Continuous Shortest Path Problems with Time Window Constraints

by

Chia-We Chang

B.Sc., Mathematics and Computer Science, Simon Fraser University, 1994

A THESIS SUBMITTED IN PARTIAL FULFILLMENT

OF THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF SCIENCE

in the Department

of

Computing Science

SIMON FRASER UNIVERSITY

August 14, 1997

APPROVAL

Name: Chia-We Chang

Degree: Master of Science

Title of thesis: Continuous Shortest Path Problems with Time Window Constraints

Examining Committee: Dr. Tiko Kameda
Chair

Dr. Arvind Gupta
Senior Supervisor

Dr. Ramesh Krishnamurti
Senior Supervisor

Dr. Perry Fizzano
External Examiner

Date Approved: August 14 1997

Abstract

The shortest path problem with time window constraints and costs (SPW-Cost) consists of finding a least cost route between a source and a sink in a network $G = (N, A)$ such that a vehicle visits each node within their specified time windows $[a_i, b_i]$. Each arc $(i, j) \in A$ has a positive duration d_{ij} and an unrestrictive cost c_{ij}.

This problem has appeared as a sub-problem of many vehicle routing and scheduling problems, most of which are known to be NP-hard.

In this thesis, we will study a variant of SPW-Cost called Continuous Shortest Path Problem with Time Window Constraints (Continuous-SPW). Unlike SPW-Cost where a vehicle is allowed to wait at a node for a time window to open, in Continuous-SPW the vehicle must move continuously in the network only passing through the nodes whose time windows are open.

We will determine the complexity of this and other versions of Continuous-SPW for restricted classes of graphs that are of practical interest. Our goal is to construct sequential algorithms and determine their running time complexities. We will also provide a parallel algorithm for the general Continuous-SPW and show how these results can be extended to handle SPW-Cost problems.

Acknowledgments

I would like to express my eternal gratitude to my supervisors, Professor Arvind Gupta and Professor Ramesh Krishnamurti, for their support, motivation, and patience guiding me through my research. Furthermore, I am very grateful for their contributions to Chapter 3 of this thesis.

I would also like to extend my gratitude to Erick Wong for his contribution to Chapter 4 of this thesis, and Tamara Dakic for many helpful discussions.

Dedication

To my parents who have been always supportive of my work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Let us consider an Automated Vehicle Guidance(AVG) system in a manufacturing plant in which there are several vehicles that service a set of stations interconnected by a set of lanes. A major problem in such a system is determining a conflict-free path for a vehicle dispatched from a source station to arrive at the destination station as early as possible without disrupting other active travel schedules [CT91].

This is an elaborate example of a problem known as *Shortest Path Problem with Time Window Constraints (SPW)* . It consists of finding a shortest path between a source and a sink in a network $G = (N, A)$ while respecting specified time windows $[a_i, b_i]$ at each visited node. Each arc $(i, j) \in A$ has a positive duration d_{ij}.

This problem appeared as a sub-problem of a vehicle routing problem studied in [GK95] where they provide an efficient sequential and parallel algorithm to solve it exactly.

When there is a cost c_{ij}, possibly negative, associated with each arc $(i, j) \in A$ and the objective is to minimize the cost of the route, the problem is known as *Shortest Path Problem with Time Windows and Cost (SPW-Cost)* .

SPW-Cost first appeared as a sub-problem in the construction of school bus routes, where the number of routes needed to complete all tasks must be found while minimizing total costs [DSD84]. It also appeared as a sub-problem in the time constrained vehicle routing problem with capacity constraints on vehicles, where a set of minimum cost routes, originating and terminating at a central depot must be determined [Sor86]. Here, vehicles servicing the nodes have a capacity which cannot be exceeded.

A common characteristic of all these practical vehicle routing and scheduling problems, including SPW-Cost, is that they are **NP**-hard and therefore there are no known algorithms to solve them exactly and efficiently in polynomial time. Solution metholologies currently capable of solving problems of realistic size range from simple heuristics, to optimization-based heuristics, to optimization methods [DDSS93]. Because of their practical importance, finding efficient solutions to SPW-Cost is of paramount importance.

In this thesis, we will study variants of SPW-Cost called *Continuous Shortest Path Problem with Time Window Constraints (Continuous-SPW)* . Unlike SPW-Cost where a vehicle is allowed to wait at a node until the time window opens, in Continuous-SPW the vehicle must continuously travel visiting nodes strictly within their specified time windows. This problem has applications in the area of networking where a continuous stream of data transfers between nodes is required, such as video data, and where intermediate nodes are open only at specific time intervals.

We will be looking at the complexity of this and other versions of Continuous-SPW under restricted classes of graphs. We will construct sequential algorithms for Continuous-SPW under different requirements. We will then provide a parallel algorithm for the general Continuous-SPW. Finally, we will show how these results can be extended to handle SPW-Cost.

Organization of this Thesis

Chapter 2 introduces some definitions and background materials that are needed in later chapters. Chapter 3 discusses the complexity and difficulty of Continuous-SPW under general and restrictive settings. Chapter 4 contains sequential algorithms for Continuous-SPW under different requirements. Chapter 5 contains a parallel algorithm for Continuous-SPW. Chapter 6 explores some relaxations of Continuous-SPW requirements, providing insight into the difficulty of the problem. Chapter 7 describes extension of solutions to Continuous-SPW to solve SPW-Cost. Chapter 8 presents a summary of all the results and a list of related open problems.

Chapter 2

Background

In this chapter, we will introduce basic definitions and concepts used throughout the thesis. At the end of each section, references are given where a more complete treatment of the subject can be found.

2.1 Graph Definitions

A *graph* G is an ordered pair (V_G, E_G) where V_G is a nonempty set of vertices $\{v_1, v_2, \ldots, v_n\}$ (sometimes called nodes) and E_G is a subset of edges $\{v_i, v_j\}$ where v_i, v_j are vertices in V. When there is no ambiguity as to which graph G we refer to, the subscripts G in V_G, E_G are dropped. Graphs are generally drawn with labeled points representing the vertices and lines joining pair of points representing the edges.

A *loop* is an edge of the form $\{v, v\}$.

A vertex v_i is *adjacent* to vertex v_j whenever $\{v_i, v_j\}$ is in E. If a vertex v has d adjacent vertices, then the *degree* of v is d. Vertices which have loops are counted twice in the degree.

A *path* is a sequence of vertices $v_1, v_2, ..., v_k$ such that for $i = 1, ..., k-1$, $\{v_i, v_{i+1}\} \in E$. The *length* of this path is $k-1$. If the vertices $v_1, v_2, ..., v_k$ appear uniquely in the sequence, then the path is *simple*.

We say that the *distance* between two vertices u, v is k if the shortest path from u to v has length k.

A *cycle* is a path $v_1, v_2, ..., v_k$ where $v_1 = v_k$. The integer $k-1$ is the length of this cycle. If all vertices $v_1, v_2, ..., v_{k-1}$ appear uniquely, then the cycle is simple.

A *Hamilton* cycle is a simple cycle containing all vertices of the graph. A graph which has a Hamilton cycle is *Hamiltonian*.

A graph is *planar* if it can be *embedded* (*ie.* drawn) on a plane such that the edges intersect only at their endpoints.

A graph $G = (V, E)$ is *bipartite* if the set of vertices V can be partitioned into disjoint sets S_1, S_2 such that no pair of vertices in the same set are adjacent.

A graph with no cycles is a *tree*. Vertices of degree 1 in a tree are called *leafs*.

A *directed* graph $D = (V, A)$ is a graph where the edges are directed; that is, A is a subset of arcs (v_i, v_j) in $V \times V$; sometimes the word *digraph* is used instead of directed graph.

A directed path $v_1, v_2, ..., v_k$ in a directed graph $D = (V, A)$ is a path such that for $i = 1, ..., k-1$, $(v_i, v_{i+1}) \in A$.

A directed cycle $v_1, v_2, ..., v_k$ is a cycle such that arc $(v_i, v_{i+1}), i = 1, ..., k-1$ is in the arc set.

A directed graph with no directed cycles is *acyclic*.

A *network* is a directed graph with two distinguished vertices called *source* and *sink*.

For more details on graph theory, see [BM76].

2.2 Graph Algorithms

One of the most fundamental techniques that forms the basis of many other graph algorithms is graph searching or traversal. The most basic graph searching is *Breadth-First Search (BFS)*.

Starting at a vertex u in the graph, the order in which BFS visits the vertices is as follows: vertex u is visited first, then the vertices adjacent to u that have not been visited before are visited, then the vertices adjacent to those vertices are visited, and so forth. BFS essentially expands its frontier of visited vertices by visiting all the vertices at distance k from u before visiting any vertices at distance $k+1$ from u. The vertices that are at distance k from vertex u are said to be at *level k of the breadth-first search*.

Algorithm 1: Breadth-First Search.
Input: Graph $G = (V, E)$, vertex u in V.
Output: Sequence of visited vertices.

```
(1)  Set Q ← {u}.
(2)  while Q ≠ o
(3)     v ← dequeue Q.
(4)     Visit vertex v.
(5)     foreach vertex w adjacent to v
(6)        if w has not been visited
(7)           Q ← Q + {w}.
```

For more details see [CLR90].

2.3 Models of Computation

In order to analyse and compare algorithms, we need to look at the underlying model of computation used to solve the problem. Although there are many models of computation, we will only look at models that are of interest to us in this thesis.

For sequential algorithms, we will use the *Random-Access Machine (RAM)* model. Here, we have a single stream of instructions operating on a single stream of data executed sequentially.

For parallel algorithms, we will use the *Parallel Random-Access Machine (PRAM)* model. Here, we have Shared-Memory Single Instruction stream, Multiple Data stream (SIMD) Computers that have a number of identical processors, each with its own local memory operating under a single instruction. The processors communicate through a shared common memory where memory writes can be either concurrent writes or exclusive writes (CW,EW), and memory reads can be either concurrent reads or exclusive reads (CR,ER). In the exclusive reads (write) policy, only one processor can read (write) to a memory location at any given time, whereas in the concurrent read (write) policy, more than one processor can read (write) to a memory location simultaneously.For more details and examples, see [Akl89]

The overwhelming majority of computers today adhere to these model.

2.4 Complexity Classes

Many problems can be categorized into different complexity classes. Two such classes that we are interested in are: the class $\boldsymbol{P}$ and the class $\boldsymbol{NP}$.

Problems in **P** are those for which there is an algorithm that makes deterministic steps and solves the problem in polynomial time in the input length. For example, the problem of finding the shortest path between two vertices is in **P**. Given a graph $G = (V, E)$, length $l(\epsilon) \in \mathbb{Z}^+$ for each $\epsilon \in E$, vertices $a, b \in V$, and a positive integer B, the problem is to determine whether there is a simple path from a to b in G having total length at most B. A deterministic algorithm for this problem is

Dijkstra's Shortest Path Algorithm (see [CLR90]).

Problems in **NP** are decision problems for which there is an algorithm that makes nondeterministic steps and outputs yes in polynomial time in the input length when the problem instance has a yes solution. For example, the problem of finding the longest path between two vertices is in **NP**. Given a graph $G = (V, E)$, length $l(e) \in \mathbb{Z}^+$ for each $e \in E$, vertices $a, b \in V$, and a positive integer B, the problem is to determine whether there is a simple path from a to b in G having total length at least B. A nondeterministic algorithm can generate a sequence of vertices nondeterministically and then check if this sequence is a simple path of length at least B.

We say that a problem π_1 *reduces* to problem π_2 when there exists a transformation that maps any instance of problem π_1 to an equivalent instance of problem π_2. If all problems in **NP** can be reduced to a problem π in polynomial time, then we say that problem π is ***NP-hard***. If, in addition, problem π is in **NP**, then it is a ***NP-complete*** problem. The *Hamilton Cycle Problem* is an example of an **NP**-complete problem. Given a graph G, the problem is to determine whether there is a Hamilton Cycle in G.

Because of the nondeterministic property, it is believed that far more problems can be solved with nondeterministic algorithms in polynomial time than with deterministic algorithms. However, to date, it is still unknown whether there exists a problem which is solvable by a nondeterministic algorithm in polynomial time, but not by a deterministic algorithm in polynomial time.

Presently, all **NP**-hard problems can only be solved by deterministic algorithms in exponential time. So when we show that a problem is **NP**-hard, we are providing strong evidence that it is a very hard problem; one that is unlikely to be solved in

polynomial time by any deterministic algorithm. For more details, see [GJ79]

2.5 Linear Diophantine Equation

Linear equations of the form $ax + by = c$ with integer coefficients a, b, integer value c, and integer variables x, y are called *Linear Diophantine Equations*. We shall see next how this linear equation can be solved for the variables x, y.

Given integers a, b, we define the *greatest common divisor (gcd)* of a, b to be the greatest integer d such that d divides a and d divides b. We use the notation (a, b) to mean $\gcd(a, b)$, and $d \mid a$ to mean d divides a.

Observation 2.5.1. For integers a, b, d, if $d \mid ab$ and $(d, a) = 1$ then $d \mid b$.

Observation 2.5.2. For integers a, b, c, d, if $(a, b) = d$ and $c \mid a$, $c \mid b$ then $c \mid d$.

Observation 2.5.3. For integers a, b, c, if $a \mid c$, $b \mid c$ and $(a, b) = 1$ then $ab \mid c$.

Observation 2.5.4. For integers a, b, c, if $c \mid a$, $c \mid b$ then $c \mid (a + b)$

For integers a, b, $\gcd(a, b)$ can be computed in polynomial time using an algorithm known as the *Euclidean Algorithm*. Assuming that $a \leq b$, by the division algorithm we can find integers q_1, r_1 such that

$$a = q_1 b + r_1, \quad 0 < r_1 < b$$

By observation 2.5.4 we see that $(a, b) = (b, r_1)$. Thus by recursively computing (b, r_1) for k steps with $b > r_1 > \ldots > r_k$ until $r_{k-1} = r_k q_k$, we have $(a, b) = (b, r_1) = \ldots = (r_{k-1}, r_k) = r_k$.

Since at each step computation of (b, r) from $a = qb + r$, either $b \leq a/2$ or else $b > a/2$ and $r = a - bq = a - b < a/2$, we have $(b, r) \leq a/2$: that is, the bound on the gcd is reduced by at least half. Hence, $\gcd(a, b)$ can be determined in $O(\log b)$ steps.

Observation 2.5.5. From the above algorithm, we can obtain integers s, t such that $at + bs = (a, b)$. This is computed by an algorithm known as the *Extended Euclidean Algorithm*.

Lemma 2.5.1. Equation $ax + by = c$ has integer solutions x, y if and only if $(a, b) \mid c$. Furthermore, if x_0, y_0 is a solution, then all integer solutions x, y can be expressed as

$$x = x_0 - \frac{b}{(a,b)}t, \quad y = y_0 + \frac{a}{(a,b)}t, \qquad t \text{ integer}$$

Proof. If x_0, y_0 is a solution, then $(a, b) \mid ax_0$, $(a, b) \mid by_0$ and thus by Observation 2.5.4 $(a, b) \mid c$.

Conversely, if $(a, b) \mid c$, then $c = m(a, b)$ for some integer m. From Observation 2.5.5, there are integers r, s such that $ar + bs = (a, b)$; hence, $x = mr$, $y = ms$ is a solution to the equation $ax + by = c$.

Now, let us suppose x, y is any solution to the equation. Then

$$0 = c - c = a(x - x_0) + b(y - y_0) \tag{2.1}$$

Since $a \mid a(x - x_0)$ and $a \mid 0$, we have $a \mid b(y - y_0)$. Observation 2.5.1 together with $(\frac{a}{(a,b)}, \frac{b}{(a,b)}) = 1$ implies that for some integer t,

$$\frac{a}{(a,b)}t = y - y_0 \quad , \quad y = y_0 + \frac{a}{(a,b)}t \tag{2.2}$$

Substituting 2.2 into Equation 2.1 we get

$$0 = a(x - x_0) + \frac{ab}{(a,b)}t \quad , \quad x = x_0 - \frac{b}{(a,b)}t$$

□

As a generalization, we define the gcd of integers $a_1, a_2, ..., a_n$ to be the gcd of a_1 and the gcd of $a_2, ..., a_n$; that is, $\gcd(a_1, a_2, ..., a_n) = \gcd(a_1, \gcd(a_2, ..., a_n))$.

Finally, we define the *least common multiple (lcm)* of integers a, b to be the smallest integer m such that $a \mid m$ and $b \mid m$. We use $\{a, b\}$ to denote $\text{lcm}(a, b)$.

Lemma 2.5.2. For integers a, b, $ab = lcm(a, b) \cdot gcd(a, b)$.

A more complete and in-depth treatment of linear Diophantine equations with applications in other areas can be found in [Sch86].

2.6 Theory of Congruences

For integers a, b, m, we say that a is *congruent* to b modulo m (denoted by $a \equiv b \pmod{m}$) whenever $m \mid (a - b)$.

Observation 2.6.1. For integers a, b, c, and d

1. If $a \equiv b \pmod{m}$ and $b \equiv c \pmod{m}$, then $a \equiv c \pmod{m}$
2. If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$, then $a + c \equiv b + d \pmod{m}$.
3. If $a \equiv b \pmod{m}$ and $c \equiv d \pmod{m}$, then $ac \equiv bd \pmod{m}$.
4. If $ac \equiv bc \pmod{m}$ and $(c, m) = d$, then $a \equiv b \pmod{m/d}$.

We can see that congruence acts like equality in many ways.

For linear congruence of the form $ax \equiv b \pmod{m}$ with integer values a, b, m and integer variable x, a solution in x exists if and only if there are integers x, k such that $ax = b + km$. Hence, the problem of solving this equation is essentially the same as that of solving linear Diophantine equations. Thus,

Observation 2.6.2. The linear congruence $ax \equiv b \pmod{m}$ has a solution in x if and only if $(a, m) \mid b$.

Theorem 2.6.1. The Chinese Remainder Theorem. The system of congruences

$$x \equiv a_i \pmod{m_i}, \qquad i = 1, 2, \ldots, k, \tag{2.3}$$

where $(m_i, m_j) = 1$ if $i \neq j$, has a unique solution for x modulo $m_1 m_2 \cdots m_k$.

Proof. We show by induction on k that system (2.3) has a solution.

The result is obvious when $k = 1$. Let us consider the case $k = 2$. If $x \equiv a_1 \pmod{m_1}$, then all solutions for x are of the form $x = a_1 + k_1 m_1$ for an integer k_1. If in addition $x \equiv a_2 \pmod{m_2}$, then

$$a_1 + k_1 m_1 \equiv a_2 \pmod{m_2} \qquad \text{for every } k_1,$$

or

$$k_1 m_1 \equiv a_2 - a_1 \pmod{m_2}$$

Because $(m_1, m_2) = 1$, we know from Observation 2.6.2 that this congruence, with k_1 as the unknown, has a unique solution s modulo m_2. Thus $k_1 = s + k_2 m_2$ for some k_2 and

$$x = a_1 + (s + k_2 m_2) m_1 \equiv a_1 + s m_1 \pmod{m_1 m_2}$$

satisfies both congruences.

Now, suppose that system (2.3) has a solution $\pmod{m_1 m_2 \cdots m_{k-1}}$. Then there is a solution s to the system

$$x \equiv a_i \pmod{m_i} \qquad i = 1, 2, \ldots, k-1.$$

But the system

$$x \equiv s \pmod{m_1 m_2 \cdots m_{k-1}}$$
$$x \equiv a_k \pmod{m_k}$$

has a solution modulo $m_1 m_2 \cdots m_{k-1} m_k$, just as in the case $k = 2$, since we have $\gcd(m_1 m_2 \cdots m_{k-1}, m_k) = 1$.

Moreover the solution is unique. If r and s are both solutions of the system, then

$$r \equiv s \equiv a_i \pmod{m_i} \qquad i = 1, 2, \ldots, k,$$

so $m_i \mid (r - s)$, $i = 1, 2, \ldots, k$. Because the moduli are pairwise relatively prime, by observation 2.5.3 we have $m_1 m_2 \cdots m_k \mid (r - s)$. But

$$-m_1 m_2 \cdots m_k < r - s < m_1 m_2 \cdots m_k$$

whence $r - s = 0$. □

An efficient algorithm to solve system (2.3) can be found in [AHU74]. For further details, see [AHU74, CLR90, Dud78].

Chapter 3

Complexity of Continuous-SPW

In this chapter, we will show that an efficient algorithm to solve Continuous-SPW is unlikely to exist. We will show that Continuous-SPW is **NP**-hard by providing a polynomial time reduction algorithm from a known **NP**-complete problem to the Continuous-SPW. We will show that Continuous-SPW remains **NP**-hard even when it is restricted to simple classes of graphs such as bipartite graphs, planar graphs, grid graphs and monotone grid graphs. The incentives for studying these classes of graphs come from these applications: in modeling Continuous-SPW for packet routing over a network, many of the network connections topology, such as the star topology, the n-dimensional cube topology, and the tree topology can be represented as bipartite graphs; in modeling pick-up service for trucks in a city, the roads can be represented as grid graphs; and in modeling delivery of components over coveyor belt system in a factory, the system of belts can be represented as monotone grid graphs.

For the remaining part of this thesis, let us use the notation $G = (V, A, D, T, a, b)$ for an instance of the Continuous-SPW, with a network consisting of a vertex set $V = \{v_1, v_2, ..., v_n\}$, an arc set $A = \{e_1, e_2, ..., e_m\}$, vertices a, b in V for which we want

to compute the shortest path between them, and a set of distances $D = \{d_1, d_2, ..., d_m\}$ associated with the arcs in A, and a set of time windows $T = \{[a_1, b_1]_1, [a_2, b_2]_2, ..., [a_n, b_n]_n\}$ associated with the vertices in V (the a_i's are the release times, and the b_i's are the deadlines).

Theorem 3.0.2. Continuous-SPW is **NP**-hard.

Proof. Our reduction is from the Hamilton Cycle Problem. We recall that the Hamilton Cycle Problem is an **NP**-complete problem that asks if there exists a Hamilton Cycle in a given graph.

Let $H = (N, E)$ be a graph instance of the Hamilton Cycle Problem with vertex set $N = \{v_0, ..., v_{n-1}\}$ and edge set $E = \{e_0, ..., e_{m-1}\}$. Let a be any vertex in H. We construct an instance $G = (V, A, D, T, a, b)$ of the Continuous-SPW as follows. The vertex set V consists of the vertex a, a new vertex $v_n = b$, and $n - 1$ copies $N_1, N_2, ..., N_{n-1}$ of the vertex set N. The arc set A consists of arcs $(a, v), v \in N_1$ such that $\{a, v\}$ is in E, of arcs $(u, v_n), u \in N_{n-1}$ such that $\{u, a\}$ is in E, and of arcs $(x, y), x \in N_i, y \in N_{i+1}, i = 1, ..., n-2$ such that $\{x, y\}$ is in E. We assign the distance of each arc $(x, y) \in A$ to 2^y. We assign the time window of each vertex $v \in N_i, i = 1, ..., n-1$, and of vertex a, to $[0, 2^n]$, while we assign the time window of vertex v_n to $[2^{n+1} - 1, 2^{n+1} - 1]$. For an example, see Figure 3.1.

We can easily see that the above construction takes $O(nm + n^2)$ time since there are $(n-1)n + 2$ vertices in V and at most $(n-2)m + 2n$ arcs in A.

The purpose of this construction is to correspond a Hamilton Cycle in H to a feasible (a, b)-path of length n in G such that all vertices of N appear uniquely.

Let us suppose H has a Hamilton Cycle $C = v_{i_1}, ..., v_{i_n}, v_{i_1}$, and vertex a is some v_{i_j} in C. Then we claim that the path $a, v_{i_{j+1}}, v_{i_{j+2}}, ..., v_{i_{j-1}}, b$ is a feasible (a, b)-path in G. To see this, we notice that each vertex $v_i \in N_j, i = 0, ..., n-1, j =$

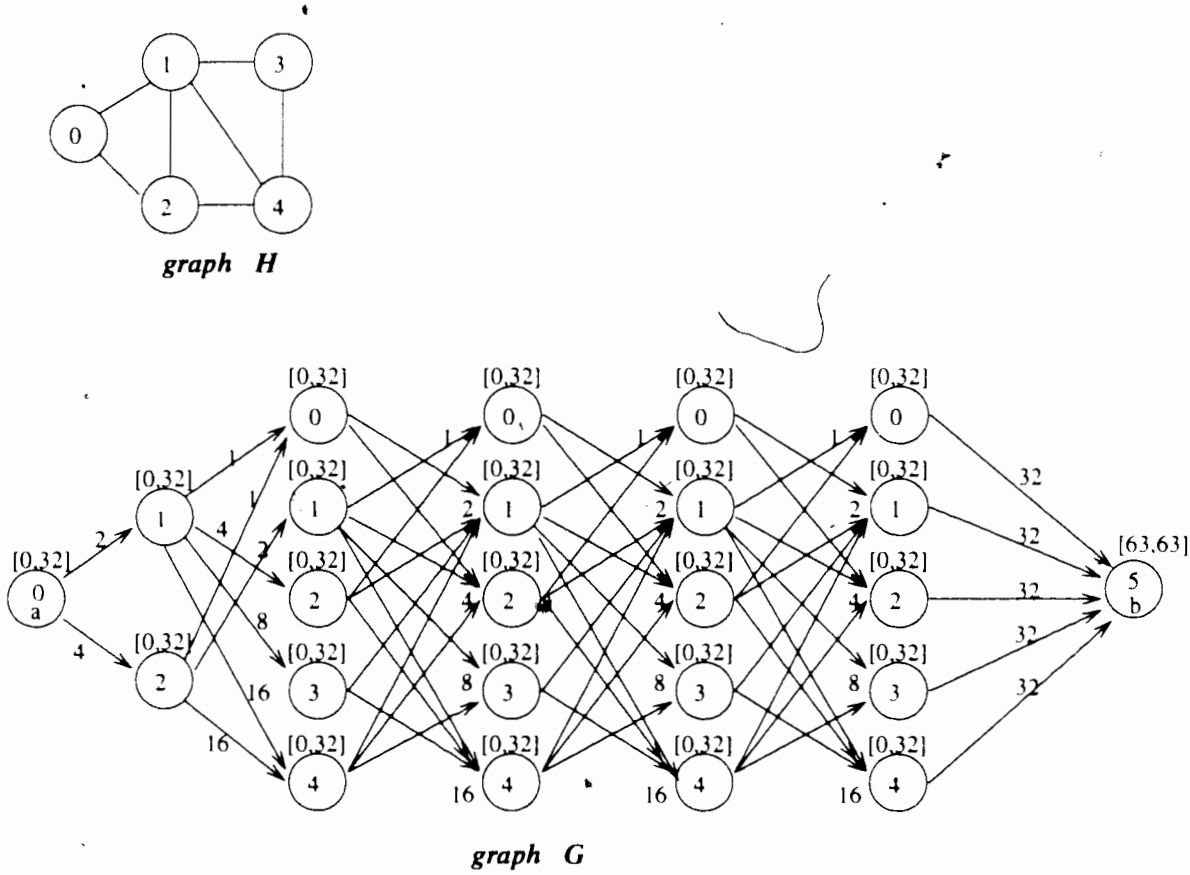


Figure 3.1: Construction of a Continuous-SPW Instance from a Hamilton Cycle Problem Instance

$1, \ldots, n-1$ has essentially an unbounded time window; thus, all the (a, v_{i_k})-subpaths, $k = j+1, \ldots, n, j-1$, are feasible paths in G. Since each vertex v_i appears uniquely in the cycle C, and the distance of arcs into v_i is 2^i, the total distance along the (a, b)-path must be $2^0 + 2^1 + \cdots + 2^n$, or $2^{n+1} - 1$, which is within b's time window. Hence, it is a feasible path in G.

Conversely, let us suppose G has a feasible (a, b)-path $P = a, v_{i_2}, \ldots, v_{i_{n-1}}, b$ with distances $d_1, d_2, \ldots, d_n, 2^n$, where $d_i = 2^j, 1 \leq i \leq n$ for some j, along the arcs, respectively. Because P is a feasible (a, b)-path, we must have $d_1 + d_2 + \cdots + d_n + 2^n = 2^{n+1} - 1$. However, this equation is feasible if and only if each $2^i, 1 \leq i \leq n$ appears uniquely in the sum. To see this, consider the binary representation

$$\overbrace{111\ldots111}^{n}{}_2 = 2^{n+1} - 1$$

Each $2^i, 1 \leq i \leq n$ must appear at least once, and since there are only n variables in the summation, each $2^i, 1 \leq i \leq n$ must appear exactly once.

Therefore, each vertex in the path P appears uniquely and the cycle $a, v_{i_2}, \ldots, v_{i_{n-1}}, a$ is a Hamilton Cycle in H. □

Collorary 3.0.2.1. Continuous-SPW problem remains **NP**-hard when restricted to bipartite graphs.

Proof. The graph constructed in the reduction preserves the bipartiteness property. To see this, suppose that the vertex set N has a bipartition X, Y. Then we can bipartition the vertex set V by bipartition the vertex sets $N_1, N_2, \ldots, N_{n-1}$ the same as N and add the vertices a, b to the partition where a is in the X, Y bipartition of N. Since Hamilton Cycle Problem is **NP**-complete even when the graph H is bipartite [GJ79], the result follows. □

Next, let us consider Continuous-SPW for the class of planar graphs.

Theorem 3.0.3. Continuous-SPW problem remains **NP**-hard when its input is restricted to planar graphs.

Proof. We are going to use the graph G constructed in the reduction for the general graph case in Theorem 3.0.2 and transform it to a planar graph instance F of Continuous-SPW.

We make the graph G planar in the previous reduction by adding a new vertex at every pair of crossing arcs in the graph; that is, if there are arcs $(x_i, y_i), i = 1, \ldots, k$ that intersect with arc $(u, v), u \in N_L, v \in N_{L+1}$, then we add vertex x_{vy_iL} to each crossing with time window $[0, 2^n]$. Each arc $(x_{vy_iL}, x_{vy_{i+1}L}), \quad i = 1, \ldots, k-1$ between the new vertices has distance 2^{-2n^2L}, while arcs $(u, x_{vy_1L}), (x_{vy_kL}, v)$ have distances $2^v - 2^{-vL} - (k-1)2^{-2n^2L}$ and 2^{-vL}, respectively (see Figure 3.2).

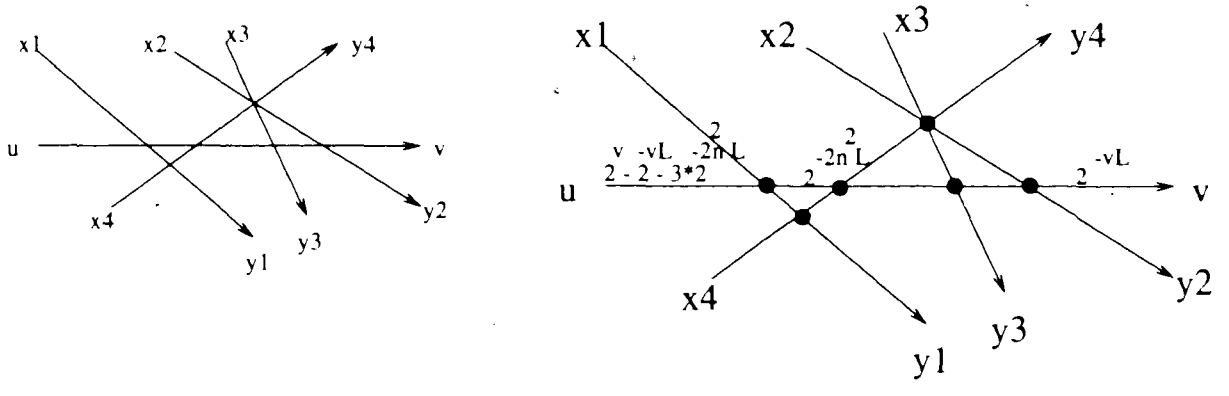


Figure 3.2: Constructing a Planar Instance of Continuous-SPW

Since each arc has at most n^2 crossings, we can see that the above transformation takes polynomial time.

The main point in the assignment of the distances to the arcs in this scheme is to force the existence of a feasible (a, b)-path P in the planar construction if and only

if the path P with the new vertices deleted is a feasible path in the general instance graph G. By assigning very small distances between the new vertices in the crossing, we essentially introduce fractional values that are canceled only when they are added together to the total distance of paths in F that correspond to paths in G. We show next that the planar instance F has a feasible (a,b)-path if and only if the instance H is Hamiltonian.

If H has a Hamilton Cycle, then we can easily see that there is a corresponding (a,b)-path in F since for an arc (u,v) in G such as the one shown in Figure 3.2, the corresponding (u,v)-path in F has the same distance value.

Now suppose that the planar instance F has a feasible (a,b)-path. Each arc along the path has a distance of $2^i - 2^{-iL} - (k-1)2^{-2n^2L}$, 2^{-iL}, or 2^{-2n^2L}, for some integers i, k, L. Because vertex b has an integer time window $[2^{n+1}-1, 2^{n+1}-1]$, the total distance of the (a,b)-path must be $2^{n+1}-1$. This implies that for each arc in the (a,b)-path with distance 2^{-iL} for some i, L, there must be a corresponding arc in the path with distance $2^i - 2^{-iL} - (k-1)2^{-2n^2L}$ and corresponding $(k-1)$ arcs with distances 2^{-2n^2L}, for some k. The reason is that each distance 2^{-iL}, and 2^{-2n^2L} are unique; the smallest value $2^{-n\cdot n}$ is greater than $n^2 \cdot 2^{-2n^2\cdot 1}$, and $2^{-2n^2 i}$ is greater than $n^2 \cdot 2^{-2n^2(i+1)}$. If we look at their binary representations, the terms 2^{-iL}, $i = 1, \ldots, n$, $L = 1, \ldots, n$ are distinct in the first n^2-bits of the fractional part, while the terms 2^{-2n^2L} are distinct between $2n^2$-bits and $2n^3$-bits of the fractional part.

Hence, the underlying (a,b)-path without the crossing vertices corresponds to an (a,b)-path in G, which corresponds to a Hamilton Cycle in H.

□

Theorem 3.0.4. Continuous-SPW problem remains **NP**-hard when restricted to grid graphs.

Proof. From Theorem 3.0.3, it suffices to modify a planar Continuous-SPW instance to a planar Continuous-SPW instance in which each vertex has both outdegree and indegree of at most 2. For each vertex u that has arcs $a_1, a_2, \ldots, a_i$ into u with distances $d_1, d_2, \ldots, d_i$ and arcs $a_{i+1}, a_{i+2}, \ldots, a_{i+o}$ out of u with distances $d_{i+1}, d_{i+2}, \ldots, d_{i+o}$, respectively, we replace the vertex with a double comb like subgraph consisting of $i + o$ new vertices $y1, y2, \ldots, y_{i+o}$ as shown in Figure 3.3.

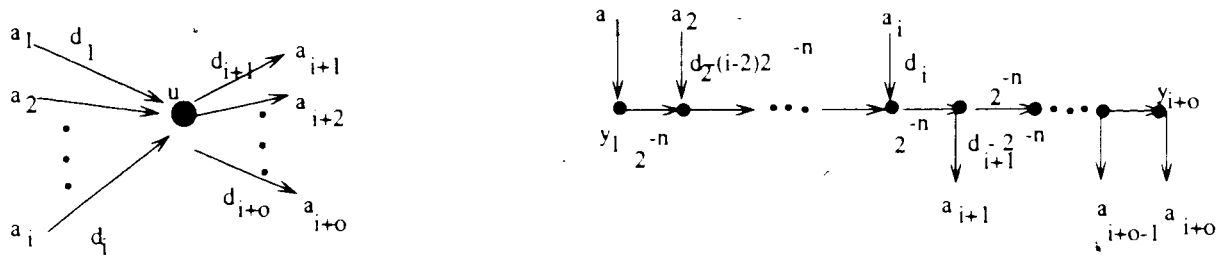


Figure 3.3: Constructing a Grid Graph Instance of Continuous-SPW

The vertices $y_1, y_2, \ldots, y_{i+o}$ in the comb subgraph has time window $[0, 2^n]$, the same as all other internal vertices of the planar graph F. The arcs $(y_j, y_{j+1}), j = 1, \ldots, i + o - 1$, has distance 2^{-n}, while the arcs $a_k, k = 1, \ldots, i$ has distance $d_k - (i - k)2^{-n}$, and the remaining arcs $a_l, l = i + 1, \ldots, i + o$ has distance $d_l - (l - i)2^{-n}$.

We can see that this transformation into a grid graph can be accomplished in polynomial time. Using Theorem 3.0.3, we see that a feasible (a, b)-path exists if and only if the instance H is Hamiltonian. □

Next, we show that even with further restriction that all arcs in the grid graph can be oriented only in two possible directions (eg. right and down), that is, the grid graph is monotone, the problem remains **NP**-hard.

Theorem 3.0.5. Continuous-SPW problem remains NP-hard even when the grid graphs is monotone.

Proof. We will use the *Partition Problem* for our reduction. In this problem, we are given a finite set A and a size $s(a) \in Z^+$ for each $a \in A$. The problem is to find a subset $A' \subset A$ such that $\sum_{a \in A'} s(a) = \sum_{a \in A - A'} s(a)$ and $|A'| = |A|/2$.

Let us consider such an instance of the Partition Problem with set $A = \{a_1, a_2, ..., a_n\}$. We construct a monotone grid graph for the Continuous-SPW instance in a diamond structure such as the one shown in Figure 3.4 where $M = 2^{\lceil \log(\sum_{a \in A} s(a)) \rceil + 1}$ and the time windows of all vertices but vertex b are unbounded.

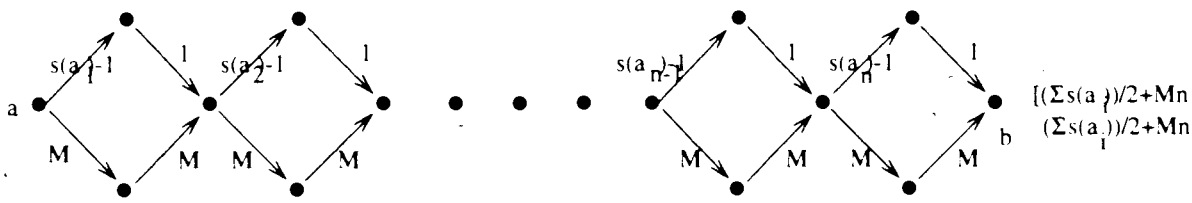


Figure 3.4: Reducing Partition Problem Instance to Monotone Grid Graph Instance of Continuous-SPW

Here again, we use an appropriate assignment of distances to arcs so that a feasible (a, b)-path exists if and only if there is a solution to the Partition Problem. Since M is larger than the sum of all the item sizes, a feasible (a, b)-path must contain exactly n arcs with distance M; that is, the path must contain exactly n of the bottom arcs in the diamond structure, and exactly n of the top arcs that sum to $(\sum_{a \in A} s(a))/2$.

Hence, there is a (a, b)-path to the Continuous-SPW instance if and only if there is a solution to the Partition Problem instance. □

These results strongly suggest that we are unlikely to find a polynomial time algorithm for the problem.

Chapter 4

Sequential Algorithms

Although we have shown in Chapter 3 that Continuous-SPW is **NP**-hard even for quite restrictive graphs, in many practical situations, it is often the case that the distances along the arcs or the intervals of the time windows at the vertices are small, bounded by a polynomial in the size of the input instance. For example, consider our previous model of pick-up service for trucks in a city with roads represented as grid graphs. In that model, we can see that the distances of roads between intersections can easily be bounded by a constant.

In cases such as those shown above, we want to determine if there exist algorithms that solve the problems in polynomial time when the maximum of the input values are bounded. Algorithms that exhibit this type of behavior are called *pseudo-polynomial time algorithms*.

In this chapter, we will show by construction that a pseudo-polynomial time algorithm for Continuous-SPW exists.

We will construct a set of algorithms that runs in polynomial time with respect to the input length when the maximum time window interval is bounded. We will then

construct an algorithm that runs in polynomial time with respect to the input length when the maximum distance is bounded.

4.1 Algorithms for Bounded Time Window Intervals of Continuous-SPW

Let us consider a Continuous-SPW instance $G = (V, A, D, T, x, y)$ in which the underlying graph is acyclic. Suppose we want to determine for all possible start times, and for all vertices x in V, a shortest feasible (x, y)-path. Let $M = \max_{[a,b]\in T}\{b - a\}$.

An algorithm for this problem proceeds as follows. Start at vertex y and work backwards in a breadth-first traversal. For each traversal along an arc (x, v), compute the earliest arrival time to y from u, for each start time t in the time window $[a, b]_x$ of vertex x. A sketch of the algorithm is shown in Figure 2.

Algorithm 2: Backward BFS Traversal.
Input: Continuous-SPW instance $G = (V, A, D, T, x, y)$
Output: Shortest path distance $d_{uy}(t)$ for $u \in V$ and start times $t \in [a, b]_u$

(1) Set $d_{uy}(t) = +\infty$ for all $u \in V, t \in [a, b]_u$.
(2) Set $d_{yy}(t) = 0$ for all $t \in [a, b]_y$.
(3) Set queue $Q = \{y\}$.
(4) **while** $Q \neq \{\}$
(5) $v \leftarrow$ dequeue Q
(6) **foreach** vertex u adjacent to v
(7) **foreach** t in $[a, b]_u$
(8) compute $d_{uy}(t) = \min\{d_{uy}(t), d_{uv} + d_{vy}(t + d_{uv})\}$, $d_{vy} \in D$
(9) enqueue u to Q

We can easily see that the complexity of this approach is $O(n + mM)$; each arc is traversed once in Step 6, and Step 7 takes $O(M)$ time for each iteration in the loop.

Let us go through the example shown in Figure 4.1. The output produced at each

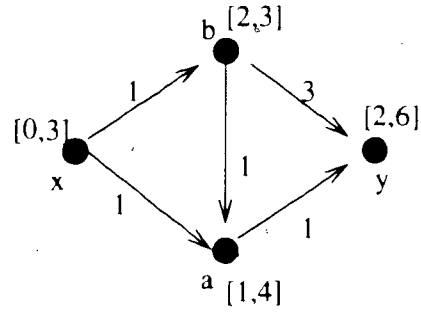


Figure 4.1: An Instance of Continuous-SPW for the Sequential Algorithm

step is shown in Table 4.1.

Breadth Level 1				Breadth Level 2						Breadth Level 3	
$d_{ay}(t)$		$d_{by}(t)$		$d^a_{by}(t)$		$d^b_{xy}(t)$		$d^a_{xy}(t)$		$d_{xy}(t)$	
1	2	2	5	2	4	0	$+\infty$	0	2	0	2
2	3	3	6	3	5	1	5	1	3	1	3
3	4					2	6	2	4	2	4
4	5					3	$+\infty$	3	5	3	5

Table 4.1: Output of Continuous-SPW Algorithm

In the first backward BFS iteration starting at vertex y, the shortest (b, y)-path and the shortest (a, y)-path is computed and remembered, for all the feasible start times at vertices b and a. In the second iteration, we find an (x, y)-path that uses vertex a, an (x, y)-path that uses vertex b, and a shorter (b, y)-path that uses vertex a. In the last iteration we find another (x, y)-path that uses both vertices b and a.

Remark 4.1.1. We note that it is necessary in the algorithm to use backward BFS because a shortest (u, w)-path passing through vertex v does not necessarily consist of the shortest (u, v)-path. Consider for example, the graph in Figure 4.2. Using BFS from vertex x starting at time 0 we would reach vertex c at time 2 by going through

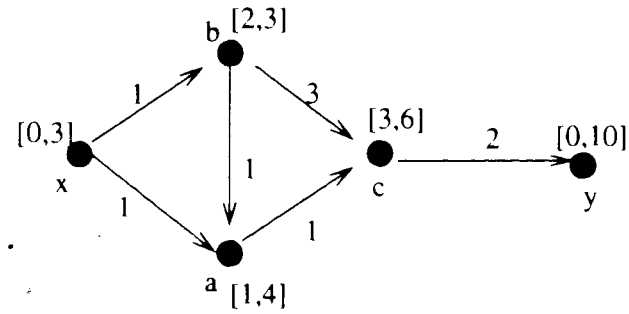


Figure 4.2: An Instance of Continuous-SPW for BFS Sequential Algorithm

vertex a. However, this does not yield a feasible (x, y)-path.

Remark 4.1.2. We can perform Algorithm 2 n times for each vertex b in the graph to obtain the all pairs shortest feasible (a, b)-paths.

Remark 4.1.3. We can perform the all pairs source and sink algorithm version M times for each arrival time in the sink's time window to find shortest feasible paths for all combinations of source,sink,start time, and arrival time in $O((n + mM)nM)$.

Remark 4.1.4. We can easily extend the procedure to handle instances in which the underlying graph contains cycles by computing the transitive closure. This yields an algorithm with running time complexity of $O((n + mM)M)$.

Table 4.2 is a summary of the time complexity for the different Continuous-SPW sequential algorithms.

Continuous-SPW Algorithms	Complexity
n source, one sink, all start times, shortest arrival time	$O(n + mM)$
n source, n sink, all start times, shortest arrival time	$O(n(n + mM))$
n source, one sink, all start times, all arrival times	$O(M(n + mM))$
n source, n sink, all start times, all arrival times	$O(nM(n + mM))$

Table 4.2: Complexity of Continuous-SPW Algorithms

4.2 Algorithms for Bounded Distances of Continuous-SPW

In the previous section, we constructed algorithms that run in polynomial time with respect to the input length and the maximum time window interval. Even though we expect to find these values bounded by a polynomial in the input length in many practical situations, we would still like to investigate how the problem behaves when the distances are bounded, but the time window intervals are of exponential size in the length of the input. In particular, we are interested in constructing a pseudo-polynomial time algorithm with respect to the input length and the maximum distance.

Let us assume, without loss of generality, that all distances are unit 1 (we can always subdivide an arc in the input graph to paths composed of unit distance arcs whose total path-length is equal to the initial arc distance).

Given an input instance $G = (V, A, T, s, t, t_0)$ with vertex set V, arc set A, set of time window intervals T, source s, sink t, and initial time t_0 at s, we want to find an (s, t)-path starting at time t_0.

If the underlying graph in the problem is acyclic, then clearly the problem reduces to finding a feasible $(s, t)-$ path, which can be accomplished using a BFS algorithm

on the graph in G.

So let us suppose that the underlying graph contains cycles. Then we can simulate a waiting time w at a vertex u by traversing along the cycles through u of length w. If we can pre-compute and describe in polynomial time all the waiting times possible at each vertex, then we can use a BFS algorithm on G to solve Continuous-SPW. Specifically, if we have computed all the arrival times of a (s, a)-path in G, and (a, b) is an arc in G, then we can compute and describe all the arrival times of (s, b)-paths using arc (a, b) as follows: the arrival times of (s, b)-paths is the arrival times of (s, a)-paths plus 1 that are feasible, together with all the waiting times possible at vertex b.

Hence, our problem is reduced to that of finding a polynomial time description of possible waiting times at each vertex.

Computing Possible Waiting Times

Suppose we want to compute all possible waiting times feasible at a vertex v. If vertex v is not in any cycle of the graph, trivially no waiting is possible at v. So let us suppose that there are simple cycles of length $c_1, c_2, ..., c_k$ that go through vertex v. Then any waiting time w possible at v must be a feasible linear combination of $c_1, c_2, ..., c_k$: that is,

$$w = x_1c_1 + x_2c_2 + ... + x_kc_k \tag{4.1}$$

where $x_1, x_2, ..., x_k$ are non-negative integers. This is a linear Diophantine equation with non-negative integers constraints. Therefore, a polynomial description to the solution space of Equation (4.1) corresponds to a polynomial description of all the feasible waiting times at vertex v.

Unfortunately, determining whether an integer w' is in the solution space of Equation (4.1) is an **NP**-hard problem. It is shown in [Sch86] that the *Integer Programming Problem*, known to be **NP**-complete [GJ79], reduces to the linear Diophantine equations with non-negative integer variables. There is, however, a pseudo-polynomial time algorithm for Equation (4.1) that runs in polynomial time in the input length, and the maximum value of $c_1, c_2, ..., c_k$ and w. In our case, w can take on exponential values and therefore we cannot use the pseudo-polynomial time algorithm.

In what follows, we will show that if we restrict the problem by requiring that for a fixed number of values $c_{i_1}, c_{i_2}, ..., c_{i_j}$, the $\gcd(c_{i_1}, c_{i_2}, ..., c_{i_j})$ is 1 then there is a pseudo-polynomial time algorithm to determine what integer values are in the solution space of Equation (4.1) with respect to the input length, and the maximum values of only $c_1, c_2, ..., c_k$. This requirement allows us to solve our waiting time computation in polynomial time in the maximum distance values only since each of $c_1, c_2, ..., c_k$ are cycle lengths.

Theorem 4.2.1. If for a fixed $j, 1 \leq j \leq k, \gcd(c_1, c_2, ..., c_j) = 1$ in Equation (4.1), then there is an algorithm that can describe the solution space of Equation 4.1 in polynomial time with respect to the $\max\{c_1, c_2, ..., c_j\}$.

Proof. We begin with the case for two variables; that is, $j = 2$. Our linear Diophantine equation will be:

$$x_1 c_1 + x_2 c_2 = w \tag{4.2}$$

where (c_1, c_2) is not necessarily 1. We recall in Chapter 2, Section 2.5 that for a given initial integer solution x_1^0, x_2^0, all solutions of 4.2 can be expressed as

$$x_1 = x_1^0 - \frac{c_2}{(c_1, c_2)}t, \qquad x_2 = x_2^0 + \frac{c_1}{(c_1, c_2)}t, \qquad t \text{ integer}$$

If we let $x_1^0 = ms$, and $x_2^0 = mr$, where r, s are integers such that $sc_1 + rc_2 = (c_1, c_2)$ and where $m = w/(c_1, c_2)$, then all non-negative integer solutions x_1, x_2 must satisfy:

$$x_1 = ms - \frac{c_2}{(c_1, c_2)}t \geq 0, \qquad x_2 = mr + \frac{c_1}{(c_1, c_2)}t \geq 0$$

This implies that an integer solution exists if and only if t can take on a value in the interval,

$$[\frac{-mr}{c_1}(c_1, c_2), \frac{ms}{c_2}(c_1, c_2)]$$

Recall that $\{c_1, c_2\}$ is the least common multiple of c_1, c_2. If $w \geq \{c_1, c_2\}$, then by Lemma 2.5.2,

$$\frac{ms}{c_2}(c_1, c_2) + \frac{mr}{c_1}(c_1, c_2) \geq sc_1 + rc_2 \geq 1$$

Since the interval is of size ≥ 1, a non-negative integer solution always exists. For integer $w < \{c_1, c_2\}$ we can do a brute force search of all possible x_1, x_2 values; there are only polynomial number of possible values in c_1, c_2 — not more than $c_1 c_2$ possible values. Therefore, we have

Theorem 4.2.2. There is a polynomial time algorithm in c_1, c_2 for the linear Diophantine equation (4.2).

See also [HW76, Kan80] for an alternative algorithm. Next, we consider the case for three variables. The linear Diophantine equation is

$$x_1 c_1 + x_2 c_2 + x_3 c_3 = w \tag{4.3}$$

Naturally, we would like to apply Theorem 4.2.2.

Claim 4.2.2.1. Equation 4.3 has an integer solution if and only if

$$x_1 + x_{23}(c_2, c_3) = w \tag{4.4}$$

has an integer solution x_1, x_{23}.

Proof. Let y_1, y_2, y_3 be a solution to Equation (4.3). Then $y_2c_2 + y_3c_3 = k$ for some integer k. Since (c_2, c_3) divides y_2c_2 and y_3c_3, it must also divide k. Thus $k = m(c_2, c_3)$ for some integer m, and clearly $x_1 = y_1$, $x_{23} = m$ is a solution for Equation (4.4).

Conversely, let us suppose that z_1, z_{23} is a solution to Equation (4.4). Using the Extended Eucledian algorithm, we can find integers z_2, z_3 such that $z_2c_2 + z_3c_3 = (c_2, c_3)$. Then we can see easily that $x_1 = z_1$, $x_2 = z_2z_{23}$, $x_3 = z_3z_{23}$ is a solution to Equation (4.3) □

Hence, we can determine the solution space of non-negative integers for Equation (4.3) recursively by determining the solution space of Equation (4.4). Hence, we have

Theorem 4.2.3. There is a polynomial time algorithm in c_1, c_2, c_3 that solves linear Diophantine equation (4.3).

Now, we can generalize to Theorem 4.2.1 by repeat application of Theorem 4.2.3. We should notice, however, that the range of feasible values for x_{23} in Equation (4.4) may not be continuous. The reason is that $x_2c_2 + x_3c_3 = v$ is guaranteed to be feasible only when $v \geq \{c_2, c_3\}$. Hence, Equation (4.3) always has a non-negative integer solution when $w \geq \{c1, \{c_2, c_3\}\}$. When $w < \{c_1, c_2, c_3\}$ then we can again use brute force search to find all possible $c_1c_2c_3$ values or, alternatively, find a proper subset S of c_1, c_2, c_3 in which the gcd of those values is 1 and their lcm is smaller than $\{c_1, c_2, c_3\}$. Hence, Theorem 4.2.1 follows. □

All that remains is to compute the values $c_1, c_2, ..., c_j$ in polynomial time. A simple greedy approach is to compute the shortest cycle lengths progressively; that is, determine the values in the order $c_1 \leq c_2 \leq ... \leq c_j$. Each time we find a cycle c_i, we compute all gcds $g_1, g_2, ..., g_l$ of all subsets of $c_1, c_2, ..., c_i$ such that $g_1 \leq g_2 \leq ... \leq g_l$

and describe the solution space generated.

Since we only need to consider a constant number of cycles before we obtain a set of lengths with gcd 1, we will have eventually either described all the waiting times within the vertex's time window, or else, during traversal of the cycles, have reached the upper bound of some vertex z's time window. In the former case, we completed the vertex's waiting time description in polynomial time. In the latter case, vertex w will never be visited again for the purpose of computing waiting times; thus, a vertex is eliminated in polynomial time.

Remark 4.2.1. In practical situations, it is very difficult to determine precisely whether an input instance has a graph in which we expect to find a bound on the number of cycle length combinations going through a vertex with gcd 1. However, if our input is an undirected graph, or that there are bidirectional arcs between any pair of adjacent vertices in the graph, then the gcd requirement can easily be achieved. The reason is that bidirectional arcs at each vertex provide provide a length 2 cycle, so the problem of finding cycle lengths with gcd 1 becomes that of finding odd cycle lengths.

Therefore, when the network has bidirectional arcs between adjacent vertices, we have a pseudo-polynomial time algorithm in the input length and the maximum distance value that solves Continuous-SPW.

We now show how to solve the problem for general graphs where we no longer require that a constant number of cycle lengths have gcd 1.

Our strategy is to construct a cycle C with length d from the composition of simple cycles of lengths $a_1 \leq a_2 \leq \ldots \leq a_k$ through the vertex such that $\gcd(d, a_1) = 1$. We can then apply Theorem 4.2.2 to obtain a pseudo-polynomial time algorithm for Continuous-SPW provided that d is polynomial in the length of the input instance.

Hence, we want

$$d = \sum_{i=1}^{k} a_i x_i, \quad 0 \leq x_i \leq b, \quad \text{where} \quad \gcd(a_1, d) = 1 \tag{4.5}$$

Let us suppose that a_1 has a prime decomposition $p_1^{e_1} p_2^{e_2} \cdots p_j^{e_j}$. Let $\pi = p_1 p_2 \cdots p_j$. Then $\gcd(a_1, d) = 1$ if and only if $d \not\equiv 0 \pmod{p_i}, i = 1, \ldots, j$. That is,

$$\begin{aligned} a_1 x_1 + a_2 x_2 + \cdots a_k x_k &\equiv c_1 \pmod{p_1} \\ a_1 x_1 + a_2 x_2 + \cdots a_k x_k &\equiv c_2 \pmod{p_2} \\ &\vdots \\ a_1 x_1 + a_2 x_2 + \cdots a_k x_k &\equiv c_j \pmod{p_j} \end{aligned} \tag{4.6}$$

where $c_i \neq 0, i = 1, \ldots, j$. By the Chinese Remainder Theorem, for each setting of the c_i's, there is a unique solution $d \pmod{\pi}$. Since there are $m = \Pi_{i=1}^{j}(p_i - 1)$ settings of the c_i's such that no $c_i = 0$, we can enumerate all integers $0 < d_1, d_2, \ldots, d_m < \pi$ such that $d_i \not\equiv 0 \pmod{p_s}, i = 1, \ldots, m, s = 1, \ldots, j$. So all we have to show is that there is a non-negative linear combination of $a_1, a_2, \ldots, a_k$ which yields one of the $d_1, d_2, \ldots, d_m$.

Fortunately, we can generate all possible linear combinations of $a_1, a_2, \ldots, a_k$ values which are less than π in no more than $O(k\pi)$ steps.

Let us construct a digraph $J = (V, A)$ with vertex set $V = \{0, 1, 2, \ldots, \pi - 1\}$ and arc set $A = \{(x, y) : x + a_i \equiv y \pmod{\pi} for some i\}$. Clearly, we can construct J in no more than $O(k\pi)$ steps. Moreover, we can see that any vertex u reachable from vertex 0 corresponds to a linear combination of $a_1, a_2, \ldots, a_k$ that yields u. Hence, finding d for Equation (4.6) reduces to finding a $(0, d_i)$-path in J, for some $1 \leq i \leq m$. This can be done using BFS traversal on J starting at vertex 0.

Let us go through an example to illustrate how the possible waiting times are computed. Suppose there are three cycles of length 6, 10, 15 at vertex u as shown in

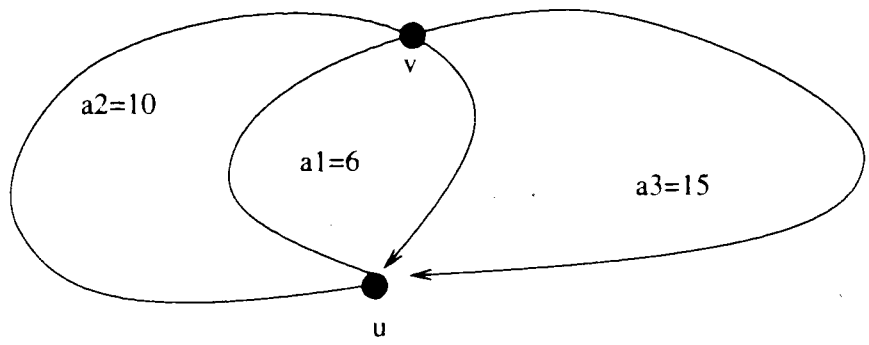


Figure 4.3: Computing Possible Waiting Times

Figure 4.3. For simplicity, we assume that the vertices have unbounded time windows. The shortest cycle at vertex u has length $6 = 2 \cdot 3$. So we want to compose a cycle of length d as a linear combination of the cycles with length 10 and length 15 such that

$$d \not\equiv 0 \pmod 2, \quad \text{and} \quad d \not\equiv 0 \pmod 3 \tag{4.7}$$

Listing all values $1, 2, 3, 4, 5 \pmod 6$, we see that the only values for d that satisfies Equation (4.7) are 1 and 5. Next, we construct a digraph as shown in Figure 4.4. .

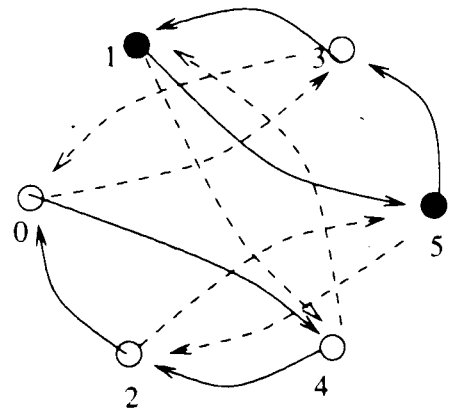


Figure 4.4: A Directed Graph Representation of Cycle Length Combinations

Solid arcs represent addition of 10 modulo 6, while broken arcs represent addition

of 15 modulo 6. We can see that there are many combinations for $(0,1)$-paths and $(0,5)$-paths in the digraph, such as $0+10+15 \pmod 6$, and $0+10+15+10 \pmod 6$.

All the steps so far can be done in polynomial time with respect to $\max\{a_1, a_2, ..., a_k\}$. All that is left for us to show is that prime decomposition of a_1 can also be computed in polynomial time with respect to $\max\{a_1, a_2, ..., a_k\}$.

A simple approach that takes $O(a_1^{1/2})$ time to decompose integer a_1 is to remove all integers from the set $2, 3, ..., a_1^{1/2}$ that divides a_1. However, a better alternative algorithm that uses randomization is the *Pollard's Rho Heuristic*. It has a expected running time of $O(a_1^{1/4})$ [CLR90].

Hence, we have an algorithm for the Continuous-SPW that runs in polynomial time with respect to the input length and the maximum input distance.

Chapter 5

Parallel Continuous-SPW Algorithm

In this chapter, we will design a parallel algorithm for Continuous-SPW using a technique known as pointer doubling.

The pointer doubling technique is used in many parallel algorithms [CLR90]; in particularly, the parallel algorithm for SPW in [GK95] uses pointer doubling. Because of the similarities between Continuous-SPW and SPW, we will introduce the implementation in [GK95] and follow it closely, adding any extensions necessary for Continuous-SPW.

An instance $G = (V, A, D, T, s, t)$ of SPW consists of a network with vertex set $V = \{v_1, ..., v_n\}$, arc set $A = \{a_1, a_2, ..., a_m\}$, distance set $D = \{d_1, d_2, ..., d_m\}$, set of time windows $T = \{t_1, t_2, ..., t_n\}$, a source s and sink t. SPW asks for a shortest $(s, t)-$path such that a vehicle visits vertices in the path within their time windows. This is similar to Continuous-SPW, except that the vehicle is allowed to wait at a vertex for a time window to open.

Allowing a vehicle to wait makes SPW a much easier problem to solve. In particular, for SPW there is a polynomial time algorithm with respect to the input length even when all distances and time windows take on exponential values. The reason is that the following property holds: an (a, b)-subpath of a shortest path is the shortest path from a to b. If the vehicle arrives earlier than b's time window, it can simply wait.

Hence, an algorithm for SPW only has to look for simple paths. Furthermore, a shortest (s, t)-path can always be obtained by composing shortest subpaths previously computed (this is especially useful in the parallel case). The algorithm in [GK95] takes advantage of this property by composing paths P_1, P_2 in parallel, given that P_1, P_2 has been computed. The resulting path has length up to twice the length of P_1 or P_2 so that in i steps, all paths of length up to 2^i are computed.

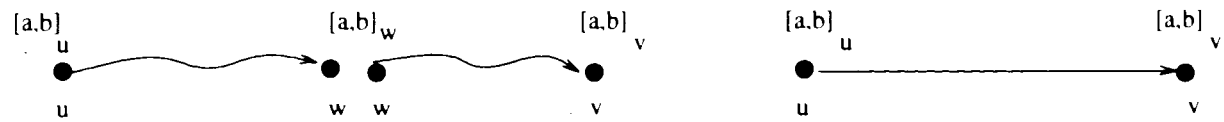


Figure 5.1: Pointer Doubling Technique

Because there are time constraints involved, a table τ_{uv} is used for each computed (u, v)-path describing start and arrival times of a vehicle using the path. This table is updated during path compositions and it is shown that the table size never exceeds $O(n)$. Intuitively, this holds because only the shortest arrival time needs to be known for any given start time.

An outline of the parallel algorithm is given below.

It is shown in [GK95] that Algorithm 3 has complexity of $O(\log^2 n)$ time using $O(n^4)$ processors in a CREW PRAM.

Algorithm 3: Parallel Algorithm for SPW.
Input: SPW instance $G = (V, A, D, T, s, t)$
Output: Tables τ_{uv} for each $u, v \in V$
(1) **In parallel step** compute initial τ_{uv} for each arc $(u, v) \in A$
(2) Loop for $\log n$ rounds
(3) **In parallel step** for every u, v in V
(4) let $S = \{w \in V : (u, w), (w, v) \in A\}$
(5) **foreach** $w \in S$
(6) compute τ_{uv}^{w} by composition from τ_{uw} and τ_{wv}
(7) set $\tau_{uv} = \min\{\tau_{uv}, \min\{\tau_{uv}^{w} | w \in S\}\}$
(8) add arc (u, v) to G.

In the case of Continuous-SPW, the property of shortest subpath may not hold as we have seen in Chapter 4, Figure 4.2; thus, the vehicle cannot arrive earlier than a vertex's time window and wait. Therefore, computing only the shortest path during compositions is not sufficient; all feasible arrival times for every start time must be computed. This implies that a table τ_{uv} for each (u, v)-path has size at least M^2, where M is the maximum time interval in G.

Using a $0-1$ matrix to represent feasibility paths in τ_{uv}, we can implement path composition simply as a $0-1$ matrix multiplication; that is, we set $\tau_{uv}(t_1, t_2) = 1$ if and only if there is a (u, v)-path starting at time t_1 and arriving at time t_2, and we set composition τ_{uv}^{w} to be $\tau_{uw} \times \tau_{wv}$. An algorithm for Continuous-SPW can no longer simply look for simple paths.

Let us analyse the complexity of Algorithm 4. Initialization in Step 1 can be done in $O(1)$ time using $O(M^2)$ processors. Composition in Step 6 is a $0-1$ matrix multiplication step which can be done in $O(\log M)$ time with $O(M^3/\log M)$ processors using the *Four Russian's Matrix Multiplication* [Cha92]. Finally, path computation in Step 7 can be done in $O(\log n)$ time with $O(nM^2)$ processors.

Hence, the total complexity is $O(\log^2 M)$ time and $O(n^3 M^3/\log M)$ processors in a CREW PRAM -- total work of processor and time product of $O(n^3 M^3 \log M)$. This is

Algorithm 4: Parallel Algorithm for Continuous-SPW
Input: Continuous-SPW instance $G = (V, A, D, T, s, t)$
Output: tables τ_{uv} for vertices $u, v \in V$

(1) **In parallel step** set $\tau_{uv} = \mathbf{0}$, for vertices u, v in V.
(2) Loop $\log M$ times
(3) **In parallel step** for every u, v in V
(4) let $S = \{w \in V : (u, w), (w, v) \in A\}$
(5) **foreach** $w \in S$
(6) compute $\tau_{uv}^{w} = \tau_{uw} \times \tau_{wv}$
(7) compute $\bigvee_{w \in S} \tau_{uv}^{w}$
(8) add arc (u, v) to G.

quite efficient compared to the sequential algorithm in Chapter 4 which has complexity of $O(nM^2(n + mM))$ time for graphs that contain cycles. When $m \in O(n^2)$, this is $O(n^3M^3)$.

We can see that for instances where the graph is acyclic, however, the parallel algorithm takes more work by a factor of $O(M)$ over the sequential algorithm. Finding an efficient parallel algorithm for acyclic graphs that uses this additional information is still an open problem. However, one obvious improvement that can be done to Algorithm 4 is to reduce the size of tables τ_{uv} to $O(MnL)$ where $L = \max\{d \in D\}$. Any (s, t)-path in the acyclic Continuous-SPW is simple, so with start time t_1, a vehicle cannot reach other vertices later than nL. This implies that the complexity of path composition in Step 6 can be reduced to $O(\log L)$ time using $O(Mn^2L^2/\log L)$ processors. This is a major improvement in processor time product when nL is small compared to M (especially when L is polynomial in n, m). This only leaves the case where $O(nL) = M$ open.

Chapter 6

Relaxation

An alternative method of dealing with problems that are **NP**-hard is to look for constraint relaxations in the hope of being able to construct efficient polynomial time algorithms that provide approximate solutions.

In this chapter, we will look at the time window relaxation of Continuous-SPW. The first relaxation of Continuous-SPW we examine is the *Continuous Shortest Path Problem with Deadlines (CSPd)*, followed by the *Continuous Shortest Path Problem with Release Times (CSPr)*.

Given a Continuous-SPW instance $G = (V, A, D, T, s, t)$, our objective is

1. to implement a polynomial time algorithm that finds an initial feasible (s, t)-path in the relaxed problem ,

2. to approximate the solution to Continuous-SPW using the solution to the relaxed problem .

6.1 Continuous Shortest Path Problem with Deadlines

We define CSPd as a restricted version of Continuous-SPW where time windows are of the form $[0, b_i]$; that is, time constraints which consist only of the deadlines b_i's. Thus, a vehicle can always visit a vertex early before its deadline and therefore, it never has to wait.

To accomplish Objective 1, our algorithm will look for shortest (s,t)-paths in CSPd that satisfies the deadlines. It looks for shortest paths because arriving early at a vertex is no worse than arriving at a later time as long as this is before its deadline.

Algorithm 5: Algorithm for CSPd
Input: Continuous-SPW instance $G = (V, A, D, T, s, t)$ with time windows $[0, b_i]$
Output: Shortest path distances $d_{su}(t)$ for $s, u \in V$ and start time t

(1) Set $d_{ss} = 0$.
(2) Set $d_{su} = +\infty$ and $d_{su}(t) = +\infty$ for all $u \in V, t \in [0, b]_s$.
(3) Set $Q = \{s\}$.
(4) **while** $Q \neq \phi$
(5) $v \leftarrow$ dequeue Q
(6) **foreach** arc (v, u) in A
(7) Let d'_u be the deadline at u
(8) $d_{su} = \min\{d_{su}, d_{sv} + d(v, u)\}$ where $d_{sv} + d(v, u) < d'_u$
(9) **if** $d_{sv}(t) < +\infty$ for some $t \in [0, b]_s$
(10) Set $d_{su}(t) = d_{su}$
(11) **else**
(12) Set $d_{su}(t) = +\infty$
(13) enqueue u to Q

Polynomial time solvability is achieved because shortest path extensions are computed and kept in Step 7; thus, only simple paths are constructed.

The algorithm successively computes, in breadth-first traversal from vertex s, all shortest feasible paths to other vertices. It maintains for each pair of s, u vertices, the

latest start time at vertex s for which vertex u is reachable. The entire process takes at most $O(n+m)$ time.

Let us consider an example shown in Figure 6.1.

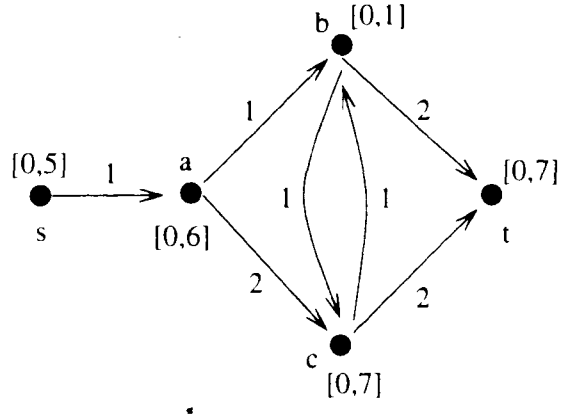


Figure 6.1: CSPd Algorithm

Computation done in each step is shown in Table 6.1.

Breadth Level 1		Breadth Level 2				Breadth Level 3	
$d_{sa}(t)$	d_{sa}	$d_{sb}(t)$	d_{sb}	$d_{sc}(t)$	d_{sc}	$d_{st}(t)$	d_{st}
$[0,5]$	1	$[0,5]$	$+\infty$	$[0,4]$	3	$[0,2]$	5

Table 6.1: Output of CSPd Algorithm

Despite the fact that we now have a polynomial time algorithm to find a feasible solution to the CSPd problem, we would prefer to find the longest (s,t)-path that satisfies deadline times as this would yield an approximation to Continuous-SPW. Unfortunately, this problem turns out to be **NP**-hard.

To see the complexity of CSPd, it is sufficient that we consider the reduction from the Hamilton Cycle Problem to Continuous-SPW in Chapter 3, Theorem 3.0.2.

In that reduction, the vertices in the vertex set $V_i, i = 1, \ldots, n-1$ and the vertex a have essentially unbounded time windows of $[0, 2^n]$, while the time window of vertex b is $[2^{n+1} - 1, 2^{n+1} - 1]$. For our CSPd, we assign vertex b's time window to $[0, 2^{n+1} - 1]$. See Figure 6.2. Notice vertex b's time window is $[0, 63]$ instead of $[63, 63]$.

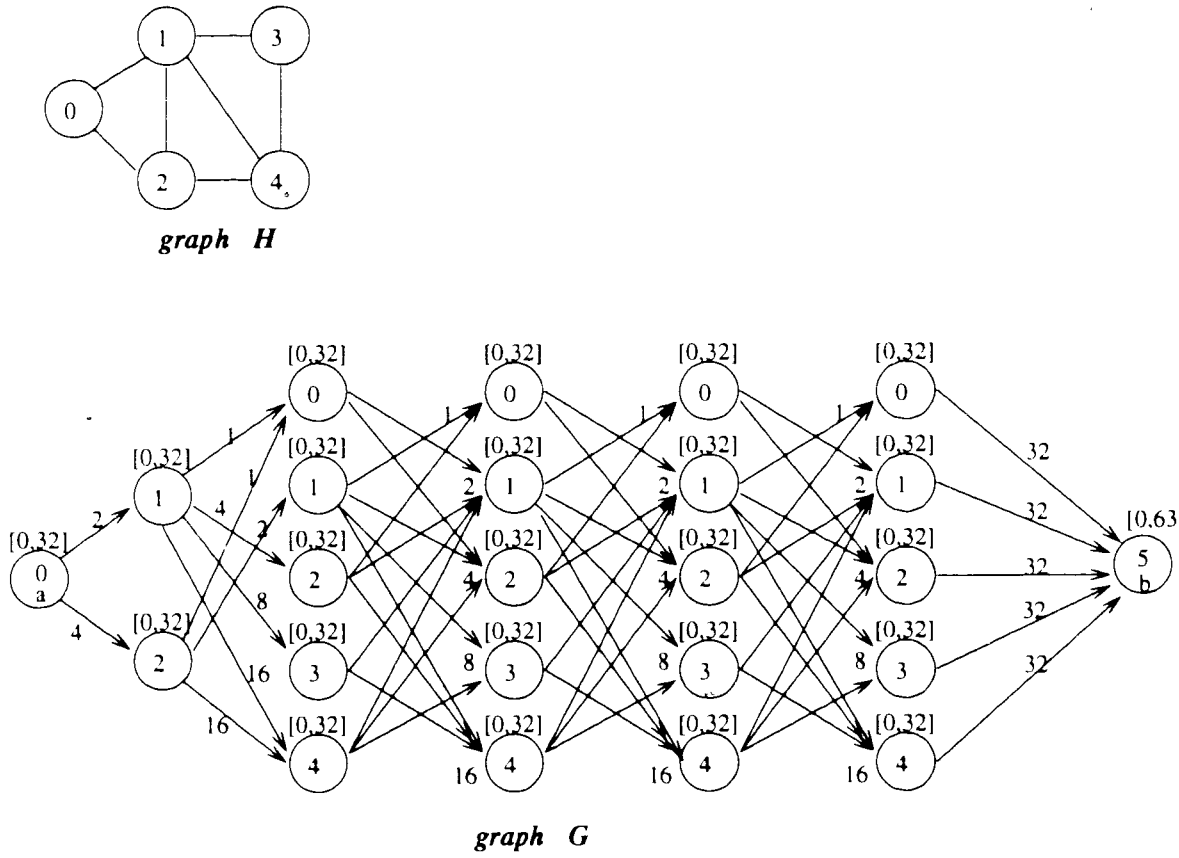


Figure 6.2: CSPd Reduction

Then we ask if there exists an (s, t)-path in the graph constructed this way that satisfies all time window constraints (only deadlines are concerned) whose total distance is at least $2^{n+1} - 1$. This is equivalent to asking if there is such an (s, t)-path whose distance is exactly $2^{n+1} - 1$. Hence, a solution to CSPd corresponds to a solution for the Hamilton Cycle Problem.

6.2 Continuous Shortest Path Problem with Release Times

Like the CSPd problem, we can define CSPr as Continuous-SPW with the restriction that the time windows be of the form $[a_i, +\infty]$; that is, time constraints which consist of only release times.

In a manner analogous to CSPd, we accomplish Objective 1 by implementing an algorithm that finds the longest (s,t)-paths in CSPr. The reason is that arriving late at a vertex is no worse than arriving at an earlier time after the release time. Let $M = \max\{a : [a, +\infty] \in T\}$.

Algorithm 6: Algorithm for CSPr
Input: Continuous-SPW instance $G = (V, A, D, T, s, t)$ with time windows $[a_i, +\infty]$
Output: Longest path distances $d_{su}(t)$ for $s, u \in V$ and start time t

(1) Set $d_{ss} = 0$. Set $d_{su} = 0$ and $d_{su}(t) = -\infty$ for all $u \in V, t \in [0, M]$.
(2) Set $Q = \{s\}$
(3) **while** $Q \neq \emptyset$
(4) $v \leftarrow$ dequeue Q
(5) **foreach** arc (v, u) in A
(6) let r_u be release time of vertex u.
(7) **if** vertex u is in a cycle
(8) $d_{su} = +\infty$
(9) **else**
(10) $d_{su} = \max\{d_{su}, d_{sv} + d(v, u)\}$ where $d_{sv} + d(v, u) \geq r_u$.
(11) **if** $d_{sv}(t) > -\infty$ for some $t \in [0, M]$
(12) set $d_{su}(t) = d_{su}$
(13) **else**
(14) set $d_{su}(t) = -\infty$
(15) enqueue u to Q

Unlike CSPd problem, we need to take care of cycles in the input graph. At each

level of the propagation with vertices v_i, we keep track of the longest distance (s, v_i)-path. If at any point in time we determine that a vertex v_j is in a cycle, then we can traverse along the cycle indefinitely; thus, we set longest distance $d_{sv_j} = +\infty$, and proceed to propagate this distance.

We can determine when a vertex is in a cycle by testing membership of the vertex along traversed paths from the source s. This testing can be performed in $O(n)$ time. Hence, we can see that we do not have to traverse more than $O(n)$ levels to find a feasible solution yielding total time complexity of $O(nm)$. For example, consider the graph in Figure 6.3. Computation performed at each step is shown in Table 6.2..

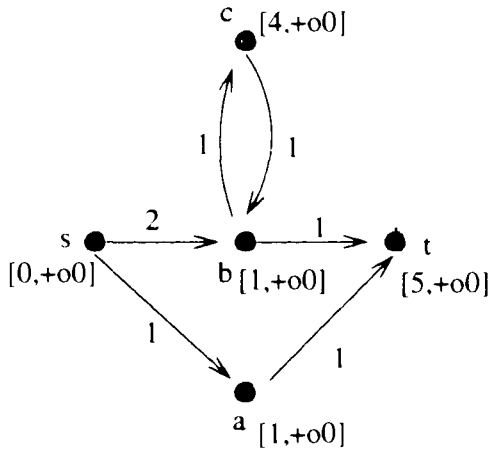


Figure 6.3: CSPr Algorithm

Next, we would like to approximate Continuous-SPW by finding a shortest feasible (s, t)-path that satisfies all release times of visiting vertices. Again, this problem turns out to be **NP**-hard.

We will use the reduction in Chapter 3, Theorem 3.0.2 with these changes: we assign the time window of vertex b in the constructed graph G to $[2^{n+1} - 1, +\infty]$ and we assign the time window of vertices in the vertex set $N_i, i = 1, \ldots, n-1$ and the

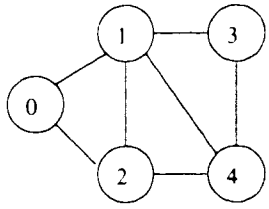
Breadth Level 1				Breadth Level 2				Breadth Level 3	
$d_{sa}(t)$	d_{sa}	$d_{sb}(t)$	d_{sb}	$d_{sc}(t)$	d_{sc}	$d_{st}(t)$	d_{st}	$d_{sb}(t)$	d_{sb}
$[0, +\infty]$	1	$[0, +\infty]$	2	$[1, +\infty]$	3	$[3, +\infty]$	2	$[0, +\infty]$	$+\infty$

Breadth Level 4	
$d_{st}(t)$	d_{st}
$[0, +\infty]$	$+\infty$

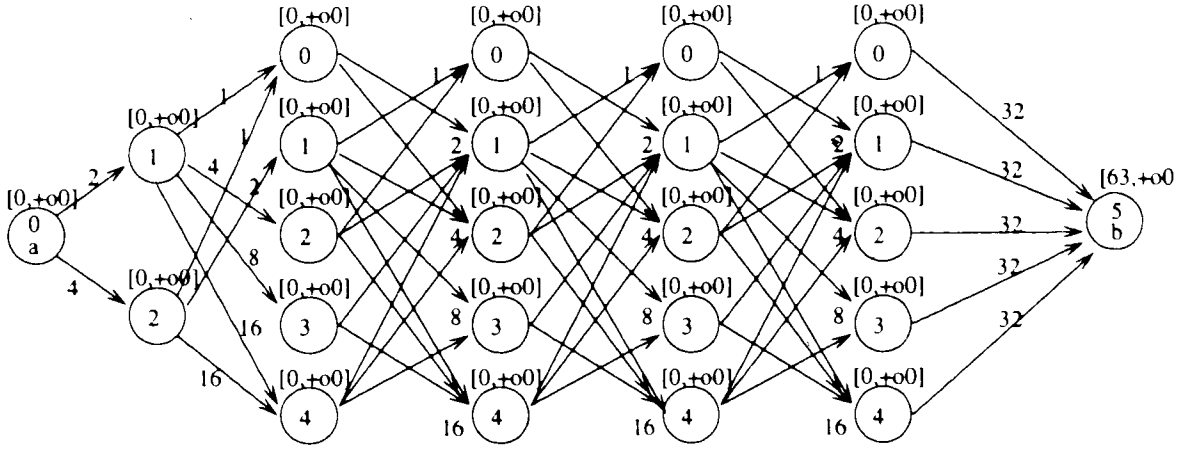
Table 6.2: Output of CSPr Algorithm

vertex a to $[0, +\infty]$. See Figure 6.4.

Then we ask for a shortest path in the graph constructed this way for CSPr whose distance is $2^{n+1} - 1$. Clearly, a solution to this problem corresponds to a solution to the Hamilton Cycle Problem.



graph H



graph G

Figure 6.4: CSPr Reduction

Chapter 7

Applications

In this chapter, we will show how the results and algorithms developed so far can be applied to SPW-Cost, a cost optimization version of SPW. Specifically, we will show an efficient reduction from SPW-Cost to Continuous-SPW.

7.1 Literature on SPW-Cost

An instance $G = (V, A, D, T, C, s, t)$ of SPW-Cost is an optimization problem of SPW on $G = (V, A, D, T, s, t)$ such that each arc (u, v) in the graph has an associated cost c_{uv}, possibly negative, in C. The objective is to determine the least cost (s, t)-path that meets the time window requirement when the vertices are visited. The vehicle is allowed to wait at a vertex until its time window opens.

SPW-Cost was first introduced as a subproblem of the *Multiple Traveling Salesman Problem with Time Windows* [DSD84].

Although the continuity constraint is no longer in the SPW-Cost problem, it is still **NP**-hard [DDSS93]. SPW-Cost problem includes the Multiple Knapsack Problem as

a special case.

For the remaining part of this section, let us the notation $H = (V, A, D, T, C, s, t)$ for an instance of SPW-Cost with network (V, A), source s, sink t, a set of distances $d_{uv} \in D$ and costs $c_{uv} \in C$ associated with each arc $(u, v) \in A$, and a set of time windows $[a_v, b_v]_v \in T$ associated with each vertex $v \in V$.

There are generally two families of algorithms proposed to solve SPW-Cost: one uses dynamic programming and the other involves Lagrangian relaxation.

For algorithms involving Langragian relaxation, SPW-Cost for simple paths can be formulated as a *mathematical program* in the following way.

$$\text{Minimize} \sum_{(i,j)\in A} c_{ij} X_{ij} \tag{7.1}$$

subject to:

$$\sum_{j\in V} X_{ij} - \sum_{j\in V} X_{ji} = \begin{cases} 1, & i = s \\ -1, & i = t \\ 0, & i \in V \end{cases} \tag{7.2}$$

$$X_{ij} \geq 0, \qquad \forall (i,j) \in A \tag{7.3}$$

$$X_{ij}(T_i + d_{ij} - T_j) \leq 0, \qquad \forall (i,j) \in A \tag{7.4}$$

$$a_i \leq T_i \leq b_i, \qquad \forall i \in V. \tag{7.5}$$

where X_{ij}, $(i,j) \in A$ is a variable for the flow on arc (i,j), T_i is a variable for the start of service at vertex i.

The objective function (7.1) seeks to minimize the total travel cost. Constraints (7.2) - (7.3) define flow condition on the graph G. Equation (7.5) defines the time window constraints. Compatibility requirements between flow and time variables is captured by (7.4).

Although this program has a nonlinear formulation, it has an appealing characteristic in that if the problem is feasible, then there is an optimal integer solution. To see this, we note that constraint (7.4) indicates that if $X_{ij} > 0$, then $T_i + d_{ij} \leq T_j$. If the flow values are fractional, then the optimal solution of value Z^* is composed of paths of cost c_p, each with positive flow ϕ_p; i.e., $Z^* = \sum_p c_p \phi_p$, where $\sum_p \phi_p = 1$. Assigning a unary flow to the arcs of the minimum cost path $c_{\min}$, then $c_{\min}$ satisfies the time constraints and constitutes an optimal integer solution since

$$Z^* \leq c_{\min} = \sum_p c_{\min} \phi_p \leq \sum_p c_p \phi_p = Z^*.$$

See [Min75] for a proposed algorithm that involves the solution of a shortest path problem with costs modified by the addition of a multiplier associated with some supplementary constraints. It produces a feasible solution and a lower bound on the value of the optimal solution.

For dynamic programming algorithms, SPW-Cost can be formulated as follows. Let $F(v,t)$ be the minimum cost of a (s,v)-path for $v \in V$ servicing vertex v at time $\leq t$. This cost $F(v,t)$ can be computed by solving these recurrence equations:

$$F(s, a_0) = 0 \tag{7.6}$$

$$F(v,t) = \min_{(u,v)\in A} \{F(u,t') + c_{uv} \mid t' \leq t - d_{uv}, a_u \leq t' \leq b_u\},$$

$$\text{for all } v \in V \text{ and } a_v \leq t \leq b_v \tag{7.7}$$

The optimal solution is given by

$$\min_{a_d \leq t \leq b_d} F(d,t) \tag{7.8}$$

The first proposed dynamic programming approach appeared in [Jok66]. It solves the recurrence equations using a generalization of the FIFO rule for the shortest path problem. Other rules with better running times have been investigated in [DS88].

7.2 Using Continuous-SPW

In this section, we will show that solving SPW-Cost is no harder than solving Continuous-SPW. In particular, we will show that an instance of SPW-Cost can be transformed to an optimization instance of Continuous-SPW with costs associated to each arc. We will then show how we can adapt an algorithm for the Continuous-SPW to solve this optimization version of Continuous-SPW.

Let us consider an SPW-Cost instance H as described previously. We construct a continuous instance $G = (V_G, A_G, T, D_G, C_G, s, t)$ of the optimization version of Continuous-SPW in the following steps.

We take the graph in H and subdivide each arc $e_{ij} = (v_i, v_j)$ in A_H into paths of length 2; that is, a new vertex v_{ij} is added along each arc e_{ij} such that arcs $(v_i, v_{ij}), (v_{ij}, v_j)$ are added in A_G with corresponding distances $d_{ij} - 1$, 1 and costs c_{ij}, 0, respectively. The time window for v_{ij} is [0, max time]. In addition, a loop (v_{ij}, v_{ij}) with cost 0 and distance 1 is added to v_{ij} (see Figure 7.1).

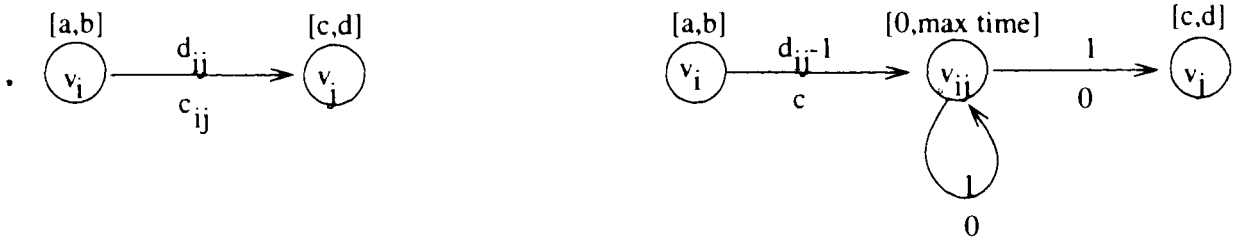


Figure 7.1: Waiting at a Vertex Simulation

This subdivision essentially allows the simulation of a wait at vertex v_{ij} for the Continuous-SPW optimization version before v_j's opening time window by looping through v_{ij}.

We can easily see that this construction takes at most $O(m)$ time. More importantly, we can easily see that the techniques used to solve Continuous-SPW can be applied for SPW-Cost with only minor changes. Rather than optimizing the distance from vertices to the sink t, we optimize the cost from vertices to t. For example, let us take Algorithm 2 in Chapter 4 and adapt it to solve SPW-Cost.

Algorithm 7: Algorithm for Continous-SPW with costs
Input: Continuous-SPW with costs instance $G = (V, A, D, T, C, x, y)$
Output: Least cost path $c_{uy}(t)$ for $u \in V$ and start times $t \in [a, b]_u$

(1) Set $c_{yy} = 0, d_{yy}(t) = 0$ for all $t \in [a, b]_y$.
(2) Set $c_{uy} = +\infty, d_{uy}(t) = +\infty$ for all $u \in A, t \in [a, b]_u$.
(3) Set queue $Q = \{y\}$.
(4) **while** $Q \neq \phi$
(5) $v \leftarrow$ dequeue Q
(6) **foreach** vertex u adjacent to v
(7) **foreach** t in $[a, b]_u$
(8) compute $c_{uy}(t) = \min\{c_{uy}(t), c_{uv} + c_{vy}(t + d_{uv})\}$, $d_{vy} \in D, c_{uv} \in C$
(9) $d_{uy}(t) = d_{uv} + d_{vy}(t + d_{uv})$ if $c_{uv} + c_{vy}(t + d_{uv}) > c_{uy}$
(10) enqueue u to Q

We see that in Step 8 the least cost (u, y)-path, $u \in V$ determines the choice of the distance from u to y.

In fact, we can generalize the problem to multiple resource optimization as it is done in [Des88]. Details of this generalization using Continuous-SPW is left open for future research.

Chapter 8

Conclusion

In this thesis we have obtained a number of results for a new problem, the continuous shortest path problem with time windows. We showed that this problem was motivated by the fact that the simpler version appeared as a subproblem of vehicle routing problems with time windows. Due to the economic implications in the area of transportation systems, scheduling, and factory automation, there has been extensive research in the vehicle routing problems, despite its complexity.

8.1 Summary of Results

The main results are summarized below.

1. In Chapter 3, we showed that Continuous-SPW is **NP**-hard for general graphs, bipartite graphs, planar graphs, grid graphs, and finally monotone grid graphs.

2. In Chapter 4, we developed a number of sequential pseudo polynomial time algorithms for Continuous-SPW when the input values are bounded.

3. In Chapter 5, we developed a parallel pseudo-polynomial time algorithm for Continuous-SPW.

4. In Chapter 6, we showed that polynomial time algorithms for Continuous-SPW exists when either the upper bound or the lower bound of time windows are relaxed. We showed that any optimization of these relaxation to approximate the actual Continuous-SPW is NP-hard.

5. In Chapter 7, we described extensions to algorithms for Continuous-SPW to solve the cost optimization version of Continuous-SPW. The resulting algorithms can then clearly be used to handle the simpler cost optimization version of SPW where the continuity requirement is dropped.

8.2 Summary of Open Problems

Some problems that emerged during our study of Continuous-SPW were left unanswered. Below is a list of open problems:

1. In Chapter 4, we constructed a pseudo-polynomial time algorithm for the bounded distance version of Continuous-SPW. We would like to see an implementation and the running time analysis of the algorithm in comparison with the pseudo-polynomial time algorithms for the bounded time windows version.

2. We would like to see a parallel algorithm for the bounded distance version of Continuous-SPW as we have done in Chapter 5.

3. We would also like to see a even more efficient pseudo-polynomial time algorithm for the Knapsack Problem by applying of the techniques of Chapter 4 used to solve the linear Diophantine equations (4.1).

4. In Chapter 7, we showed how the algorithms for Continuous-SPW can be extended to handle the optimization problem of Continuous-SPW with single resource constraint — time window constraint. We would like to generalize the extension to optimization problems with multiple resource constraints.

5. We would also like to see other solution methods to develop efficient approximation algorithms — for example randomized algorithms.

6. Another variatiant to Continuous-SPW that is very much of interest and in active research is the *Continuous Shortest Path Problem with Periodic Time Window Constraints (CSPPW)*. As the name implies, the time windows in CSPPW are period.

Bibliography

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley Publishing Company, 1974.

[Akl89] Selim G. Akl. *The Design and Analysis of Parallel Algorithms.* Prentice Hall, 1989.

[BM76] J. A. Bondy and U. S. R. Murty. *Graph Theory with Applications.* North-Holland, 1976.

[Cha92] Pranay Chaudhuri. *Parallel Algorithms, Design and Analysis.* Prentice Hall, 1992.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, Mass., 1990.

[CT91] W. Kim Chang and J.M.A. Tanchoco. Conflict-free shortest-time bidirectional avg routeing. *INT. J. PROD. RES.*, 29(12):2377-2391, 1991.

[DDSS93] Jacques Desrosiers, Yvan Dumas, Marius M. Solomon, and François Soumis. *Time Constrained Routing and Scheduling.* North-Holland, 1993.

[Des88] Martin Desrochers. An algorithm for the shortest path problem with resource constraints. Technical report, GERAD, 1988.

[DS88] Martin Desrochers and François Soumis. A generalized permanent labelling algorithm for the shortest path problem with time windows. *INFOR*, 26(3):191–211, 1988.

[DSD84] J. Desrosiers, F. Soumis, and M. Desrochers. Routing with time windows by column generation. *Networks*, 14:545–565, 1984.

[Dud78] Underwood Dudley. *Elementary Number Theory*. W. H. Freeman and Company, 1978.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to NP-Completeness*. W.H. Freeman and Company, San Francisco, California, 1979.

[GK95] Arvind Gupta and Ramesh Krishnamurti. Parallel algorithms for vehicle routing problems. *TBA*, 1995.

[HW76] D.S. Hirschberg and C. K. Wong. A polynomial-time algorithm for the knapsack problem with two variables. *Association for Computing Machinery*, 23(1):147-154, 1976.

[Jok66] H. C. Joksch. The shortest route with constraints. *Mathematical Analysis and Applications*, 14:191-197, 1966.

[Kan80] Ravindran Kannan. A polynomial algorithm for the two-variable integer programming problem. *Journal of Association of Computing Machinery*, 27(1):118-122, 1980.

[Min75] M. Minoux. Plus court chemin avec contraintes: algorithmes et applications. In *Annales des Télécommunications*, volume 30, pages 1–12, 1975.

[Sch86] Alexander Schrijver. *Theory of Linear And Integer Programming.* John Wiley and Sons Ltd., 1986.

[Sor86] B. Sornesen. *Interactive Distribution Planning.* PhD thesis, Technical University of Denmark, 1986.

Index