in current layer, and $\mathcal{N}_{next}$ to save the nodes of the next layer for the next iteration. d_{max} is used to restrict the maximum layers that the algorithm probes. In each loop of line 6, we pop a node y_b from $\mathcal{N}_{curr}$ and iterate through each of its neighbor z in G. We check if there exists a node copy z_c in G_P such that the time on z_c matches the threshold δ. If we do not find the copy and index z is not reachable, then we move on to the next neighbor index. If we do not find the copy and z is reachable, we create a new copy z_c of index z and connect it to the network G_P (line 12 - 20). If we find the copy z_c, then we calculate the waiting time t_{wait} using Definition 1. If $t_{wait} \geq \delta$, we will repeat lines 12 - 20 to create a new copy and connect it to the network G_P. Otherwise, no new copy will be created. We add the neighbor node copies probed in the current layer to $\mathcal{N}_{next}$ (line 20, 25). The graph update algorithm recursively probes a "deeper" neighborhood on one additional layer (line 4, 5, 26) until the maximum depth d_{max} is reached.

Note that when connecting the new copy z_c to the graph, we first loop over all the incoming edges of index z in G, and for each neighbor v such that there is an edge from v to z, we only connect the latest copy v_l of index v with z_c (line 15 - 16). We then loop over all the outgoing edges of index w in G, and for each neighbor w such that there is an edge from z to w, we only connect z_c with the earliest copy w_e of that index (line 18 - 19). By doing this, the number of edges between node copies is minimized.

The solution check step in Figure 6.1 is performed as follows. We traverse the solution path P. For each edge (x_a, y_b) on P, we calculate the waiting time t_{wait} on edge (x_a, y_b) using Definition 1. If $t_{wait} \geq \delta$, the solution check will return "yes", meaning the graph

Algorithm 9: Graph Update

Input: $G_P(V_P, E_P)$, current solution path P, maximum depth d_{max}
Output: Updated graph $G_P(V_P, E_P)$

```
1  initialize depth:= 0, N_curr := ∅, N_next := ∅
2  foreach node x_a ∈ P do
3      N_next := {x_a};
4      while depth < d_max do
5          N_curr := N_next;
6          foreach node y_b ∈ N_curr do
7              foreach neighbor index z of node y in G do
8                  find the neighbor of z_c with index z;
9                  if not found then
10                     if z is not reachable from y_b then continue;
11                     else
12                         create a new copy z_c;
13                         V_P := V_P ∪ z_c; E_P := E_P ∪ (y_b, z_c);
14                         foreach node index v ∈ V such that (v, z) ∈ E do
15                             v_l := latest copy that is reachable to z_c in G_P;
16                             E_P := E_P ∪ (v_l, z_c);
17                         foreach node index w ∈ V such that (z, w) ∈ E do
18                             w_e := earliest copy that is reachable from z_c in G_P;
19                             E_P := E_P ∪ (z_c, w_e);
20                             N_next := N_next ∪ w_e;
21                 else
22                     calculate wait time t_wait on edge (y_b, z_c);
23                     if t_wait ≥ δ then
24                         do line 12 - line 20;
25                     else add neighbors of y_b in G_P to N_next;
26         depth:=depth+1;
27 return G_P(V_P, E_P);
```

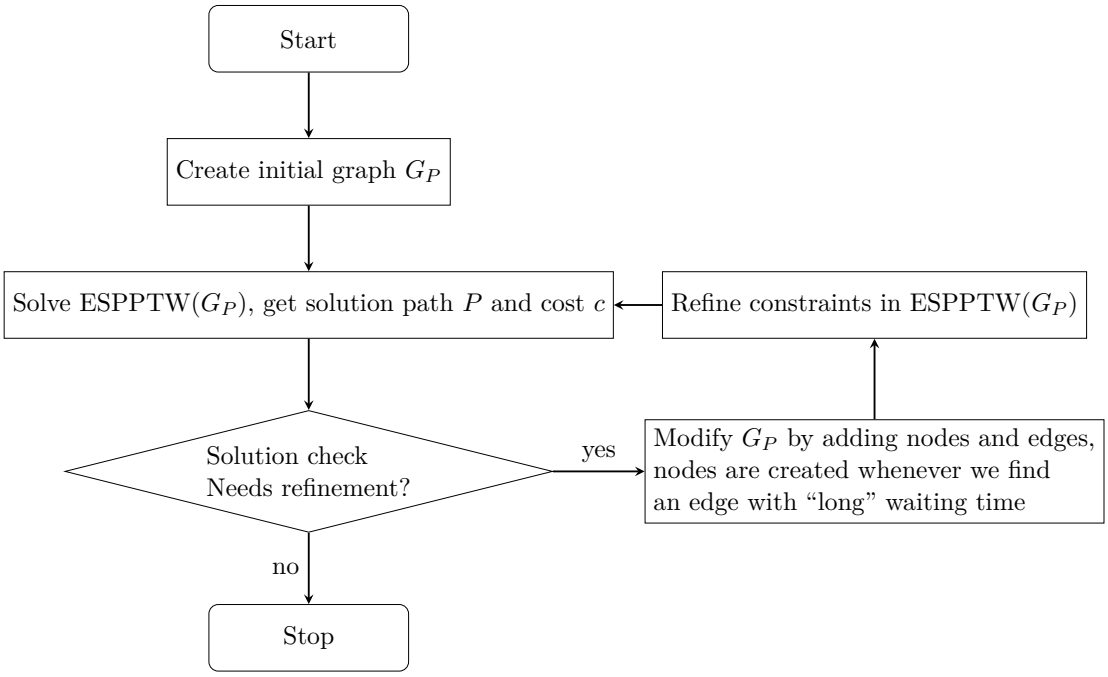


Figure 6.1: Flow Chart illustrating the ATWD based IP Approach

needs further refinement. When the algorithm terminates, the extra waiting time on each edge along the solution path will be smaller than δ.

Note that the solution will not be optimal even with respect to the threshold parameter δ unless we set $d_{max} \geq |V|$. Setting $d_{max} \geq |V|$ will let the graph update procedure exhaustively discretize the whole graph. We compare the performance of two different algorithms in the experiment section.

6.3 Experiments

6.3.1 ATWD based IP Solution

We generate a time-constrained graph G with negative cycles using Solomon's *0100_RC201*. We use the same test graph generation approach as introduced in Chapter 5. We randomly selected 10 pairs of source and destinations, and test the ATWD based IP approach on each of the pairs, using three different d_{max} values: $d_{max} = 1, 2, 3$ and six different thresholds: $\delta = 1, 2, 4, 10, 20, 30$. We also experiment with solving the discrete IP formulation on a fully time-expanded graph $G_T(V_T, E_T)$. The comparison is to demonstrate that the ATWD method is able to compute a good quality result in a more efficient way, without constructing the entire time-expanded network. We present the aggregated results in Table 6.1.

	IP with ATWD					IP with BFS Discretization			
δ	d_{max}	Precision	V_P	E_P	Time(s)	Precision	V_T	E_T	Time(s)
1	1	0.92	505	9149	1.589	1	6820	89263	24.835
	2	0.99	766	16545	6.186				
	3	1	861	18824	6.951				
2	1	0.89	480	8642	1.532	1	5101	68037	13.141
	2	0.99	733	15615	5.831				
	3	1	846	18149	6.814				
4	1	0.87	477	8585	1.476	0.99	3347	43648	9.112
	2	0.98	694	14362	5.294				
	3	0.99	800	16832	5.444				
10	1	0.76	445	8405	1.431	0.97	1551	22779	5.924
	2	0.95	650	13301	3.712				
	3	0.96	732	14177	4.891				
20	1	0.74	431	7201	1.224	0.91	910	14335	3.423
	2	0.87	508	9499	2.043				
	3	0.91	579	11281	2.878				
30	1	0.68	343	5186	0.713	0.79	690	9365	2.632
	2	0.73	412	7069	1.196				
	3	0.79	480	8681	2.218				

Table 6.1: Comparing ATWD based IP solution against solving discrete IP formulation with pre-constructing time-expanded network using breadth-first search, on graph generated from *0100_RC201*, with different δ and d_{max}.

As shown in Table 6.1, ATWD based IP solution solves the ESPPTW faster than ATWD-LCA-k-cycle algorithm. This can be observed by cross comparing this table with Table 5.1. The main reason is that the partially time-expanded graph G_P constructed using ATWD based IP approach is much smaller than the graph constructed by ATWD-LCA-k-cycle, since label correcting algorithm exhaustively discretizes the whole graph if l_{max} is not specified. Comparing the ATWD IP solution with solving the discrete IP formulation on fully time-expanded graph $G_T(V_T, E_T)$, ATWD based IP solution is able to find the optimal solution by constructing a partial of the entire time-expanded network. The difference is more noticeable when δ becomes smaller.

When $d_{max} = 1$, the algorithm cannot find the optimal solution. The best precision comes when $\delta = 1$ at 0.92. When $d_{max} = 2, \delta = 1$, the algorithm finds 9 optimal solutions out of 10, and the averaged precision is 0.99. When $d_{max} = 3$, the algorithm always finds the optimal solutions in 10 different tests, even when we set $\delta = 2$.

When $d_{max} = 1$, the algorithm will only probe the neighborhood of current solution 1 hop away. As we increase the value of d_{max}, the algorithm starts to probe further away from the current solution path. If the nodes on optimal solution are all probed, then the

resulting path found when algorithm terminates will be optimal. Hence as we increase d_{max}, the solution will likely to get better.

6.4 Conclusion

In this chapter, we present an ATWD based IP solution. Our solution is an iterative procedure, which refines the graph according to the current IP solution. Since the discrete version of the ESPPTW IP formulation is much easier to solve compared to the continuous version, the iterative approach runs faster than the continuous IP solution. Additionally, we apply the strategy of discretizing the "neighborhoods" of the current solution, which allows the optimal solution to be found while only the partial graph is discretized. We demonstrate that ATWD based IP solution constructs a much smaller graph than the label correcting algorithms, thereby it outperforms the ATWD-LCA-k-cycle algorithm (presented in Chapter 5) in our experiments. We believe that the iterative refinement approach can be combined with other IP formulations to solve similar time-constrained problems, such as dynamic network flow problems, vehicle routing problems.

Chapter 7

Conclusions

In this thesis, we present the Adaptive Time Window Discretization (ATWD) approach, which adopts the time-expanded network idea to construct a partially time-expanded network. ATWD method can be easily combined with existing label setting algorithms, label correcting algorithms to solve Shortest Path Problems with Time Windows (SPPTW). We demonstrate that existing solutions for handling negative cycles in the graph, including k-cycle elimination, IP formulation can also be combined with ATWD to solve Elementary Shortest Path Problem with Time Windows (ESPPTW). We experiment with different ATWD based algorithms and show that ATWD method is able to efficiently solve SPPTW with an acceptable precision. Proposed algorithms for solving ESPPTW on graphs with negative cycles, including ATWD based label correcting algorithm with k-cycle elimination, and ATWD based IP solution, have high scalability and can be applied on graphs with large sizes.

7.1 Applying ATWD on Graphs with Real Times

ATWD takes a threshold parameter δ which controls the granularity of the time-expanded network, and setting δ closer to 0 will generally gives a more optimized result. We prove that when the graph contains integral times, the ATWD based algorithms are able to find the optimal solution with $\delta = 1$. If the graph contains real times, however, the algorithms can still be used for generating good quality solutions with a minor change.

The discretization strategy that is used in this thesis is a "uniform discretization": the discrete time step remains the same between any adjacent node copies. While this strategy performs well on graphs with integral times, it is not very effective for graphs with real times. We present an "adaptive discretization" strategy in algorithm 10. Compared with

algorithm 2 introduced in Chapter 2, Section 2.4.3, the adaptive discretization strategy creates a new node copy if there does not exist any node copy within δ time later (line 4). If no such node copy is found, then we use the exact arrival time to create a new node copy. This strategy tries to assign exact arrival times as much as possible. It eliminates the overhead of time rounding up, which introduces extra waiting. Note that if the adaptive discretization strategy is used, the maximum number of node copies may exceed $\lceil (e_v - l_v)/\delta \rceil$.

Algorithm 10: GetCopyIndex(Adaptive)

Input: Node index v, arrival time t, copies set $\mathcal{C}$, threshold δ.
Output: Copy index a of the node copy

```
1 if t > l_v then a := −1;
2 else
3 |   if t ≤ e_v then t := e_v;
4 |   if find a node copy v_a ∈ C such that v_a.t − t ∈ [0, δ) then
5 |   |   return v_a.c
6 |   a := len(C);
7 return a;
```

7.2 SPPTW with Soft Time Windows

In this thesis, we focus on SPPTW/ESPPTW with hard time windows: late arrivals are not permitted on any node. Another problem definition is to allow late arrivals, with penalties. Time windows specified at each vertex of the time-constrained graph can referred to as soft time windows. The proposed ATWD based algorithms can be easily modified to solve SPPTW/ESPPTW on graphs with soft time windows. Specifically, two modifications need to be done when constructing the time-expanded graph. Firstly, whenever we add a node copy that is outside of the time window, we add the penalty to each edge connecting to that node; Secondly, with hard time windows, the algorithm only need to connect a node copy with at most one node copy for each neighbor index. With soft time windows, this strategy remains the same when we connect a node copy with another node copy that is within its time window. However, for node copies that are outside of its time window, we need to keep every incoming and outgoing edge if it is connecting to another "late" node copy. The reason is that different node copies that are outside of the time window represent different penalties, hence, we cannot do any optimization in this case.

7.3 Assigning Time Windows to Edges

In this thesis, we do not consider time windows on edges: each edge can be visited at any time. If the edge is assigned with a time window, then it is also needed to be considered as one of the time window constraints while solving the problem.

The ATWD based algorithms can be easily modified to solve SPPTW/ESPPTW on graphs with time windows on edges. Whenever the algorithm explores an edge (v, w) in the time-constrained graph G, we verify if the time window on edge allows one to reach to the neighbor node. If the travel time exceeds the boundary of the time window, then the node w will not be discretized.

7.4 Future Work

For SPPTW, ATWD provides the benefit of a fully time-expanded network without constructing the entire graph. We believe the ATWD method can also be embedded in existing algorithms for solving other problems, such as dynamic network flow problems, vehicle routing problems with time windows (VRPTW). A future research direction of this thesis is to investigate an ATWD based IP solution for VRPTW, where a similar refine-and-solve process can be applied. Another research direction is to embed the ATWD method into dynamic programming algorithms or IP formulations for the dynamic network flow problems.

Bibliography

[1] Michael J Bannister and David Eppstein. Randomized speedup of the bellman–ford algorithm. In *2012 Proceedings of the Ninth Workshop on Analytic Algorithmics and Combinatorics (ANALCO)*, pages 41–47. SIAM, 2012.

[2] John E Beasley and Nicos Christofides. An algorithm for the resource constrained shortest path problem. *Networks*, 19(4):379–394, 1989.

[3] Richard Bellman. On a routing problem. *Quarterly of applied mathematics*, 16(1):87–90, 1958.

[4] Natashia Boland, John Dethridge, and Irina Dumitrescu. Accelerated label setting algorithms for the elementary resource constrained shortest path problem. *Operations Research Letters*, 34(1):58–68, 2006.

[5] Natashia Boland, Mike Hewitt, Duc Minh Vu, and Martin Savelsbergh. Solving the traveling salesman problem with time windows through dynamically generated time-expanded networks. In *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 254–262. Springer, 2017.

[6] Guy Desaulniers, Jacques Desrosiers, Marius M Solomon, François Soumis, and Daniel Villeneuve. A unified framework for deterministic time constrained vehicle routing and crew scheduling problems. In *Fleet management and logistics*, pages 57–93. Springer, 1998.

[7] Guy Desaulniers and Daniel Villeneuve. The shortest path problem with time windows and linear waiting costs. *Transportation Science*, 34(3):312–319, 2000.

[8] Martin Desrochers, Jacques Desrosiers, and Marius Solomon. A new optimization algorithm for the vehicle routing problem with time windows. *Operations Research*, 40(2):342–354, 1992.

[9] Martin Desrochers and François Soumis. A generalized permanent labelling algorithm for the shortest path problem with time windows. *INFOR: Information Systems and Operational Research*, 26(3):191–212, 1988.

[10] Jacques Desrosiers, Yvan Dumas, Marius M Solomon, and François Soumis. Time constrained routing and scheduling. *Handbooks in operations research and management science*, 8:35–139, 1995.

[11] Jacques Desrosiers, François Soumis, and Martin Desrochers. Routing with time windows by column generation. *Networks*, 14(4):545–565, 1984.

[12] Jacques Desrosiers, Francois Soumis, Martin Desrochers, and Michel SauveGerad. Methods for routing with time windows. *European Journal of Operational Research*, 23(2):236–245, 1986.

[13] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[14] Michael Drexl and Stefan Irnich. Solving elementary shortest-path problems as mixed-integer programs. *OR spectrum*, 36(2):281–296, 2014.

[15] Michael Drexl and Eric Prescott-Gagnon. Labelling algorithms for the elementary shortest path problem with resource constraints considering eu drivers' rules. *Logistics Research*, 2(2):79–96, 2010.

[16] Irina Dumitrescu and Natashia Boland. Improved preprocessing, labeling and scaling algorithms for the weight-constrained shortest path problem. *Networks: An International Journal*, 42(3):135–153, 2003.

[17] Québec). Département de génie industriel École polytechnique (Montréal and DJ Houck. *Traveling Salesman Problem as a Constrained Shortest Path Problem: Theory and Computational Experience*. Ecole polytechnique de Montréal, 1978.

[18] Faramroze G Engineer, George L Nemhauser, and Martin WP Savelsbergh. Dynamic programming-based column generation on time-expanded networks: Application to the dial-a-flight problem. *INFORMS Journal on Computing*, 23(1):105–119, 2011.

[19] Dominique Feillet, Pierre Dejax, Michel Gendreau, and Cyrille Gueguen. An exact algorithm for the elementary shortest path problem with resource constraints: Application to some vehicle routing problems. *Networks: An International Journal*, 44(3):216–229, 2004.

[20] Mirko Ferrati and Lucia Pallottino. A time expanded network based algorithm for safe and efficient distributed multi-agent coordination. In *52nd IEEE Conference on Decision and Control*, pages 2805–2810. IEEE, 2013.

[21] Frank Fischer and Christoph Helmberg. Dynamic graph generation for the shortest path problem in time expanded networks. *Mathematical Programming*, 143(1-2):257–297, 2014.

[22] Lisa Fleischer and Martin Skutella. The quickest multicommodity flow problem. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 36–53. Springer, 2002.

[23] Robert W Floyd. Algorithm 97: shortest path. *Communications of the ACM*, 5(6):345, 1962.

[24] Lester R Ford Jr. Network flow theory. Technical report, Rand Corp Santa Monica Ca, 1956.

[25] Lester R Ford Jr and Delbert R Fulkerson. *Flows in networks*. Princeton university press, 2015.

[26] Michael L Fredman and Robert E Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[27] William J Guerrero, Nubia Velasco, Caroline Prodhon, and Ciro-Alberto Amaya. On the generalized elementary shortest path problem: A heuristic approach. *Electronic Notes in Discrete Mathematics*, 41:503–510, 2013.

[28] Gabriel Y Handler and Israel Zang. A dual algorithm for the constrained shortest path problem. *Networks*, 10(4):293–309, 1980.

[29] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[30] Koki Ho, Olivier L De Weck, Jeffrey A Hoffman, and Robert Shishko. Dynamic modeling and optimization for space logistics using time-expanded networks. *Acta Astronautica*, 105(2):428–443, 2014.

[31] Toshihide Ibaraki, Shinji Imahori, Mikio Kubo, Tomoyasu Masuda, Takeaki Uno, and Mutsunori Yagiura. Effective local search algorithms for routing and scheduling problems with general time-window constraints. *Transportation science*, 39(2):206–232, 2005.

[32] Stefan Irnich and Guy Desaulniers. Shortest path problems with resource constraints. In *Column generation*, pages 33–65. Springer, 2005.

[33] Stefan Irnich and Daniel Villeneuve. The shortest-path problem with resource constraints and k-cycle elimination for k $\geq$ 3. *INFORMS Journal on Computing*, 18(3):391–406, 2006.

[34] Donald B Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal of the ACM (JACM)*, 24(1):1–13, 1977.

[35] Ekkehard Köhler, Katharina Langkau, and Martin Skutella. Time-expanded graphs for flow-dependent transit times. In *European Symposium on Algorithms*, pages 599–611. Springer, 2002.

[36] Antoon WJ Kolen, AHG Rinnooy Kan, and Harry WJM Trienekens. Vehicle routing with time windows. *Operations Research*, 35(2):266–273, 1987.

[37] Leonardo Lozano, Daniel Duque, and Andrés L Medaglia. An exact algorithm for the elementary shortest path problem with resource constraints. *Transportation Science*, 50(1):348–357, 2016.

[38] Stefano Pallottino and Maria Grazia Scutella. Shortest path algorithms in transportation models: classical and innovative aspects. In *Equilibrium and advanced transportation modelling*, pages 245–281. Springer, 1998.

[39] Warren B Powell and Zhi-Long Chen. A generalized threshold algorithm for the shortest path problem with time windows. In *Network Design: Connectivity and Facilities Location*, pages 303–318, 1997.

[40] Luigi Di Puglia Pugliese and Francesca Guerriero. Shortest path problem with forbidden paths: The elementary version. *European Journal of Operational Research*, 227(2):254–267, 2013.

[41] Giovanni Righini and Matteo Salani. Symmetry helps: Bounded bi-directional dynamic programming for the elementary shortest path problem with resource constraints. *Discrete Optimization*, 3(3):255–273, 2006.

[42] Giovanni Righini and Matteo Salani. New dynamic programming algorithms for the resource constrained elementary shortest path problem. *Networks: An International Journal*, 51(3):155–170, 2008.

[43] Alfonso Shimbel. Structure in communication nets. In *Proceedings of the symposium on information networks*, pages 119–203. Polytechnic Institute of Brooklyn, 1954.

[44] Matthew R Silver and Olivier L De Weck. Time-expanded decision networks: A framework for designing evolvable complex systems. *Systems Engineering*, 10(2):167–188, 2007.

[45] Solomon et al. Solomon benchmarking problems. `http://w.cba.neu.edu/~msolomon/problems.htm`. Accessed: 2020-11-27.

[46] Marius M Solomon. Algorithms for the vehicle routing and scheduling problems with time window constraints. *Operations research*, 35(2):254–265, 1987.

[47] Marius M Solomon and Jacques Desrosiers. Survey paper—time window constrained routing and scheduling problems. *Transportation science*, 22(1):1–13, 1988.

[48] Leonardo Taccari. Integer programming formulations for the elementary shortest path problem. *European Journal of Operational Research*, 252(1):122–130, 2016.

[49] Ignacio Vitale and Rodolfo Dondo. On alternative formulations to the shortest path problem with time windows and capacity constraints. In *I Simposio Argentino de Informática Industrial e Investigación Operativa (SIIIO 2019)-JAIIO 48 (Salta)*, 2019.

[50] Duc Minh Vu, Mike Hewitt, Natashia Boland, and Martin Savelsbergh. Dynamic discretization discovery for solving the time-dependent traveling salesman problem with time windows. *Transportation Science*, 54(3):703–720, 2020.