Multi Constraint Shortest Path Routing

Arian Khosravi, Maryam Daneshi, Tyler Cadigan

January 4, 2009
## Contents

1 Introduction .................................................. 3  
   1.1 Motivation .............................................. 3  
   1.2 Problem Definition ................................... 4  
   1.3 Problem Formulation ................................... 4  

2 Related Work .................................................. 4  
   2.1 Multi-Constraint Shortest Path ......................... 5  
   2.2 Feedback Arc Set ...................................... 5  

3 Exact Solution for General Graphs ...................... 6  
   3.1 Dynamic Programming Approach ......................... 7  
      3.1.1 Approach 1 ....................................... 7  
      3.1.2 Approach 2 ....................................... 7  
      3.1.3 Approach 3 ....................................... 8  
      3.1.4 Final Approach .................................. 8  
   3.2 Proof of Concept ....................................... 9  
      3.2.1 DFA and Regular Expressions .................. 9  
      3.2.2 Deterministic Finite Automata ................ 9  
      3.2.3 Regular Expression ............................ 10  
      3.2.4 All Paths Problem and it’s relation to Language Theory 11  
   3.3 Time Complexity ..................................... 11  
      3.3.1 Approach 1 ....................................... 11  
      3.3.2 Final Approach .................................. 12  
   3.4 Space Complexity ..................................... 13  
      3.4.1 Approach 1 ....................................... 13  
      3.4.2 Final Approach .................................. 13  

4 Heuristic Approach ........................................... 13  
   4.1 K-Shortest Path ....................................... 14  
   4.2 Loop Removal .......................................... 14  

5 Implementation and Results .................................. 16  
   5.1 Exact Solution ........................................ 16  
      5.1.1 Network Topology ................................ 17  
      5.1.2 Weight Distribution ............................. 18  
   5.2 Heuristic Solution ..................................... 18  

6 Future Work .................................................. 19
A Appendix

A.1 What is Wrong With Approach 1?! ...................... 20
A.2 Membership of All Paths Problem in NP-Hard ............. 21
1 Introduction

Shortest path routing is an important problem in communication networks. Quality-of-Service (QoS) routing is a term used for service aware routing for path computation in data networks. In this work we look at methods for developing algorithms that compute a set of optimal paths in a directed network graph from a given source to a destination-based on several network parameters and service requirements[3].

1.1 Motivation

In communication networks, the term QoS refers to resource reservation control mechanisms. They guarantee a certain level of performance for data flow and/or application throughput. QoS guaranteeing is important if the network capacity is insufficient and all links don’t can not provide all or part of the requirements of traffic. Some of the QoS parameters are bit rate (commonly known as bandwidth), delay, jitter, packet dropping probability, etc. In the area of Traffic Engineering, QoS parameters can be categorized into three main sections: additive (e.g. delay), min/max (e.g. bandwidth) and multiplicative (e.g. loss) constraints [9]. Since multiplicative constraints are reducable to additive constraints, we may exclude them from our model. These parameters are used to represent an application’s expected lower bounds. If we look at the network as a directed multi-weighted graph, this problem can be modeled as finding the optimal path between any pair of a source and a destination with respect to these QoS parameters. Each edge in the graph represents a network link that has certain properties. We can define these properties as weights of the edges in the graph. Finding the shortest path problem for more than one additive/multiplicative weight is known to be NP-hard. Significant work has been done in finding approximate solutions for this problem. This work focuses on finding an exact algorithm for the problem by using parameterized complexity [5]. In the following sections we formally define the problem, show the work we undertook and present our findings. This includes two approaches we propose for solving the problem. In the following the section we give the formal definition of the problem which we will try to find a solution for it throughout the paper.
1.2 Problem Definition

Here we consider networks as directed graphs. We define the problem of finding all optimal paths between a pair of source-destination nodes based on multiple constraints in a directed graph. Each edge of the graph has a vector of positive values representing the capacity of that link for different QoS parameters. The problem is finding a set of optimal paths between a pair of nodes that satisfy a set of constraints. This set of constraints may have any number of additive, multiplicative or Min/Max functions as its members. The final path should satisfy each of the input constraints based on the function and constraint specified for it.

1.3 Problem Formulation

In this section we give the formal definition of the Multi Constraint QoS problem:

**Input:** A weighted directed network \( N = (V, A) \) with a source node \( s \in V \) and a sink node \( t \in V \). Each edge \( e \in A \) is associated with an \( r \)-dimensional vector \( v(e) = [v_1(e), ..., v_r(e)] \) of positive integers. Further, given are an \( r \)-dimensional requirement vector \( c = [c_1, ..., c_r] \) consisting of positive integers, and a function vector \( f = [(f_1, j_1), ..., (f_r, j_r)] \), where each pair consists of a function \( f_i \) (representing a network requirement such as delay and bandwidth), and \( j_i \in \leq, \geq \).

**Output:** Does there exist a path \( p \) in \( N \) from \( s \) to \( t \) such that \( p \) satisfies every requirement \( c_i \) in \( c \), \( (1 \leq i \leq r) \) if \( f_i(p) \leq c_i \) for \( j_i = \leq \), and \( f_i(p) \geq c_i \) otherwise?

2 Related Work

The problem of finding a shortest path based on multiple constraints has been studied for quite a long time. Most of these solutions are based on some sort of approximation either in the way the problem has been defined or the precision of the return solution. Dr. Ulrike Stege and Dr. Sudhakar Ganti started working on finding an exact algorithm for this problem. They proved that even though is problem in NP-Complete it is still fixed parameter tractable for topologically sortable graphs, where the problem parameter is the number of constraints \( (r) \) on each link. Thus there is a need of new approaches for graphs that contain loops, which are very common in network graphs. We decided to take this solution and extend it to the case of gen-
eral graphs. We present two different approaches. The first one is an exact solution that works on general directed graphs described in Section 3. The other one is a heuristic approach for converting general graphs into DAGs, based on the k-shortest path algorithm. This way we are able to use the previous algorithm by Stege and Ganti. Before presenting our solutions, we are going to review some work that has been done for the multi-constraint shortest path problem and also the problem of converting a general directed graph into an acyclic one with minimum number of edge removals.

2.1 Multi-Constraint Shortest Path

As we mentioned earlier the problem of finding a path subject to multiple constraints is known to be NP-Complete. Many approximation and heuristic approaches has been suggested for this problem. A complete survey is given in [8]. There are some exact algorithms for the finding shortest path based on two additive constraints. What these approaches tend to do is they give weights to different constraints and combine them linearly and come up with a new constraint. They further use a famous shortest path algorithm (e.g.-Dijkstra’s Shortest Path Algorithm) to find the shortest path. This approach is approximating the weight on each edge of the graph. The way we define weights for constraints totally affects the final solution. What makes Dr. Stege’s exact algorithm novel is the way it looks at each constraint individually and in the end the solution (if exists) is optimum for any constraint. In our work we look at two approaches. One is using the existing exact algorithm for directed acyclic graphs. We tried to look at the problem of converting a multi-weighted directed graph into an acyclic version with minimum number of edge removals with respect to weights. In the following section we introduce this problem.

2.2 Feedback Arc Set

The problem of converting a general graph to a loop free has been an open problem for a long time. There are different versions of this problem. For the cases where the graph is directed or undirected and/or the vertices(edges) are weighted or unweighted. It has been shown in [10] that almost all versions of this problem are NP-hard. The set of is the edges are removed from a graph that makes it acyclic is called Feedback arc set and the problem of finding the minimum sized feedback set has been viewed widely in the
literature\[4\]. Another version of the problem deals with removing minimum number of vertices and their incident edges such that the resulted graph is acyclic. This problem is called feedback vertex set problem. It has been shown that these two instances are reducible in polynomial time. It means from a solution for one of these instances, the other can also be derived. The formal definition of feedback arc set problem is as follow:

Given an arc weighted graph \((G, w), W = (V, E)\) and the set \(C\) of all cycles in \(G\), the minimum weighted feedback arc set problem can be formulated as the following integer programming problem:

\[
\text{MFAS} = \min \sum_{e \in E(G)} w(e)x_e
\]

s.t. \(\sum_{e \in S} x_e \geq 1, \forall S \in C, x_e \in \{0, 1\} \forall e \in E(G)\)

In this work we have mainly studied the solutions for minimum feedback arc set problem. One of the pioneering papers on feedback arc set problems is by Ramachandran \[11\], where it is proved that finding a minimum feedback arc set in an arc-weighted reducible flow graph is as difficult as finding a minimum cut in a flow network. The algorithm in this paper has complexity \(O(mn^2 \log \frac{w^2}{m})\) where \(m = |E(G)|\) and \(n = |V(G)|\).

Another approximation algorithm has been proposed for special types of graphs like planar graph by Stamm \[6\] which is a 2-approximation algorithm for the minimum weighted arc set problem.

TODO: Arian

As we can see this problem has been an open problem for quite a while. The problem is even harder for the our case which deals with multi-weighted graphs.

\section{3 Exact Solution for General Graphs}

Let \(G(E, V)\) be an unweighted directed graph. Notice that \(G\) may contain cycles. For any given pair \(s, t \in V\) our algorithm will find all possible simple paths from \(s\) to \(t\). A simple path is an ordered sequence of vertices in \(V\) where every vertex appears at most once and there is an edge between any two consecutive vertices.

We define the Path Expression recursively, based on simple paths and two operations on Path Expressions:
- **Concatenation:** \( P_1 \times P_2 \)
- **Union:** \( P_1 + P_2 \)

where \( P_i \) is a simple path or a Path Expression and the result is a new Path Expression.

### 3.1 Dynamic Programming Approach

#### 3.1.1 Approach 1

This is the exact approach used in converting a FDA to a regular expression. Although the final set of paths includes both repetitions and invalid paths that contain loops, it is worth mentioning, just to give the reader the basic idea. We define our optimal sub-problem as follows:

\[
\text{OPT}[a, b, k] : \text{All Path Expressions from } a \text{ to } b \text{ containing at most } k \text{ intermediate vertices.}
\]

Thus the recursive formula for filling the dynamic programming table is:

\[
\text{OPT}[a, b, k] = \text{OPT}[a, b, k - 1] + \sum_{m=1}^{k} \sum_{i \in V - \{a, b\}} \text{OPT}[a, i, m - 1] \times \text{OPT}[i, b, k - m]
\]

(1)

And the base cases for this recursive formula are:

\[
\text{OPT}[u, v, 0] = \begin{cases} 
\{e\} & \text{if } e = (u, v) \in E_G \\
\emptyset & \text{otherwise.}
\end{cases}
\]

(2)

#### 3.1.2 Approach 2

It is easy to see the above solution can be simplified without reducing the generality of our final answer. It can be proved by induction that by changing the term “at most” to “exactly” \( k \) intermediate vertices in the definition, our optimal sub-problem is still valid:

\[
\text{OPT}[a, b, k] : \text{All Path Expressions from } a \text{ to } b \text{ containing exactly } k \text{ intermediate vertices.}
\]
Thus the recursive formula for filling the dynamic programming table is:

\[ \text{OPT}[a, b, k] = \sum_{m=1}^{k} \sum_{i \in V - \{a, b\}} \text{OPT}[a, i, m - 1] \times \text{OPT}[i, b, k - m] \] (3)

And the base cases for this recursive formula are:

\[ \text{OPT}[u, v, 0] = \begin{cases} \{e\} & \text{if } e = (u, v) \in E_G \\ \{\} & \text{otherwise.} \end{cases} \] (4)

### 3.1.3 Approach 3

We take one step further in simplifying the solution by defining our optimal sub-problem to be the set of all possible unique paths for a fixed source node \( s \) to an arbitrary node \( b \) with exactly \( k \) intermediate vertices in the middle:

\[ \text{OPT}[b, k] : \text{All Path Expressions from } s \text{ to } b \text{ containing exactly } k \text{ intermediate vertices.} \]

In which \( s \) is the fixed source node. Thus the recursive formula for filling the dynamic programming table is:

\[ \text{OPT}[b, k] = \sum_{q \in V - \{s, b\}, e = (q, b) \in E_G} \text{OPT}[q, k - 1] \times e \] (5)

And the base cases for this recursive formula are:

\[ \text{OPT}[v, 0] = \begin{cases} \{e\} & \text{if } e = (s, v) \in E_G \\ \{\} & \text{otherwise.} \end{cases} \] (6)

### 3.1.4 Final Approach

Finally we add two more restrictions to the algorithm, reducing both its time and space complexity. The first restriction is to simply exclude paths containing loops. The second one is to use the rule proposed in [Stege, Ganti] to eliminate paths that are dominated by others, i.e. to remove paths that may never contribute to the final answer based on any optimality criteria.

We define our optimal sub-problem as follows:

\[ \text{OPT}[b, k] : \text{Pruned set of all simple paths from } s \text{ to } b \text{ containing exactly } k \text{ intermediate vertices.} \]
First, by checking the loop free condition in every step we get rid of paths containing loops. Also we need to prune the set of answers using the pruning rule from the FPT algorithm. Thus the recursive formula for filling the dynamic programming table is:

\[
\text{OPT}[b,k] = \bigcup_{q \in \text{pre}(b)} \bigcup_{\mathcal{P} \in \text{OPT}[q,k-1], b \notin \mathcal{P}} \mathcal{P} \times e, e = (q,b) \in E_G \tag{7}
\]

Where \(\text{pre}(b)\) is the set of all vertices that have outgoing edges to \(b\). Note that we have used the union operation here to show the elimination of repeated paths at every step. The base cases for this recursive formula are:

\[
\text{OPT}[v,0] = \begin{cases} \{e\} & \text{if } e = (s,v) \in E_G \\ \{\} & \text{otherwise.} \end{cases} \tag{8}
\]

### 3.2 Proof of Concept

The above method for enumerating all possible paths from a source to a destination is originally used in the Theory of Computations and Machine Language where we need to convert a Deterministic Finite Automata to a Regular Expression. To investigate the similarities and differences between the “All Paths From \(s\) to \(t\) in a Directed Graph” and “Converting a DFA to a Regular Expression” we need to introduce the latter as well:

#### 3.2.1 DFA and Regular Expressions

Let’s define each of these concepts individually and then try to put them together and describe the conversion procedure.

#### 3.2.2 Deterministic Finite Automata

In theory of computation a DFA is a finite state machine where for every input symbol there is one outgoing transition from every state to another state. Here is a formal definition of the DFA \(M\) as a 5-tuple \((S, \Sigma, T, s, A)\):

- \(S\): a finite set of states.
- \(\Sigma\): a finite set of symbols called the alphabet.
$T : S \times \Sigma \rightarrow S$: a transition function.

$s \in S$: a start state.

$A \subseteq S$: a set of accept states.

Every DFA $M$ recognizes a regular language $L(M)$ which consists of a set of words. Every word $X = x_0x_1 \ldots x_n$ is a sequence of symbols ($x_i \in \Sigma$) and it is recognized by DFA $M$ if there exists a sequence of states is $S$ such as $s_0, s_1, \ldots, s_n$ with the following conditions:

\begin{align*}
  s_0 &= s \\
  s_{i+1} &= T(s_i, x_i), \text{ for } i \in \{0, 1, \ldots, n-1\} \\
  s_n &\in A
\end{align*}

\section{3.2.3 Regular Expression}

Formally a regular expression is an expression consisting of constants and operators. This expression is then used to express a set of strings, known as its corresponding regular language. Here is the list of defined constants:

- $\emptyset$ denoting the empty set
- $\epsilon$ denoting a string with no symbols
- $a \in \Sigma$ denoting a character in the alphabet

And the list of operations:

**Concatenation** $RS$ denotes the set $\{\alpha \beta | \alpha \in R, \beta \in S\}$

**Alternation** $R | S$ denotes the set union of $R$ and $S$.

**Kleene star** $R^*$ denotes the smallest superset of $R$ that contains $\epsilon$ and is closed under string concatenation.

Now that we have defined these two concepts we are able to make comparisons regarding their computational powers. It has been proved in the literature that these two structure are convertible to each other, therefore have the same computational power [7, 1]. Since these conversions preserve the resulting regular language, if a word is recognized by a DFA it also matches the corresponding regular expression.
3.2.4 All Paths Problem and it’s relation to Language Theory

We now try to prove an equivalence relation between paths in a directed graph and words that belong to the language defined by a DFA. Here is how we construct a DFA $M$ from our directed graph $G(V,E)$:

1. $S = V$: i.e. each states in $M$ represents a vertex in $V$
2. $\Sigma$: a set of $|E|$ unique symbols every one of which represent an edge in $E$
3. $T(s_i,e) = s_j$: where there is an edge from $v_i$ to $v_j$ in graph $G$ and $e \in \Sigma$ is the label denoting that edge
4. $s$: the source vertex
5. $A = \{t\}$: where $t$ is the destination vertex

A word in this language is a set of consecutive symbols of the alphabet, and since those symbols represented edges in the graph, a word is therefore, equal to a path. Looking back at (9) it is easy to see that if there is a path from $s$ to $t$ in $G$ its corresponding word is recognized by our DFA. Therefore all possible paths from $s$ to $t$ in graph $G$ is equivalent to all recognizable words for this DFA, i.e. the regular language it is representing.

3.3 Time Complexity

3.3.1 Approach 1

The size of the dynamic programming table is $O(n^3)$. For each element in the table we need to consider $O(n^2)$ cases for all possible pairs $(m,i)$ where $m$ indicates how to break the number of intermediate vertices in two parts and $i$ is the new vertex to use as a connection between two sub-paths. Each concatenation takes $O(1)$ if we simply concatenate two Path Expressions $OPT[a,i,m-1]$ and $OPT[i,b,k-m]$ (Similar to what we do when we convert a DFA to a regular expression) [TODO: has to be rephrased! then what do they use that leads to $O(n^5)$ running time!]. Alternatively we could perform pairwise concatenation between paths in each of these path expressions with the running time $O(S(OPT[a,i,m-1])) \times S(OPT[i,b,k-m]))$, where $S(P)$ is the size of the path expression $P$ in terms of simple paths. In this case each path expression would be in the standard format of $P_1 | P_2 | \ldots | P_l$ in which every $P_i$ is a simple path. Therefore the total running
Running Time = \[
\begin{cases}
\O(n^5) & \text{case I} \\
\O(n^5) \times \O(S(OPT[a, i, m - 1]) \times S(OPT[i, b, k - m])) & \text{case II}
\end{cases}
\]

By computing \(S(OPT[a, b, k])\) in the next section we will be able to show that:

\[
\O(S(OPT[a, i, m - 1]) \times S(OPT[i, b, k - m])) = \O(n^{k-1})
\]

Since the resulting answer in the first case could be a path expression of any form, it is not suitable for our application. This is because we are looking for a set of explicit paths in the end. In addition we are unable to apply our pruning algorithms (introduced in the following chapters) on complicated path expressions.

### 3.3.2 Final Approach

The table size in this approach is \(\O(n^2)\) [TODO: Discuss with Ulrike]. The following list, contains all the operations needed to compute element \(OPT[b, k]\) of the table:

- For every incoming edge of \(b\), consider the incident vertex \(q\). \(\O(n)\)
- For every path \(P\) in set \(OPT[q, k - 1]\), which according to the pruning rule may be \(\O(2^r - 1)\), where \(r\) is the number of constraints on each edge.
- Check whether or not \(b \in P\) that takes linear time \(\O(n)\) since length of path \(P\) is \(k - 1\).

After constructing this new set of paths, we need to perform the pruning by deleting paths that are dominated by others. Thus we need to perform pairwise comparisons between elements in this set. The maximum size of the set is \(\O(nS(k - 1))\) where \(S(k - 1)\) is the space complexity of element \(OPT[x, k - 1]\). Since we need pruning at every step this size is also \(\O(2^r - 1)\) according to the space complexity analysis. Thus the pruning step at the current step involves \(\binom{n(2^r - 1)}{2}\) comparisons. Since every comparison takes
the running time of pruning is $O(rn^2(2^r - 1)^2)$.

Therefore the total running time of filling the whole table is:

$$O(n^2 \times \{(n \times (2^r - 1) \times n) + rn^2(2^r - 1)^2\}) = O(n^4 \times \{2^r - 1 + r(2^r - 1)^2\})$$

(12)

3.4 Space Complexity

3.4.1 Approach 1

Let $T(k)$ be the number of paths with length at most $k + 1$, i.e. using at most $k$ intermediate vertices other than the source and the sink. We can conclude the following recursive relation for $T(k)$ according to our previous recursive algorithm for OPT$[a, b, k]$:

$$T(k) = \begin{cases} 
T(k - 1) + \sum_{m=1}^{k} T(m - 1) \times T(k - m) & k > 0, \\
O(1) & k = 0.
\end{cases}$$

(13)

The base case is when $k = 0$, i.e. we do not use intermediate vertices for constructing the path between source and sink. Since we do not allow multiple edges between two vertices in the initial graph, this number can at most be equal to one. One can solve this recurrence and come up with the following closed form formula for $T(k)$:

$$T(k) = O(n^k) \Rightarrow O(S(OPT[a, i, m - 1]) \times S(OPT[i, b, k - m])) = O(n^{k-1} = O(n^n)$$

(14)

3.4.2 Final Approach

Every path consists of a number of vertices $O(n)$ and a set of parameters $O(r)$. Since the pruning algorithm bounds the number of possible paths between every two vertices to $2^r - 1$, every element in table can take up $O(nr2^r - 1)$ space. Since the table contains $n^2$ elements, the total space complexity of this dynamic programming approach is $O(n^3r2^r - 1)$.

4 Heuristic Approach

If we were able to design an efficient algorithm that converts a general multi-weighted graph in to a directed acyclic graph, then we could use the revised
graph as an input of the existing FPT algorithm. The idea is based on the k-shortest path algorithm.[2] We use k-shortest path to remodel the graph into a single weighted version. Then we use one of the known algorithms for minimum feedback arc set problem for directed weighted graph.

### 4.1 K-Shortest Path

Eppestein has suggested an algorithm in [2] that finds the top k-shortest paths in a directed weighted graph. This algorithm can also work if we define an upper bound for the cost of the shortest paths (Paths that have cost less than a certain value). The algorithm runs in time $O(m + n \log n + k)$, where $m$ is the number of edges and $n$ is the number of vertices of the graph.

We use the k-shortest paths algorithm to find a new weight for each edge of the graph. The idea is as follow:

For each constraint (therefore each member of the capacity vector on edges) that we have (say $r$) we run the k-shortest path algorithm individually and find top $k$ path that satisfies the specific constraint. We define a new variable for each edge of the graph called frequency. This value on each edge shows the number of times this edge has been a part of a shortest path for any of the constraint. For each constraint we separately run the k-shortest path algorithm and update the frequency value of the edges. At the end, we build a graph with a value (frequency) on each edge which represents the effect of this edge in shortest paths for all constraints. Frequency is highlighting the edges that are important in the graph. Those that have appeared in more shortest paths have a higher frequency value.

We use this weight and try to break the loops of the graph with minimum weight of the removed edges. In the next section we show an algorithm that breaks the loops of a single weighted directed graph. Here is the pseudocode of this heuristic approach:

### 4.2 Loop Removal

Loops are broken in $G$ by identifying a loop and then removing the edge that has been used in the least k-shortest paths. We create a new graph $G'$ iteratively based on the frequency the edge appears in the k-shortest path, highest to lowest. For each edge that is added to $G'$ a Depth-First-Search (DFS) is performed, if there is found to be a back edge than it is known that the edge inserted created a loop. Furthermore from the method of insertion to $G'$ it is known that the edge added is the least frequently used in the
Algorithm 1 Algorithm for minimum weighted cycle removal

**Input:** A weighted directed network $N = (V, A)$ with a source node $s \in V$ and a sink node $t \in V$. Each edge $e \in A$ is associated with an $r$-dimensional reality vector $v(e) = [v_1(e), ..., v_r(e)]$ of positive integers. Furthermore, given are an $r$-dimensional requirement vector $c = [c_1, ..., c_r]$ consisting of positive integers, and a function vector $f = [(f_1, j_1), ..., (f_r, j_r)]$, where each pair consists of a function $f_i$ (representing a network requirement such as delay and bandwidth), and $j_i \in \leq, \geq$. and parameter $k$

**Output:** Minimum Feedback arc set of $G$

```
for all edge in $G$ do
    define $f = 0$
end for
for all Constraint in vector $C$, $c_i$ do
    Find $K$-shortest path
end for
for all edge in $c_i$-$k$-shortest path do
    increase $e_f$
end for
sort each edge in $G$ on $e_f$ in decreasing order
for all edge in $G$ do
    add edge, $uv$, to $G'$
    run Depth-First-Search on $G'$ starting at vertex $v$
    if Depth-First-Search finds back edge then
        remove $uv$ from $G'$
    end if
end for
```
k-shortest paths. Thus the edge to be removed to break the cycle is the edge just inserted. Once every edge from G has been tried, G’ contains the largest subgraph of G such that there are no cycles.

5 Implementation and Results

5.1 Exact Solution

We proposed an exact algorithm for finding all possibly good paths from a source to a destination in a directed graph (see Section 3). Although we showed that this problem is fixed parameter tractable, the exponential running time is still a major problem for large graph instances, which is usually the case for real network graphs.

Despite this relatively high exponential upper bound for running time of our algorithm, it performs quite fast if a parameter with additive behavior is involved in the network. As it is illustrated in [?] the actual running time of our algorithm seems to be growing in a constant and slow rate (compared to the all paths). We can explain this behavior by looking at the number of paths at each step [?]. Once $k$ exceeds the shortest path hop count between source and sink, the number of paths begin to grow in both algorithms, however this growth does not last in the dynamic programming approach where we prune dominated paths. The number of paths in this case seem to converge to a constant value, meaning that there are no new paths with more vertices in the middle that are not dominated by our previous set of paths. Considering the additive behavior of our constraints, it is quite straightforward to verify that longer paths are less probable to remain in the final pruned set of paths.

One can also argue that this kind of behavior depends on many network parameters such as the network topology and the weight distributions. We try to verify our results by using different realistic models both for the network topology and the weight distributions.
5.1.1 Network Topology

As an initial assumption we started working with random graphs, in which there is an edge between every ordered pair of vertices with a constant probability $p$. For our test cases we set this parameter to a value that leads to average edge per vertex density of approximately 10, for a network of 100 nodes (a typical backbone network in North America). However in such random graphs, the shortest path between any two nodes often consists of very few number of intermediate nodes. However in real networks due to the geographical constraints, links usually connect nodes locally. This leads to node further from each other to have more number of hops in the middle of the shortest path between them.

To simulate this topology, we simply built up a network in which every node is connected to other nodes within it’s range. It is also important to mention that network nodes are distributed uniformly in this scenario.

5.1.2 Weight Distribution

Future work.

5.2 Heuristic Solution

We proposed a heuristic algorithm to convert a general directed graph to a directed loop-free graph.

To do this, we used the k-shortest path approach for ranking edges of graph and based on their ranks, we remove loops from graphs.

We used the k-shortest path implementation using Yan’s algorithm already available online and modified to fit our problem. Also we implemented the break loop algorithm in C++. The output of these sets of processes results in a DAG which we use as an input the FPT algorithm.

There are major factors affecting the order of approximation these heuristic approaches have. One is the value of $k$ for the k-shortest path. If we have a big $k$, then we are close to finding all possible paths between the source and destination. This way the ranking of edges may be more realistic and illustrate how important an edge is. Still there is no guarantee that by having a big number for $k$ (having a bigger running time) we still have the optimal solution.
Another important factor is how the far the constraint values are from the actual paths cost. If constraints are bigger than the total cost of paths, then the k-shortest path is facing a big set of possible solutions and k-shortest path may not be a good indicator of the importance of edges in graph.

6 Future Work

This project still needs further investigation. As we have mentioned in previous sections, there are still more we need to work on. We have listed some of the tasks we are planning to do in future:

- Verify the $O(2^r - 1)$ upper bound for the number of non-dominated paths in a set.
- Explore different network topologies, and evaluate their effects on the performance of our algorithms.
- Further investigate the running time of algorithms and compare the trade-off between the running time and approximation.
- Reduce memory usage in proposed solutions.
- Refactor code to lower memory accesses for data structures.
- Unify individual components of proposed solution to reduce I/O time of writing and reading from file.
- Generate/locate larger selection of random directed graphs.

References


A Appendix

A.1 What is Wrong With Approach 1?!

1. In the recursive when we break the $k$ intermediate vertices into three groups; $m – 1$ vertices in the prefix, 1 vertex as the bridge and the remaining $k – m$ vertices in the suffix; it is possible that those intermediate vertices overlap. This is because we are considering all possible $m – 1$ and $k – m$ vertices in those two cases. This will lead to some solutions that contain loops. Although this may seem like a problem, but if we can make sure that the whole set of solutions is bounded by some polynomial upper bounds, the loop-free subset will be bounded too. And we will eventually be able to recognize and eliminate paths with loops from our set of solutions in polynomial time.

2. The time and space complexity seem to be exponential when we expand the Path Expressions (instead of using string concatenation), however this number is actually bounded by the number of available paths in a real network graph, which is significantly thinner than a
complete directed graph. Using string concatenation is another option which has its own difficulties, including detecting and eliminating redundant paths using regular expression comparisons.

A.2 Membership of All Paths Problem in NP-Hard

We prove that the longest path problem from $s$ to $t$ in a directed graph is NP-Hard, consequently the all paths problem is also NP-Hard. Assume there exist a polynomial time algorithm for finding the longest path from $s$ to $t$ in graph $G(V, E)$. Therefore one can design a polynomial time algorithm that answers the following question:

- Longest($G, s, t$): Is the length of the longest path from $s$ to $t$ equal to $|V| - 1$.

Based on this tool we can design the following algorithm that solves the Hamiltonian cycle problem in polynomial time:

- Hamiltonian($G$): For all $uv \in E$ \{ if Longest($G, v, u$) is yes, answer is yes \} answer is false.