8-2010

Detours admitting short paths

Reshma Koganti
University of Nevada, Las Vegas

DETOURS ADMITTING SHORT PATHS

by

Reshma Koganti

Bachelor of Electronics and Communication Engineering
Jawaharlal Nehru Technological University, Hyderabad, India
2007

A thesis submitted in partial fulfillment
of the requirements for the

Masters of Science Degree in Computer Science
School of Computer Science
Howard R. Hughes College of Engineering

Graduate College
University of Nevada, Las Vegas
August 2010

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

THE GRADUATE COLLEGE

We recommend the thesis prepared under our supervision by

Reshma Koganti

entitled

Detours Admitting Short Paths

be accepted in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science
School of Computer Science

Laxmi Gewali, Committee Chair

John Minor, Committee Member

Yoohwan Kim, Committee Member

Rama Venkat, Graduate Faculty Representative

Ronald Smith, Ph. D., Vice President for Research and Graduate Studies
and Dean of the Graduate College

August 2010

ABSTRACT

Detours Admitting Short Paths

by

Reshma Koganti

Dr. Laxmi Gewali, Examination Committee Chair
Professor of Computer Science
University of Nevada, Las Vegas

Finding shortest paths between two vertices in a weighted graph is a well explored problem and several efficient algorithms for solving it have been reported. We propose a new variation of this problem which we call the Detour Admitting Shortest Path Problem (DASPP). We present an efficient algorithm for solving DASPP. This is the first algorithm that constructs a shortest path such that each edge of the shortest path admits a detour with no more than $k-$hops. This algorithm has important applications in transportation networks. We also present implementation issues for the detour admitting shortest path algorithm.

ACKNOWLEDGMENTS

I am extremely happy to take this opportunity to acknowledge my debts and gratitude to those who were associated with the preparation of this thesis. Words fail to express my profound regards from the inmost recess of my heart to my advisior Dr. Laxmi Gewali for the invaluable help, constant guidance and wide counseling extended to me right from the selection of the research topic to the successful completion of this thesis. Academic assistance apart, I would like to thank him sincerely for his infinite patience, and he was extremely generous with his time which was the driving force for me to successfully complete this thesis.

I am extending my sincere thanks to Dr. John Minor, Dr. Yoohwan Kim, Dr. Rama Venkat for their direct and indirect contribution throughout this investigation. Finally and most importantly, I thank my dear friend Chandu Venu, parents, Mr. Ratnalu Koganti and Mrs. Jaya Lakshmi Koganti and my brother, Praveen Kumar Koganti for their love and support.

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGMENTS iv

LIST OF FIGURES vi

LIST OF ALGORITHMS vii

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 LITERATURE REVIEW 3
Standard Shortest Path Algorithms 3
Faster All-Pair Shortest Path Algorithms. 4
Turn-Angle Constrained Shortest Paths 7
Visibility Constrained Shortest Paths 8

CHAPTER 3 DETOURS ADMITTING SHORTEST PATHS 11
Problem Formulation 11
Recognizing Detour Admitting Paths 13
Constructing Detour Admitting Shortest Path 17
Non-Overlapping k-Detour Admitting Paths 21

CHAPTER 4 IMPLEMENTATION 26
Dijkstra's Shortest Path Interface 26
Interface description 26
Program menu items 27
Shortest Path Generation 28
k-Bounded Detour Interface 31
Interface Description 32
Breadth First Search Tree (BFS) generation 34
Detour Generation 35

CHAPTER 5 CONCLUSION 38

REFERENCES 40

VITA 42

LIST OF FIGURES

Figure 1 Dijkstra's approach for Shortest Path extension. 5
Figure 2 All fringe edges cab be LSP. 6
Figure 3 Transform G to G′. 8
Figure 4 Illustrating a Shortest Watchman Path. 9
Figure 5 Illustrating shortest Watchman Route Inside a Rectilinear Polygon. 10
Figure 6 Illustrating a shortest s-t path in a geometric graph 13
Figure 7 Execution Trace of BFS 14
Figure 8 Illustrating the proof of Lemma 3.1 17
Figure 9 Showing edges not admitting 5-detour 18
Figure 10 Reduced graph and the shortest path that admits 5-detour 20
Figure 11 A detour can overlap with the shortest path 21
Figure 12 Non-overlapping shortest path tree 21
Figure 13 Illustrating partially built short path tree by Dijkstra's algorithm . 22
Figure 14 GUI Layout 27
Figure 15 The Initial Display of GUI for graph construction. 28
Figure 16 Snap-Shot of File-menu pull down. 29
Figure 17 Prompting user for File selection. 29
Figure 18 GUI Displaying Geometric Graph. 30
Figure 19 Display of Shortest Path Tree and the corresponding Graph. 30
Figure 20 Snap-shot of Export to XFig format. 33
Figure 21 Interface for k-detour computation. 33
Figure 22 Snap-shot of Bounded BFS-tree 34
Figure 23 Generation of Bounded BFS Tree 36
Figure 24 A shortest path not admitting 5 hop-detour. 36
Figure 25 A shortest path admitting 8hop-detour. 37

LIST OF ALGORITHMS

Algorithm 1 Dijkstra's Shortest Path 4
Algorithm 2 k-Hop Detour Algorithm 16
Algorithm 3 OutputDetour 17
Algorithm 4 Detour Admitting Shortest Path Algorithm 19
Algorithm 5 Non-Overlapping Detour Admitting Short Path Algorithm . . . 24

CHAPTER 1

INTRODUCTION

The problem of computing shortest paths in a weighted graph has been considered by many researchers [9, 10, 11, 15]. Algorithms for computing the shortest path have been reported for the last fifty years [11]. Dijkstra published the first provably correct algorithm [11] for solving the shortest path problem. Many variations of the shortest path problems have been suggested [1, 5, 9, 16, 26]. In almost all such variations, Dijkstra's concept of relaxation has been used as an important ingredient. Use of sophisticated data structures have been suggested [8] for speeding up the time complexity of Dijkstra's algorithm.

In real world applications, there are limitations of the shortest path itself which can be briefly listed as follows.

- The shortest path may not be reliable in a dynamically changing environment. If an edge e of the shortest path connecting source vertex s to target vertex t is broken unexpectedly then the path obtained by replacing the broken edge e by a detour could be very long. In some situations the detour for some edges of the shortest path may not exist.

- For applications in trajectory planning for aerial vehicles, a shortest path may not be acceptable. If two consecutive edges of the shortest path make a sharp angle, then the implied turn angle between them could be very tight and may not be feasible as a trajectory for aerial vehicles. What is needed is a shortest path without sharp turns.

- Even a shortest path without sharp turns may not be useful in a dynamic environment where edges can appear or disappear in an unpredictable manner. If an edge e of the shortest path disappears then it may not be feasible to recompute the new shortest path, if the time for recomputation is comparable to the time required to traverse edge e. In such situations, it is necessary to

have multiple short length paths connecting the source vertex to the target vertex.

In this thesis, we are proposing a variation of the shortest path problem with increased reliability in trajectory planning applications. We consider the detour property in addition to total length in developing the algorithm. We refer to such paths as detour admitting shortest paths. We show how to modify the standard Dijkstra's algorithm so that the resulting shortest path also admits k-detours (detour with at most k edges) for all edges of the path.

The thesis is organized as follows. Chapter two presents a review of existing variations on the shortest path algorithm that include (i) Sharp-Turn constraints (ii) Multiple shortest paths, and (iii) Update of the shortest paths. In chapter three, we present the main contributions of the thesis. We propose a polynomial time algorithm to compute a shortest path that admits k-detours for all its edges. The time complexity of the proposed algorithm is $O(|E|.(|E| + |V| + d^k_{max}) + |V|^2)$, where $|E|$ and $|V|$ denote the set of edges and the set of vertices of the input graph G. Further more, d_{max} denotes the value of the maximum degree of vertices in G. We consider another version of the detour admitting short path problem that does not allow overlap between detours and the short path. For this version we present an algorithm that runs in $O(|V||E|(d^k_{max} + |E| \log |E|))$ time. This algorithm produces paths of short lengths but is not necessarily of shortest length. In chapter four, we describe the implementation of several algorithms that include (i) Dijkstra shortest path algorithm (ii) k-bounded Breadth First Search (BFS) algorithm and (iii) Detection of k-detour algorithm. The implementation is done in the JAVA programming language with a user-friendly front-end Graphical User Interface (GUI). Finally in chapter five, we discuss extensions and further research problems related to detour admitting shortest paths.

CHAPTER 2

LITERATURE REVIEW

In this chapter we present a review of shortest path algorithms under various constraints. We first describe an overview of Dijkstra's shortest path algorithm in which the shortest path is computed without any constraint. We also address the problem of efficiently computing all-pair shortest path problems. In particular, we examine the use of the "Locally Shortest Path (LSP)"property for developing faster all-pair shortest path algorithms. We then address the problems of computing shortest paths under other constraints that include (i) Turn-Angle and (ii)Visibility properties.

Standard Shortest Path Algorithms

The problem of computing the shortest path between two nodes in a given network is a well investigated problem and several algorithms have been reported in the literature [1, 9, 10, 11, 15, 27]. The shortest path problem can be formally stated as follows.

Given: A weighted graph $G(V, E)$ and source vertex s.
Question: Find the length of the shortest path connecting s to all other vertices in the graph.

One of the most widely used algorithm for computing shortest paths is Dijkstra's shortest path algorithm [11]. This algorithm is based on the greedy paradigm [8]. The algorithm maintains shortest paths from source vertex to a selected set of vertices R. Initially only the source vertex is included in R. The algorithm picks the new vertex to include in R by examining all candidate vertices in $V - R$. The vertex that minimizes the length of shortest path from s using vertices in R as the intermediate vertices is taken as the next vertex w to add to R. This greedy process continues until all vertices are included in R. The result is the shortest path tree

rooted at source vertex s.

This algorithm can be sketched as follows:

Dijkstra's Shortest Path Algorithm

Input: Weighted graph $G(V, E)$, source nodes, number of vertices

Output: Implicit representation of shortest path tree rooted at s

Algorithm 1 Dijkstra's Shortest Path

```
1: {Cost[][] array holds weights of edges}
2: {dist[] used to record distance from s to other nodes }
3: {path[i] holds node index of the previous node}
4: {of ith vertex in the shortest path}
5: {The number of vertices n=|V|}
6: boolR[n];
7: {represent processed nodes}
8: for i = 1 to n do
9:       R[i] = false;
10:      dist[i] = cost[s][i];
11: end for
12: R[s] = true;
13: dist[s] = 0.0; {put s in R}
14: for num = 2 to n do
15:      u = 2;
16:      for i = 3 to n do
17:           if ((R[i]== false) and (dist[i] < dist[u])) then
18:                u = i;
19:           end if
20:           R[u] = true; {put u to R}
21:           for w = 1 to n do
22:                if ((R[w]==false) and (dist[w] > dist[u] + cost[u][w])) then
23:                     dist[w] = dist[u]+ cost[u][w];
24:                     path[w] = u;
25:                end if
26:           end for
27:      end for
28: end for
```

Faster All-Pair Shortest Path Algorithms.

To compute the shortest paths between all pairs of vertices using Dijkstra's algorithm, a straightforward approach is to repeat the execution of the algorithm by taking each of the $|V| = n$ vertices as the source vertex. This approach clearly

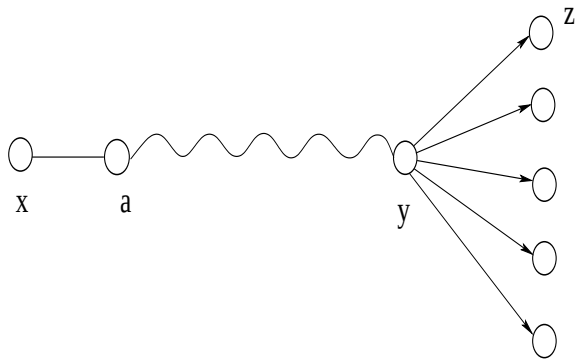

Figure 1 Dijkstra's approach for Shortest Path extension.

takes $O(n^3)$ time. Computing all pair shortest paths in time less than $O(n^3)$ (i.e $o(n^3)$) is a very interesting problem and some progress has been achieved in recent years [9, 10]. The technique presented in [9, 10] is based on the use of the optimal-substructure property of the shortest path which can be stated as follows.

OS-Property(Optimal substructure property): Every sub-path of a shortest path must be the shortest path connecting corresponding end vertices. In other words, OS-Property states that if v_{i_1},v_{i_2},... v_{i_k} is the shortest path connecting v_{i_1} to v_{i_k} then v_{i_p}, v_{i_p+1}...v_{i_r} must also be the shortest path connecting v_{i_p} to v_{i_r}, where $1 < p < r$.

Dijkstra's algorithm computes the shortest path tree in a greedy manner by adding a new vertex to the partially constructed shortest path tree. Consider the partially constructed shortest path connecting vertex x to the vertex y as shown in Figure 1, where the outgoing edges from vertex y are drawn as directed edges. We refer to these edges as *fringe edges*.

Dijkstra's algorithm examines all the fringe edges incident at y to include the next vertex. Demetrescu and Italiano [9, 10] argue that it may not be necessary to examine all the fringe edges to include as the next vertex. They argue that it is necessary to examine only those fringe edges yz such that both π_{xy} and π_{az} are the sub paths of π_{xz}, where π_{vw} denotes the shortest path connecting vw, and a is the vertex as depicted in Figure 1.

A path whose all proper sub-paths are the shortest path is called *locally shortest*

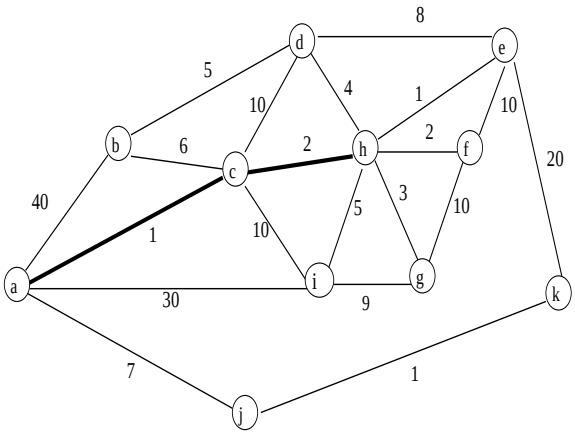

Figure 2 All fringe edges cab be LSP.

paths(LSP) [9, 10]. So, if the path obtained by adding a fringe edge is a LSP then only those fringe edges need to be considered. However, it turns out that some graphs can be constructed for which all fringe edges can appear as LSP. In Figure 2, the shortest path connecting a to h is shown. We can observe that all fringe edges incident at vertex h are LSP.

Reference [9] reports that in the worst case there can be $O(mn)$ LSP paths in some network, where m is the number of edges in the graph. Thus even if an algorithm keeps track of LSPs, the time complexity of all pair shortest path algorithm based on the Dijkstra's algorithm still remains $O(n^3)$.

Even though the number of locally shortest paths (LSP) could be $O(n^3)$ in general, there are some classes of graphs for which the number of LSPs is significantly small. As reported in [9], the number of LSPs in a road network is $O(n^2)$. Furthermore, for the network consisting of Internet Autonomous Systems, the number of LSP's is also close to $O(n^2)$.

The notion of LSP has been successfully used for developing an algorithm for maintaining all pair shortest paths in a dynamically changing networks. In a dynamically changing networks, edge weights may change as time progresses. For example, in traffic networks, if edge weights are assigned a weight reflecting the traffic flow, then the resulting network changes with time. When an edge weight

changes, recomputing the shortest paths all over again would be very time consuming. The concept of LSP has been used to update all pair shortest paths when the weights of some edges change [9].

If edge weights are allowed to only increase as time progresses then the idea of LSPs have been used effectively to develop efficient shortest path updating algorithms [9].

If the edge weights are allowed to have a sequence of weight increases, then during each update the number of paths that can stop being locally shortest are known to be $O(n^2)$ and the number of new paths that can start being locally shortest are also known to be $O(n^2)$ [9]. Using these ideas, updates of locally shortest paths can be done in $O(n^2 \log(n))$ time per edge weight increase.

Similarly updating the LSP's for a sequence of weight decrease can be done in $O(n^2 \log n)$ per edge weight decrease.

If edge weights are allowed to be a sequence of both weight increase and decrease, LSP can no longer be used directly. If the increased and decreased edge weights are intermixed, there may be $O(mn)$ changes in the set of locally shortest paths during each update.

Turn-Angle Constrained Shortest Paths

An interesting variation of shortest path problem has been introduced in [1]. This variation deals with requirements for turn angles in the shortest path. Specifically, the problem is to compute a shortest path in a geometric graph such that the turn angles between consecutive edges of the path should not be more than a certain given angle.

Angle constrained shortest paths have applications for trajectory planning of aerial vehicles. If a path has sharp turn angles then such paths can not be used as trajectories for flying aerial vehicles. In fact, most aerial vehicles can not have turn angles larger than 30° [16].

An efficient algorithm for computing angle-constrained shortest paths in geo-

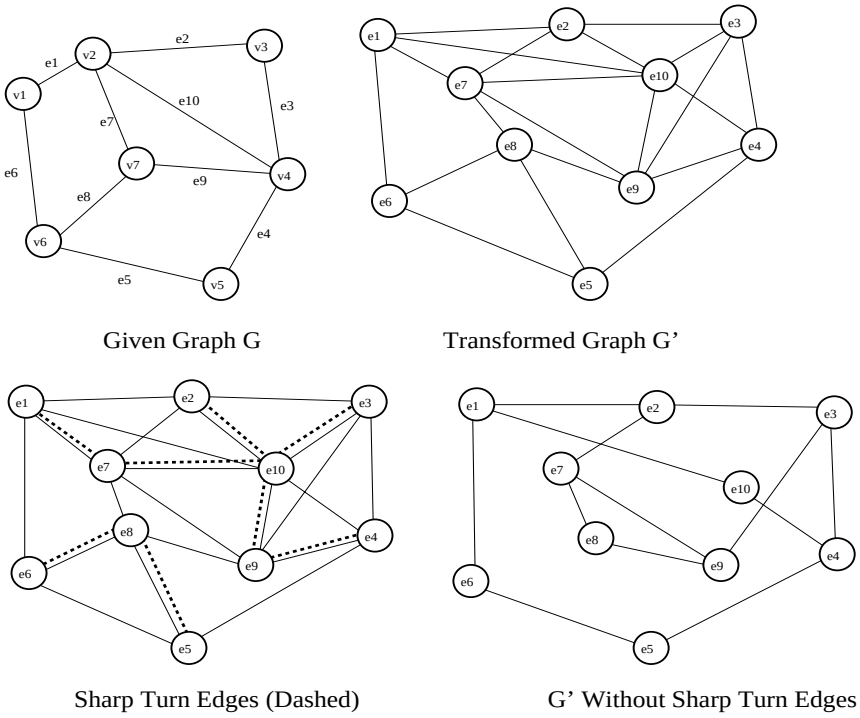

Figure 3 Transform G to G′.

metric graphs is reported in [1]. Note that in a geometric graph, edges satisfy the triangle inequality. The paper describes a graph transformation technique which can be briefly described as follows.

Given a geometric graph $G(V, E)$, a transformed graph $G'(V', E')$ is constructed such that each edge $e_i \epsilon$ E becomes a vertex of G' and two nodes in G' are connected by an edge if the corresponding edges in G' are consecutive and the implied turn angle is less than a predefined threshold value.

Angle constrained shortest paths can be obtained by applying Dijkstra's algorithm to the transformed graph. The resulting algorithm runs in $O(|E| \log |V|)$ time [1]. An example of graph transformation is shown in Figure 3.

Visibility Constrained Shortest Paths

The problem of computing shortest paths satisfying visibility requirements has been considered in robotics and computational geometry [2, 3, 4, 6, 7, 12, 13, 14, 17,

18, 19, 20, 21, 24, 23, 25]. Given a set of polygonal obstacles in two dimensions, a shortest path from which all points in the free-space is visible is called the *Shortest Watchman Path*. Note that free-space is the space not occupied by the obstacles. Figure 4 shows a collection of obstacles and a shortest watchman path. The path is drawn with dashed edges.

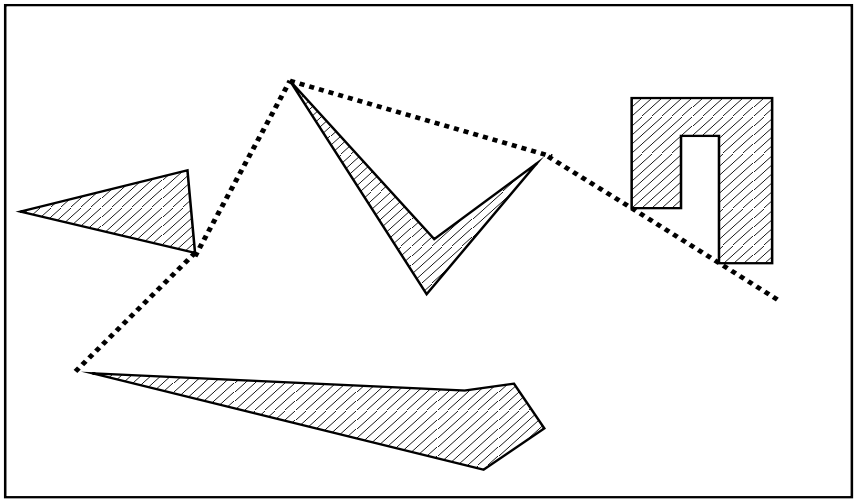

Figure 4 Illustrating a Shortest Watchman Path.

It is known that the problem of computing the shortest watchman path/route in the presence of polygonal obstacle is NP-hard [6]. Computing watchman routes (closed path) inside a polygon has also been considered [5]. It is known that the shortest watchman route inside a rectilinear polygon can be computed in $O(n \log n)$ time, where n is the number of vertices in the polygon. An example of the shortest watchman route inside a rectilinear polygon is shown in Figure 5.

Computing shortest watchman routes inside simple polygons (not necessarily rectilinear) is a much more difficult problem. The first polynomial time algorithm for constructing shortest watchman routes inside a simple polygon was reported in [5]. Other variations of the watchman route problem and the development of incremental and approximation algorithms are reported in [12, 13, 14, 17, 21]. It is noted that construction of shortest watchman paths and routes have been con-

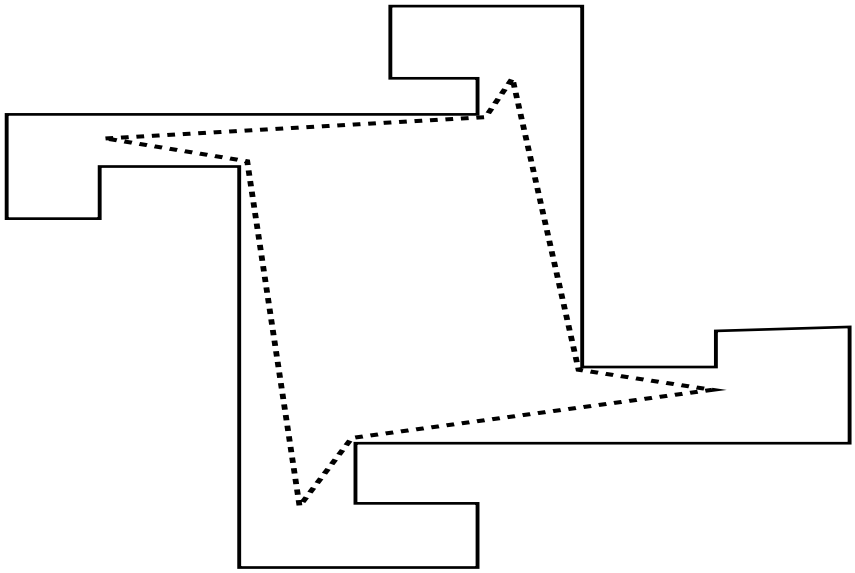

Figure 5 Illustrating shortest Watchman Route Inside a Rectilinear Polygon.

stantly investigated for the last twenty five years and same interesting results are reported in [12, 13, 14, 17, 18, 19, 20, 21, 24].

CHAPTER 3

DETOURS ADMITTING SHORTEST PATHS

In this chapter we propose a new variation of the shortest path problem that has rich applications inF transportation networks. This variation considers the construction of shortest paths satisfying detour properties. We first consider the notion of detours in a shortest path. While some paths could admit detours for all of its edges, others may not admit at all. After formulating the notion of bounded detours in a path, we proceed to the development of efficient algorithms for computing shortest paths that admit detours for all its edges. We also explore the development of detour admitting path construction algorithms such that no detour overlaps with the shortest path. This variation of the shortest path problem seems to be very hard and we propose a polynomial time algorithm for obtaining the solution.

Problem Formulation

Consider a path PT_1 = v_{i_1}, v_{i_2}, v_{i_3} ... v_{i_k} connecting vertex v_{i_1} to vertex v_{i_k} in a weighted geometric graph $G(V, E)$ with set of vertices V = $v_1, v_2, ..., v_n$ and sets of edges E = $e_1, e_2, ..., e_m$. Note that edges of a geometric graph $G(V, E)$ satisfy the triangle inequality. Figure 6 shows a path connecting source vertex $s = v_1$ to target vertex $t = v_7$. The path is drawn high-lighted with thick edges. In fact this path is also the shortest path connecting vertex v_1 to vertex v_7 in $G(V, E)$.

Definition 3.1 *A* **detour** *for an edge $e = (v_{i_k}, v_{i_{k+1}})$ is a path connecting v_{i_k} to $v_{i_{k+1}}$ that does not include edge e. In Figure 6, edge e_2 = (v_2, v_3) has 2-hop detour (v_2, v_9, v_3). The edge e_2 has many other detours including the one with 4-hops ($v_2, v_{23}, v_{22}, v_{24}, v_3$). On the other hand edge (v_5, v_6) does not have any detour with less than 8 hops.*

The notion of detour can be extended from a single edge to a sub-path (sequence of edges) in a straightforward manner. The detour for a sub-path $P_{i,j}$ con-

necting vertex v_i to vertex v_j is a path connecting v_i to v_j that does not include any edge in the path. In Figure 6, a detour for sub-path (v_1, v_2, v_3, v_4) is given by $(v_1, v_{10}, v_9, v_{11}, v_{12}, v_4)$. In our investigation we are interested in detours with smallest number of edges. Such detours are also shortest detours if distance is measured in terms of the number of edges in the path. The shortest detour for edge (v_{i_k}, v_{i_j}) is denoted by $dt(i_k, i_j)$. Similarly the shortest detour for sub-path P_1 is denoted as $dt(PT_1)$.

The problem of computing shortest detours has important application in path planning in transportation networks. Consider the problem of planning the shortest path between two given nodes in a road network. The shortest such path can be computed by using Dijkstra's shortest path algorithm [8, 11, 22]. Such a shortest path may not be reliable in situations when an unexpected traffic-jam occurs. Suppose we have computed a shortest path P_1 for travel in a road network. If one of the edges e in P_1 ceases to be functional due to a traffic-jam or road accident then we can not use the precomputed path for the intended travel. We need to make a detour around edge e to reach the destination. For certain kinds of road networks some edges of the shortest path may not support a detour or the detour may be too long. This can be clarified by inspecting the shortest path in Figure 6. Suppose edge (v_5, v_6) breaks down in the highlighted shortest path. No detours of length shorter than 8 hops exist for this edge. In fact the shown network can be easily modified so that no detours are available for edge (v_5, v_6). This observation motivates us to consider the problem of computing the shortest path connecting two given nodes in a geometric network such that the resulting shortest path should admit (if possible) detours of short lengths for all its edges. The problem can be formally defined as follows.

Detour admitting shortest path problem (DASPP)

Given : (i) A weighted graph $G(V, E)$.

(ii) Start node s, target node t.

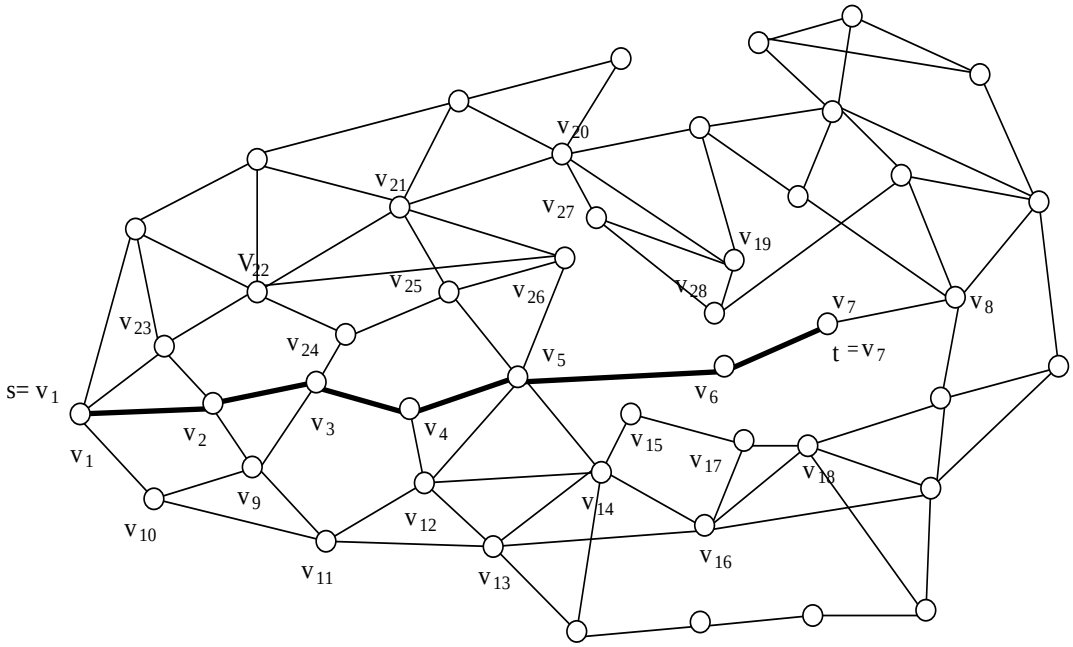

Figure 6 Illustrating a shortest s-t path in a geometric graph

(iii)An integer k.

Question : Find a shortest path connecting s to t in G such that the shortest path admits a detour of at most k hops for all its edges.

Recognizing Detour Admitting Paths

We now proceed to develop an algorithm for checking whether a given path admits k-bounded detours. For this purpose we need to use a variation of the Breadth First Search (BFS) algorithm. We start with a short overview of BFS. BFS explores a graph starting from a given source node s. The nodes are visited in the order of their shortest distance $d(v)$ from s. Note that distance $d()$ is measured in term of the number of edges. BFS first explores nodes at 1 edge away from s. Next it explores nodes 2 edges away from s. In general, nodes at distance k are explored only after visiting nodes at distance $k-1$. BFS uses a queue to record vertices at the front of the exploration. The front vertices are the ones that are at the boundary of explored and unvisited vertices. At first the source vertex is entered in the queue.

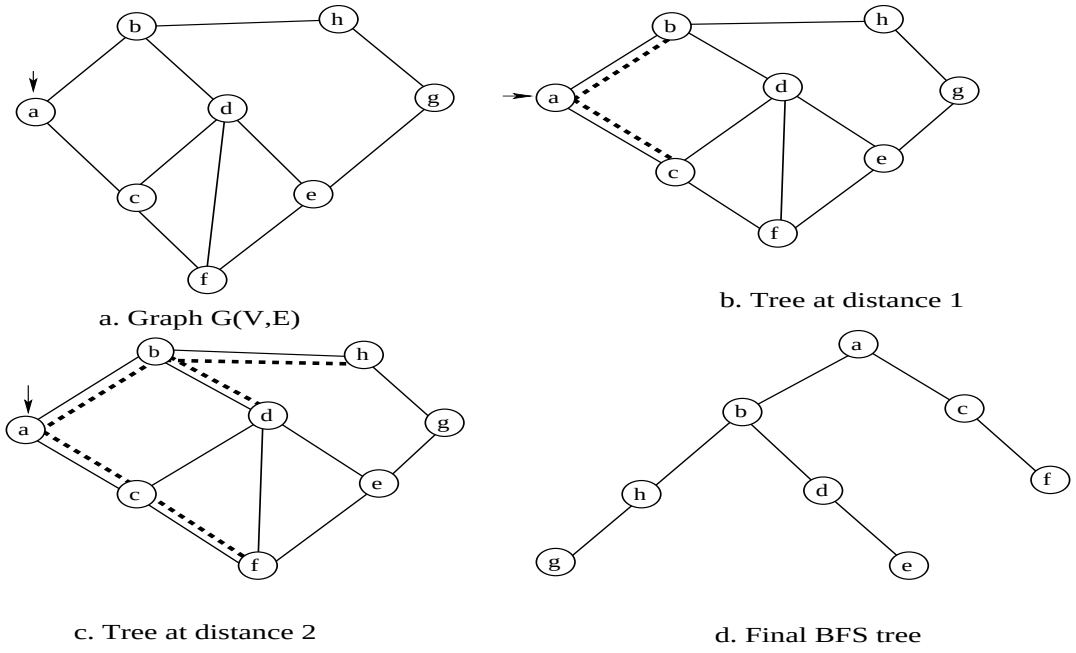

Figure 7 Execution Trace of BFS

The algorithm proceeds for exploration by taking a node v from the queue. When node v is visited, its adjacent vertices that have not been visited before are inserted into the queue. Due to the first-in-first-out property of the queue, the vertices of the graph are explored in the order of their distances from the source vertex. The application of BFS produces a exploration tree called a BFS-tree. A partial snapshot of the execution of BFS on a graph is illustrated in Figure 7 where tree-edges implicitly constructed by BFS are shown by shades. The details of BFS can be found in [8].

We can modify the standard BFS algorithm [8] to obtain a k-bounded detour detection algorithm for an edge $e = (v_i, v_j)$. We need to pass k (maximum number of allowed hops) as a parameter. When the first vertex u removed from the queue has distance from the source vertex s equal to k, the search stops. The search also stops when one of the leaf nodes in the partially constructed BFS tree is the node v_j. The algorithm takes the weighted graph G, designated edge e, and hop-count limit k as its parameters and outputs boolean value true if there is a detour of at most

k-hops for edge $e = (v_i, v_j)$. If the detour does not exist then it returns 'false'. The main ingredient of the algorithm is to grow the BFS-tree so that vertex v_j becomes one of the leaves of the constructed BFS tree. If vertex v_j is not found even after exploring k level nodes then the algorithm concludes that no k-detour exits for edge $e = (v_i, v_j)$.

All unprocessed nodes are initially marked 'white'. Distance from the root node v_i (the start vertex of edge e) to other nodes w's, discarding edge e, are maintained on the node record as $w.d$. Initially, $w.d$ is set to ∞ for all nodes other than node v_i. When search tree construction is progressing, the pointer to the parent node of w, in the partially constructed BFS tree, is recorded as $w.\pi$. The search-tree construction stops when ever vertex v_j (the end vertex of edge e) is encountered or the value of $w.d$ is greater than k. A formal description of the algorithm is listed as k-Hop Detour Algorithm (Algorithm 2).

If the execution of the k-Hop Detour Algorithm returns true then the k-detour path corresponding edge e can be extracted by following the parent vertices starting from the end vertex v_j of edge e. This is listed below as Algorithm Output Detour (Algorithm 3).

Theorem 3.1: *Given (i) a path P_1, (ii) integer k, and (iii)the underlying weighted geometric graph $G(V, E)$, we can determine whether or not path P_1 is k-detour supporting in $O(d_{max}^k\ r)$ time, where d_{max} is the maximum degree of node in G, and r is the number of edges in the path.*

Proof. We apply the k-detour algorithm for all edges of the path. If the algorithm returns true for all edges then the path is k-detour admitting. One execution of k-detour algorithm takes $O(d_{max}^k)$ time, which can be argued as follows. In the process of finding a k-detour for edge e, it first explores $O(d_{max})$ edges and each d_{max} edge has at most $O(d_{max})$ more edges to explore. Such explorations continues for at most $k-1$ levels. Adding up the total work done for all levels of exploration we come up with a geometric progression: $d_{max}^1 + d_{max}^2 + d_{max}^3 ... d_{max}^{k-1}$. This sum is

Algorithm 2 k-Hop Detour Algorithm

```
1: bool k-detour(G, e, k)
2: s = e.getStart();
3: w = e.getEnd();
4: for u ϵ G.V − s do
5:     u.color = white;
6:     u.d = ∞ ;
7:     u.π = nil;
8: end for
9: s.color = gray;
10: s.d = 0;
11: s.ϵ = nil;
12: Q= ∅;
13: Remove e from G;
14: bool detourPresent = false;
15: ENQUEUE(Q, s)
16: while Q is not empty do
17:     u = DEQUEUE(Q);
18:     if u== w then
19:         detourPresent = true;
20:         break;
21:     end if
22:     if u.d== k then
23:         break;
24:     end if
25:     for v ϵ G.Adj[u] do
26:         if v.color= white then
27:             v.color = gray;
28:             v.d = u.d + 1;
29:             v.π = u;
30:             ENQUEUE(Q, v);
31:         end if
32:         u.color=black;
33:     end for
34: end while
35: return detourPresent;
```

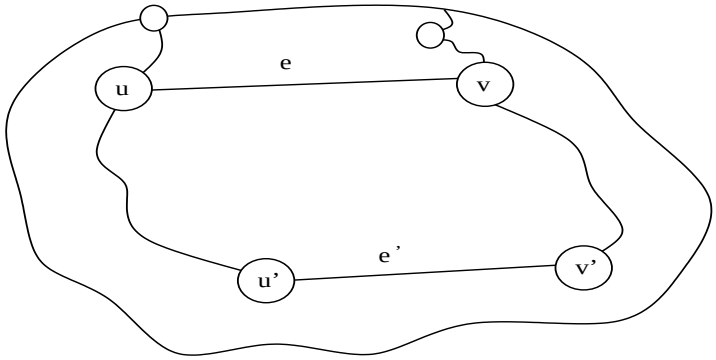

Figure 8 Illustrating the proof of Lemma 3.1

less or equal to $O(d^k)$. If the path P_1 has r edges then the total time to check detour for all edges of the path becomes $O(d^k_{max}\ r)$.

Algorithm 3 OutputDetour

```
u = e.getStart();
v = e.getEnd();
while v ≠ u do
    output(v);
    v = u.π;
end while
```

Constructing Detour Admitting Shortest Path

We observed in the previous section that all edges of a network need not support a k-detour. It is then interesting to explore that if an edge e does not have a k-detour then can this edge be in the k-detour of some other edge? The following lemma settles this inquiry.

Lemma 3.1: *If an edge $e = (u, v)$ is not k-detour supported, then e can not be in the $k-$detour of any other edge.*

Proof. Assume to the contrary that e is in the k-detour of some other edge $e^{'} = (u^{'}, v^{'})$. Then the path $u - u^{'} - v^{'} - v$ (refer to Figure 8) has at most k-hops and this becomes the k-hop-detour for edge e - a contradiction.

One approach to computing a detour admitting shortest path is to first identify

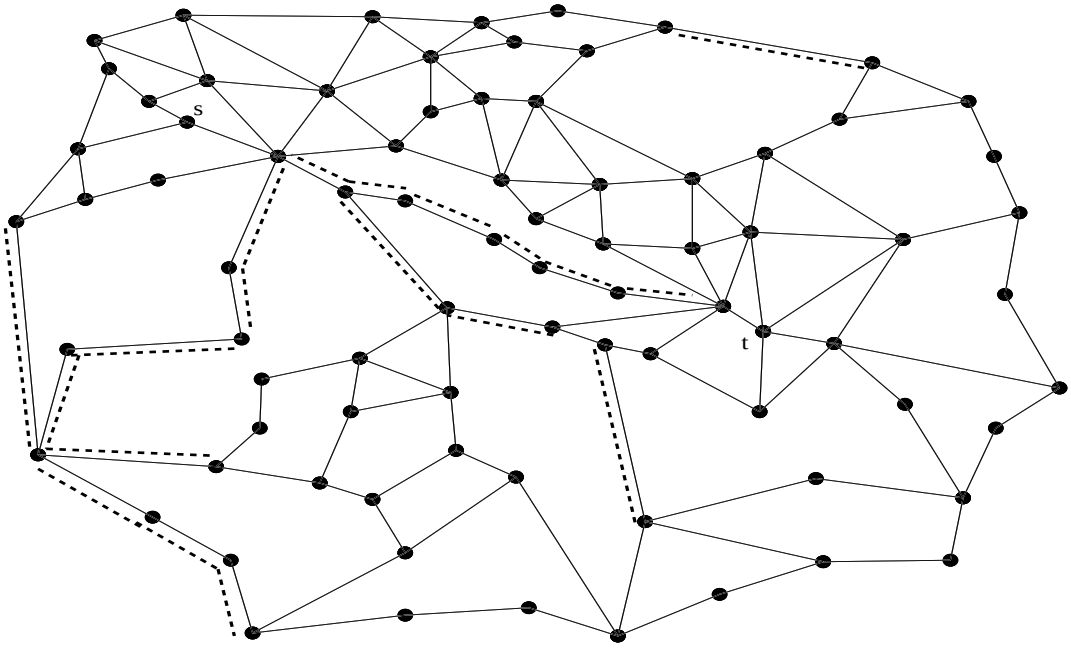

Figure 9 Showing edges not admitting 5-detour

all edges that do not support k-detours. In Figure 9, edges that do not support 5-detours are marked with dashed edges. Let $G^{'}(V^{'}, E^{'})$ denote the **reduced graph** obtained by deleting all dashed edges. It is noted that as a direct consequence of Lemma 3.1 we can safely remove such edges without compromising k-detours for other edges. We can then execute Dijkstra's shortest path algorithm on $G^{'}(V^{'}, E^{'})$ to find the shortest k-detour admitting path. Figure 20 shows the reduced graph after deleting edges not admitting 5-detours from the graph of Figure 9. The shortest path on the reduced graph admits 5-detours for all edges. In Figure 10, the shortest path from s to t admitting 5-detours is drawn with thick edges. A formal description of the algorithm based on this approach is listed as Algorithm 4.

Theorem 3.2: *Detour Admitting Shortest Path Algorithm (Algorithm 4) can be executed in* $O(|E|(|E| + |V| + d^k_{max}) + |V|^2)$ *time.*

Proof. Removal of edges not admitting a k-detour (lines 8 thru 12) takes $O(|E|(|E|+ |V| + d^k_{max})$ time. Distance initialization (lines 14 thru 17) takes $(|V|)$ time. The nested for-loops for path update (lines 20 thru 34) takes $O(|V|^2)$ time. Hence the

Algorithm 4 Detour Admitting Shortest Path Algorithm

1: ShortestPathWithDetour($int\ s$, $float\ cost[][]$, $float\ dist[]$, $int\ path[]$, $int\ n$, $int\ k$, $int\ t$)
2: {Cost[][] array holds weights of edges}
3: {dist[] used to record distance from s to other nodes }
4: {path[i] holds node index of the previous node}
5: {of ith vertex in the shortest path}
6: {The number of vertices n=$|V|$}
7: { Remove edges not admitting detour}
8: **for** edge $e_{ij}\ \epsilon\ E$ **do**
9: **if** k-detour(G,e_{ij},k) == $false$ **then**
10: $cost[i][j]$= ∞;
11: **end if**
12: **end for**
{Distance Initialization}
13: $bool\ \ R[n]$;
14: **for** $i = 1$ to n **do**
15: $R[i]$ = $false$;
16: $dist[i]$ = $cost[s][i]$;
17: **end for**
18: $R[s]$ = $true$;
19: $dist[s]$ = 0.0; {put s in R}
20: **for** $num = 2$ to n **do**
21: $u = 2$;
22: **for** $i = 3$ to n **do**
23: **if** (($R[i]$== $false$) and ($dist[i] < dist[u]$)) **then**
24: $u = i$;
25: **end if**
26: **end for**
27: $R[u] = true$; {put u to R}
28: **for** $w = 1$ to n **do**
29: **if** (($R[w]$==$false$) and ($dist[w] > dist[u]$ + $cost[u][w]$)) **then**
30: $dist[w]$ = $dist[u]$+ $cost[u][w]$;
31: $path[w] = u$;
32: **end if**
33: **end for**
34: **end for**

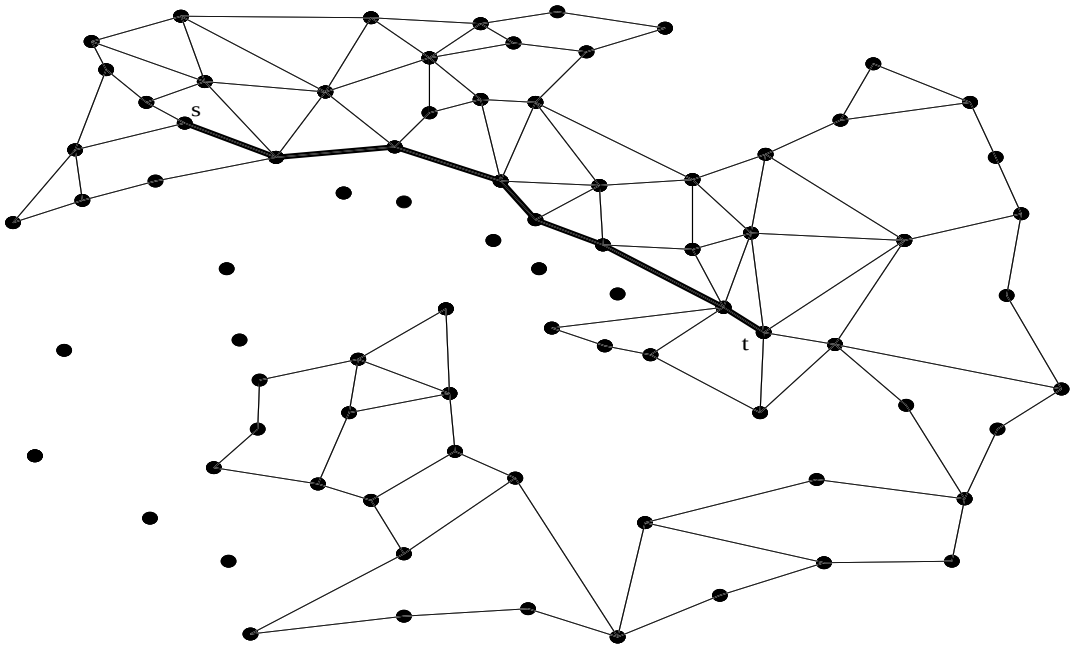

Figure 10 Reduced graph and the shortest path that admits 5-detour

total time adds up to $O(|E|(|E| + |V| + d^k_{max}) + |V|^2)$.

Theorem 3.3: *Algorithm 3 correctly computes the shortest path in which each edge admits a k-detour.*

Proof. Observe that if there is a path in the original graph G connecting s and t that admits k-detours on all its edges then that path is also retained in the reduced graph G'. This follows from the fact that only those edges are removed from G that do not admit k-detours. Further more, Lemma 3.1 confirms that the deleted edges can not be the edges of any k-detours. The fact that the path computed by G' is indeed the shortest one follows directly from the correctness of the standard Dijkstra's shortest path algorithm.

Observation 3.1: *In the solution obtained by using Algorithm 4, the k-detour of an edge could include edge(s) of the shortest path. This is shown in the Figure 11. In the figure, if edge v_2v_3 is broken, the shortest detour for this edge is $v_2v_9v_3$ which does not include the shortest path. On the other hand, the shortest detour for edge v_3v_4 is (v_3, v_2, v_8, v_4) which*

overlaps with the shortest path in edge (v_3, v_2).

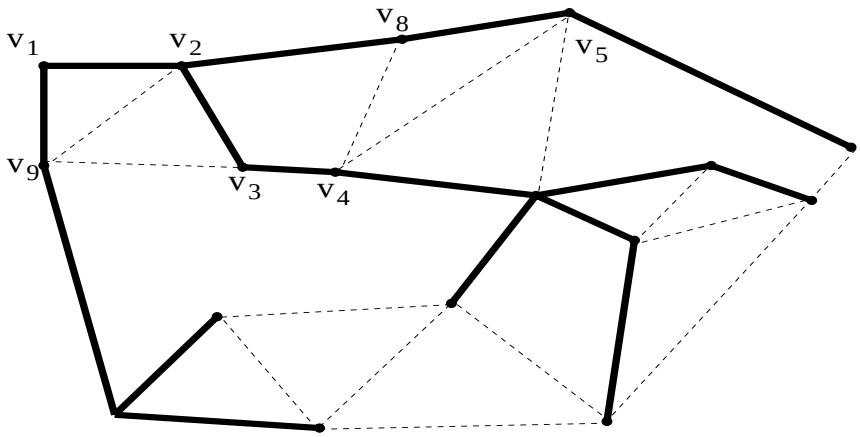

Figure 11 A detour can overlap with the shortest path

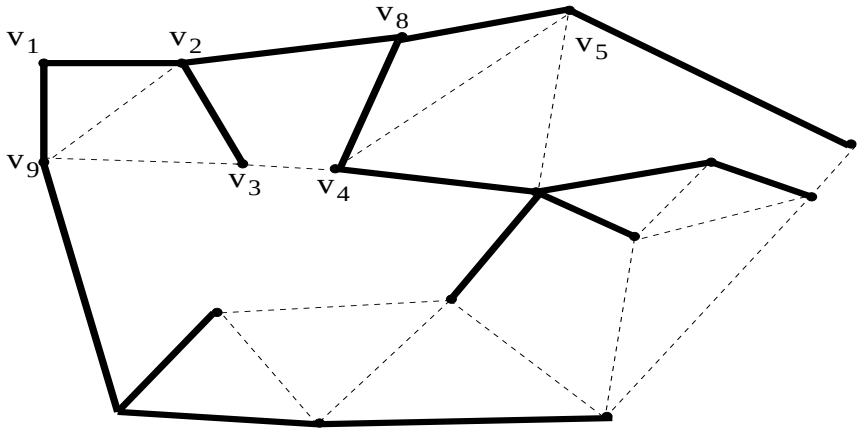

Figure 12 Non-overlapping shortest path tree

Non-Overlapping k-Detour Admitting Paths

It is very interesting to construct a short length path connecting two vertices such that the path admits k-detours and the detour for each edge of the path does not overlap with the path itself. The problem can be formally stated as follows.

Overlap-Free Detour Admitting Short Path Problem (OF-DASPP)

Given : (i) A weighted graph $G(V, E)$.

(ii) Start node s, target node t, integer k

Question : Find a short path connecting s to t in G such that the path admits a detour of at most k hops for all its edges and no detour overlaps with the path.

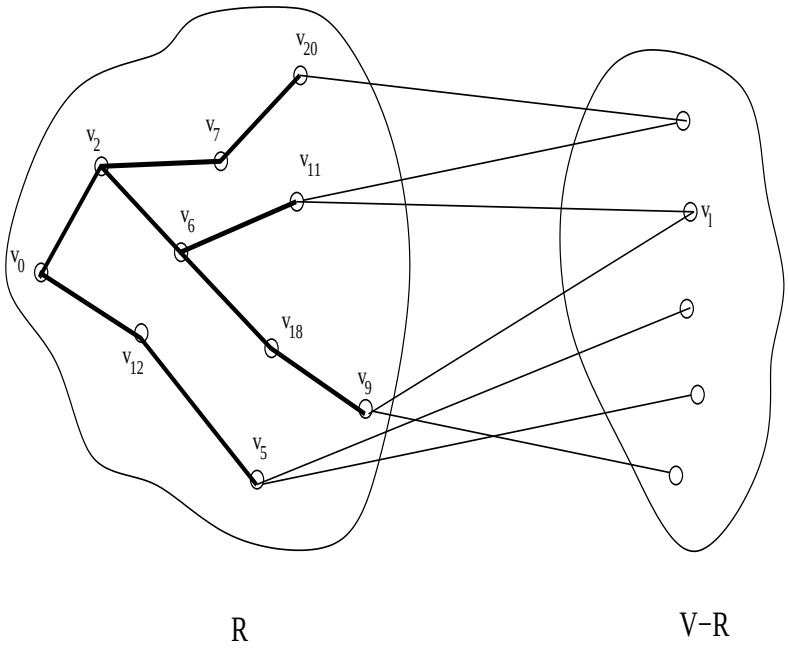

Figure 13 Illustrating partially built short path tree by Dijkstra's algorithm

Let us now explore how the Dijkstra's shortest path algorithm can be adjusted so that the k-detour for each edge does not overlap with the path. For this purpose we need to look in detail how Dijkstra's shortest path algorithm maintains its shortest path tree as nodes are processed one by one. A snap-shot of the state of Dijkstra's algorithm when a few nodes are processed can be pictured as shown in Figure 13. In the figure the set of processed nodes R are in the left side and the remaining unprocessed nodes *V-R* are in the right side. The **fringe edges** are the edges connecting vertex set R with vertex set *V-R*. While the vertex of the fringe edge in R is called its**start vertex**, the one in *V-R* is called its **forward vertex**. The set of processed nodes are arranged in a shortest path tree rooted at the source node s. To update the shortest path tree in R, Dijkstra's algorithm picks node w in *V-R* that has the smallest distance from source vertex s for all choices of w in

V-R. The fringe edge corresponding to the smallest distance node w in *V-R* is refereed to as the **light edge**. To grow the partially constructed shortest path tree, Dijkstra's algorithm adds the light edge, and updates distances to the remaining vertices in *V-R*. But the selected light edge may not support a k-detour. This means we need to check the candidate fringe edge e for overlap with edges in the shortest path from the source node s to the leaf node corresponding to e. The fringe edge that minimizes the distance to node w in *V-R* and such that its k-detour does not intersect with the corresponding shortest path is called a **non-overlapping** fringe edge.

To develop an algorithm to solve OF-DASPP, we modify the standard Dijkstra's algorithm to essentially reject those fringe edges that overlap with the partially constructed shortest path. The fringe edges are examined in the increasing order of its distance to the forward node of the fringe edge from the source node. If a fringe edge does not admit a k-detour or the k-detour overlaps with the shortest path, then it becomes **invalid**. When a fringe edge E_{uy} (u is the start node and y is the forward node) is selected we first compute its shortest detour which we denote by $path1$. Next we extract shortest path $path2$ from s to u by following the parent nodes from u to s. The two paths $path1$ and $path2$ are examined for any overlapping edges. If there is no overlap between them then the fringe edge e_{uy} is added to the partially constructed shortest path tree to grow it. If all fringe edges are found invalid then the solution does not exist. A formal sketch of the algorithm is listed as Algorithm 5 (Non-Overlapping Detour Admitting Short Path Algorithm).

The time complexity of the algorithm can be done in a straightforward manner. Distance initialization (line 3 thru line 6) can be done in $O(n)$ time. There can be $O(|E|)$ edges in the set of fringe edges for each instance of partial shortest path tree. Hence one execution of line 10 takes $O(|E| \log |E|)$ time. Computation of k-detour (line 14) needs $O(d^k_{max})$ time. Overlap between two paths (line 17) takes $O(|E| \log |E|)$ time. Time for shortest path extraction (line 16) takes $O(|E|)$ time.

Algorithm 5 Non-Overlapping Detour Admitting Short Path Algorithm

```
1: ShortestPathWithDetour(int s, float cost[][], float dist[], int path[], int n)
   {Distance Initialization}
2: bool R[n];
   {Initialize distance from source node s to other nodes}
3: for i = 1 to n do
4:     R[i] = false;
5:     dist[i] = cost[s][i];
6: end for
7: R[s] = true; {put s in R}
8: dist[s] = 0.0; {Distance from source node to itself is zero}
9: for num = 2 to n do
10:    Let L be the list of fringe edges sorted by distance to forward vertices
11:    bool found = false;
12:    for j = 1 to L.size() do
13:        e_uy = L[j];
14:        path1 = k-detour(G, e_uy, k); {obtain detour-path}
15:        if (path1 != null) then
16:            path2 = pathToRoot(y); {shortest path to node y}
17:            if (not overlap(path1, path2)) then
18:                found =true; {path1 and path2 do not overlap}
19:            end if
20:        end if
21:    end for
22:    if found then
23:        R[u] = true; {put u to R - k-detour is present}
24:        for w = 1 to n do
25:            if ((R[w]==false) and (dist[w] > dist[u] + cost[u][w])) then
26:                dist[w] = dist[u]+ cost[u][w]; {update shortest distance}
27:                path[w] = u; {record parent node}
28:            end if
29:        end for
30:    else
31:        stop; {Solution does not exist}
32:    end if
33: end for
```

Thus total time for the first inner-for-loop is $O(|E|(d_{max}^k + |E| \log |E|)$. Time for the second inner-forloop is $O(n)$. The total time for the entire algorithm adds up to $O(n|E|(d_{max}^k + |E|log|E|))$. Hence we have the following theorem.

Theorem 3.4: *Overlap-Free Detour Admitting Short Path can be computed in* $O(n|E|(d_{max}^k + |E| \log |E|))$ *time.*

It is noted that the above algorithm produces a path of short length but it does not guarantee that the path is shortest.

CHAPTER 4

IMPLEMENTATION

This chapter describes an implementation and study of the construction of shortest paths satisfying detour properties. The programs were implemented in Java,Version 1.5.

The implementation of the Detour Admitting Shortest Paths problem is carried out in two stages. In the first stage, Dijkstra's Shortest Path Algorithm is applied to the graph $G(V, E)$, while the second stage deals with the shortest path admitting $k-$bounded detour by using a variation of the Breadth First Search (BFS) algorithm.

Dijkstra's Shortest Path Interface

The implementation is done by permitting the user to generate a graph or read any pre-designed graph from a file consisting of V vertices and E edges. The graph can be edited by adjusting edges and vertices. The source vertex can be fixed at any vertex in the graph $G(V, E)$. Once the graph is finalized, the user can execute the program to generate the Dijkstra's shortest path from the selected source vertex. The shortest path tree (SPT) rooted at the source vertex can be displayed.

Interface description

The main Graphical User Interface (GUI) window is implemented by extending the JFrame class component in javax.swing which consists of three panels, as shown in Figure 14. The menu bar panel is added to the JFrame on the top, which contains the File menu. All other panels contained within the JFrame object are constructed by using the JPanel class. The whole panel is classified as left, center and right, where the left panel contains the radio buttons that are used to select the color of the graph. It consists of three colors red, blue and green. The center

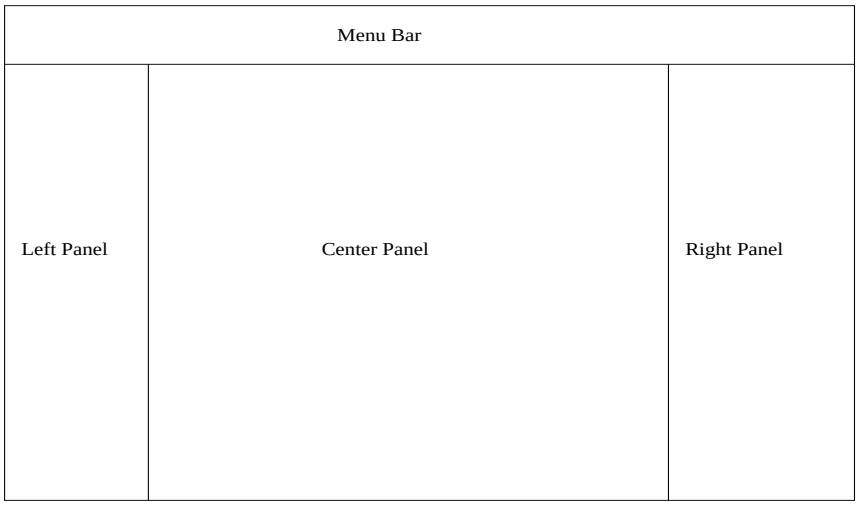

Figure 14 GUI Layout

panel contains the main display area where the graph is drawn and manipulated. Finally, the right panel contains the check boxes and buttons which are used to select and manipulate the edges and vertices of the graph. Initially, a simple planar graph with only one face (triangle) is displayed. Figure 15 shows the actual GUI as presented to the user at the start of the program. The initial triangle planar graph can be grown to a bigger planar graph by adding a sequence of edges, by splitting edges and by splitting faces. Edge addition, edge split and face split can be done one at a time. For convenience, we highlight the currently selected face.

There are several checkboxes and buttons present to manipulate and generate the graph. We will describe the functionality of each checkbox and button in Table 1 and Table 2 respectively.

Program menu items

A File tab is represented as a menu item in the program. The File menu items enable the user to (i) read and open previously saved graph files, (ii) save a generated graph to a file and (iii) export the generated graph to the file in $.eps$ format. A brief description of the File items is provided in Table 3. Figure 16 and Figure 17

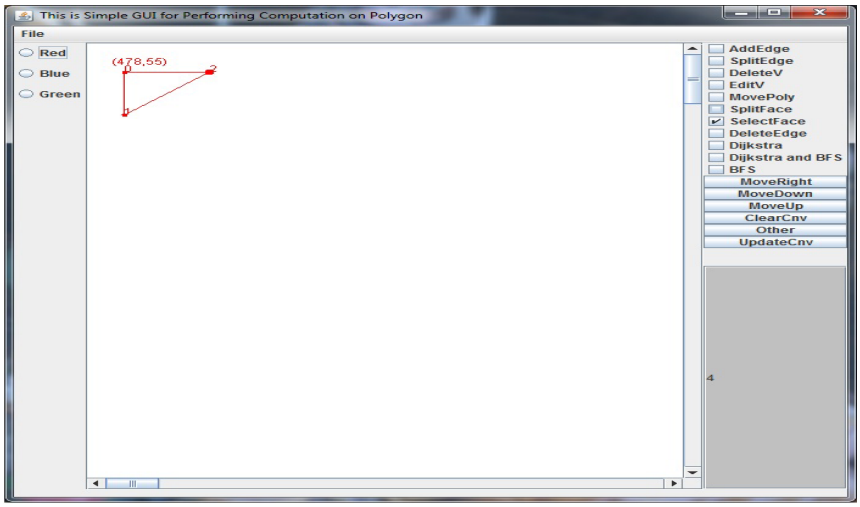

Figure 15 The Initial Display of GUI for graph construction.

show the GUI representation of the File menu item and selection panel to choose or save the graph $G(V, E)$ respectively.

Shortest Path Generation

The shortest path tree of the generated or selected graph can be produced by selecting source vertex s. The program generates all shortest paths from the selected source vertex s to all other vertices. The shortest path tree is generated by applying Dijkstra's Shortest Path Algorithm as discussed in Chapter 2. The resulting shortest paths are highlighted with thick solid edges while the original graph is shown as dashed lines. Figure 18 and Figure 19 show the GUI presented for the user in which the Dijkstra checkbox is enabled and the Shortest Path Tree (SPT) is generated. Note the tree generated is in a new GUI Frame which contains a file Menu option "Export to XFig" which brings up a file save panel so that the user can export the shortest path tree generated in a eps file format. This is shown in the Figure 20.

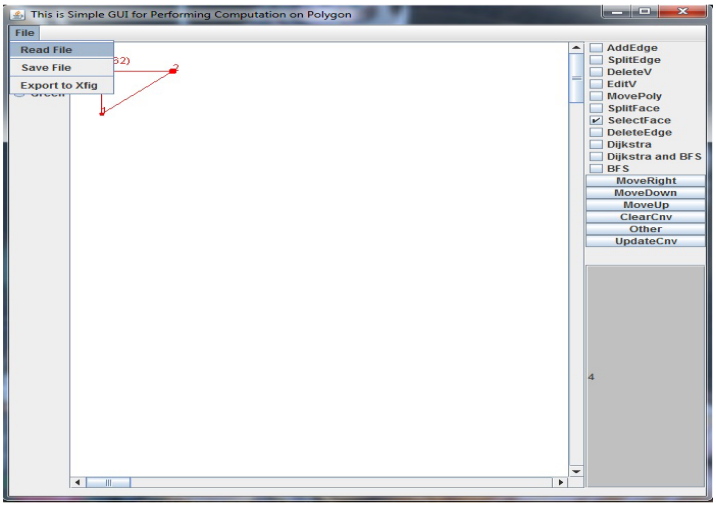

Figure 16 Snap-Shot of File-menu pull down.

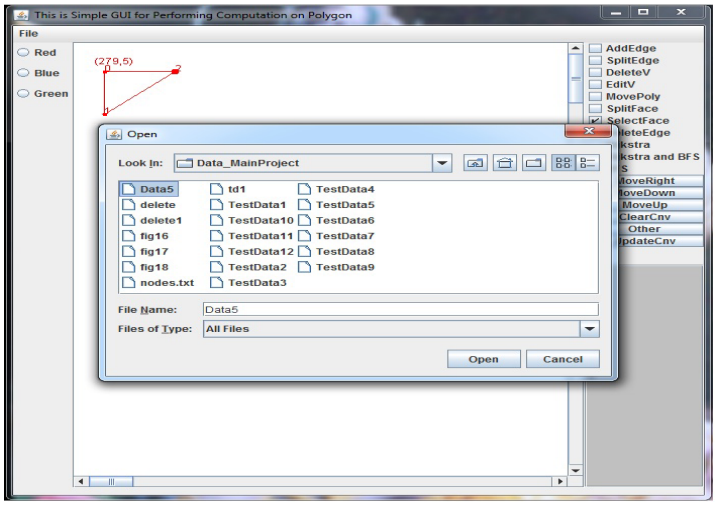

Figure 17 Prompting user for File selection.

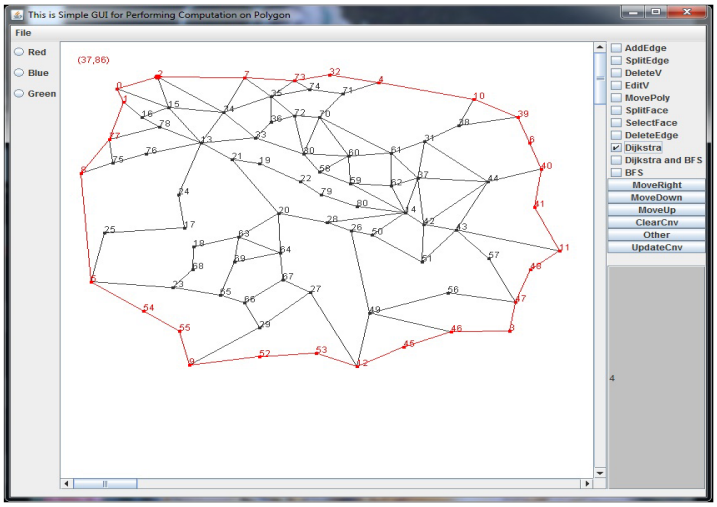

Figure 18 GUI Displaying Geometric Graph.

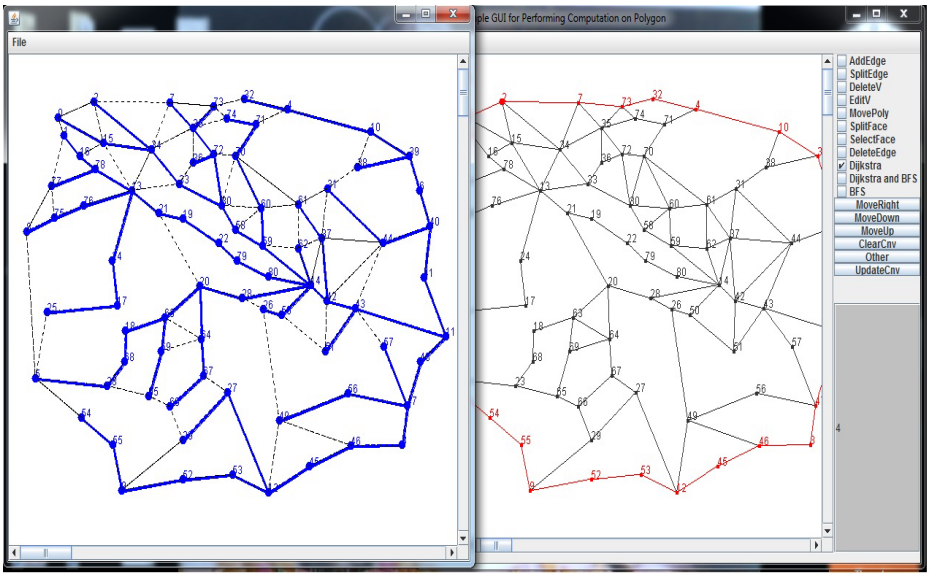

Figure 19 Display of Shortest Path Tree and the corresponding Graph.

Table 1 CheckBox description.

1	Add Edge	Add an edge to any vertex v_i of a selected face of the graph. It also shows a pre-drawn edge with a dashed green lines to verify the edge is in its correct position.
2	Split Edge	Splits the selected edge into two parts by generating a new vertex to the selected edge.
3	DeleteV	Deletes the selected vertex of a graph by updating the values to the connecting vertices.
4	EditV	Can move the position of selected vertex by dragging mouse.
5	MovePoly	The graph can be moved to any selected mouse cursor location on the center panel.
6	Split-Face	Used for splitting the face by joining two vertices with an edge.
7	SelectFace	Used to select the face of a graph where the mouse cursor position is placed.
8	DeleteEdge	Used to remove the selected edge from the graph.
9	Dijkstra	Used to select a source vertex by using a mouse cursor position and draws a shortest path tree from the source vertex to every vertex in the graph by using Dijkstra's algorithm in a separate GUI Frame.
10	Dijkstra with BFS	Used for (i) Selecting source vertex s, (ii) Displaying shortest path tree(SPT) rooted at s, (iii) Selecting edge for displaying bounded BFS-tree and (iv) For displaying *k-detour* for the selected edge of the shortest path tree.
11	BFS	BFS tree of the graph is shown in a separate GUI Frame by selecting a root vertex by using a mouse cursor position until a maximum hop count of k is reached.

k-Bounded Detour Interface

The k-bounded detour provides an implementation that allows the user to know if the shortest path is admitting k-bounded detours. This is found by using a variation of the Breadth First Search (BFS) algorithm. The program finds a possibly overlapping $k-$detour admitting shortest path from the user specified positions of the source vertex s to the target vertex t.

Table 2 Buttons description.

1	MoveRight	Used to move the graph to the right of the center Panel by 10 pixels
2	MoveDown	Used to move the graph to the down of the center Panel by 10 pixels
3	MoveUp	Used to move the graph to the up of the center Panel by 10 pixels
4	ClearCanv	Used to clear the center Panel by deleting the graph.

Table 3 File Menu Items description.

1	Read File	Brings up a file selection panel, user can choose a pre generated graph file.
2	Save File	Brings up a file save panel, user can save a new generated file or replace an existing file.
3	Export to XFig	Brings up a file save panel. The user can save a new generated graph in eps format.

Interface Description

The *k-detour* admitting shortest path program has the same main display window as that of the previous program implementation, which is done by extending the JFrame class component in javax.swing. Figure 21 shows the program GUI which is presented to the user. The program has a checkbox created by name *Dijkstra and BFS* and a textField which accepts the maximum number of hop count in executing the BFS tree. The program's output has two different JFrame display windows where one shows the shortest path tree from the selected source vertex s to target vertex t along with bounded Breadth First Search (BFS) Trees from the selected BFS source vertex bs. The second display window shows the shortest path from s to t along with the detour from the selected source if the BFS target vertex bt is reachable within the given maximum hop.

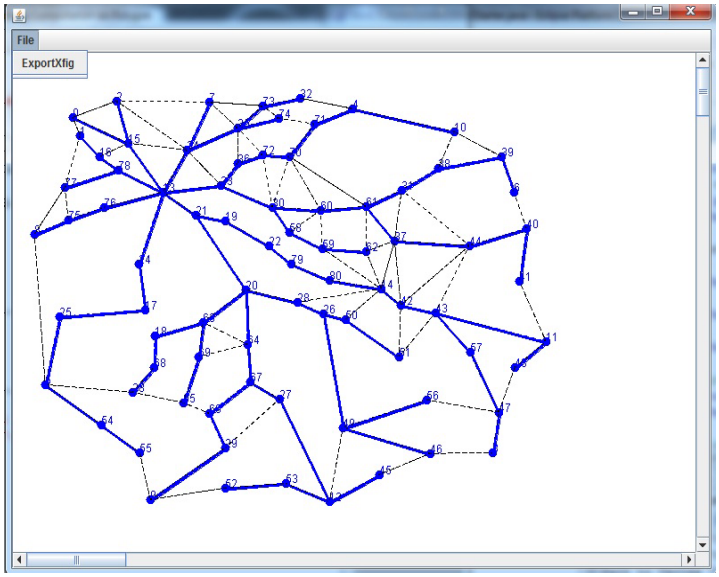

Figure 20 Snap-shot of Export to XFig format.

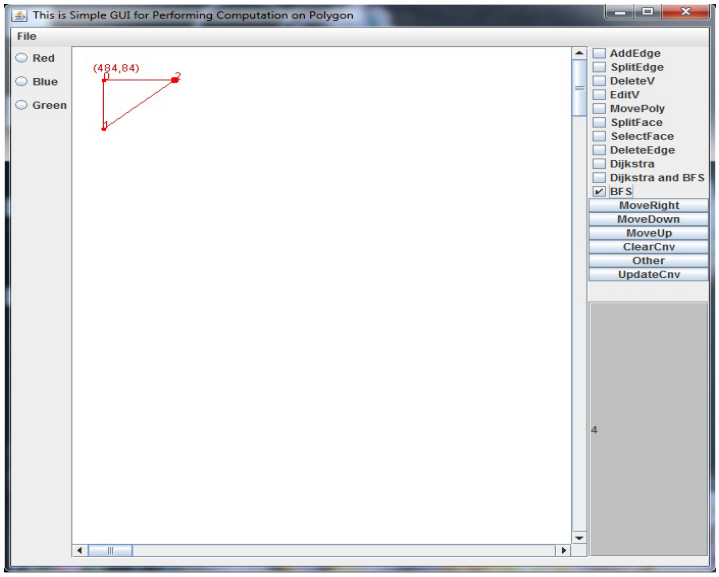

Figure 21 Interface for k-detour computation.

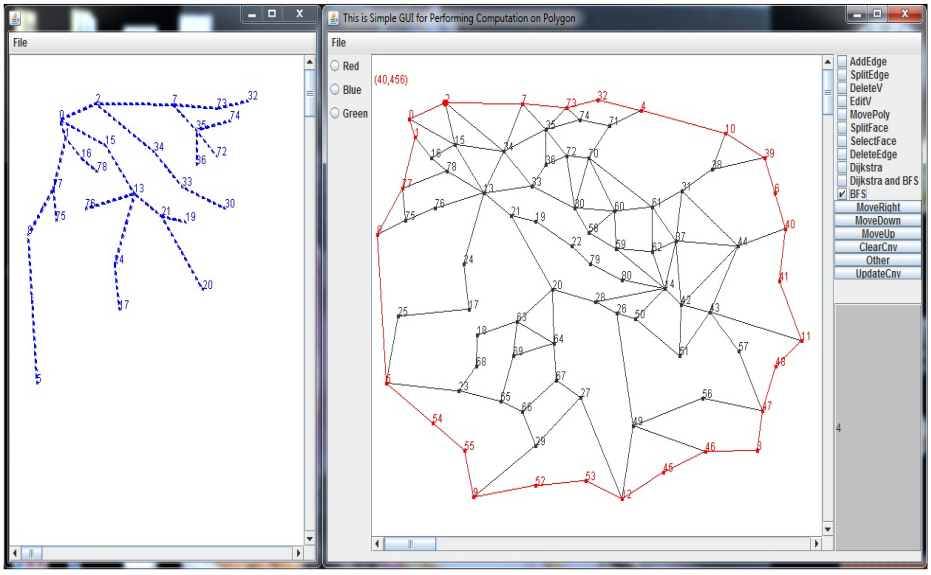

Figure 22 Snap-shot of Bounded BFS-tree

Breadth First Search Tree (BFS) generation

The variation of the Breadth First Search algorithm is implemented to recognize detour admitting paths. The BFS program is implemented by recording the source vertex s in the Queue. The program explores the graph starting from a given source vertex s taken out from the queue. When vertex s is visited, its adjacent vertices that have not been visited are inserted into the queue. The Breadth First Search Tree of the generated or selected graph can be produced by selecting a source vertex s. The BFS program has the function **public void bfs(int bfsSource)**. The function is used to accept the bfsSource as the parameter, by selecting a source vertex s, using a mouse cursor position. The BFS class has the main implementation of the BFS algorithm, which stores the BFS tree in a queue. This queue is then given to the JFrame where the BFS tree is drawn. The JFrame GUI has a single panel, which is used as a display panel on which the BFS tree is drawn. Figure 22 shows the GUI with the BFS checkbox enabled and the center display panel with BFS tree. The JFrame GUI has a File menu. The File menu has a single function *Export to XFig* that allows the user to save the generated BFS tree with the extension $.fig$.

Detour Generation

A variation of the above Breadth First Search algorithm is implemented to recognize detour admitting paths. The variation includes a maximum number of allowed hops as a parameter to the BFS algorithm. The program is implemented by using variations of two algorithms, Dijkstra's algorithm and BFS algorithm. The variations include the source vertex, the target vertex and maximum number of allowed hops for the Dijkstra's and BFS algorithms respectively. The shortest path between source vertex s and the target vertex t is found by using Dijkstra's algorithm. Similarly the detour is found from the BFS source vertex bs to the BFS target vertex bt with the variation of the above BFS algorithm. This algorithm records the source vertex in the Queue. The program explores the graph starting from a given source vertex s taken out from the queue. When vertex s is visited, its adjacent vertices that have not been visited before are inserted into the queue. When the first vertex s removed from the queue has distance from the source vertex s equal to k, the search stops. The search also stops when one of the leaf vertices in the partially constructed BFS tree is the target vertex t. The Breadth First Search Tree of the generated or selected graph can be displayed by selecting a source vertex s along with the given hop count in textField. Figure 23 shows the program interface for selecting hop-count and BFS. Figure 24 and Figure 25 show execution snap-shots of detour generation.

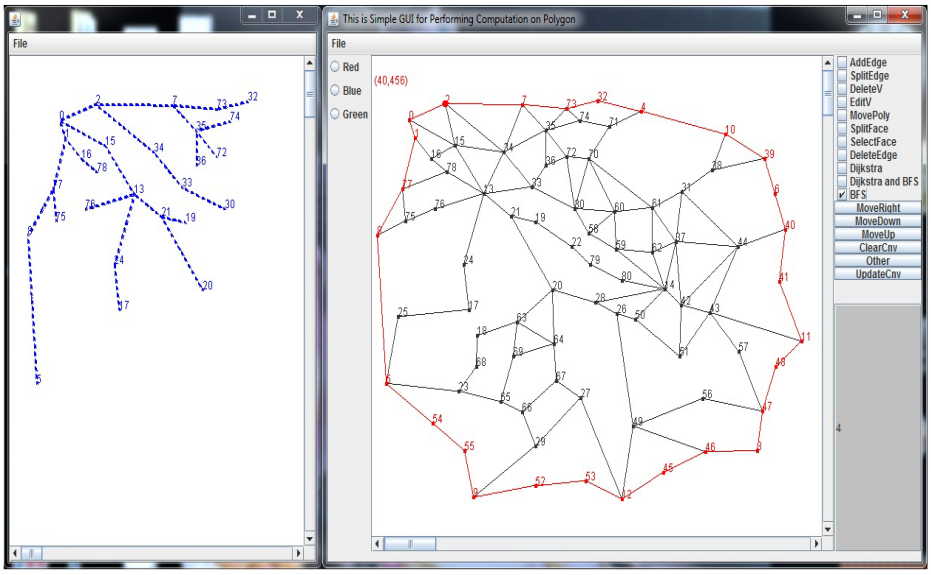

Figure 23 Generation of Bounded BFS Tree

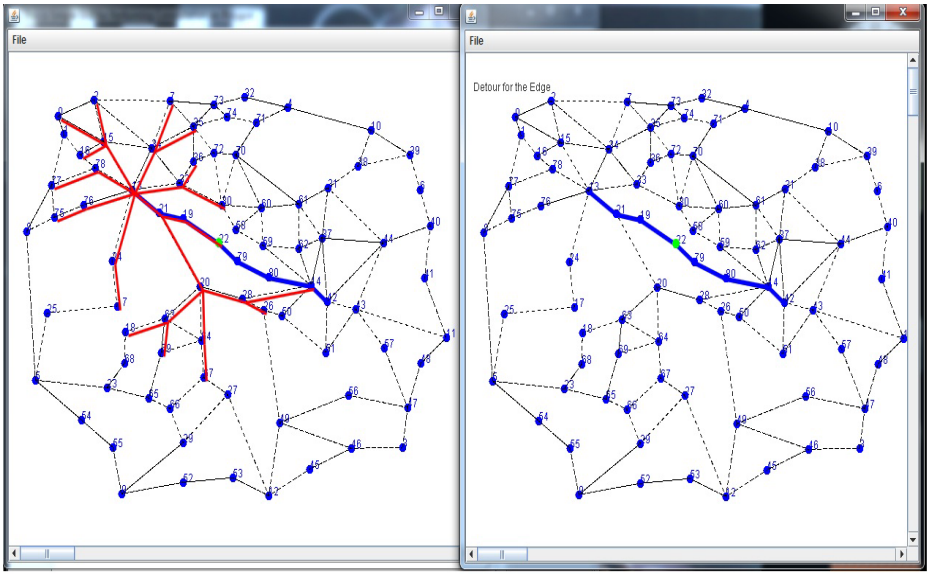

Figure 24 A shortest path not admitting 5 hop-detour.

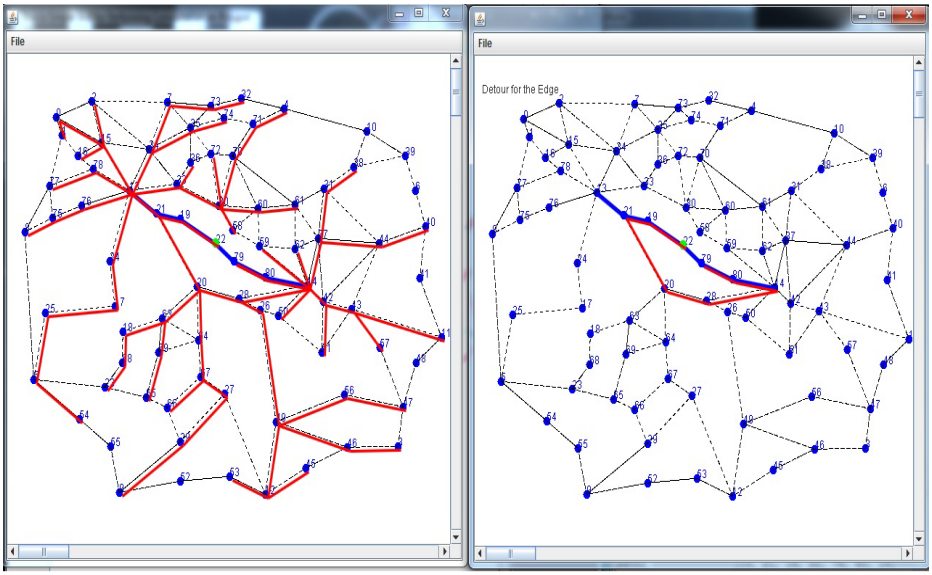

Figure 25 A shortest path admitting 8hop-detour.

CHAPTER 5

CONCLUSION

We presented a comprehensive review of the development of an efficient algorithm for solving the standard shortest path problem and approaches for updating the shortest path when some edge weights change. We reviewed the development of an efficient algorithm for solving the shortest path problem under various constraints. The constrained shortest path problems include (i) Computation of turn-constrained shortest path, (ii) Construction of multiple disjoint shortest path and (iii) Formation of shortest path under visibility constraints.

We introduced a new variation of the shortest path problem: computation of detour admitting shortest paths between two nodes in a weighted graph. The detour admitting shortest path problem seeks to construct a shortest path connecting two given vertices s to t such that each edge of the path has a detour of at most $k-$edges, where k is a given integer. To solve the detour admitting shortest path problem we first considered a variation of the standard Breadth First Search (BFS) algorithm called the $k-$bounded Breadth First Search (*k-bfs*) algorithm. It is remarked that (*k-bfs*)algorithm constructs a bfs tree whose height is no more than k. The *k-bfs* algorithm is used to detect whether or not a given path is a $k-$detour admitting path.

We presented a modification of the standard Dijkstra's algorithm for constructing a shortest path connecting two given nodes u and v such that each edge of the path admits a $k-$detour. Our approach for developing this algorithm is to first identify all edges (*singular edges*) of the graph that do not admit $k-$detours. We then applied Dijkstra's algorithm on the reduced graph obtained by removing all singular edges. We formally proved that the proposed algorithm correctly computes $k-$detour admitting shortest path.

The $k-$detour shortest path algorithm is such that the detour of the edges could

overlap with one or more edges of the shortest path. Motivated by this observation, we presented another variation of the shortest path problem called the overlap-free detour admitting shortest path problem (OF-DASPP). We present a polynomial time algorithm for solving OF-DASPP. The algorithm runs in $O(n|E|(d_{max}^k + |E| \log |E|))$ time.

We addressed implementation issues of the shortest path algorithm and its variations. We developed a program in Java that can be used to compute the shortest path, and to check for $k-$detours for the edges of the shortest path. The program has a friendly front-end GUI, so that users can construct custom geometric networks by interacting with mouse clicks and drags. For clarity of presentation, we considered only planar graphs for the implementation.

Several modifications and extensions of the proposed problems arise. The time complexity of the algorithm for solving OF-DASPP is rather high. It would be interesting to develop a faster algorithm.

We did not implement the algorithm for solving OF-DASPP. It would shed more insight on OF-DASPP, if we could experiment with the performance of the algorithm on several randomly generated weighted graphs.

A related problem that has potential applications for unmanned aerial vehicles can be stated as follows. Construct a shortest path connecting two vertices in a geometric graph such that the shortest path does not have any sharp turns and that each edge of the path admits a $k-$detour. Development of an efficient algorithm for solving this problem would be interesting.

Furthermore, in our implementation we only considered planar graphs. It would be fruitful to study the performance of proposed algorithms by implementating them on non-planar graphs.

REFERENCES

[1] Boroujerdi, A. and Uhlmann, J. *An efficient algorithm for computing least cost paths with turn constraints*. Journal of Information Processing Letters,67, pp:317-321 ,1998.

[2] Carlsson, S., Jonsson, H. and Nilsson, B.J. *Optimum guard covers and m-watchmen routes for restricted polygons*.In Dehne, F., Sack, J.R. and Santoro, N., edt. WADS, LNCS,Vol: 519, Springer Heidelber, pp:367-378 ,1991.

[3] Carlsson, S., Jonsson, H. and Nilsson, B.J. *Optimum guard covers and m-watchmen routes for restricted polygons*. Journal of Computational Geometry Applications,3, pp:85-105 ,1993.

[4] Carlsson, S., Jonsson, H. and Nilsson, B.J. *Finding the shortest watchman route in a simple polygon*. Journal of Discrete Computational Geometry,22, pp:377-402 ,1999.

[5] Chin, W. and Ntafos, S. *Shortest Watchman Routes in Simple Polygons*. Journal of Discrete Computational Geometry,6, pp:9-31 ,1991.

[6] Chin, W. and Ntafos, S. *Optimum watchman routes*. Journal of Information Processing Letters,28, pp:39-44 ,1999.

[7] Chin, W.P. and Ntafos, S. *Optimum Watchman Routes*. Journal of Proceedings of the ACM Symposium on Computational Geometry, pp:24-33, 1985.

[8] Cormen, T.H. , Leiserson, C.E., Rivest, R.L. and Stein, C. *Introduction to Algorithms*. McGraw-Hil, 2001.

[9] Demetrescu, C. and Italiano, G.F. *Engineering Shortest Path Algorithm*. Journal of Lecture Notes in Computer Science 3059, pp:191-198, 2004.

[10] Demetrescu, C. and Italiano, G.F. *Experimental Analysis of Dynamic All Pairs Shortest Paths Algorithm*. Journal of ACM Transactions on Algorithms Vol.2, 4, pp:578-601, 2006.

[11] Dijkstra, E.W. *A Note on two Problems in Connection with Graphs*.Journal of Number-Mathematics, pp:267-271, 1959.

[12] Fajje, Li. and Klette, R. *An Approximation Algorithm for Solving Watchman Route Problem*. Robvis, Lecture Notes in Computer Science, pp:189-206 ,2008.

[13] Gewali, L.P. and Ntafos, S. *Watchman routes for multiple guards*.In Proc. Canad. Conf. Comput. Geom. , pp:126-129 ,1991.

[14] Gewali, L.P. and Ntafos, S. *Watchman Routes in the presence of a pair of convex polygons*.In Proc. Canad. Conf. Comput. Geom. , pp:127-132 ,1995.

[15] James, A.S. and Reif, J.H. *Shortest path in the plane with polygonal Obstacles*. Journal of ACM, Vol:41, 5, pp:983-1012 ,September 1984.

[16] Krozel, J. and Lee, C. and Mitchell, J.S.B. *Turn-Constrained Route Planning for Avoiding Hazardous Weather*. Journal of Air Trafic Control quarterly, Vol:14, 2, pp:159-182 ,2006.

[17] Nilsson, B.J *Guarding art galleries*. Methods for mobile guards, PhD thesis, Sweden, Lund University, 1995.

[18] Nilsson, B.J. and Wood, D. *Optimum watchmen routes in spiral polygons*.In Proc. Canad. Conf. Comput. Geom. , pp:269-272 ,1990.

[19] Ntafos, S. *The robber route problem*. Journal of Information Processing Letters,34, pp:59-63 ,1990.

[20] Ntafos, S. *Watchman routes under limited visibility*. Journal of Computational Geometry,1, pp:149-170 ,1992.

[21] Ntafos, S. and Gewali, L. *External watchman routes*. Journal of Visual Computation,10, pp:474-483 ,1994.

[22] O'Rourke, J. *Computational Geometry in C*. Cambridge University Press, 1998.

[23] Tan, X. *Approximation algorithm for the watchman route and zookeeper's problem*.In Wang, J. edt. CoCOON, LNCS,Vol: 2108, Springer Heidelber, pp:201-206 ,2001.

[24] Tan, X. *Approximation algorithms for the watchman route and Zookeeper's problems*. Journal of Discrete Applied Mathematics,136, pp:363-376 ,2004.

[25] Tan, X. *A Linear-Time 2-Approximation Algorithm for the Watchman Route Problem for simple Polygons*. Journal of Theoretical Computer Science, 384, pp:92-103, 2007.

[26] Tan, X. and Hirata, T. *Constructing shortest watchman routes by divide-and-conquer*.In Ng, K.W. edt. ISAAC, LNCS,Vol: 762, Springer Heidelber, pp:68-77 ,1993.

[27] Urrutia, J. shortest paths and network optimization.In Urrutia, J. edt. Handbook of Computational Geometry, Elsevier Amsterdam , pp:973-1027 ,2000.

VITA

Graduate College
University of Nevada, Las Vegas

Reshma Koganti

Home Address:
4255 Tamarus Street Apt 270
Las Vegas, Nevada 89119

Degree:
Bachelors of Electronics and Communication Engineering
Jawaharlal Nehru Technological University
Hyderabad, India

Thesis Title: Detours Admitting Short Paths

Thesis Examination Committee:
Committee Chairperson: Dr. Laxmi Gewali, Ph.D.
Committee Memeber: Dr. John Minor, Ph.D.
Committee Memeber: Dr. Yoohwan Kim, Ph.D.
Graduate Faculty Representative: Dr. Rama Venkat, Ph.D.