MINSUM SINK LOCATION AND EVACUATION PROBLEM ON DYNAMIC CYCLE NETWORKS

MD KHAIRUL BASHAR
Bachelor of Computer Science & Engineering, Chittagong University of Engineering & Technology, 2011

A Thesis
Submitted to the School of Graduate Studies
of the University of Lethbridge
in Partial Fulfillment of the
Requirements for the Degree

MASTER OF SCIENCE

Department of Mathematics and Computer Science
University of Lethbridge
LETHBRIDGE, ALBERTA, CANADA

MINSUM SINK LOCATION AND EVACUATION PROBLEM ON DYNAMIC CYCLE NETWORKS

MD KHAIRUL BASHAR

Date of Defence: December 13, 2018

Dr. Robert Benkoczi Supervisor	Associate Professor	Ph.D.
Dr. Daya Gaur Committee Member	Professor	Ph.D.
Dr. Saurya Das Committee Member	Professor	Ph.D.
Dr. Howard Cheng Chair, Thesis Examination Committee	Associate Professor	Ph.D.

Abstract

In this thesis, we study 1-sink location and k-sink evacuation problem on dynamic cycle networks. We consider the 1-sink location problem is to find the optimal location of the 1 sink, while the k-sink evacuation problem is to find the optimal evacuation protocol for the given locations of the k sinks. Both results minimize the sum of the evacuation times of all the supply located at the vertices to the sink/s of a given cycle network of n vertices. We present an efficient algorithm with a useful data structure that finds the optimal location of the 1 sink in $O(n)$ time when the capacity of the edges are uniform. If the edges have arbitrary capacities, we solve the problem in $O(n \log n)$ time by an extension of the data structure. We also propose an $O(n)$ time algorithm to solve the k-sink evacuation problem with uniform edge capacity.

Acknowledgments

I would like to express my deepest gratitude to my friends in Lethbridge without whom I couldn't imagine passing the duration of this program so smoothly. I must thank first my family and most of all, my beautiful Shahanaz, the most kind, caring and accepting person I have ever seen who has supported me in every imaginable way. Only I know how important is her presence, love, smile and understanding in my life. I am also utmost grateful to my parents for their trust, patience, confidence and unconditional love for me.

I am thankful to Fuad for his unbelievable support to find accommodation in Lethbridge and to Mostafiz for the awesome snow ride on the day I arrived. I must thank Arif for helping me out on my first car and for the valuable times to put me on the driver seat. Asif, for his motivation, suggestion and awesome jokes that always relieved my stress. Shammi, for her inspiration and confidence in me. Shahina, for allocating time to review my thesis on her busy days. Zerin, the cutest and smiley girl, for her endless support to my family in our toughest days. I am also thankful to my friends Salahuddin, Farzana, Sultanul, Asif, Shauli, Rafee, Sharmin, Ratna, Rajib, Peash, Shamanta, Asif, Polash for those wonderful UNO nights, movie shows, late night dinners, tours and many more quality times.

Finally, thanks are due to my supervisor Dr. Robert Benkoczi, without whom this thesis wouldn't have existed. Through his guidance, encouragement, and optimism, he made the times I spent in my thesis more effective and fruitful. I am thankful to my committee members Dr. Daya Gaur and Dr. Saurya Das for their valuable comments and suggestions. I also owe my gratitude to all the members of the Optimization Research Group of the University of Lethbridge for their support over the duration of this program. Thank you all.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

Maximizing efficiency and minimizing cost has become a primary objective to the economists and industrial engineers in both the public (e.g., schools, hospitals, fire stations) and private sectors (e.g., agriculture, retail facilities) to design a useful model nowadays. This type of research deals with advanced analytic techniques to help make decisions towards an optimal solution, which is known as the operations research. It has many branches dealing with the optimization of different type of activities. Facility location problems (FLPs) are one of these branches concerned with the facility placement.

FLPs ideally consist of a set of demand points which need to be served and a set of designated spots where the facilities can be placed. Further, FLPs have a set of constraints that influence the transportation criteria of the supply from the demand points to the facility points, where the supply can be a set of evacuees or fluid-like material. The objective of an FLP is to find the optimal location of the facilities so that the supplies at the demand points can be sent to one of the facilities in such a way that a specific objective function is minimized.

FLPs often deal with many real-life problems, such as emergency evacuation planning, industrial or urban planning, traffic or vehicle routing. For example, if we need to build a hospital in a particular residential area, the facility location problem can deal with identifying the optimal location of the hospital that can minimize the residents' transportation cost.

In this model, the residences are the demand points, and the hospital is the facility point.

In this thesis, we studied two problems in facility location. One is the sink location problem, where we need to find the optimal location of the sink/s (facilities), and the other is the evacuation problem, where we need to find an optimal evacuation protocol for a set of given sink locations.

1.1 Overview of the covered problems

The sink location and the evacuation problems can be modelled in dynamic networks, which is first introduced by Ford et al. [6]. A dynamic network is a graph network where each vertex has an associated supply, and each edge has a length and a capacity. The supply evacuates to a subset of the vertices called sinks. The capacity of an edge limits the flow of supply that enters the edge in the unit time, and the length determines the time the flow traverses the edge from one end to another. In this thesis, we consider the dynamic network with the continuous flow of supply, where the supply at a vertex can be a real number. Or the supply can be regarded as fluid-like material, where each infinitesimally small amount of supply is an evacuee. The evacuation time of an evacuee is the time to send that evacuee to one of the sinks. The cost of a sink is defined as the sum of the evacuation times of all the evacuees. As a simple illustration, let us consider a dynamic path network as follows.

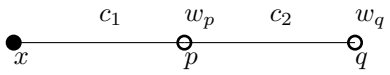


Figure 1.1: Illustration of evacuation on a dynamic path network. The sink is at x, and all the supplies at the vertices p and q evacuate to the sink.

In Figure 1.1, the evacuation starts with w_p and w_q amount of supply at the vertices p and q respectively. Let all the edges have the same length l and the time to travel a unit distance is τ. An infinitesimally small amount of supply dw takes τl units of time to cross each of the edges. Since the amount of supply that can enter an edge in the unit time is

bounded by the capacity of that edge. The capacity of the edge (x, p) is c_1, so c_1 amount of supply can enter the edge in the unit time from vertex p. Therefore, the last supply at p leaves p at time w_p/c_1 and reaches sink x at time $\tau l + (w_p/c_1)$. Note that the first supply from vertex q reaches vertex p at time τl. At time τl, if there is no supply at p, *i.e.*, $\tau l \geq w_p/c_1$ then the supply from q can continuously reach sink x without being delayed at p, so no congestion occurs. But, if $\tau l < w_p/c_1$, then the first supply from q reaches p before the last supply at p leaves p, so congestion occurs at p. Now, if we plot the arrival flow rate at sink x from all the vertices as a function of time, we get the sequence of supplies arriving at the sink as follows.

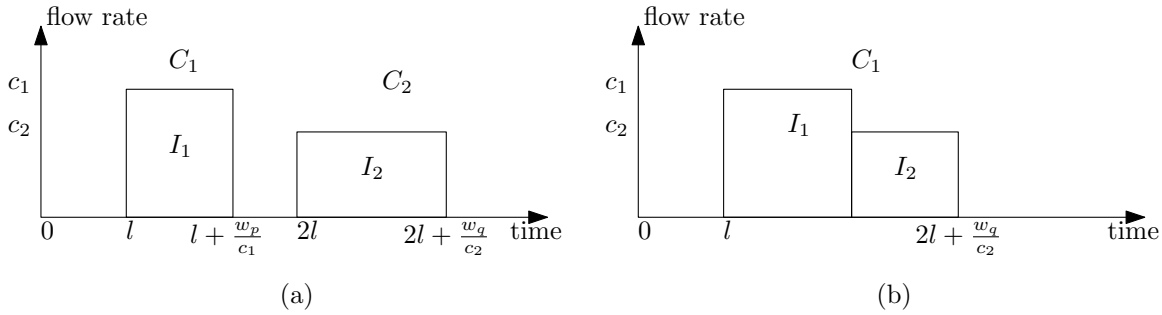


Figure 1.2: Sequence of supplies arriving at sink x as an illustration of evacuation on the path graph of figure 1.1 in two cases, when $c_1 > c_2$. (a) No congestion occurs; (b) Congestion occurs at p.

Definition 1.1. ***Cluster***

Let us consider a dynamic network N with a sink located at some point $x \in N$. For sink x, if we plot the arrival flow rate from all the vertices of N to the sink as a function of time, we get a sequence of clusters, where a cluster is a maximal time interval for which the arrival flow rate at sink x is continuously greater than zero.

In Figure 1.2, we call each of the contiguous time interval of flow rate c_1 or c_2 or both, a cluster. We also call a cluster the set of vertices whose supplies are evacuated within a cluster. In Figure 1.2(a), vertex p forms cluster C_1 and vertex q forms C_2, where, the areas of the clusters C_1 and C_2 are the supplies w_p and w_q respectively. In Figure 1.2(b), the supply of q leaves the vertex with rate c_2 but a part of the supply gets congested at p and

leaves vertex p with rate c_1. However, the rest of the supply of q that does not have to wait at p reaches the sink with rate $c_2 (< c_1)$. Because of the congestion, the set of vertices p and q forms the cluster C_1, where the area of C_1 is the sum of the supplies w_p and w_q.

Definition 1.2. ***Cluster Head***

Cluster head is the first vertex of a cluster whose supply is the first to reach the sink without being blocked at any intermediate vertices in the evacuation path between the vertex and the sink.

In Figure 1.2(b), the cluster head of cluster C_1 is vertex p. A cluster consists of sequence of sections.

Definition 1.3. ***Section***

A section of a cluster is a contiguous and maximal time interval of the cluster for which the flow rate is constant. For any two adjacent sections of a cluster, the flow rates are different.

In Figure 1.2(b), cluster C_1 consists of two sections I_1 and I_2, while in Figure 1.2(a), both clusters C_1 and C_2 have only one section.

In this thesis, a cluster is represented by C_i and a section by I_j, where $1 \leq i \leq j \leq n$. The cluster head of a cluster C_i is denoted by η_i. The flow of a section must originate from a vertex. The *first vertex* of a section I_j is denoted by u_j, from which the supply corresponding to that section can reach the sink without being delayed at any intermediate vertices. The flow rate (or height) of a section I_j is determined by the minimum edge capacity of the evacuation path between the sink and the first vertex of the section, denoted by h_j. The supply carried by section I_j is represented by σ_j.

Moreover, if there are two sections I_j and I_{j+1} belongs to two adjacent clusters (as shown in Figure 1.2(a)) then the time interval of flow rate 0 between I_j and I_{j+1} is called a "gap." On a similar note, if the sections I_j and I_{j+1} belongs to the same cluster (as shown in

Figure 1.2(b)) then the region on top of section I_{j+1} of height $h_j - h_{j+1}$ and duration $\delta_{t_{j+1}}$ is called a "step".

However, we define two possible type of clusters depend on the number of sections in the cluster; one we call the "Simple cluster" and the other is "Mixed cluster."

Definition 1.4. ***Simple Cluster***

A cluster of uniform height is called the simple cluster. A simple cluster consists of only one section. In Figure 1.2(a), $C1$ and C_2 are the simple clusters.

Definition 1.5. ***Mixed Cluster***

A cluster is called mixed cluster when it contains multiple sections of different heights. In Figure 1.2(b), cluster C_1 is mixed clusters.

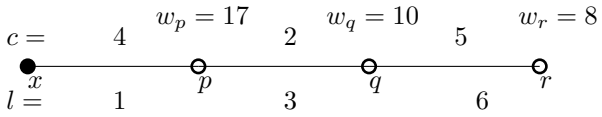


Figure 1.3: An example of a dynamic path network

Example 1.6. Figure 1.3 shows an example of a dynamic path network of three supply vertices p, q and r, and a sink x. The capacity of an edge is labelled on the top, and the length is labelled below of that edge. The time to travel a unit distance by an infinitesimally small supply is 1. The first supply of vertex p reaches the sink at time 1. If the last supply of p reaches sink x at time t', then

$$(t' - 1)4 = 17$$

So,

$$t' = 5.25$$

Therefore, the last supply of p leaves vertex p at time 4.25 while the first supply from q arrives at p at time 3. Without loss of generality, we consider the portion of supply from

q arriving at p in between time 3 to 4.25 waits to start evacuating through the edge (p,x) until time 4.25. Thus, the amount of supply arrives and waits at p is $1.25 \times 2 = 2.5$, while the rest of the supplies from q pass through the edge (p,x) without waiting at p.

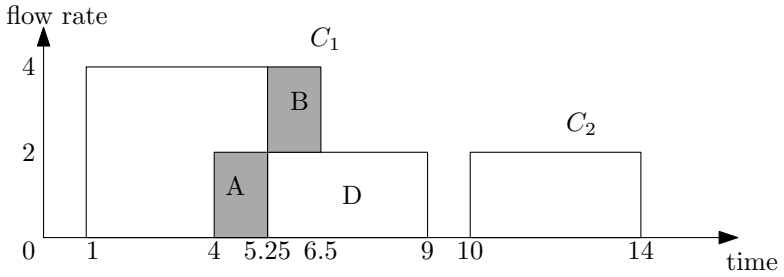


Figure 1.4: Illustration of the congestion of the supply evacuating to sink x

In Figure 1.4, time 4 is the time the first supply from q would have reached the sink if it had not been delayed at vertex p. Therefore, the area of region A represents the portion of supply of q waiting at p until all the supply of p leaves p, which is the 2.5 amount of supply. This 2.5 amount of supply starts to leave vertex p at time 4.25 with the higher capacity 4 (the area of region B) along with the rest of the supply from q that arrives at p at time 4.25 and afterwards (the area of region D). Thus, the areas of regions A and B represents the same supply. The time 5.25 is the time the first supply of q reaches sink x with the higher capacity 4. The time interval in which the supply of q arrives sink x at rate 4 is $(2.5+2.5)/4 = 1.25$. Therefore, the time t at which the rest of the supply of q starts to arrive at the sink with capacity 2 is

$$t = 5.25 + 1.25 = 6.5$$

where the last supply of q reaches sink x at time 9. Unlike q, the supply of vertex r continuous through to x without being delayed at any of the vertices p and q but with the lowest capacity in its evacuation path, which is 2. As a result, the total amount of supply that arrives at sink x forms two clusters or three sections as shown in Figure 1.5.

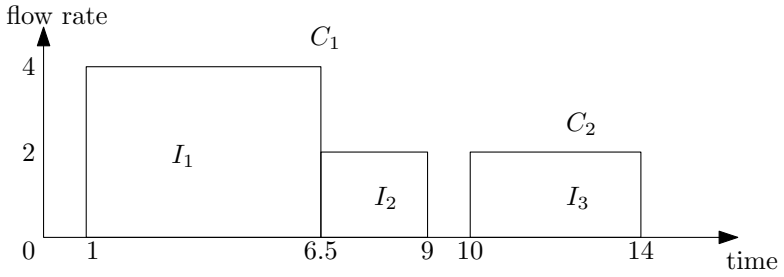


Figure 1.5: Sequence of clusters as a function of time arriving at sink x for the path network showed in figure 1.3

Now, for the dynamic path network in Figure 1.1, the cost of sink x is the sum of the costs of all the sections. The cost of a section is defined as the sum of the evacuation times of all the infinitesimally small amount of supply carried by the section. Figure 1.6 shows the area of a section I, which represents the total amount of supply evacuates at rate c or the height of the section.

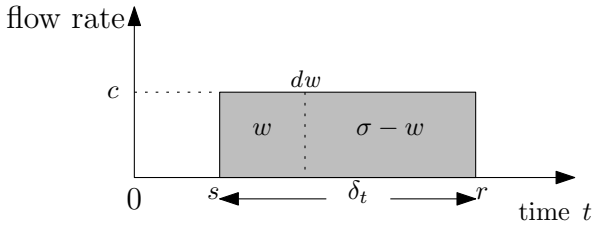


Figure 1.6: Area of a section I with start and end time at the sink

The start time (resp. end time) of a section with respect to a sink is the time when the first supply (resp. last supply) of the section reaches the sink, denoted by s (resp. r). Thus, the duration of a section is the difference between the end time and start time, denoted by $\delta_t (= r - s)$. If the area of section I is denoted by $\sigma (= c\delta_t)$, then the end time r is $s + (\sigma/c)$. In Figure 1.6, w is a portion of supply in the total supply σ, where, the infinitesimally small dw-th supply reaches the sink at time $s + (w/c)$. Therefore, the cost of section I is given by [9]

$$\int_0^{\sigma} (s + \frac{w}{c}) dw$$

$$= \left[sw + \frac{w^2}{2c} \right]_0^{\sigma}$$

$$= s\sigma + \frac{\sigma^2}{2c} \tag{1.1}$$

From equation (1.1), the average evacuation cost for an evacuee of section I is $s + (\sigma/2c)$, where $\sigma/2c$ is the average waiting time per evacuee before it starts to evacuate. We adopt equation (1.1) as the main equation to compute the cost of a section. For a clear understanding, if we plot the cost of the section I as a function of time, we get a trapezoid as shown in Figure 1.7.

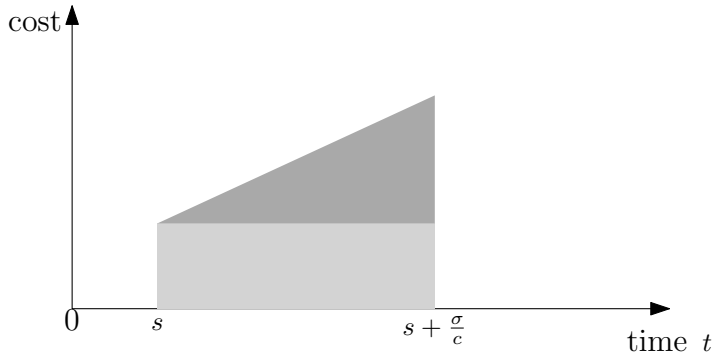


Figure 1.7: Total cost of section I as function of time

The area of the trapezoid is the cost of I, where the area of the triangle (dark gray part) is the total waiting cost (second part of the right hand side of equation (1.1)) and the area of the rectangle (light gray part) is the total travelling cost (first part of the right hand side of equation (1.1)).

Furthermore, the sink location problems are formulated in one of the two objective functions, minisum and minimax. The minisum objective function deals with the goal of minimizing the sum of the evacuation times of all the infinitesimally small supplies located at the vertices to the sinks, which is also known as the total cost criterion. The minimax, by contrast, aims to minimize the maximum evacuation cost of an infinitesimally small amount of supply from a supply vertex to a sink, which is also known as the maximum cost criterion. In Figure 1.1, all the supply located at the vertices p and q evacuate to sink x by forming multiple clusters/sections. Let I_x denote the cost of a section with respect to sink

x, and $sum(I_x)$ and $max(I_x)$ are defined as the summation and maximum of the costs of all the sections respectively. Now, if we are given the control to change the location of the sink in such a way that a specific objective function is minimized, then the minisum (resp. minimax) objective is to minimize $sum(I_\nu)$ (resp. $max(I_\nu)$ for $\nu \in S$, where S is a set of all feasible locations of the sink.

In this thesis, we focus on a specific sink location problem called **1-sink location problem on dynamic cycle networks with minsum objective function**. A cycle graph is represented by a set of vertices connected in a single cycle in such a way that all the vertices are of degree two so that the number of vertices and edges in the graph are equal.

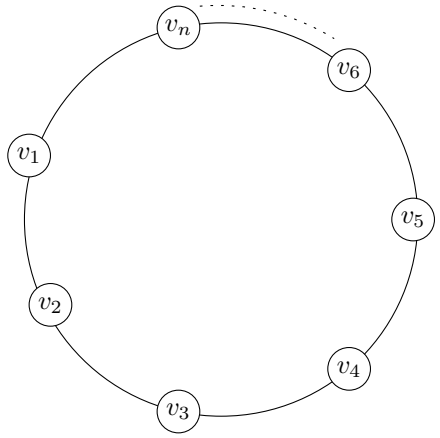


Figure 1.8: Cycle graph

Therefore, the problem is to find the the optimal location of the 1 sink that satisfies the minsum objective function on a given cycle network (see section 2.1 for detail problem definition). Note that in this thesis, we call the problem **Minsum** 1**-sink location**, in short.

On the other hand, the specific evacuation problem we covered in this thesis is called *k*-**sink evacuation problem on dynamic cycle networks with minsum objective function**. So, in this problem, we are given a dynamic cycle network with the location of k sinks, where we need to identify the optimal evacuation protocol that minimizes the sum of the

evacuation times of the supplies at the vertices to the sinks (see section 5.1 for detail problem definition). We call this problem **Minsum k-sink evacuation,** in short.

1.2 Literature review

The evacuation and the sink location problems have been studied in recent years motivated by the mass evacuation planning strategies in different cities due to natural disasters such as earthquakes, tsunamis, hurricanes etc., where dynamic graph networks are used to model evacuations. Cities can be modelled as dynamic graph networks, where, people in a building represent the supply at a vertex and roads connecting the buildings represent edges. The problem is to locate a set of facility points (sink location problem) or to identify the optimal evacuation protocol (describing who goes to which facility point) if the facility points are already there (evacuation problem), so that the total cost to evacuate all the evacuees from the buildings (vertices) is minimum.

The minsum k-sink location problem is equivalent to the classical k-median problem (see Definition 1.7) when the edge capacities are sufficiently large so that the supply at the vertices can move to their closest sink without being delayed at any of the vertices. In [11], Kariv and Hakimi formulated the k-median problem as a decision problem and proved that for a general graph the k-median problem is NP-Hard. A problem is said to be NP-Hard if it is at least as hard as the hardest problems in the class of NP. In computational complexity theory, NP is a set of all decision problems for which the result can be a yes or no. The "yes" answer should have adequate verifiable proof, and the proof has to be verified in polynomial time.

Definition 1.7. ***k-median problem***

Given an undirected graph $G = (V,E)$ of n vertices, identified as a set of supply points. The k-median problem is to find a set $S \subseteq V$ of k facility points that minimizes the sum of the

weighted distances between the supply points and the nearest of the selected facility points. Unlike the minisum k-sink location problem, the edges of graph G are uncapacitated. If the distance between a facility point p and a supply vertex v is represented by $d(v,p)$ and w represents the supply at vertex v, then the cost of the k-median problem is given by

$$\sum_{v \in V} \Big(Min_{p \in S} \big(d(v,p) w \big) \Big) \tag{1.2}$$

The cost of a median is defined as the sum of the weighted distances between the median and the supply points, which are served by the median.

Definition 1.8. ***k-median as a decision problem***

Given an undirected graph $G = (V,E)$ with a positive integer k and a positive real value z. The problem is to identify, does there exist a set of medians S such that $S \subseteq V$ and $|S| \leq k$ for which the sum of the costs of the k medians is not greater than z?

Let us consider an undirected graph $G = (V,E)$ of n vertices, whose edges and vertices have unit length and supply respectively. In graph G, the cost of the optimal k-median is $n-k$.

Now, we consider a dynamic graph network $N = (G, w, l, c, \tau)$, where graph G is the same graph we consider for the k-median problem. Therefore, in network N, w is a function that associates each vertex $v \in V$ with the unit supply and l is a function that associates each edge $e \in E$ with the unit length. Positive constants c and τ represent the uniform capacity of the edges and the time to travel the unit length respectively. In network N, the cost of the optimal k-sink is $n-k$, if no supply has to wait at any vertex of N. So, in that case, the capacity c is greater than or equal to the maximum supply at a vertex, which is 1. Thus, the solution of the k-sink problem on network N is identical to the solution of the k-median problem on graph G, when the uniform capacity $c \geq 1$.

Now, if we convert the k-median problem to a decision problem, then if the answer is "yes"

then it is also "yes" for the k-sink problem and vice versa.

As, in [11], Kariv and Hakimi proved that the k-median problem is NP-hard even in a planar graph with edges of unit length and the maximum degree of a vertex is three. Therefore, minsum k-sink location problem is also NP-Hard as the k-median problem is a special case of the k-sink problem.

Because of its hardness, special cases of the evacuation and sink location problems are actively being studied in recent years. More specifically, Mamada et al. [12] proposed an $O(n\log^2 n)$ algorithm that solves the minmax 1-sink location problem on dynamic tree networks. Bhattacharya et al. [3] solved the minmax k-sink location problem on dynamic path networks in $O(\min\{n+k^2\log^4 n, n\log^3 n\})$ time when the edges have arbitrary capacities and in $(\min\{n+k^2\log^2 n, n\log n\})$ time when the edge capacities are uniform. In the arbitrary case, Golin et al. [1] derived an $O(kn\log^2 n)$ algorithm for solving the minmax k-sink location problem on dynamic path networks.

The sink location problem with minsum objective has been studied in [8] [9] [2]. Higashikawa et al. [8] were the first to show an $O(n)$ algorithm that solves the 1-sink location problem on dynamic path networks in the uniform edge capacity case. In the same paper, they also solved the k-sink in $O(kn^2)$ time, but later they improved the result to an $O(n^2\min\{\sqrt{k\log n}+\log n, 2^{\sqrt{\log k\log\log n}}\})$ time algorithm in [9]. The authors proved that the location of the sinks that minimizes the total evacuation cost of all the evacuees of the network must be at the vertices of the network. In both papers, the authors used some properties strictly applicable to the uniform edge capacity case; therefore, in our observation, we cannot extend their idea to solve the problem in the arbitrary case. In [9], the authors also proposed an $O(kn)$ algorithm that solves the minmax k-sink location problem on dynamic path networks. Recently, Benkoczi et al. [2] showed the first polynomial time

algorithm, where they solved the k-sink on dynamic path networks in $O(kn^2 \log^2 n)$ time in the arbitrary case and improved the running time to $O(kn \log^3 n)$ when the edge capacities are uniform. They applied a dynamic programming approach inspired by the recursive formulation showed by Hassin and Tamir [7] for solving the k-median problem on path graphs. The authors of [2] also proposed an efficient data structure to get the local cost of all the intermediate vertices in a given sub-path.

At best of our knowledge, there is nothing known for the minsum objective in any general graphs except the path graph, which prompted us to do further research on this problem. We motivated to satisfy the minsum objective function from the desire of minimizing the total evacuation cost of the evacuees, which reduces the average psychological stress for each evacuee. Also, we studied the dynamic cycle network, which we believe can be a significant extension of the results found by Higashikawa et al. [9] and Benkoczi et al. [2]. Furthermore, our results can be used towards solving the problem in cactus graph networks, as a cactus graph consists of multiple cycles and defines as a connected graph where any two cycles have at most one vertex in common.

On the other hand, evacuation problems in different variants have extensively been studied by several researchers but only with the minmax objective function. Ford and Fulkerson [6] were the first to introduce the dynamic graph networks, where they show a polynomial time algorithm that gives the maximum dynamic flow from a single source to a single sink in a given time T. Another variant of the dynamic flow problem is the quickest transshipment problem, where the sources have specific supplies and sinks have specific demands. In contrast to the evacuation problem, the network is a directed graph network. However, Hoppe and Tardos [10] provided a polynomial time algorithm that sends the right amount of supply from multiple sources to multiple sinks in minimum time. Burkard et al. [4] studied a different variant of evacuation problem, which is called the quickest flow problem

(QFP). The QFP is to determine the minimum units of time that are necessary to transmit a given amount of supply (flow units) in a given directed network from a given source to a given sink. They derived a strongly polynomial time algorithm of time complexity $O(m^2 \log^3 n(m + n \log n))$, where n and m are the numbers of vertices and edges in the network respectively. Therefore, after all of these variants, we study the evacuation problem on cycle networks with the minsum objective function and came up with some useful results which can be a good start to this type of problems with the minsum objective.

1.3 Contributions

To the best of our knowledge, no previous study of the sink location problem or the evacuation problem on dynamic cycle networks with minsum objective function has been carried out in the literature we surveyed. In this thesis, we present efficient algorithms for the problems listed below.

- **Minsum 1-sink location problem on dynamic cycle networks**

  We proposed an efficient algorithm with a useful data structure to solve the problem in $O(n)$ time when the edge capacities are uniform. In our proposed solution, the cost per operation is amortized $O(1)$. Moreover, by implementing some extensions in the data structure, we solve the problem in $O(n \log n)$ time in the arbitrary edge capacity case, where the cost per operation is amortized $O(\log n)$. We named our data structure the "Cluster Head Tree" (see chapter 3).

- **Minsum k-sink evacuation problem on dynamic cycle networks**

  Minsum 1-sink evacuation problem is trivial because if we fixed the location of the 1 sink then all the supply goes to the fixed sink and there is no choice to be made. Therefore, we studied the evacuation problem with multiple sinks and came up with an $O(n)$ algorithm that solves the problem for k sinks.

The sink location problem with minsum objective has been studied only on path networks,

so our results advance the research on algorithms for more general networks. Further, we are the first to solve the evacuation problem with minsum objective, which we think can be a significant start towards solving the problem on other special graphs.

1.4 Thesis organization

The rest of this thesis is organized in the following order. In the second chapter, we define the minsum 1-sink location problem on dynamic cycle networks and present the common part of our algorithm that solves the problem in both uniform and arbitrary edge capacity cases. We also describe the general approach to compute clusters due to the congestion at the vertices, which is followed by some essential properties of the problem. Finally, we discuss the complexity of the algorithm comparing to the straightforward approach. In chapter three, we describe the construction of our proposed data structure in the uniform case followed by an extension for the arbitrary case. Chapter four shows the implementation of the data structure to solve the minsum 1-sink location problem on dynamic cycle networks. Finally, in chapter five, we define the minsum k-sink evacuation problem and describe our proposed $O(n)$ algorithm to solve the problem. In the concluding chapter, we discuss our results and future works.

Chapter 2

Algorithm for Minsum 1-sink on Cycle Networks

2.1 Problem Definition

Consider a dynamic network $N = (G, w, l, c, \tau)$, where $G = (V, E)$ be an undirected cycle graph. Graph G has an ordered set of vertices $V = \{v_1, v_2, ..., v_n\}$ and a set of edges $E = \{e_1, e_2, ..., e_n\}$, where, n is the total number of vertices. In graph G, vertices are ordered either clockwise or counter-clockwise. For a simple illustration, unless stated otherwise, we consider the vertices as ordered counter-clockwise, as shown in figure 2.1. An edge between two adjacent vertices v_i and v_{i+1} is represented by $e_i \in E$ for $1 \leq i \leq n-1$; for $i = n$, the edge between the vertices v_n and v_1 is represented by e_n.

In network N, w is the function that associates a vertex v_i with a positive supply w_i which represents the initial amount of supply located at the vertex ($w_i : V \to R_+$). An edge e_i has a positive length l_i ($l_i : E \to R_+$) and a positive capacity c_i ($c_i : E \to R_+$). The length of an edge determines the time required by an evacuee to traverse the edge, and the capacity is the upper limit of the flow rate that can enter the edge in the unit time. Finally, the time required by each evacuee to travel a unit of distance is represented by the positive constant τ.

Moreover, the sub-path from a vertex v_i to vertex v_j in clockwise (resp. counter-clockwise) direction is denoted by $P^c[i, j]$ (resp. $P^k[i, j]$), and $d^c(i, j)$ (resp. $d^k(i, j)$) denote its length. The capacity or flow rate of a sub-path $P^c[i, j]$ (resp. $p^k(i, j)$) is the minimum edge capacity

in that path, denoted by $c^c(i,j)$ (resp. $c^k(i,j)$.

In our model, the sink has infinite capacity, therefore, the evacuation time for any supply already located at the sink is zero and its corresponding cost is also zero. We assume that all evacuees located at the vertices start to evacuate to the nearest sink at the same time. Moreover, the evacuee flow is confluent, which means, all the evacuees, initially located at a non-sink vertex along with the evacuees who arrive there later must evacuate along the same path to the sink (either in the clockwise direction or in the counter-clockwise direction). The confluent flow is desired in the context of evacuation planning to avoid confusion between the evacuees located at the same vertex regarding who moves in which direction.

However, in the case of cycle graph G, the supply at a vertex $v \in V$ can reach the sink x either in the clockwise direction or the counter-clockwise direction. Moreover, the flow is confluent, every vertex evacuates its entire supply in on the two directions. Since all vertices visited by some evacuation flow must use the same outgoing edge to evacuate, there exists one edge, called split edge. The vertices on one side of the split edge evacuate clockwise, and the vertices on the other side evacuate counter-clockwise.

Let $Z_x(e_i)$ denote the total cost of sink x with respect to the split edge $e_i \in E$, where $L(x,i)$ (resp. $R(x,i)$) represents the total cost of evacuating all the supply at the vertices that evacuate clockwise (resp. counter-clockwise) to sink x. Then,

$$Z_x(e_i) = L(x,i) + R(x,i) \tag{2.1}$$

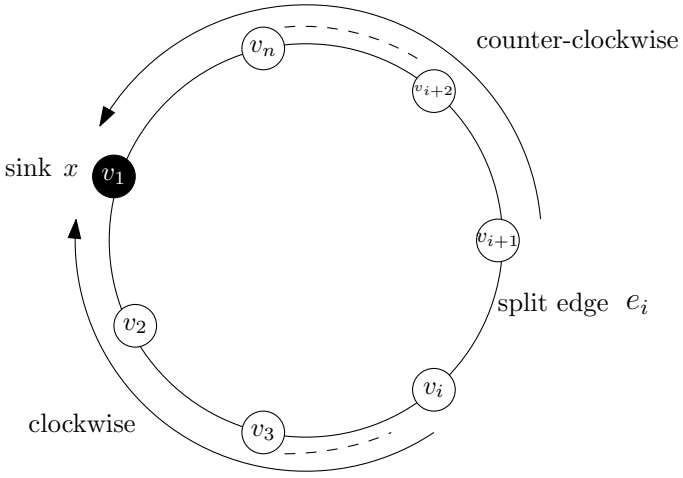


Figure 2.1: Evacuation path to sink x with respect to split edge e_i

In Figure 2.1, $L(x,i)$ (resp. $R(x,i)$) is the total cost to evacuate the supplies of the vertices from v_i to v_2 (resp. v_{i+1} to v_n) to sink x.

Now, the optimal cost of x is the total cost of x with respect to the optimal split edge.

Definition 2.1. ***Optimal split edge***

Let $G = (V,E)$ be an undirected cycle graph with the set of vertices V and the set of edges E. The optimal split edge $e_i \in E$ for the sink located at vertex $x \in V$ is the split edge for which the sum of the evacuation times required to send all the evacuees located at the vertices of G to x is minimum.

If Z_x denotes the optimal cost of sink x then it is defined by the equation below.

$$Z_x = Min_{e_i \in E}(Z_x(e_i)) \tag{2.2}$$

Finally, we need the optimal location of the 1-sink which minimizes the total evacuation cost over all the other sink locations on graph G. Regarding the location of the sink, we get the following theorem from Higashikawa et al. [9] in the context of minsum sink location problem.

Theorem 2.2. *If a sink location minimizes the total evacuation cost, then the sink must be located at a vertex.*

Proof. Let us consider a cycle graph $G = (V,E)$ of n vertices with e_n as the split edge. Let us also consider an optimal sink location at x on an edge $e_i \in E$ for $1 \leq i \leq n-1$. The shortest path from a supply vertex v to x visits either vertex v_i or v_{i+1}, $\forall v \in V$. Let the length of edge e_i is l and the distance of sink x from vertex v_i is d. Now, the cost for sink x is given by

$$Z_x(e_n) = L(x,n) + R(x,n)$$

where, $L(x,n)$ is given by

$$L(x,n) = L(i+1,n) + \tau(l-d) \sum_{i+1 \leq j \leq n} w_j$$

and $R(x,n)$ is given by

$$R(x,n) = R(i,n) + \tau d \sum_{1 \leq j \leq i} w_j$$

Therefore,

$$Z_x(e_n) = L(i+1,n) + R(i,n) + \tau l \sum_{i+1 \leq j \leq n} w_j + d\tau \Big(\sum_{1 \leq j \leq i} w_j - \sum_{i+1 \leq j \leq n} w_j \Big) \tag{2.3}$$

Without loss of generality, if we assume $\sum_{1 \leq j \leq i} w_j \leq \sum_{i+1 \leq j \leq n} w_j$, then $d = l$. So, we move the sink from x to vertex v_{i+1}, which gives a cost $Z_{i+1}(e_n) \leq Z_x(e_n)$. Therefore, it is proved that the sink on vertex v_{i+1} is optimal, since the assumption was that point x is optimal.
Thus, by Theorem 2.2, the following location model defines the minsum 1-sink location problem.

$$Minimize_{x \in V}(Z_x) \tag{2.4}$$

As defined in equation (2.4), we need to identify the optimal split edge (defined in equation (2.2)) for each sink location to get the optimal location of the 1-sink.

2.1.1 Properties of the minsum 1-sink on cycle networks

Let us consider a cycle graph $G = (V, E)$ of n ordered vertices, where a sink is located at some vertex $x \in V$.

Now, $L(x,i)$ (resp. $(R(x,i))$ is the sum of the costs of all such sections that evacuate clockwise (resp. counter clockwise) to the sink x with respect to the split edge e_i. Let I_x denote the cost of a section I to the sink x. So, the total clockwise cost $L(x,i)$ is given by

$$L(x,i) = \sum_{I \in I_L} I_x \tag{2.5}$$

where I_L is the set of sections that evacuates clockwise. The total counter-clockwise cost $R(x,i)$ is similar. In equation (2.5) the cost of a section I_x is computed by equation 1.1.

2.2 Brute force approach to solve the Minsum 1-sink location problem

In the brute force approach, we try every combination of the sink location and the split edge to find the optimal location of the 1 sink. We compute the total cost $Z_i(e_j)$ for $\forall v_i \in V$ with respect to each edge $e_j \in E$ as the split edge. Algorithm 1 shows the main procedure which is common for both the uniform and the arbitrary edge capacity cases. Algorithm 2 and 3 show the procedure to calculate the total evacuation cost of an arbitrary pair of sink location and split edge for the uniform and the arbitrary edge capacity cases respectively.

Algorithm 1: BRUTE FORCE APPROACH

1 **Input data:** cycle graph $G = (V, E)$ of n vertices; supply w_i at each vertex v_i; length l_j and capacity c_j for each edge e_j and constant τ. All are defined as in section 2.1.
2 **Output data:** Minsum cost Z and the location of optimal 1-sink.
3 $Z \leftarrow \infty$ //set the minsum cost as infinite
4 **for** i = *1 to n* **do**
5 $Z_i \leftarrow \infty$ // set the optimal cost of V_i
6 **for** j = *1 to n* **do**
7 //get the $L(i, j)$ and $R(i, j)$ by calling the function in Algorithm 2 or 3.
8 $Z_i(e_j) = L(i, j) + R(i, j)$ //cost of v_i with respect to split edge e_j
9 **if** $Z_i(e_j) < Z_i$ **then**
10 $Z_i \leftarrow Z_i(e_j)$
11 **end**
12 **end**
13 **if** *(*$Z_i < Z$*)* **then**
14 $Z \leftarrow Z_i$
15 $location \leftarrow i$
16 **end**
17 **end**

We describe Algorithm 1 in three nested steps as below.

- In line 4, the algorithm sets each vertex in the given cycle graph G as the sink location one by one and compute the optimal cost Z_i for $\forall v_i \in V$ (line 4 to 17). Note that there are $O(n)$ choices for the sink location in graph G. At the end of this process, the optimal 1-sink should be at the vertex v_i with minimum Z_i. .

- In line 6, the algorithm sets each edge as the split edge one by one to find the optimal split edge for $\forall v_i \in V$ (line 6 to 12). The optimal split edge is the split edge for which

we get the optimal Z_i. For each sink location, there are also $O(n)$ choices for the split edge.

- In line 8, the algorithm calls algorithm 2 (uniform case) or 3 (arbitrary case) to get the cost $Z_i(e_j)$, which individually takes $O(n)$ time to compute the cost.

Overall, the brute force approach takes $O(n^3)$ time for the three nested steps to find the optimal 1-sink on a cycle graph.

The algorithm to compute $L(i,j)$ (computation of $R(i,j)$ is similar) for $v_i \in V$ and $e_i \in E$ is as follows.

Algorithm 2: CALCULATE $L(i,j)$ FOR UNIFORM CAPACITY CASE

1 **Input data:** cycle graph $G = (V,E)$ of n vertices with the sink location i and split edge j; supply w_p at each vertex $v_p \in V$; length l_q and capacity c_q for each edge $e_q \in E$, and constant τ.
2 **Output Data:** returns cost $L(i,j)$.
3 $\eta \leftarrow i+1$ // set $i+1$ as the first cluster head
4 $cost \leftarrow 0$ //initialize the cost as zero
5 **for** k = i+2 *to* j **do**
6 **if** *the supply of k get congested at* η **then**
7 $w_\eta \leftarrow w_\eta + w_k$
8 **else**
9 //k becomes the cluster head of a new cluster
10 $cost \leftarrow cost + w_\eta s_\eta + \frac{w_\eta^2}{2c}$ // adding the cost of the previous cluster
11 $\eta \leftarrow k$// assign k as the new cluster head
12 **end**
13 **end**
14 $cost \leftarrow cost + w_\eta s_\eta + \frac{w_\eta^2}{2c}$ // adding the cost of the final cluster
15 *return cost*

We describe Algorithm 2 as follows,

- In line 3, we set the vertex v_{i+1} as the cluster head of the first cluster.
- After that, starting from the vertex v_{i+2}, we check for each vertex whether it gets congested at the current cluster head or not. If it is congested, then we extend the cluster by adding the supply in line 7. On the other hand, if it doesn't get congested, then the vertex forms a new cluster. So, we add the cost of the previous cluster in line 10 and assign vertex k as the cluster head of the new cluster in line 11. This process continues until we reach vertex v_j.
- Finally, we add the cost of the final cluster in line 14 and return the total cost of sink v_i in the clockwise direction for the sub-path $P^c[j, i+1]$.

Algorithm 3: CALCULATE $L(i,j)$ FOR ARBITRARY CAPACITY CASE

1 **Input data:** cycle graph $G=(V,E)$ of n vertices with the sink location i and split edge j; supply w_p at each vertex $v_p \in V$; length l_q and capacity c_q for each edge $e_q \in E$, and constant τ.

2 **Output Data:** returns cost $L(i,j)$.

3 $u \leftarrow i+1$ // set $i+1$ as the first section

4 $cost \leftarrow 0$ //initialize the *cost* as zero

5 **for** k = *i+2 to j* **do**

6 **if** *the first supply of k gets congested at u* **then**

7 **if** *k completely merges with u* **then**

8 $\sigma_u \leftarrow \sigma_u + w_k$ //extends the current section by adding the supply of k

9 **else**

10 //k partially merge with u

11 $\sigma_u \leftarrow \sigma_u + \alpha$ //α is the amount of supply from k merges with u

12 $cost \leftarrow cost + \sigma_u s_u + \frac{\sigma_u^2}{2h_u}$ //adding the cost of the current section

13 $w_k \leftarrow w_k - \alpha$ //deducting the partial supply α from the supply of k

14 $u \leftarrow k$ //k becomes a new section

15 **end**

16 **else**

17 //k becomes a new section

18 $cost \leftarrow cost + \sigma_u s_u + \frac{\sigma_u^2}{2h_u}$ //adding the cost of the current section

19 $u \leftarrow k$ //k becomes a new section

20 **end**

21 **end**

22 $cost \leftarrow cost + \sigma_u s_u + \frac{\sigma_u^2}{2h_u}$ //adding the cost of the final section

23 *return cost*

Algorithm 3 is similar to Algorithm 2 except from line 9 to 14. For the arbitrary case,

we check whether the supply of a vertex k completely merges with the current section or partially. If it is merged partially, then we calculate the partial amount of supply α and extend the current section by adding the α supply to it in line 11. After that we add the cost of the current section to the result in line 12 and assign vertex k as a new section with supply $w_k - \alpha$.

2.3 Efficient Algorithm to solve the minsum 1-sink location problem

We propose an efficient algorithm over the brute force approach by identifying some lemmas and designing a useful data structure. For better understanding, we split our goal into two objectives.

Objective 1:

We design an efficient algorithm which is optimized over the brute force Algorithm 1. In this thesis, we might refer to this efficient algorithm as the "Common Algorithm," unless stated otherwise. The algorithm is designed to solve the minsum 1-sink location problem on cycle networks with both the uniform and arbitrary edge capacity cases.

Objective 2:

We design a data structure which helps to compute the total evacuation cost $Z_i(e_j)$ for the sink at some vertex $v_i \in V$ and the split edge at some edge $e_j \in E$ efficiently. We design the data structure for the uniform case, then extend the data structure to solve the problem in the arbitrary case.

2.3.1 Technique to solve Objective 1

The vertices are ordered counter-clockwise in our model, therefore, we move both the location of the sink and the split edge for each sink location in the counter-clockwise direction. We now prove the following two lemmas to find the optimal split edge for all the

vertices of V efficiently.

Lemma 2.3. *Let G be a cycle graph with $n \geq 3$ vertices. If $Z_i(e_{j-1}) \leq Z_i(e_j)$ then $Z_i(e_j) \leq Z_i(e_{j+1})$*

Proof: According to equation (2.1) and (2.5), the cost of v_i with respect to split edge e_{j-1} is,

$$Z_i(e_{j-1}) = \sum_{I \in I_L} L_i(I) + \sum_{I \in I_R} R_i(I) \tag{2.6}$$

where, I_L (resp. I_R) is the set of sections between vertices v_{j-1} to v_i in clockwise (resp. v_j to v_i in counter-clockwise) direction.

So, the cost of v_i respect to split edges e_j and e_{j+1} are as follows.

$$Z_i(e_j) = \sum_{I \in I_L + P[v_j, v_j]} L_i(I) + \sum_{I \in I_R - P[v_j, v_j]} R_i(I) \tag{2.7}$$

$$Z_i(e_{j+1}) = \sum_{I \in I_L + P[v_j, v_{j+1}]} L_i(I) + \sum_{I \in I_R - P[v_j, v_{j+1}]} R_i(I) \tag{2.8}$$

We find from equation (2.10) and (2.11) that the cost to evacuate all the supply located at all other vertices remains the same except for vertex v_j.

Let $s_i^c(j)$ (resp. $s_i^k(j)$) denote the start time of the vertex v_j when the supply moves clockwise (for split edge e_j) (resp. counter-clockwise (for split edge e_{j-1})) direction to the sink v_i. The start time of a vertex is the time when the first unit of that vertex reaches the sink.

As $Z_i(e_{j-1}) \leq Z_i(e_j)$, so from equation (2.10) and (2.11),

$$\sum_{I \in I_L} L(I) + \sum_{I \in I_R} R(I) \leq \sum_{I \in I_L + P[v_j, v_j]} L(I) + \sum_{I \in I_R - P[v_j, v_j]} R(I)$$

or,

$$w_j s_i^k(j) + \frac{w_j^2}{2c^k(j,i-1)} + \sum_{I \in I_L} L(I) + \sum_{I \in I_R - P[v_j, v_j]} R(I) \leq w_j s_i^c(j) + \frac{w_j^2}{2c^c(j-1,i)} + \sum_{I \in I_L} L(I) + \sum_{I \in I_R - P[v_j, v_j]} R(I)$$

so,

$$s_i^k(j) + \frac{w_j}{2c^k(j,i-1)} \leq s_i^c(j) + \frac{w_j}{2c^c(j-1,i)} \tag{2.9}$$

Now, according to the fundamental constraint of evacuation planning, if the supply at v_j and v_{j+1} both evacuate in counter-clockwise direction then the relation between their start time as follows.

$$s_i^k(j) \geq s_i^k(j+1) + \frac{w_{j+1}}{c^k(j+1,i-1)} \tag{2.10}$$

and if the both v_j and v_{j+1} evacuates in clockwise direction then the relation as follows.

$$s_i^c(j+1) \geq s_i^c(j) + \frac{w_j}{c^c(j-1,i)} \tag{2.11}$$

Finally, by combining equation (2.13), (2.14) and (2.15), we get,

$$s_i^c(j+1) \geq s_i^k(j) + \frac{w_j}{2c^k(j,i-1)} - \frac{w_j}{2c^c(j-1,i)} + \frac{w_j}{c^c(j-1,i)}$$

or,

$$s_i^c(j+1) \geq s_i^k(j+1) + \frac{w_{j+1}}{c^k(j+1,i-1)} + \frac{w_j}{2c^k(j,i-1)} + \frac{w_j}{2c^c(j-1,i)}$$

So,

$$s_i^c(j+1) \geq s_i^k(j+1) \tag{2.12}$$

From equation (2.11) and (2.12), the cost to evacuate all the supply located at all the other vertices remains the same except for vertex v_{j+1}, and we find in equation (2.16) that the start time of the vertex v_{j+1} in clockwise direction is greater than or equal to the start time

in counter-clockwise direction. Therefore, the cost of v_i respect to split edge e_j is less than or equal to split edge e_{j+1}, and the lemma is proved. We can see in line 15 of Algorithm 4 that the algorithm stops moving to the next split edge for the current sink location when the lemma 2.3 satisfies in line 11.

Lemma 2.4. *Let G be a cycle graph with $n \geq 3$ vertices. If $Z_i(e_{j-1}) > Z_i(e_j)$ then $Z_{i+1}(e_{j-1}) > Z_{i+1}(e_j)$*

Proof: Here, we use the same procedure that we used to prove Lemma 2.3. When we calculate the cost of v_i respect to the adjacent split edges e_{j-1} and e_j, the cost to evacuate all the supply located at all other vertices remains the same except for vertex v_j.
As we consider $Z_i(e_{j-1}) > Z_i(e_j)$, so similar to equation (2.13), the time to evacuate v_j in counter-clockwise direction is greater than the time in clockwise direction.

$$s_i^k(j) + \frac{w_j}{2c^k(j,i-1)} > s_i^c(j) + \frac{w_j}{2c^c(j-1,i)} \tag{2.13}$$

Now, according to the fundamental constraint of evacuation planning, if the supply at v_j evacuates counter-clockwise then for arbitrary edge capacities,

$$c^k(j,i-1) \geq c^k(j,i)$$

so, we can write,

$$\frac{w_j}{2c^k(j,i)} \geq \frac{w_j}{2c^k(j,i-1)} \tag{2.14}$$

and, the start time of v_j respect to sink v_{i+1} is greater than the start time respect to sink v_i.

$$s_{i+1}^k(j) > s_i^k(j) \tag{2.15}$$

We can combine the equation (2.18) and (2.19) as below,

$$s^k_{i+1}(j) + \frac{w_j}{2c^k(j,i)} > s^k_i(j) + \frac{w_j}{2c^k(j,i-1)} \tag{2.16}$$

Similarly, if v_j evacuates in clockwise direction, we find the inequality as follows.

$$s^c_i(j) + \frac{w_j}{2c^c(j-1,i)} > s^c_{i+1}(j) + \frac{w_j}{2c^c(j-1,i+1)} \tag{2.17}$$

Now, from equation (2.17), (2.20) and (2.21), we get the inequality as follows,

$$s^k_{i+1}(j) + \frac{w_j}{2c^k(j,i)} > s^c_{i+1}(j) + \frac{w_j}{2c^c(j-1,i+1)} \tag{2.18}$$

where, left hand side is the total time to evacuate v_j for the split edge e_{j-1} and right hand side is for the split edge e_j.
Therefore, the cost of v_{i+1} with respect to split edge e_{j-1} is greater than with respect to the split edge e_j and the lemma is proved.

So, we come to a conclusion from Lemma 2.4 that if the optimal split edge of $v_i \in V$ is $e_j \in E$ then the optimal split edge of $v_{i+1} \in V$ should be e_j or in counter-clockwise order from e_j to e_i. We can see in line 16 of Algorithm 4, the algorithm sets the starting split edge for v_{i+1} to the optimal split edge of v_i. So, for the next location of the sink, it does not start from the beginning of start index of split edges.

Algorithm 4: COMMON ALGORITHM FOR MINSUM 1-SINK ON CYCLE NETWORKS

Input Data: cycle graph G of n vertices; supply w_i at each vertex v_i; length l_j and capacity c_j for each edge e_j and constant τ. See section 2.1.

Output Data: Location of optimal 1-sink $x \in V$ and minsum cost Z for the optimal location of the 1-sink.

$Z \leftarrow \infty$ //set the minsum cost to infinity

$j \leftarrow 1$ //initialize j to select the first split edge

for i = *1 to n* **do**

  $Z_i \leftarrow \infty$ //set the optimal cost of v_i

  $done \leftarrow false$

  while *(!done)* **do**

    $Z_i(e_j) = L(i,j) + R(i,j)$ //cost of v_i respective to e_j

    //get the $Z_i(e_j)$ using the data structure (showed in chapter 4)

    if $(Z_i(e_j) < Z_i)$ **then**

      $Z_i \leftarrow Z_i(e_j)$ //set the optimal cost of v_i, e_j is the optimal split edge

      $j \leftarrow j+1$

    else

      $done \leftarrow true$ //finalize the optimal cost of i

      $j \leftarrow j-1$ // set $j-1$ as the starting split edge for $i+1$.

      if $(Z_i < Z)$ **then**

        $Z \leftarrow Z_i$ //set Z_i as the minsum cost

        $location \leftarrow i$ //set i as the optimal location

      end

    end

  end

end

2.4 Running time analysis of Algorithm 4

Let the time to compute the cost of a sink $v_i \in V$ with respect to a split edge $e_j \in E$ (computation of $Z_i(e_j)$ in algorithm 4 line 9) is T_n. The operation to place a sink at the vertex $v_i \in V$ costs 1 and the operation to consider an edge as a split edge costs 1. So, the brute force approach (Algorithm 1) takes $O(n^2.T_n)$ time to find the optimal location of the 1 sink.

Lemma 2.5. *Given an undirected cycle graph $G = (V,E)$ of n vertices. We can find the optimal split edges for all the vertices of G in $O(n.T_n)$ time, where T_n is the time to compute each $Z_i(e_j)$.*

Proof: In our efficient procedure (Algorithm 4), the number of operation to place the sink at each vertex remains the same but the number of operation to consider each edge as the split edge for a particular sink location is reduced. If the algorithm starts with the sink at vertex v_i; therefore, it computes the cost as follows,

$$Z_i(e_i)$$

$$Z_i(e_{i+1})$$

.....

$$Z_i(e_j)$$

$$Z_i(e_{j+1})$$

At this point, if $Z_i(e_j) \leq Z_i(e_{j+1})$ then according to Lemma 2.3 the optimal split edge for v_i is e_j and $Z_i(e_j)$ is the optimal cost (see line 15). Now the algorithm starts searching for the optimal cost of the sink at the vertex v_{i+1} but rather than starting with the edge e_{i+1} it considers the optimal split edge of v_i as the first candidate for v_{i+1} according to Lemma 2.4. Note that for each sink location, there is one cost computation for each sink and optimal

split edge. There are $O(n)$ such computations. Then, for every candidate sink location, there are additional cost computations with the sink and other split edges that turn out not to be optimal. However, since computations involving a new sink start with the optimal split edge for the previous sink, there are $O(n)$ computations with non-optimal sink edges.

Now, according to Lemma 2.4, the optimal split edges of the vertices (sinks) move in the same direction as the vertices move. So, if the optimal split edge of v_{i-1} crosses the optimal split edge of v_i while moves in the counter-clockwise direction it contradicts the Lemma 2.4 while it moves in the clockwise direction. Thus, we come to a conclusion that the algorithm traverses all the edges at most twice except the repeated edges, which costs at most $2n$.

Finally, after combining Lemma 2.3 & 2.4, we argue that the operation to identify the optimal split edge of all the vertices of V costs $2n + 2n = 4n$. Therefore, the number of operations to find the optimal split edge for each sink location is amortized constant. The final time complexity of algorithm 4 is $O(n.T_n)$.

Chapter 3

Cluster head tree (CH-tree)

The data structure we designed is based on dynamic path networks, that allows us to maintain the information regarding the congestion on a path or a cycle network. The hierarchical construction of the data structure permits it to answer a batch of sequential queries, where the answer to a single query depends on the clusters sequence with respect to that specific sub-path. The same data structure is also described in the M.Sc. thesis of Rajib Das [5], who has an equal contribution in designing the data structure. Rajib Das used this data structure to solve the sink location problems with minimax objective function.

For simplicity, first, we show the construction and the use of the data structure for the uniform capacity case. Then we show an extension of the data structure for the arbitrary edge capacity case. The complexities of the arbitrary case over the uniform case are listed below.

- A cluster may have multiple sections of different heights.
- Merging of clusters/sections does not occur orderly with the order of the clusters/-sections. Two clusters/sections can merge anytime when some certain conditions are met.
- The height, duration or number of evacuees carried by a section may vary with the considered sub-path.

Therefore, we need some enhancement in the data structure so that it can overcome these

complexities and answer the batch of queries efficiently. We name the data structure the "Cluster Head tree" (CH-tree, for short).

3.1 Structure of the CH-Tree in uniform capacity case

The cluster head tree (CH-tree), denoted by T, is defined for a dynamic path network. Let us consider an undirected path graph $P = (V,E)$, where $V = \{v_1, v_2, v_3, ..., v_n\}$ is an ordered set of n vertices and $E = \{e_1, e_2, ..., e_{n-1}\}$ is the set of edges. Let $N = (P, w, l, c, \tau)$ be a dynamic path network; where all the other preliminaries of N are same as mentioned in Section 2.1, unless stated otherwise. Now, we want to compute the evacuation cost of the supply on a sub-path $P[i, j]$ to a sink x located on vertex v_{i-1}, where $i < j$. For this, we need to identify the cluster sequence in the sub-path $P[i, j]$. More precisely, we need the cluster heads of the clusters containing the supply from v_j.

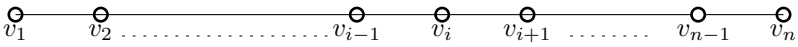


Figure 3.1: An undirected path graph P of n ordered vertices

The CH-tree stores the cluster head of every cluster on the path for evacuation flow in a fixed direction. Without loss of generality, in this thesis, we present the structure of the CH-tree in the descending order of the vertices, where the flow is oriented from v_n to v_1. At the end of the construction, the algorithm develops a forest that contains less than or equal to n trees with respect to the sink location at v_1. Each tree in the forest represents individual clusters where the root of the tree is the cluster head. For a better demonstration, we create CH-tree T by connecting the root of the trees in the forest to a dummy root ρ (see Figure 3.2). Therefore, The nodes [1] of the tree T are the vertices of P except the root node. Thus, vertex v_i in the path corresponds to node i in the CH-tree. Let $T(i)$ denote a sub-tree in T rooted at node i, where node $x(! = \rho)$ is the parent of i and node y is the node with the largest index. Then, $x < i$ and sub-tree $T(i)$ contains all the node with indices between i to

[1] We use the term "node" to distinguish between the tree T and path P. A node in T is the same vertex in P except the dummy root ρ in T.

y.

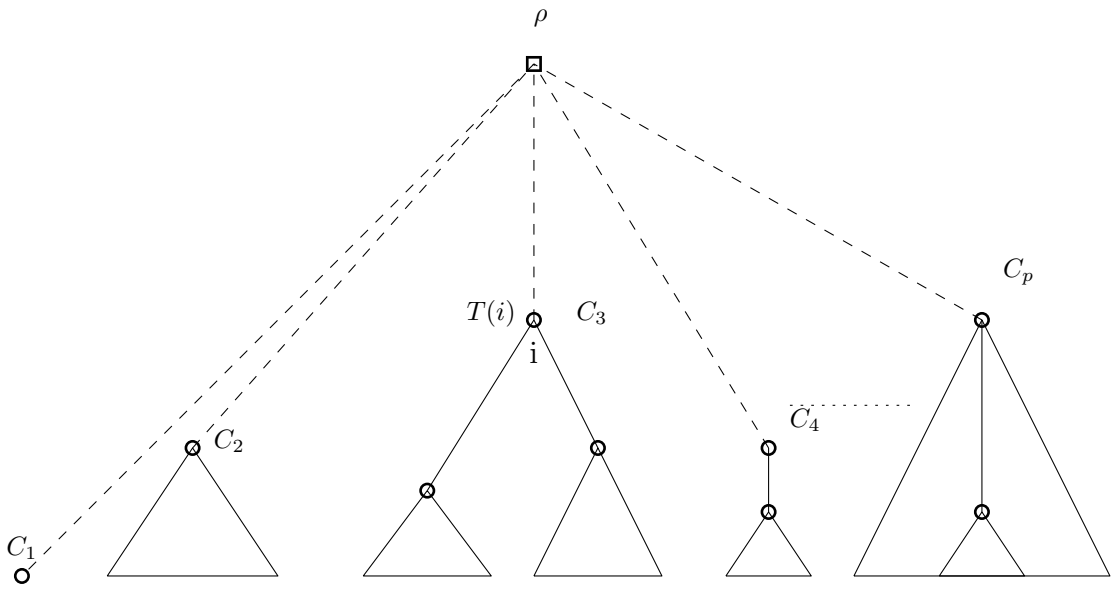


Figure 3.2: CH Tree T on the path graph P

3.2 Construction of the CH-Tree in the uniform capacity case

In this section, we describe an $O(n)$ algorithm that constructs the CH-tree on the dynamic path network N. The algorithm that we use to construct the CH tree in uniform case follows the template of the algorithm proposed by Higashikawa et al. [9] for the 1-sink location problem on dynamic path networks. The main idea is to place the sink at each vertex of the path graph P in sequential order and construct the tree according to the updated cluster information with respect to each location of the sink.

Basically, the algorithm places the sink at every vertex $v_i \in V$ for $1 \leq i \leq n-1$ in descending order of i and constructs the tree T by computing the cluster sequence on the sub-path $P[n, i+1]$. First, the algorithm places the sink at v_{n-1}, and then compute the clusters in the evacuation path $P[n, n-1]$. As there are no intermediate vertices between v_n and v_{n-1}, vertex v_n forms the first cluster C_1. The supply carried by the cluster is $\sigma_1 = w_n$. The algorithm adds the node v_n in the tree T. Thus, $v_n (= \eta_1)$ is the cluster head of cluster C_1.

Now, suppose that we have already constructed the tree T on the sub-path $P[v_n, v_{i+1}]$ by computing the set of clusters $C_L = \{C_1, C_2, ..., C_p\}$ for the sink located at some vertex v_i satisfying $2 \leq i \leq n-1$, where $1 \leq p < n$ and C_1 is the first cluster arrives at sink v_i. Note that vertex $v_{i+1}(= \eta_1)$ is the cluster head of C_1.

Next, we show how the algorithm updates the structure of the tree T when we move the sink to vertex v_{i-1}. Thus, vertex v_i is a new vertex encountered in the front, so that the evacuation path extends as $P[n, i]$. First, the algorithm sets v_i as a new cluster C_0 in front of the already obtained cluster sequence, thus, $\sigma_0 = w_i$ and $\eta_0 = v_i$. Thus, the algorithm adds the node v_i as the leftmost node in tree T.

After that, the algorithm runs a test for the clusters in C_L in the ascending order of the clusters by equation (3.1).

$$\tau d(\eta_j, \eta_0) \leq \frac{(\sigma_0)}{c} \tag{3.1}$$

If equation (3.1) is true for a cluster C_j satisfying $1 \leq j \leq p$, then the cluster will merge with cluster C_0. Therefore, the algorithm updates the structure of the tree T by connecting the cluster head node of cluster C_j as the right most child node of the cluster head node of cluster C_0. Therefore, it also sets $\sigma_0 = \sigma_0 + \sigma_j$.

The algorithm continues to test the clusters in C_L one by one until it finds a cluster C_r such that $\tau d(\eta_r, \eta_0) \geq \sigma_0/c$ for $1 \leq r \leq p$ or it reaches the last cluster C_p to test. At the end of the testing, we get an updated set of clusters with respect to the sink at v_{i-1}. Also, the tree T is constructed for the sub-path $P[v_n, v_i]$. Now, in the next recursive step, the algorithm places the sink at v_{i-2} and repeats the same procedure to update the structure of CH-tree. The algorithm finishes constructing the tree T when we get the cluster sequence corresponding to the sink location at v_1. Finally, we connect all the cluster head nodes of the sub-trees in the forest to the dummy root ρ to create the CH-tree.

3.2.1 Complexity of the construction algorithm

Lemma 3.1. *Given a path graph $P = (V, E)$ of n vertices. If the edge capacities are uniform, we can construct CH-tree T in $O(n)$ time.*

Proof: There are n iterations to construct the CH-tree, where, at each iteration exactly one node is added to the tree. In each iteration, the algorithm does a test that checks for merging of the already obtained clusters as long as two clusters merge. For n iterations, there are $O(n)$ clusters and they can merge $O(n)$ times. Moreover, for each iteration, the sequence of tests ends with a negative result for merge. Thus, there are n such negative tests for n iterations. Finally, the time complexity of the construction algorithm is $O(n)$, where the average cost per iteration is amortized constant.

3.3 Use of the CH-Tree in uniform edge capacity case

Let us consider; we are given CH-tree T constructed on path P as described in Section 3.1. Now, we show the algorithm to answer an arbitrary query $Q(i, j)$ for $1 \leq i \leq j \leq n$, where the sink is located at vertex v_i. The answer to the query is the sum of the evacuation times of all the evacuees in the sub-path $P[i, j]$. Thus, we need to identify each cluster in the sub-path and calculate the costs of the clusters. Total cost of a sub-path $P[i, j]$ is defined by the sum of the costs of all the clusters in the sub-path, denoted by $cost(i, j)$.

First, the algorithm gets the index of node $i+1$ in T in constant time. Next, it starts to traverse the tree T top-down starting from node $i+1$. Note that the sub-tree $T(i+1)$ is the first cluster in the sub-path $P[i+1, j]$. During traversal, the rightmost leaf node of a sub-tree indicates the last vertex of its corresponding cluster.

Now, let us consider, the algorithm identifies the shape of a cluster C_m by traversing the sub-tree $T(k)$, where k is the cluster head node for $i+1 \leq k \leq j$. If $cost(m)$ denote the cost

of the cluster C_m, then it is given by equation (1.1) as follows.

$$cost(m) = s_m\sigma_m + \frac{\sigma_m^2}{2c}$$

where, $s_m = \tau d(k,i)$ is the time when the first unit of cluster C_m reaches sink v_i and $\sigma_m = \sum_{k\leq y\leq l} w_y$ is the total supply carried by cluster C_m, where l is the right most leaf node of the sub-tree $T(k)$ (or the last vertex of the cluster C_m) for $k \leq l \leq j$. After that, the algorithm updates $cost(i,j)$ as follows.

$$cost(i,j) = cost(i,j) + cost(m) \tag{3.2}$$

Claim 3.2. *In a cluster head tree, T, If a sub-tree $T(k)$ (rooted at node k) is identified as a cluster with node l as the descendant with maximum index (while traversing in the ascending order of the vertices), then the sub-tree $T(l+1)$ represents the next cluster with node $l+1$ as the cluster head.*

In the next step, the algorithm continues to traverse the tree top-down to identify the next cluster in the sub-path $P[i+1,j]$ starting from node $l+1$.

The algorithm repeats the same procedure to compute the cost of each cluster in the sub-path $P[i+1,j]$ until it reaches the last sub-tree that contains the node j.

Lemma 3.3. *Given a cluster head tree T for a path network $P = (V,E)$ of n vertices. If the edge capacities are uniform, we can compute the total cost of an arbitrary sub-path $P[i,j]$ to sink v_i in $O(j-i)$ time, where $i < j$.*

Proof: To traverse a sub-tree $T(u)$ (representing a cluster) of T, it takes $O(1)$ time to get the index of u and total $|T(u)|$ time to traverse the sub-tree $T(u)$, where $|T(u)|$ is the number of vertices spanned by u. Finally, it takes $O(1)$ time to compute the cost of $T(u)$ with respect to the sink location. Now, to compute the total cost of any sub-path $P[i,j]$ to

sink v_i, the algorithm needs to traverse all the sub-trees in the sub-path, which takes $O(j-i)$ time for $i < j$, where $j-i$ is the size of the path we are querying. So, the lemma is proved.

3.4 Structure of the CH-tree in the arbitrary capacity case

As this is an extension, we use the same names, definitions and notations with some additional notations (which are defined when needed) to describe CH-tree T in the arbitrary capacity case. The main idea of the construction remains the same as described in Section 3.2. We place the sink at every vertex of the path graph $P = (V, E)$ (see Figure 3.1) one by one from vertex v_n to v_1 and update the structure of CH-tree with respect to each location of the sink. However, unlike the uniform case, the CH-tree needs to handle two complications in order to compute the evacuation cost of the supply between two vertices v_i and v_j to a sink located at vertex v_{i-1} for $i < j$. To handle the complications, we extend the structure of the CH-tree for the arbitrary case as follows.

Label the edges of the CH-tree:

During the construction, when we move the sink from vertex v_{i+1} to v_i, several clusters can merge because of a smaller capacity of the edge e_i. More importantly, two merged clusters can be located anywhere in the already obtained cluster sequence, as shown in Figure 3.3. Therefore, to know the cluster sequence with respect to a location of the sink, we label the edges of the CH-tree with the index of the sink location for which the clusters get merged.

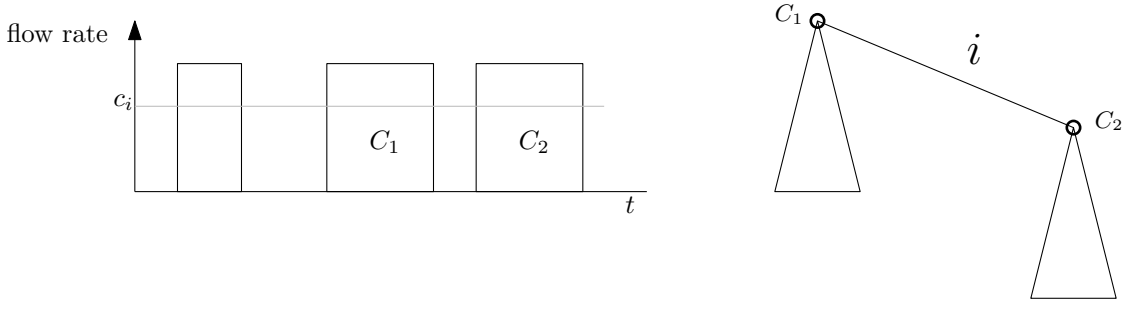


Figure 3.3: Cluster C_1 and C_2 merge because of the smaller capacity c_i

Store necessary information in the cluster head node:

In the arbitrary case, unlike the uniform case, there can be mixed clusters along with the simple clusters. Therefore, we need to know the rate of the sections, as it is variable. We observe that, at every iteration in the construction algorithm, the rate of $O(n)$ sections can change $O(n)$ times. Thus, updating this information at every iteration can lead to an $O(n^2)$ algorithm. Therefore, we choose not to store the rate (or height) of a section explicitly. Each time we need the rate of a section with first vertex x, the rate is the minimum edge capacity of the sub-path between vertex x and the sink location. To get the rate of a section efficiently, we construct a capacity tree ζ, which is a standard binary search tree. We use the tree ζ to get the minimum capacity $c(x,y)$ in the sub-path $P[x,y]$ for $\forall x,y \in \{1,2,3,...,n\}$ in $O(\log n)$ time. However, to obtain the sections in a mixed cluster, we store other necessary information in the cluster head node. The information stored in a cluster head is nothing but the information of the next section in the same mixed cluster with respect to a sink location, denoted by ϕ_u for node u. As a simple illustration, consider Figure 3.4(b), where we have two clusters C_a and C_b with respect to the sink at some vertex i as shown in Figure 3.4(a), then in Figure 3.4(c), the clusters get merged into a mixed cluster C_a with four sections I_1, I_2, I_3, I_4 with respect to the sink at vertex $i-1$.

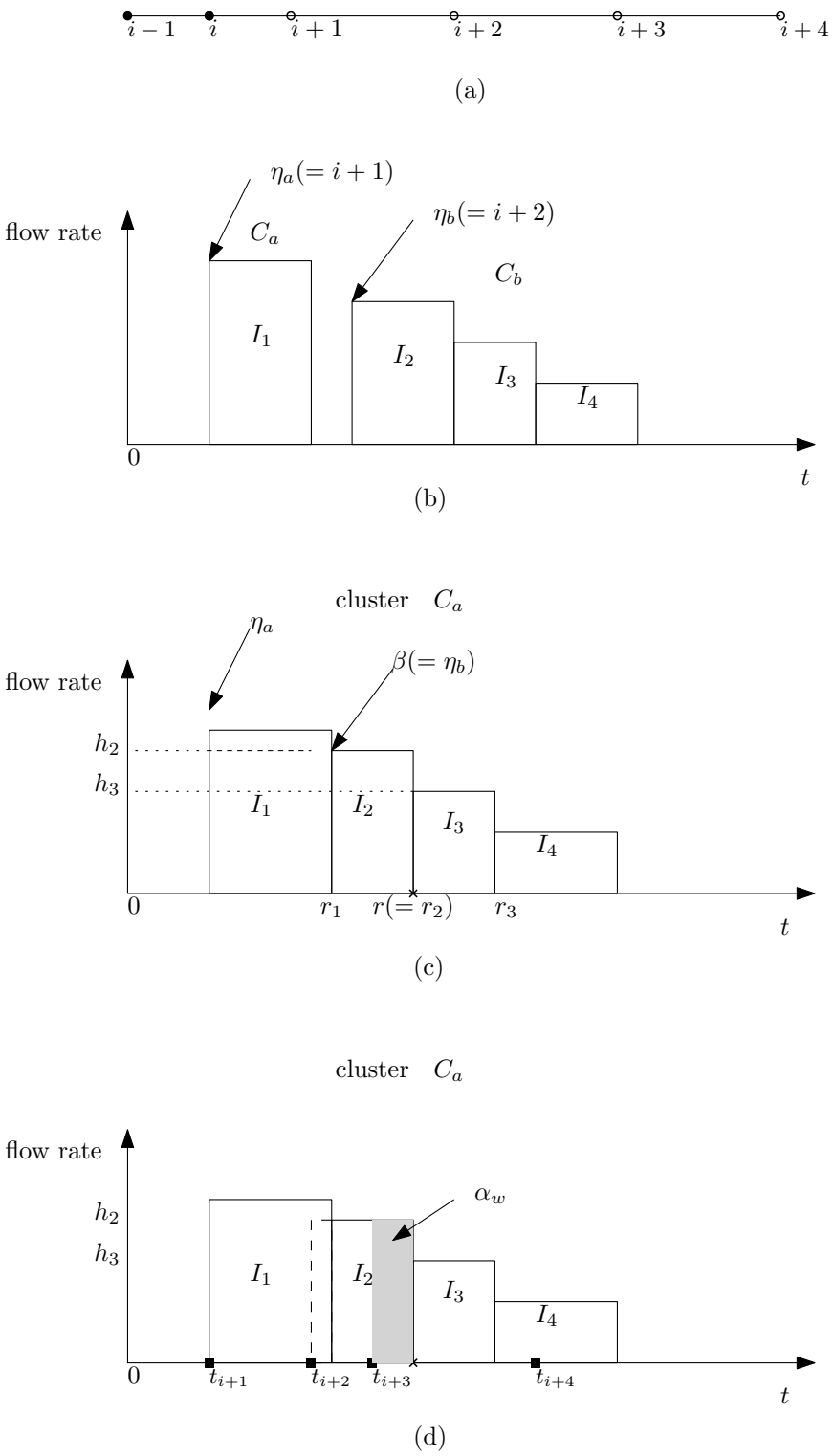


Figure 3.4: (a) A path graph; (b) sequence of clusters C_a and C_b when the sink is at i; (c) a mixed cluster C_a with three section I_1, I_2, I_3, I_4 by merging of C_a and C_b after moving the sink to $i-1$; (d) cluster C_a with the the time when the first supply from a vertex reaches sink $i-1$

For mixed cluster C_a, similarly, we do not store any information of section I_1, as the rate of I_1 can change $O(n)$ times for $O(n)$ different edge capacities. However, we can store necessary information regarding the sections I_2, I_3 and I_4 as their rates do not change as long as the cluster consists of sections I_1, I_2, I_3, I_4. Thus, we store the following information of section I_2 in the cluster head node η_a with respect to the sink location for which both clusters get merged. Note that the similar information was also stored regarding section I_3 in the head of section I_2, when I_2 was the first section of cluster C_b consisting of I_2, I_3 and I_4.

s = The index of the sink location for which the mixed cluster is created or the shape of the cluster is updated significantly.

β = The index of the cluster head of cluster C_b that C_a just merged with. In Figure 3.4(b), it is the index of the *first vertex* of section I_2, $\beta = \eta_b$.

r = The end time of section I_2 with respect to s (the index of the sink location). For cluster C_a, $r = r_2$. It is the time when the last supply of the section reaches the sink. we can compute r to be stored in η_a from the previously stored information in η_b as follows.

$$(r - r_1)h_2 + (r_3 - r)h_3 = \sum_{\eta_b \le y \le \alpha - 1} w_y + \alpha_w$$

$$r = \frac{1}{h_2 - h_3}\Big(\sum_{\eta_b \le y \le \alpha - 1} w_y + \alpha_w - r_3 h_3 + r_1 h_2 \Big) \tag{3.3}$$

where, index α and supply α_w are defined similarly as follows but stored in cluster head η_b of section C_b.

α = The index of the vertex from which a partial amount of supply evacuates as some supply of section I_2. The partial amount of supply is defined as the portion of supply in a section whose originating vertex is not included in the section, therefore, it is always the last portion of the supply in a section. For any cluster, the partial amount

of supply can be zero. As a simple illustration, let us consider Figure 3.4(d), which shows the time when the first supply of a vertex reaches sink $i-1$. In the mixed cluster C_a, the cluster head η_a is the vertex $i+1$, where, the first supply of $i+1$ reaches sink at time t_{i+1}. Similarly, the first supplies of vertices $i+2, i+3, i+4$ reach the sink at times $t_{i+2}, t_{i+3}, t_{i+4}$ respectively. Now, from the definition of α, for cluster C_a shown in Figure 3.4(a) and (b), we store the index $i+3$ as α in cluster head node η_a.

α_w = The partial amount of supply in the total supply of sections I_1 and I_2 originating from vertex α, which is given by

$$\alpha_w = (r_1 - s_1)h_1 + (r_2 - r_1)h_2 - \sum_{\eta_a \leq y \leq \alpha - 1} w_y \tag{3.4}$$

Using this stored information, if we are given the cluster head of a cluster, then we can visit the sequence of the nodes in the CH-tree which identify the different sections in the mixed cluster. Now, if we wish to compute the cost of a cluster with respect to the sink location at some vertex i, we use equation (1.1) to compute the cost of each section in the cluster, where we need three information regarding any section; flow rate, start time and total supply. We use the stored index β to compute the start time s of section, which is $\tau d(\beta, i)$. Also, the index β is used to get the rate of the section from the capacity tree ζ, while for section I_1 we use the cluster head η_a as we do not store any information regarding I_1. Lastly, we need to know the area of the section to identify the total supply in the section. Thus, in addition to the rate of the section, we need to know the duration δ_t of the section, which is $r-s$. Note that, we have stored the r of each section in the CH-tree except for section I_1. However, we can compute the r for section I_1 using equation (3.3). Moreover, during traversing the CH-tree, we use the index β to iterate each of the sections in a cluster one by one.

3.5 Construction of the CH-tree in the arbitrary edge capacity case

We mentioned in Section 3.4 that, in arbitrary case the merge between two clusters can take place anywhere in the cluster sequence. Therefore, the algorithm maintains a critical capacity tree H during the construction process of CH-tree T to identify the merging of the sections/clusters because of the new sink location.

Definition 3.4. ***Critical Capacity***

The critical capacity of a section is the expected height or flow rate of the section for which the leftover evacuees of that section fills the gap or the step with the next adjacent section. A section's critical capacity is always lower than it's current height/flow rate.

The critical capacity tree H is a max heap tree in which we store or update the critical capacity corresponding to each section. The critical capacity of a section I_i is denoted by μ_i. It takes $O(\log n)$ time to insert/remove a critical capacity into/from the H tree and $O(1)$ time to get the maximum critical capacity.

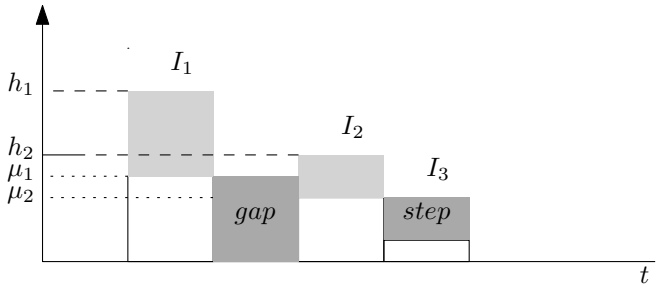


Figure 3.5: Critical Capacity of I_1 and I_2

Figure 3.5 shows that the critical capacity μ_1 of section I_1 fills the gap (dark gray) between I_1 and I_2 by the leftover evacuees (light gray) of I_1, where μ_2 fills the step between I_2 and I_3.

The critical capacity μ_i of a section I_i that fills the step with the next adjacent section I_{i+1} is given by

$$\mu_i = \frac{h_i \delta_{t_i} + h_{i+1} \delta_{t_{i+1}}}{\delta_{t_i} + \delta_{t_{i+1}}} \tag{3.5}$$

where δ_{t_i} and $\delta_{t_{i+1}}$ are the duration of section I_i and I_{i+1} respectively.
The critical capacity μ_i of a section I_i that fills the gap with the next adjacent section I_{i+1} is given by

$$\mu_i = \frac{h_i \delta_{t_i}}{\tau d(\eta_{k+1}, \eta_k)} \tag{3.6}$$

where sections I_i and I_{i+1} belong to the clusters C_k and C_{k+1} respectively.
Equation (3.6) replaces the height and duration of the next adjacent section of equation (3.5) by the gap. The gap has no height and the duration of cluster C_k should be long enough to the make the gap duration to 0.

We now show the steps of the construction of CH-tree in arbitrary capacity case. The algorithm starts with placing the sink at v_{n-1} so that the supply at v_n evacuates to the sink. Therefore, v_n becomes the first simple cluster/section with the same vertex as the cluster head. The algorithm adds vertex v_n as a node in the tree T. At this point, the critical capacity tree $H = \emptyset$ as there is only one section in T.

Now, suppose that we have already constructed the tree T in the sub-path $P[v_n, v_{i+1}]$ by computing a set of sections with respect to the sink at some vertex v_i satisfying $2 \leq i \leq n-1$. The algorithm also populates H tree by the critical capacities corresponding to the each computed sections arriving at sink v_i. We then show how the algorithm updates the cluster/section sequence information after moving the sink to vertex v_{i-1}.

First, the algorithm inserts the node v_i in the tree T as a simple cluster C_0 (section I_0) with vertex v_i (=η_0) as the cluster head. The section I_0 has the height h_0 (=c_{i-1}) and duration δ_{t_0} (=$\frac{w_i}{c_{i-1}}$). Next, the algorithm uses equation (3.6) to calculate the critical capacity μ_0 that fills the gap between sections I_0 and I_1 and inserts μ_0 into the H tree.

After that, the algorithm starts processing the critical capacities in the H tree to update

the cluster/section sequence respect to the new sink location v_{i-1}. The processing does compare the critical capacities one by one with the newly encountered departing capacity, starting from the maximum value in H. A critical capacity in H is successfully processed if it is higher than the current departing capacity. The new departing capacity is c_{i-1} for the sink location at v_{i-1}. Now, as a simple illustration, see Figure 3.6, where $I_1, I_2, I_3, I_4, ...$ are the already computed sections arriving at sink v_i. The successful processing of a critical capacity corresponding to a section either fills the gap or the step with its next adjacent section; therefore they get merged.

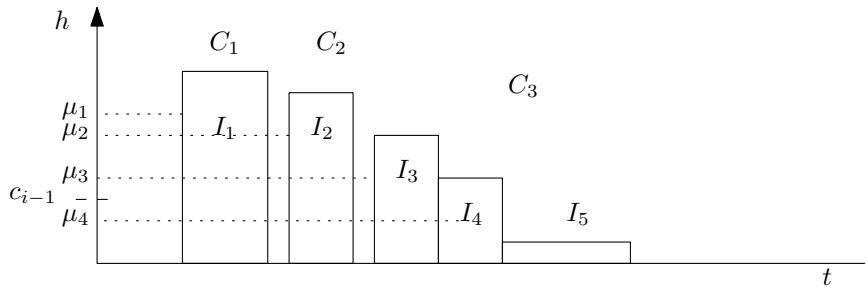


Figure 3.6: Sequence of arriving sections with their critical capacities and the new departing capacity c_{i-1}

The following two situations can arise to these sections when their corresponding critical capacity in H is higher than the new departing capacity.

Case 1:

In this case, a stretched section I_j of cluster C_k fills the gap with the next adjacent section I_{j+1} of the cluster C_{k+1} and clusters get merged. Figure 3.6 shows that the leftover evacuees of section I_1 fills the gap between I_1 and I_2 as $\mu_1 > c_{i-1}$, and as a result clusters C_1 and C_2 get merged. At this point, the algorithm updates the structure of the CH-tree T by connecting the cluster head node η_{k+1} as a child of node η_k by an edge labelled $i-1$. After a successful processing, the algorithm removes the processed critical capacity μ_j from the H tree and calculates a new critical capacity for the section I_j that can remove the step between I_j and I_{j+1} (they are in the same cluster now) using equation (3.5). In equation 3.5,

we need the height and duration of both sections. We get the height of the sections from the capacity tree ζ as described in Section 3.4. In this case, the section I_j is a simple cluster, so the Duration of I_j is given by

$$\delta_{t_j} = \frac{\sum_{v_l \in V_L} w_l}{h_j} \tag{3.7}$$

where V_L is the set of vertices got merged into simple cluster C_k (section I_j) before cluster C_{k+1} merged with C_k.

Similarly, If the section I_{j+1} is in a simple cluster, then, computing the duration is similar to equation (3.7) (ex. I_1 and I_2 in Figure 3.6). On the other hand, if the section I_{j+1} is in a mixed cluster (ex. I_2 and I_3 in Figure 3.6), we can compute the duration using equation (3.3) as shown in Section 3.4.

After computation, the algorithm inserts the computed critical capacity in H tree.

Case 2:

A stretched section I_j fills the step with the adjacent next section I_{j+1} and merges into one section. Figure 3.6 shows that the departing capacity c_{i-1} is lower than the critical capacity μ_3. So, the leftover evacuees of I_3 fills the step with I_4 and both sections get merged. The algorithm removes the processed critical capacity μ_j from the H tree and calculates the new critical capacity of section I_j (after merged with I_{j+1}) with respect to the next adjacent section. The following two situations can arise when a step between two sections get filled.

1. If the next adjacent section is in the next cluster, then the algorithm calculates the critical capacity that can fill the gap between two sections/clusters using equation (3.6)).

2. If the next section is in the same cluster, then the algorithm calculates the critical capacity that can fill the step between two sections using equation (3.5). In Figure 3.6, two sections I_3 and I_4 get merged because the departing capacity $c_{i-1} < \mu_3$.

After that, the algorithm proceeds to the next critical capacity in the H tree to process.

During processing the critical capacities in the H tree, to keep track which cluster/section gets merged with it's preceding cluster/section, the algorithm stores the information in CH-Tree T as follows.

- Let the sections I_j, I_{j+1}, I_{j+3}... belongs to the cluster C_k. If section I_{j+1} gets merged with I_j with respect to the sink location at v_i for $1 \leq i < j < n$, then the algorithm add/update the merge information at the cluster head node η_k (as showed in section 3.4).

- If a cluster C_{k+1} (simple or mixed) merges with it's preceding cluster C_k (simple) for the sink location at some vertex v_i, where $i < \eta_k < \eta_{k+1}$, the algorithm updates the structure of the CH-tree T by connecting the cluster head node η_{k+1} as a child of node η_k by an edge labeled i. The edge label indicates that the cluster C_{k+1} get merged with cluster C_k for the location of the sink at vertex v_i. If cluster C_{k+1} was a mixed cluster, then the cluster C_k becomes a mixed cluster after the merge. So, in that case, the algorithm also adds the merge information to the node η_k which contains information regarding the next section with respect to sink v_i.

So, for the sink location at v_{i-1}, the algorithm continues to process each of the critical capacities in H one by one until it finds a critical capacity less than the current outgoing capacity.

After that, the algorithm eventually sets v_{i-2} as the new sink location and repeat this process till vertex v_1.

Lemma 3.5. *Given a path graph $P = (V,E)$ of n vertices. If the edges have arbitrary capacities, we can construct CH-tree T in $O(n \log n)$ time.*

Proof: For each node u of T, the algorithm calculates the critical capacity and inserts into the max heap H. Since u is a new node, it takes $O(1)$ time to calculate the critical capacity while insertion into the heap takes $O(\log n)$ time. Next, it processes the critical capacities in the heap. It takes $O(1)$ time to get the max value from the heap while it individually takes $O(\log n)$ time for each deletion and insertion operation in the heap H during processing the max value. Also, to compute the critical capacity, it takes $O(\log n)$ time to get the height of the section from the capacity tree ζ. However, for all the node of T, the algorithm process at most n critical capacities which can be proved similarly to 3.1. Therefore, the average time to process a critical capacity value for a node u is amortized $O(\log n)$. Thus the total time for all node is $O(n \log n)$.

3.6 Use of the CH-tree in the arbitrary edge capacity case

Let us consider, we are given a CH-tree constructed on the path graph P as described in Section 3.5. Now, we show how we can answer an arbitrary query $Q(i, j)$ for $1 \leq i \leq j \leq n$. Similar to Section 3.3, the algorithm first identifies a cluster in the sub-path $P[i+1, j]$. Unlike the uniform case, CH-tree in the arbitrary case may have mixed clusters. Therefore, we also need to identify each section in a mixed cluster and compute the cost of each section to sink v_i. Finally, the result is the sum of the costs of all the sections in the sub-path $P[i+1, j]$, denoted by $cost(i, j)$.

First, the algorithm gets the index of the node $i+1$ in T in constant time. Then it starts traversing the tree top-down starting from node $i+1$. Note that the sub-tree $T(i+1)$ is the first cluster with node $i+1$ as the cluster head in the sub-path $P[i+1, j]$.

To identify the shape of a cluster during the traversal, in addition to reaching the rightmost leaf node of the sub-tree, the algorithm also considers to check the edge labels between two nodes. So, when traversing a sub-tree $T(k)$, if the algorithm reaches a node l for $k < l \leq j$;

connected to it's parent node x by an edge labeled s, where $k \leq x < l$ and $s < i$, then the algorithm identifies node $l-1$ as the end node of the sub-tree $T(k)$. After reaching a cluster head node, there are two possible situations can arise as follows.

Case 1:

In this case, the cluster is a simple cluster. The algorithm calculates the total evacuation cost of the cluster/section to sink v_i by the same procedure described in section 3.3 using the equation (3.3).

Case 2:

In this case, the cluster is a mixed cluster. The algorithm identifies the sections in the cluster one by one by following the index of α stored in the first vertex corresponding to each section. After reaching a section I_m, the algorithm checks for the node j in stored α.

- If $\alpha < j$, then node j is not in the current section. So, σ_m is given by

$$\sigma_m = \delta_{t_m} h_m \tag{3.8}$$

  where the algorithm gets the height h_m from the capacity tree ζ and for duration r_m is stored in the preceding section I_{m-1} except the first section in the cluster.

- If $\alpha > j$ then the node j belongs to the current section. So, the number of evacuees carried by the section is given by

$$\sigma_m = \sum_{\eta_m \leq y \leq j} w_y - \sum_{1 \leq p < m} \sigma_p \tag{3.9}$$

  where, I_1 is first section in the current cluster.

- If $\alpha == j$ then the node j is not in the current section I_m but the first vertex of the next section I_{m+1}, which indicates that a portion of supply of vertex v_j might evacuate

with the supply of I_m. Therefore, σ_m is given by the equation (3.8) and σ_{m+1} is given by the equation (3.9).

After reaching a section I_m the algorithm computes the total evacuation cost of the section with respect to sink v_i (denoted by $I_i(m)$) using the equation (**??**), where, $s_m = \tau d(u_m, i)$ and add the cost to the result as follows. Then, move to the next section for $\alpha \leq j$.

$$cost(i, j) = cost(i, j) + I_i(m) \tag{3.10}$$

The algorithm repeats the same procedure for each sub-tree in the sub-path $P[i+1, j]$ until it reaches the last sub-tree that contains the node j.

Lemma 3.6. *Given the CH-tree T for the path graph $P = (V, E)$ of n vertices. If the edges have arbitrary capacities, we can compute the total cost of an arbitrary sub-path $P[i, j]$ to sink v_i in $O((j-i)logn)$ time, where $i < j$.*

Proof: This can be proved similarly to lemma 3.3. The only difference is, to compute the total cost of a section, it takes $O(logn)$ time to get the height of the section from the capacity tree ζ. Thus the time to compute the total cost of an arbitrary sub-path is $O((j-1)logn)$ for $i < j$.

Chapter 4

Using the CH-tree to solve the minisum 1-sink problem on Cycle Networks

4.1 Using the CH-tree in the uniform case

Here, we show how we can implement the CH-tree on the cycle graph $G = (V, E)$ of n ordered vertices, to compute each $L(i, j)$ and $R(i, j)$ in constant time. First, we convert the cycle graph G into a path graph P_1 of n vertices by a random split edge (say e_n). Next, we connect two P_1 graphs serially by that same split edge (e_n) to get a path graph P of $2n(= m)$ vertices as follows,

Figure 4.1: Path graph P of $2n$ vertices while the split edge is e_n

where all the vertices of P have the same supply, and the edges have the same length of the cycle graph G. The path graph P has all the edges and vertices of G at-least once so that we can traverse any sub-path of G. Therefore, we construct the CH-tree on P.

In our approach, the algorithm constructs two CH trees on the path P; one for the clockwise operation, denoted by T_L and other for the counter-clockwise operation, denoted by T_R. The algorithm first constructs the tree T_L by placing the sink at every vertex $v_i \in V$ for $1 \leq i \leq m-1$ in descending order of i (clockwise operation). Then, it constructs the tree T_R in a similar fashion but in ascending order of i for $2 \leq i \leq m$ (counter-clockwise operation).

In Algorithm 4 (line 10), we use the tree T_L to calculate $L(i,j)$ and T_R to calculate $R(i,j)$. We use the same construction algorithm described in section 3.2 to construct the CH-tree T_L (construction of T_R is similar) on the path graph P.

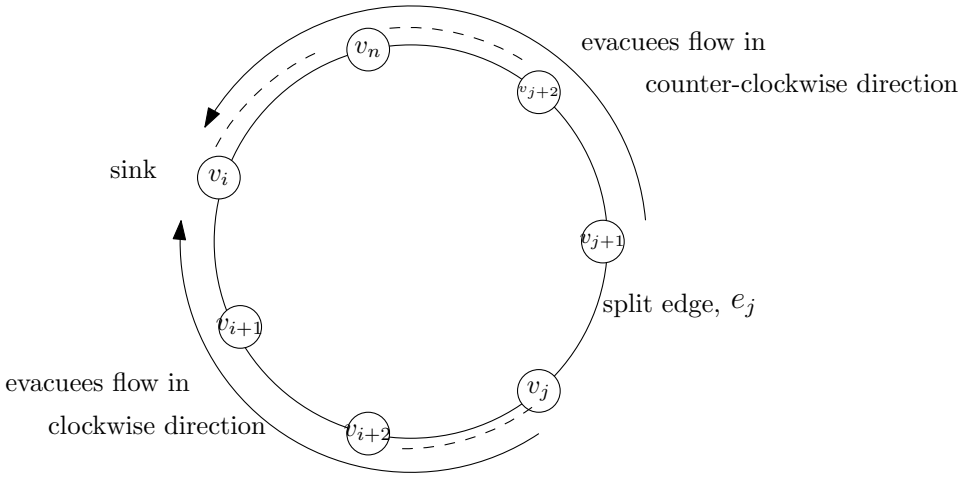


Figure 4.2: Illustration of evacuation of a cycle graph. The evacuees in the path $P^k[j+1,i]$ evacuate in counter-clockwise direction, while the evacuees in the path $P^c[j,i]$ evacuate in clockwise direction. The sink location and split edge are moving in counter-clockwise direction.

We recall our common algorithm (Algorithm 4 line 10), where it needs results for a batch of subsequent queries in both directions ($L(i,j)$ in clockwise and $R(i,j)$ in counter-clockwise) to find the optimal split edge for each sink location followed by identifying the optimal location of the 1-sink that minimizes the total evacuation cost. We use the tree T_L to answer the batch of queries like $L(i,j), L(i,j+1), L(i,j+2), L(i+1,j+1), L(i+1,j+2), \ldots\ldots\ldots$, where the split edge e_j and sink v_i moves (counter-clockwise) against the flow of the evacuees (clockwise)(see Figure 4.2). On the other hand, the tree T_R answers the batch of queries like $R(i,j), R(i,j+1), R(i,j+2), R(i+1,j+1), R(i+1,j+2), \ldots\ldots\ldots$, where the split edge e_j and sink v_i moves in the same direction with the flow of the evacuees (counter-clockwise). Here, we show how we can answer both type of batch of queries.

First, the algorithm gets the result (total evacuation cost) of the first query $L(i,j)$ in the same procedure described in section 3.3 and then, temporarily sets $cost' = L(i,j)$, when

j is incremented by 1 in the next query. For a simple illustration, let consider the algorithm identifies the set of clusters $C_L = \{C_1, C_2, C_3, ..., C_p\}$ on the sub-path $P[i+1, j]$ to get the result of the query $L(i, j)$, where C_1 is the nearest cluster to sink v_i (a similar procedure for the first query $R(i, j)$). Now, using the tree T_L we can answer the next queries as follows.

Computing $L(i, j+1)$:

If the next query is $L(i, j+1)$, where the sink remains at the same vertex but the split edge moves; the algorithm handles two situations based on the position of the node $j+1$ in T_L.

1. If the node $j+1$ is a child node of some node k for $i+1 \leq k \leq j$ then the node $j+1$ belongs to the same cluster with node j. So, the node $j+1$ includes in the last cluster C_p and the total cost is given by

$$L(i, j+1) = cost' - s_p \sum_{\eta_p \leq y \leq j} w_y - \frac{(\sum_{\eta_p \leq y \leq j} w_y)^2}{2c} + s_p \sum_{\eta_p \leq y \leq j+1} w_y + \frac{(\sum_{\eta_p \leq y \leq j+1} w_y)^2}{2c} \tag{4.1}$$

   where, η_p is the cluster head node of C_p.

2. Otherwise, if the parent node of $j+1$ is some node x for $1 \leq x \leq i$ or $x = \rho$ then the node $j+1$ is the cluster head of the new cluster C_{p+1}. So, the cost is given by

$$L(i, j+1) = cost' + s_{p+1}\sigma_{p+1} + \frac{\sigma_{p+1}^2}{2c} \tag{4.2}$$

   where, $\sigma_{p+1} (= w_{j+1})$ is the number of evacuees carried by section C_{p+1} and $s_{p+1} (= \tau d(j, i))$ is the time when the first unit reaches sink v_i.

The algorithm uses the same process to answer any next query in the batch for an incremental J.

Computing $L(i+1, j)$

Now, we show the calculation of $L(i+1, j)$ from the result of $L(i, j)$, when the sink moves.

According to the cluster computation described in section 1.1, the first cluster C_1 that evacuates to v_i has v_{i+1} as the cluster head. So, when the sink moves to $i+1$, the cluster C_1 (subtree $T(i+1)$) may breakdown into multiple clusters (sub-trees) depending on the number of child of node $i+1$. So, the algorithm traverses through the child nodes of $i+1$. If k is a child node of $i+1$ then the subtree $T_L(k)$ becomes a new cluster respect to the sink at $i+1$. So, the total cost is given by

$$L(i+1,j) = cost' - \tau d^c(i+1,i)\Big(\sum_{\eta_2 \le y \le j} w_y\Big) - C_i(1) + \sum_{I \in I_L} C_{i+1}(I) \tag{4.3}$$

where, η_2 is the cluster head node of C_2, $C_i(1)$ is the total cost of cluster C_1 and I_L is the set of new clusters created after breakdown of cluster C_1. The algorithm uses the same procedure for any next query in the batch where the sink moves to the next location. Therefore, the idea is, when the algorithm answers the query $L(i+1,j)$ after $L(i,j)$, it only needs to traverse the descendants of node $i+1$.

Now, using the tree T_R we can answer the next queries as follows.

Computing $R(i,j+1)$:

If the next query is $R(i,j+1)$, the algorithm handles two situations based on node j.

1. If the node j is the cluster head node, it deducts the cost of j from $cost'$. So, $R(i,j+1)$ is given by

$$R(i,j+1) = cost' - s_p\sigma_p - \frac{\sigma_p^2}{2c} \tag{4.4}$$

   where, $v_j(=\eta_p)$ in last cluster C_p and $\sigma_p = w_j$.

2. If the node $j+1$ is the parent of j, it re-compute the cost of the last cluster C_p.

$$R(i,j+1) = cost' - s_p \sum_{\eta_p \le y \le j} w_y - \frac{\sum_{\eta_p \le y \le j} w_y}{2c} + s_p \sum_{\eta_p \le y \le j+1} w_y + \frac{\sum_{\eta_p \le y \le j+1} w_y}{2c} \tag{4.5}$$

   The algorithm uses the same process to answer any next query using the T_R tree for

an incremental j.

Computing $R(i+1,j)$:

In this case, unlike $L(i+1,j)$ in the tree T_L, multiple clusters can merge into one cluster as a new vertex v_i encountered at the front of the evacuation path to the sink v_{i+1}. So, the algorithm traverses through the immediate child nodes of i. If k is a child node of i then the substree $T(k)$ merged with i for sink at $i+1$. Therefore, the algorithm deducts the cost of each subtree spanned by each child node of i and add the cost of $T(i)$ to $cost'$ to get $R(i+1,j)$.

$$R(i+1,j) = cost' - \sum_{I \in I_L} C_i(I) + s_1\sigma_1 + \frac{\sigma_1^2}{2c} + \tau d^k(i,i+1) \sum_{l \le y \le j} w_y \tag{4.6}$$

where, C_1 is the first cluster with v_i as the cluster head. So, $\sigma_1 = \sum_{i \le y \le l-1} w_y$, where $l-1$ is the last node of sub-tree $T(i)$. For, any next query in T_R where sink location moves, the algorithm uses the same procedure to answer the query.

Lemma 4.1. *Given the CH tree T_L for the path graph $P = (V,E)$ of n vertices. If the edge capacities are uniform, we can answer each query in a batch of subsequent query like $L(i,j), L(i,j+1), L(i,j+2), L(i+1,j+1)$.... in $O(1)$ time.*

Proof: From lemma 3.3, it takes $O(j-i)$ time to answer a single query $L(i,j)$ for $i < j$. For each next query when j is incremented by 1, it takes $O(1)$ time to answer the query. But when i move to $i+1$ in the next query, the subtree $T(i+1)$ can breakdown into multiple subtrees (as a cluster breakdown into multiple clusters) and the algorithm traverse the immediate child nodes of $T(i+1)$. However, traversing the immediate child nodes for all the values of i takes at most $O(n)$ time. Therefore, the average time to answer each query in the batch of subsequent query takes amortize $O(1)$ time by using the CH tree T_L.

4.2 Using the CH-tree in the arbitrary case

Here we show how we can implement the CH-tree on the simple cycle graph $G = (V, E)$ of n ordered vertices, to compute each $L(i, j)$ and $R(i, j)$ efficiently. We use the same procedure described in section 4.1 to get our desired path graph P (see Figure 4.1) from the cycle graph G. Here we also construct two CH-trees on the path P; one in the descending order of the vertices v_i for $1 \leq i \leq n-1$ (clockwise operation), denoted by T_L and other in the ascending order of v_i for $2 \leq i \leq n$ (counter-clockwise operation). We use the construction algorithm described in section 3.5 to construct the CH-tee T_L (construction of T_R is similar) on the path graph P.

Now, we show how we can answer the same type of batch of subsequent queries mentioned in section 4.1 using the trees T_L and T_R constructed for the arbitrary case.

First, the algorithm answer the first query (total evacuation cost) $L(i, j)$ using the same procedure described in section 3.6 and then, temporarily sets $cost' = L(i, j)$ when j is incremented by 1 in the next query. For a simple illustration, let consider the algorithm identifies the set of sections $I_L = \{I_1, I_2, I_3, ..., I_p\}$ on sub-path $P[i+1, j]$ while answering the query $L(i, j)$, where I_1 is the nearest section to the sink v_i (a similar procedure for the first query $R(i, j)$ using tree T_R). Now using the tree T_L we can answer the next queries as follows.

Computing $L(i, j+1)$:

If the next query is $L(i, j+1)$, where the sink remains at the same vertex but the split edge moves. The algorithm handles three situations based on the position of the node $j+1$.

1. If the node v_{j+1} is in the same section I_p with v_j then the total cost is given by

$$L(i, j+1) = cost' - I_i(p) + s_p \sum_{\eta_p \leq y \leq j+1} w_y + \frac{(\sum_{\eta_1 \leq y \leq j+1} w_y)^2}{2h_p} \tag{4.7}$$

So, v_{j+1} includes in the last section I_p and the algorithm updates the cost of the last section I_p to get the result of $L(i, j+1)$.

2. If the node v_{j+1} is in the same cluster C_m with v_j but in the next section I_{p+1} then the algorithm first updates the $cost'$ by updating the total cost of section I_p.

$$cost' = cost' - I_i(p) + s_p\sigma_p + \frac{\sigma_p^2}{2h_p} \tag{4.8}$$

where $\sigma_p = h_p\delta_{t_p}$. Then it adds the cost of the next section I_{p+1} to get the result of $L(i, j+1)$ as follows

$$L(i, j+1) = cost' + s_{p+1}\sigma_{p+1} + \frac{\sigma_{p+1}^2}{2h_{p+1}} \tag{4.9}$$

where, $\sigma_{p+1} = \sum_{\eta_m \le y \le j+1} w_y - \sum_{I \in I_L} \sigma_I$, where I_L is the set of section in cluster C_m before the section σ_{p+1}.

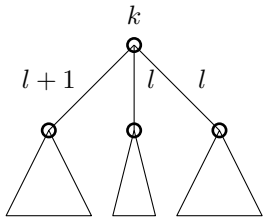
3. If the node $j+1$ is in the next section I_{p+1} that belongs to the next cluster C_{m+1} then v_{j+1} is the cluster head of the cluster. The algorithm sets $\sigma_{p+1} = w_{j+1}$ and the total cost of $L(i, j+1)$ is given by

$$L(i, j+1) = cost' + s_{p+1}\sigma_{p+1} + \frac{\sigma_{p+1}^2}{2h_{p+1}} \tag{4.10}$$

Computing $L(i+1, j)$:

Now, we show how we can answer the next query $L(i+1, j)$ after computing $L(i, j)$. According to the construction of the CH tree T_L, moving the split edge e_j does not impact the construction of the clusters but moving the sink location does. In Algorithm 4, whenever we move the sink location, a cluster can break down into multiple clusters or the number of sections in a cluster can increase.

When the algorithm calculates $L(i+1, j)$ after $L(i, j)$, the sink moves in the opposite direction of the flow. Thus, the sub-trees (or clusters) with edges or stored information with respect to sink v_i will must break down into multiple sub-trees (or clusters).



$T(k)$

Figure 4.3: A sub-tree (or cluster) $T(k)$ rooted at node k (or k is the cluster head) with three child nodes

Therefore, during the computation of $L(i, j)$, the algorithm maintains a min-priority queue γ that contains the cluster head nodes in the sub-path $P[i, j]$. In the min-priority queue, the cluster heads are sorted by the index of their minimum edge label or stored information. It takes $O(\log n)$ time to insert/delete an element in/from the priority queue. At the beginning of the computation of $L(i, j)$, the priority queue $\gamma = \emptyset$. Let us consider the Figure 4.3, the sub-tree $T(k)$ with three child nodes of root node k. Two of the edges between the child nodes and node k are labelled with index l, while another is labelled with index $l+1$. If the sub-tree $T(k)$ is in the path $P[i, j]$, then $l \geq i$. The algorithm stores the cluster head node k in γ according to the smallest index of the edge label, which is l. As the algorithm runs a batch of subsequent queries, every time the algorithm runs the query $L(i, j+1)$ after $L(i, j)$ it updates γ by inserting the index of the cluster head of any new cluster encounters in the path. Thus, the overall idea is to update the list in γ for the subsequent queries when j is incremented by 1 and process the cluster head nodes in tree γ when i is incremented by 1.

Therefore, when the algorithm needs to process a query where the sink location moves

from i to $i+1$, it starts processing the cluster heads from the top of the queue γ whose corresponding index of the sink location is i. More precisely, whenever we move the sink from i to $i+1$, the algorithm only processes the clusters which have the index i as the sink location. When a cluster head node has been processed, the algorithm removes it from the γ and inserts the cluster heads of the newly generated/modified clusters. Processing a cluster is defined as deducting the total cost of that cluster from $L(i,j)$, and then adding the cost of the newly generated cluster/s after the breakdown of the processed cluster. At the same time, the algorithm also computes the total supply $W = \sum_{C \in C_L} \sigma_C$, where C_L is the set of clusters which are processed in γ. Thus, $L(i+1,j)$ is given by

$$L(i+1,j) = cost' - \sum_{k \in \gamma} \Big(\sum_{I \in I_k} L_i(I) + \sum_{I \in I_l} L_{i+1}(I) \Big) - \tau d^c(i+1,i) \sum_{i+2 \le x \le j} w_x - W \quad (4.11)$$

where, k is a cluster in γ, and I_k is the set of sections in k and I_l is the set of sections newly revealed after the breakdown of the cluster k.

Now we show how we can answer the next queries after answering $R(i,j)$ using T_R tree.

Computing $R(i,j+1)$:

If the next query is $R(i,j+1)$, the algorithm handles three situations based on the location of j.

1. If j is the cluster head then $R(i,j+1)$ can be computed as follows.

$$R(i,j+1) = cost' - s_p \sigma_p - \frac{\sigma_p^2}{2h_p} \quad (4.12)$$

   where, j is in the last section I_p and $\sigma_p = w_j$.

2. If j is the first vertex of the section I_p except the first section of the cluster C_m then the algorithm first updates the $cost'$ by deducting the cost of the section I_p and I_{p-1}

as follows.

$$R(i, j+1) = cost' - I_i(p) - I_i(p-1) \tag{4.13}$$

Then, calculates $\sigma_{p-1} = \sum_{\eta_m \le y \le j+1} w_y - \sum_{I \in I_L} \sigma_I$, where I_L is the set of sections before the section I_{p-1}. So, $R(i, j+1)$ is given by

$$R(i, j+1) = cost' + s_{p-1}\sigma_{p-1} + \frac{\sigma_{p-1}^2}{2h_{p-1}} \tag{4.14}$$

3. If j and $j+1$ belongs to the same section I_p then we can use similar procedure of case (2) described above and obtain $R(i, j+1)$ as follows.

$$R(i, j+1) = cost' - I_i(p) + s_p\sigma_p + \frac{\sigma_p^2}{2h_p} \tag{4.15}$$

The algorithm uses the same procedure for any next query where j is incremented by 1.

Computing $R(i+1, j)$:

We use a procedure similarly to compute $L(i+1, j)$. In this case, the sink moves in the direction of the flow, so, the vertex i encountered in front of the evacuation path to $i+1$. Therefore, multiple clusters/sections can merge into one cluster/sections in the path $P[i, j]$. So, like $L(i, j)$, during the computation of $R(i, j)$ the algorithms maintains a min-priority queue γ that contains the index of the cluster heads in the sub-path $P[i, j]$. In the min-priority queue, the cluster heads are sorted by the index of their minimum edge label or stored information. Now, similar to $L(i+1, j)$, $R(i+1, j)$ is given by

$$R(i+1, j) = cost' - \sum_{k \in \gamma} \Big(\sum_{I \in I_k} R_i(I) + \sum_{I \in I_l} R_{i+1}(I) \Big) + \tau d^k(i, i+1) \sum_{i-1 \le x \le j} w_x W \tag{4.16}$$

where, k is a cluster in γ, and I_k is the set of sections in k and I_l is the set of sections newly computed after the merge operation.

Lemma 4.2. *Given the CH-tree T_L for the path graph $P = (V,E)$ of n vertices with arbitrary edge capacities. We can answer each query in the batch of subsequent query like $L(i,j), L(i,j+1), L(i,j+2), L(i+1,j+1))$... in $O(\log n)$ time.*

Proof: From lemma 3.6, it takes $O((j-i)\log n)$ time to answer the single query $L(i,j)$ for $i < j$. So, for each next query when j is incremented by 1, it takes $O(\log n)$ time to answer the query. But when the sink move from i to $i+1$, any sub-tree $T(u)$ can breakdown into multiple sub-trees. To re-calculate the cost of those sub-trees, the algorithm traverses each subtree $T(u)$ where u is stored in a min-priority queue tree γ with respect to the index of the sink location. However, re-traversing the nodes stored in γ tree for n sink locations takes $O(n)$ time (can be proved similarly to lemma 3.1. Thus average time to answer each query in the batch of subsequent query takes amortize $O(\log n)$ time.

4.3 Main Theorem

If two CH trees T_L and T_R are given for a dynamic network $N = (G, w, l, c, \tau)$, where $G = (V,E)$ be an undirected cycle graph then we solve the minsum 1-sink location problem by implementing the CH tree in our main algorithm described in section 2.3, where CH-tree is the data structure we use to achieve objective 2.

Theorem 4.3. *(a) The minsum 1-sink location problem on dynamic cycle networks with arbitrary edge capacities can be solved in $O(n \log n)$ time.*
(b) The same problem can be solved in $O(n)$ time if the edge capacities are uniform.

As a proof of Theorem 4.3, we already analyzed the time complexities in Lemmas 2.5, 4.1, 4.2.

Chapter 5

Algorithm for k-sink evacuation problem on cycle networks

In this chapter, we describe an $O(n)$ algorithm that identifies the optimal evacuation protocol for k-sink evacuation problem on simple cycle networks, mentioned in section 1.1. Note that the same approach also solves the problem on dynamic path networks. First, we define the problem rigorously with some extra terms added to the common terms we use throughout the description of the sink location problem. Then we describe the techniques we use to solve the problem efficiently followed by the overall algorithm.

5.1 Problem definition

Here, we consider the same dynamic network $N = (G, w, l, c, \tau, S)$ that we defined in section 2.1. In addition, a set of k-sink locations $S = \{x_1, x_2, ..., x_k\}$ is given, where $S \in V$ and $2 \leq k \leq n$. In graph G, vertices or sinks can be ordered either clockwise or counter-clockwise. Note that, in our description, we consider the vertices and the sinks are ordered counter-clockwise as shown in figure 5.1, unless otherwise stated.

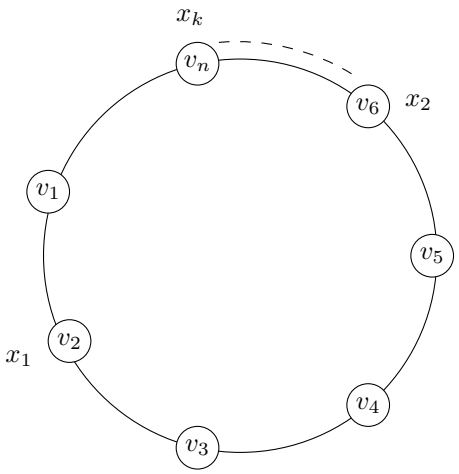


Figure 5.1: Cycle graph G of counter-clockwise ordered n vertices and k sinks,

Unlike the sink location problem, the evacuee flow is distributary in this problem, which means the evacuees at a vertex can split to different sinks to minimize the average evacuation time for each evacuee. Moreover, we are also considering the uniform capacity of the edges to solve this problem.

A simple cycle graph G with given k sink locations can be divided into k sub-problems each consists of a path graph P. Note that in this chapter, unless stated otherwise, the path graph P is defined as follows.

Definition 5.1. *A path graph $P = (V,E)$ of ordered set of vertices $V = \{v_1, v_2, ..., v_m\}$ and ordered set of edges $E = \{e_1, e_2, ...e_{m-1}\}$, where $m \geq 2$ and $k = 2$ sinks are located at the vertices v_1 and v_m.*

So, for the path graph P, we need to determine the amount of supply (≥ 0) evacuate to v_1 so that the rest evacuate to v_m (or vice verse). Let consider a point p' on P that divides the total supply of P to evacuate between two sinks, we call this point the split point. In minsum objective, the cost is define on the each evacuee located at the vertices. So, the total cost of P is define as the total cost to evacuate all the supply located at the vertices of P to the designated sinks.

Let, $sum_{p'}(P)$ denote the cost of P respect to split point p', where $L_{p'}(P)$ (resp. $R_{p'}(P)$) represents the total cost of the supplies that evacuate to the left sink v_1 (resp. right sink v_m). Then,

$$sum_{p'}(P) = L_{p'}(P) + R_{p'}(P) \tag{5.1}$$

Then the optimal cost of P is the minimum among the total costs with respect to each possible split points, denoted by $sum(P)$.

Definition 5.2. ***Optimal split point***

The split point on a path graph P is called the optimal split point for which the total cost of P is minimized.

A split point can either be on a vertex or edge, called by split vertex or split edge respectively. So, $sum(P)$ is given by

$$sum(P) = min_{p' \in (V \cup E)} \Big(sum_{p'}(P) \Big) \tag{5.2}$$

So, the optimal evacuation cost for the given cycle graph G is the summation of the results of the sub-problems.

$$sum(G) = \sum_{1 \leq i \leq k} sum(P_i) \tag{5.3}$$

5.2 Properties of the evacuation problem

Here, we show the properties that identify the optimal split point on the path graph P. The following two cases can arise based on the location of the split point.

Case 1:

In this case, the split point is on an edge $e_i \in E$, and therefore, computing $sum_{e_i}(P)$ is straightforward. The vertices at the left of e_i evacuate to v_1 and vertices at the right evacuate to v_m. Then,

$$sum_{e_i}(P) = L_{e_i}(P) + R_{e_i}(P) = \sum_{C \in C_L} L_1(C) + \sum_{C \in C_R} R_m(C) \tag{5.4}$$

where, C_L (resp. C_R) is the set of clusters evacuate to the left sink v_1 (resp. right sink v_m).

Case 2:

In this case, the split point is on a vertex $v_i \in V - \{v_1, v_m\}$. So, to minimize the total evacuation cost, the supply w_i at v_i may split into two sinks. Let, α_i (≥ 0) is the amount of supply from v_i goes to the left sink v_1 and the rest ($w_i - \alpha_i$) goes to v_m, where α_i is the optimal value for which $sum_{v_i}(P)$ is minimized.

Now we show how the algorithm calculates the value of α_i for a split vertex v_i.

Proposition 5.3. *Given the path graph P, if the vertex $v_i \in V - \{v_1, v_m\}$ is the split vertex and α_i is the split amount of supply of v_i evacuate to one of the sinks then the different values of α_i for $0 \leq \alpha_i \leq w_i$ does not impact the total evacuation cost of the clusters on P except the two clusters adjacent to the left and right of v_i.*

Let, C_l and C_r are the clusters adjacent to v_i at the left and right side respectively and the vertices $v_j = (\eta_l)$ and $v_k (= \eta_r)$ are the cluster heads, where $1 < j \leq i \leq k < m$. Then, C_l (resp. C_r) is the last cluster evacuate to v_1 (resp. v_m). Let, $X(\alpha_i)$ denote the total evacuation cost of the clusters C_l and C_r, where $X_L(\alpha_i)$ (resp. $X_R(\alpha_i)$) represents the cost to evacuate the cluster C_l (resp. cluster C_r). Then,

$$X(\alpha_i) = X_L(\alpha_i) + X_R(\alpha_i) \tag{5.5}$$

where,

$$X_L(\alpha_i) = \frac{\big(\alpha_i + (w(i,j) - w_i)\big)^2}{2c} + \big(\alpha_i + (w(i,j) - w_i)\big)\tau d(j,1)$$

so,

$$X_L(\alpha_i) = \frac{\alpha_i^2}{2c} + \alpha_i\Big(\frac{(w(i,j) - w_i)}{c} + d(j,1)\Big) + \frac{(w(i,j) - w_i)^2}{2c} + w(i,j)\tau d(j,1)$$

where, $w(i,j)$ is the sum of the evacuees in the path $P[i,j]$. In a similar manner,

$$X_R(\alpha_i) = \frac{\alpha_i^2}{2c} - \alpha_i\Big(\frac{(w(i,k)-w_i)}{c} + \tau d(k,m) + \frac{w_i}{c}\Big) + \frac{w_i^2}{2c} + w_i\Big(\frac{(w(i,k)-w_i)}{c} + \tau d(k,m)\Big) + \frac{(w(i,k)-w_i)^2}{2c} + w(i,k)\tau d(k,m)$$

As we are only interested in α_i, we can write equation (5.5) with the values of $X_L(\alpha_i)$ and $X_R(\alpha_i)$ as follows.

$$X(\alpha_i) = \frac{\alpha_i^2}{c} + \alpha_i\Big(\frac{(w(i,j)-w_i)}{c} + \tau d(j,1) - \frac{(w(i,k)-w_i)}{c} - \tau d(k,m) - \frac{w_i}{c}\Big) + constant \tag{5.6}$$

Now, by differentiating equation (5.6) by α_i, we get,

$$\alpha_i = \frac{1}{2}\Big(w(i,k) - w(i,j) + w_i\Big) + \frac{c\tau}{2}\Big(d(k,m) - d(j,1)\Big) \tag{5.7}$$

So, We use equation (5.7) to determine the optimal α_i value for the split vertex $v_i \in V - \{v_1, v_m\}$.

Finally, to identify whether a split point is on an edge or on a vertex, we go through following observations based on the value of α.

Claim 5.4. *Given the path graph P, if there are two adjacent vertices v_i and v_{i+1} such that $\alpha_i \geq w_i$ and $\alpha_{i+1} \leq 0$ then the edge e_i is the split point, where $2 \leq i \leq m-2$.*

Claim 5.5. *Given the path graph P, if there are two vertices v_{i-1} and v_{i+1} such that $\alpha_{i-1} \geq w_{i-1}$ and $\alpha_{i+1} \leq 0$ then the vertex v_i is the split vertex, where $3 \leq i \leq m-2$.*

Moreover, regarding the number of split vertex in a given path P, we prove the following lemma.

Lemma 5.6. *For the given path P, there can be at most one split vertex.*

Proof: Given the path graph P of $m \geq 4$ vertices. Let consider two split vertices v_i and v_j on the path, where $2 \leq i < j \leq m-1$. If $(w_i - \alpha_i) \leq \alpha_j$ then the $(w_i - \alpha_i)$ evacuees cross each other path for $d(i,j)$ distance. Therefore, we get an extra transport cost of $2\big((w_i - \alpha_i)\tau d(i,j)$ due to the path overlap. So, at most one split vertex can gives us the minimum total cost and the lemma is proved.

5.3 Efficient Algorithm for the k-sink evacuation problem

Here, we show the $O(n)$ algorithm to solve the problem on dynamic cycle networks. First, the algorithm decomposes the problem into k sub-problems, where each sub-problem consists of the path graph P. A sub-problem is to find the optimal split point (optimal evacuation protocol) on P that minimizes the total evacuation cost of the path graph P.

Let consider a set of path graph $\rho = \{P_1, P_2, ..., P_q\}$ such that the path $P_i \in \rho$ has exactly two sinks located at two ends, where $q = k$. Then, for each path graph P_i, the algorithm identifies the optimal split point and compute $sum(P_i)$. Finally, the summation of the q values is the optimal evacuation cost on the cycle graph G.

Here, we show the computation of $sum(P_1)$ (computation for all other $sum(P_i)$ is similar for $2 \leq i \leq m$). Basically, the algorithm computes $sum_{v_i}(P_1)$ for all the vertices where $2 \leq i \leq m-1$ in the ascending order of i (computing in descending order of i gives the same result). Note that the algorithm automatically gets the $sum_{e_i}(P_1)$ for all $e_i \in E$ in the computation process of $sum_{v_i}(P_1)$. The minimum $sum_{p'}(P_1)$ is the result for P_1, where $p' \in (V \cup E)$.

First, the algorithm calculates the α_2 for the vertex v_2 in $O(1)$ time as follows.

$$\alpha_2 = \frac{1}{2}\Big(w(2,k) - w(2,2) + w_2\Big) + \frac{c\tau}{2}\Big(d(j,m) - d(2,1)\Big) \tag{5.8}$$

Let, C_l and C_r are the last clusters on left and right of v_2 evacuate to the sinks v_1 and v_m respectively. So, in equation 5.8, v_2 and v_k are the cluster heads of C_l and C_r respectively. Note that the algorithm computes the clusters on the left side of split vertex in runtime. But to get the last cluster on right side efficiently, we do an $O(n)$ pre-processing from left to right to store the vertices responsible as a cluster head. At the end of the pre-processing, we get a total cost $sum_{e_1}(P_1)$ and a list of vertices so that we can find the first cluster head for any given path $P[x,y]$ where $2 \leq x \leq y \leq m-1$. Then, the algorithm calculates the total cost of P_1 with respect to split vertex v_2 as follows,

$$sum_2(P_1) = L_2(P_1) + R_2(P_1)$$

$$= \sum_{C \in C_L} L_1(C) + \sum_{C \in C_R} (R_m(C) \tag{5.9}$$

from equation (2.9), where C_L and C_R are the set of clusters evacuate to v_1 and v_m respectively. Calculating $L_2(P_1)$ is straightforward as there is no unknown intermediate vertices between v_2 and v_1. So,

$$L_2(P_1) = \sigma_l s_l + \frac{\sigma_l^2}{2c} \tag{5.10}$$

where, the number of evacuees carried by the section C_l, $\sigma_l = \alpha_2$ and $s_l = \tau d(2,1)$ is the time when the first unit of cluster $C_l \in C_L$ reaches sink v_1 (from equation (**??**).

On the other hand,

$$R_2(P_1) = R_{e_1}(P_1) - \alpha_2 s_r - \frac{\sigma_r}{2c}\alpha_2 \tag{5.11}$$

where, $s_r = \tau d(k,n)$ and $(\sigma_r/2c)$ is the average waiting time per evacuee for cluster C_r. So, equation (5.10) and (5.11) give us the total cost of P_1 with respect to split vertex v_2.

Now, suppose that for some integer i satisfying $2 \leq i < m-1$, we have already computed the α_i and then $sum_{v_i}(P_1)$, where C_l and C_r are the last clusters to evacuate v_1 and v_m respectively. Here we show how the algorithm computes the total cost respect to next splitting point (e_i or v_{i+1}) from the result of v_i. First, we temporary set $cost'_R = R_{v_i}(P_1)$, $cost'_L = L_{v_i}(P_1)$ and $W'_R = (w_i - \alpha_i)s_r + \frac{\sigma_r}{2c}(w_i - \alpha_i)$, $W'_L = \alpha_i s_l + \frac{\sigma_l}{2c}\alpha_i$. Then, the algorithm calculates the α_{i+1} by a equation similar to equation 5.7. Next it compares, if $\alpha_i \geq w_i$ and $\alpha_{i+1} \leq 0$ then it satisfies claim 5.4 and e_i is the split point. So, $sum_{e_i}(P_1)$ is given by

$$sum_{e_i}(P_1) = sum_{v_i}(P_1) \tag{5.12}$$

So, $sum_{e_i}(P_1)$ is the optimal cost of P_1, therefore the final result. On the other hand, If it doesn't satisfy the condition of claim 5.4 then the algorithm compares, if $\alpha_{i-1} \geq w_{i-1}$ and $\alpha_{i+1} \leq 0$ then it satisfies the condition of claim 5.2 and v_i is the optimal split point. So, $sum_{v_i}(P_1)$ is the optimal cost of P_1.

If the condition of the both claims 5.4 and 5.2 are not true then the algorithm compute $R_{v_{i+1}}(P_1)$ as follows (computation of $L_{v_{i+1}}(P_1)$ is similar).

$$R_{v_{i+1}}(P_1) = R_{v_i}(P_1) - W'_R - \alpha_{i+1}s_r - \frac{\sigma_r}{2c}\alpha_{i+1} \tag{5.13}$$

where, C_r is the last cluster on the path $P[i+2, n-1]$. So,

$$sum_{v_{i+1}}(P_1) = L_{i+1}(P_1) + R_{i+1}(P_1) \tag{5.14}$$

After that the algorithm does the same process for the next split point v_{i+2} and continue until it finds the optimal split point for P_1. The algorithm uses the same procedure to solve all the q sub-problems and thus solve the $k-sink$ on simple cycle graph.

Theorem 5.7. *The minsum $k-$sink evacuation problem on a dynamic simple cycle network*

with uniform edge capacity can be solved in $O(n)$ time.

The algorithm divides the simple cycle graph into k path graphs. Then, to find the optimal split point for each path graph, it visits each node in the path graph at most once. Thus the k optimal points for the cycle graph can be found in $O(n)$ time.

Chapter 6

Conclusion

Facility location problem is a problem in dynamic networks motivated by evacuation planning during different natural disasters all over the world. Researchers mainly work on these problems with two objective functions, one is known as the "minmax criterion" where we need to optimize the evacuation completion time, and the other is the "minsum criterion" where we need to optimize total evacuation time of all the evacuees. The minsum is harder than the minmax criteria as the objective cost function deals with the evacuation cost of all the supply vertices, not just the vertex causing maximum evacuation cost. Therefore, there is no known result for any special graph except the path graph with the minsum objective function. We motivated to work with the minsum objective function from the desire of minimizing the total evacuation cost of all the evacuees to reduce the average psychological stress for each evacuee.

In this thesis, we work on two branches of facility location problem based on the given condition of the sink/s. If the location of the sink/s is given then we need to find the optimal evacuation protocol so that the minsum objective function is obtained, we call it the "evacuation problem." On the other hand, if only the total number of the sink is given but not their location then we need to find the optimal location of the sink/s so that the minsum objective function is obtained, we call it the "sink location problem."

In sink location problem, we studied two special cases of the 1-sink location problem on

cycle networks based on the capacity of the edges. We proposed an $O(n)$ time algorithm based on a data structure that solves the problem with the uniform capacity of the edges. We extend the data structure to overcome the complexities of the cycle graph with arbitrary edge capacities and solve the problem in $O(n \log n)$ time, using the same algorithm.

The Evacuation problem has been extensively studying in different variants by several researchers. We are the first to study the evacuation problem in the minsum objective function. As a first step, we proposed an $O(n)$ time algorithm that finds the optimal evacuation protocol on cycle networks with k sink locations, when edge capacities are uniform.

6.1 Directions for future research

The algorithm and data structure presented in this thesis open several possibilities for future research. First, the 1-sink location problem can be extended to 2-sink or k-sink location problem. It's always challenging to solve the minsum sink location problem in a more general graph efficiently. Our approach can be extended to solve the 1-sink location problem on unicycle network.

The result we presented for the k-sink evacuation problem can be extended to solve the problem on dynamic unicycle network. Our proposed lemmas and theorems can be a good direction to solve the problems in more general graphs.

Bibliography

[1] Guru Prakash Arumugam, John Augustine, Mordecai J Golin, Yuya Higashikawa, Naoki Katoh, and Prashanth Srikanthan. Optimal evacuation flows on dynamic paths with general edge capacities. *arXiv preprint arXiv:1606.07208*, 2016.

[2] Robert Benkoczi, Binay Bhattacharya, Yuya Higashikawa, Tsunehiko Kameda, and Naoki Katoh. Minsum k-sink problem on dynamic flow path networks. In *International Workshop on Combinatorial Algorithms*, pages 78–89. Springer, 2018.

[3] Binay Bhattacharya, Mordecai J Golin, Yuya Higashikawa, Tsunehiko Kameda, and Naoki Katoh. Improved algorithms for computing k-sink on dynamic path networks. *arXiv preprint arXiv:1609.01373*, 2016.

[4] Rainer E Burkard, Karin Dlaska, and Bettina Klinz. The quickest flow problem. *Zeitschrift für Operations Research*, 37(1):31–58, 1993.

[5] Rajib Das. Minmax sink location problem on dynamic cycle networks. 2019.

[6] Lester R Ford Jr and Delbert Ray Fulkerson. Constructing maximal dynamic flows from static flows. *Operations research*, 6(3):419–433, 1958.

[7] Refael Hassin and Arie Tamir. Improved complexity bounds for location problems on the real line. *Operations Research Letters*, 10(7):395–402, 1991.

[8] Yuya Higashikawa, Mordecai J Golin, and Naoki Katoh. Multiple sink location problems in dynamic path networks. In *International Conference on Algorithmic Applications in Management*, pages 149–161. Springer, 2014.

[9] Yuya Higashikawa, Mordecai J Golin, and Naoki Katoh. Multiple sink location problems in dynamic path networks. *Theoretical Computer Science*, 607:2–15, 2015.

[10] Bruce Hoppe and Éva Tardos. The quickest transshipment problem. *Mathematics of Operations Research*, 25(1):36–62, 2000.

[11] Oded Kariv and S Louis Hakimi. An algorithmic approach to network location problems. i: The p-centers. *SIAM Journal on Applied Mathematics*, 37(3):513–538, 1979.

[12] Satoko Mamada, Takeaki Uno, Kazuhisa Makino, and Satoru Fujishige. An $o(nlog^2n)$ algorithm for a sink location problem in dynamic tree networks. *Exploring New Frontiers of Theoretical Informatics*, pages 251–264, 2004.