M.S. THESIS

Performance evaluation of space efficient graph algorithms

공간 효율적인 그래프 알고리즘의 성능 분석

FEBRUARY 2019

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Yeonil Yoo

M.S. THESIS

Performance evaluation of space efficient graph algorithms

공간 효율적인 그래프 알고리즘의 성능 분석

FEBRUARY 2019

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Yeonil Yoo

Performance evaluation of space efficient graph algorithms

공간 효율적인 그래프 알고리즘의 성능 분석

지도교수 Srinivasa Rao Satti

이 논문을 공학석사학위논문으로 제출함

2018 년 10 월

서울대학교 대학원

컴퓨터 공학부

유 연 일

유연일의 석사학위논문을 인준함

2018 년 12 월

위 원 장	김 선	(인)
부위원장	Srinivasa Rao Satti	(인)
위 원	Bernhard Egger	(인)

Abstract

Performance evaluation of space efficient graph algorithms

Yeonil Yoo
Department of Computer Science and Engineering
Collage of Engineering
The Graduate School
Seoul National University

Various graphs from social networks or big data may contain gigantic data. Searching such graph requires memory scaling with graph. Asano et al. ISAAC (2014) initiated the study of space efficient graph algorithms, and proposed algorithms for DFS and some applications using sub-linear space which take slightly more than linear time. Banerjee et al. ToCS **62**(8), 1736-1762 (2018) proposed space efficient graph algorithms based on read-only memory(ROM) model. Given a graph G with n vertices and m edges, their BFS algorithm spends $O(m+n)$ time using $2n+o(n)$ bits. The space usage is further improved to $n \lg 3 + o(n)$ bits with $O(m \lg n f(n))$ time, where $f(n)$ is extremely slow growing function of n. For DFS, their algorithm takes $O(m+n)$ time using $O(m \lg \frac{m}{n})$. Chakraborty et al. ESA (2018) introduced in-place model. The notion of in-place model is to relax the read-only restriction of ROM model to improve the space usage of ROM model. Algorithms based on in-place model improve space usage exponentially, to $O(\lg n)$ bits, at the expense of slower runtime. In this thesis, we focus on exploring proposed space efficient graph algorithms of ROM model and in-place model in detail and evaluate performance of those algorithms. We implemented almost all the best-known space-efficient algorithms for BFS and DFS, and evaluated their performance. Along the way, we also implemented several space-efficient data structures for representing bit vectors, strings, dictionaries etc.

Keywords: Depth first search, Breadth first search, space efficient graph algorithms

Student Number: 2015-22905

Contents

List of Figures

List of Tables

Chapter 1

Introduction

With massive growth of a data interaction of modern days, the data has grown large enough to be called as a “big data” [22, 24, 31]. 2.5 quintillion bytes of the data are created everyday, and past two years of data forms 90 percent of data in the world [31]. Studying such massive data comes with the issue of space efficiency since space usage increases as algorithms are relative to size of data but memories are limited. With such limited memories, interest of improving space efficient graph algorithm has been raised. General graph algorithms known to be linear time bound with $O(n \lg n)$ bits. Asano et al. [2] proposed a graph algorithm performing $O(m \lg n)$ time using $O(n)$ bits and another algorithm using $n + o(n)$ bits while running in polynomial time. Elmasry et al. [13] further improved algorithm to $O(m \lg \lg n)$ time using $O(n)$ bits.

In this paper, we introduce space efficient graph algorithms studied by Banerjee et al. [3] and Chakraborty et al. [6, 7], and we implement and evaluate performance of those algorithms. Several models of computations have been purposed to design space efficient algorithms. Among those models, we focus on read-only memory(ROM) model and in-place model, which will be discuss in detail later.

For the DFS problem, two versions of DFS have been studied. *lexicographically smallest* DFS(*lex*−DFS) produces unique DFS tree by traveling unvisited vertex appearance order in the adjacency list. Another version is *general*−DFS, where DFS travels regardless of adjacency list order.

1.1 Related Work

In the early ages, Munro and Paterson [27] suggested multi-pass streaming model, where input data has given in one-way read-only sequence of streaming data, so that no random access is available. Several streaming graph algorithms in the multi-pass streaming model were studied [11].

Other than ROM model and in-place model, other semi-streaming models are considered for space efficient algorithms [1, 15, 27]. Restore model is one of models that introduced by Chan et al. [8], where input may be altered in promise to be restored to original in the end. Kammer and Sajenko [21] devised space efficient BFS and DFS algorithms working in the restore model, where both traversals can be done in $O(m + n)$ time using linear words.

Additionally, Buhrman et al. [4, 5] introduced catalytic-space model, where a workspace memory consists of small amount of clean space and large occupied space. With large space being arbitrary and incompressible, large space may be used with promise to be return to original state. There is no well-known space efficient graph algorithm designed in the catalytic-space model in our knowledge.

1.2 Organization of the Paper

The rest of this thesis is organized as follows. Preliminary information on input models and succinct data structures will be introduced in Chapter 2. In Chapter 3 and Chapter 4, we elaborate on the theoretical details of space efficient algorithms. Afterwards, experiments and empirical results are given in Chapter

5. Finally, conclusion is discussed in Chapter 6.

Chapter 2

Preliminaries

In this chapter, preliminaries works upon this thesis are introduced. We start with overview of symbols that will be used through out the thesis in Table 2.1.

G	A graph $G = \{V, E\}$ where V and E are sets of vertices and edges
n, m	The numbers of vertices and edges in G
d_v	A degree of vertex v
$depth$	Depth of the current level
l	Depth of a tree

Table 2.1 Notations used in this paper

2.1 ROM Model

In the ROM model, the input is given in a read-only memory, where any modification is not possible. In order to produce any result of algorithms, the result must be written in a write-only memory. Other than read-only memory and write-only memory, workspace memory is available for limited random access memory[3, 6, 16].

2.2 In-place Model

To achieve space efficient graph algorithm, in-place model is introduced by Chakraborty et al.[6] to achieve beyond inherent space bound barrier while maintaining reasonable time bound by relaxing the limitations of ROM model. In-place model considers two input graph representations: array representation and linked list representation. Unlike ROM model, where computation may not be modified, in-place model assumes that modification is possible in limited manner. In-place model introduces rotate model and implicit model.

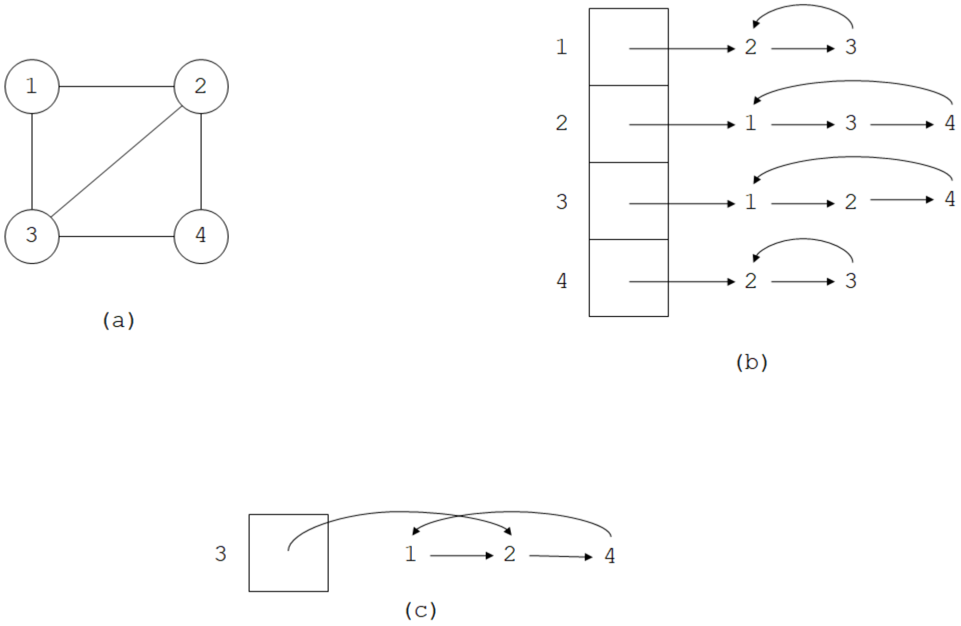

Figure 2.1 (a) An undirected graph G. (b) A circular adjacency list representation of G. (c) Result of single rotation on vertex 3.

In rotate model, we assume that only pointer points to adjacency list may be modified while maintaining adjacency list unmodified. The adjacency list is circular linked that last element is connected with first element as illustrated in Figure 2.1.

In implicit model, any two elements in the adjacency list may be swapped.

Further more, we can simulate rotate model algorithm with implicit model using Lemma 1.

Lemma 1 *[6] Let D be the maximum degree of a graph G. Then any algorithm running in t(m, n) time in the rotate model can be simulated in the implicit model in (i)* $O(D \cdot t(m,n))$ *time when G is given in an adjacency list, and (ii)* $O(\lg D \cdot t(m,n))$ *time when G is given in an adjacency array. Furthermore, let* $r_v(m,n)$ *denote the number of rotations made in v's list, and* $f(m,n)$ *be the remaining number of operations. Then any algorithm running in* $t(m,n) = \sum_{v \in V} r_v(m,n) + f(m,n)$ *time in the rotate model can be simulated in the implicit model in (i)* $O(\sum_{v \in V} r_v(m,n) \cdot d_v + f(m,n))$ *time when G is given in an adjacency list, and (ii)* $O(\sum_{v \in V} r_v(m,n) \lg d_v + f(m,n))$ *time when G is given in an adjacency array.*

2.3 Succinct Data Structure

Data structure with capability to answer *select* query is required for theorems that will be introduced. Given a bitstring O, $select_\alpha(O,i)$ queries the position of the i-th α in O.

Lemma 2 *[3, 9, 18, 26] We can store a bitstring O of length n with additional* $o(n)$ *bits such that select operation can be supported in* $O(1)$ *time. Such a structure can also be constructed from the given bitstring in* $O(n)$ *time.*

2.4 Changing Base Without Losing Space

Suppose we want to represent a vector $A[1..n]$ with each element to be some finite alphabet Σ, then optimal space for this vector is $\lceil n \lg \Sigma \rceil$. However this representation has limitation of (i) stream of symbols can not be encoded with low memory and (ii) reading or writing a single element involves reading whole vector. Dodis et al. [12] introduces solution to this problem.

Lemma 3 *[3, 12] On a Word RAM, one can represent a vector A[1..n] of elements from a finite alphabet* Σ *using* $n \lg |\Sigma| + O(\lg^2 n)$ *bits, such that element of the vector can be read or written in constant time.*

2.5 Dictionaries With Findany Operation

We consider problem where the data structure needs to maintain a set S while supporting following operations:

- insert Insert the element into the set.
- search Determine whether the element is in the set.
- delete Delete the element from the set.
- findany Find any element in the set.

Table 2.2 shows list of existing dictionaries that supports above operations along with operation time and required space. In this paper, we will be using findany structure which introduced by Banerjee et al. [3].

Structure	Ins/Del	Search	Findany	Space(bits)	W/E
CV	$O(1)$	$O(1)$	$O(n)$	n	W
BBST	$O(\lg n)$	$O(\lg n)$	$O(1)$	$O(k \lg n)$	W
DRS [20, 28]	$\frac{\lg n}{\lg \lg n}$	$O(1)$	$\frac{\lg n}{\lg \lg n}$	$n + o(n)$	W
YFT [30]	$O(\lg \lg n)$	$O(\lg \lg n)$	$O(\lg \lg n)$	$O(k \lg n)$	E
DRR [25]	$O(\lg \lg n)$	$O(\lg \lg \lg n)$	$O(\lg \lg \lg n)$	$O(k \lg n)$	E
findany [3]	$O(1)$	$O(1)$	$O(1)$	$n + o(n)$	W

Table 2.2 Comparison of existing dictionary representations. CV is characteristic vector, and BBST is balanced binary search tree. W and E denotes worst case and expected.

Using Lemma 3, we can support operations in $O(1)$ time on the data structure maintaining a collection of c disjoint sets using $n \lg c + o(n)$ bits.

Lemma 4 *[3] A collection of c disjoint sets that partition the universe of size n can be maintained using $n \lg c + o(n)$ bits to support insert, delete, search and findany operations in constant time. We can also enumerate all elements of any given set in O(k+1) time where k is the number of elements in the set. The data structure can be initialized in O(1) time.*

Additionally, similar to findany dictionary, Hagerup and Kammer [19] proposed choice dictionary which supports above operations in $O(1)$ time while occupying $n + O(n/\lg n)$ space.

Chapter 3

Breadth First Search

Breadth first search (BFS) is one of the simplest search algorithms of searching a graph, and there are many known algorithms based on BFS such as Prim's minimum-spanning tree algorithm and Djkstra's shortest path [10, 3]. We will introduce space efficient BFS based on ROM model, rotate model and implicit model.

3.1 ROM model

Theorem 1 *[3] Given a directed or undirected graph G, its vertices can be output in a BFS order starting at a vertex using* $2n + o(n)$ *bits in* $O(m + n)$ *time.*

Assume that vertices have one of four color sets. Consider unvisited vertex as *white*, finished vertex as *black*, and in-progress of exploring as $gray1$ and $gray2$. We start with adding starting vertex into $gray1$. Every *white* adjacent vertices of $gray1$ vertices are added into $gray2$, and $gray1$ vertices are moved to *black*. Repeat steps on $gray2$ set, adding all *white* adjacent vertices of $gray2$ vertices to $gray1$, and moving $gray2$ vertices to *black*. This procedure continues until there are no remaining *gray*1 or *gray*2 vertex left. With Lemma 4, we can explore BFS using $2n + o(n)$ bits in $O(m + n)$ time.

Theorem 2 *[3] Given a directed or undirected graph G, its vertices can be output in a BFS order starting at a vertex using* $n \lg 3 + O(\lg^2 n)$ *bits and in* $O(mn)$ *time.*

Consider there are three color sets: *white* for unvisited vertices, and *gray*1/*gray*2 for exploring or finished vertices. We start with adding starting vertex into *gray*1. We scan each vertices and if vertex is a *gray*1, add all its *white* adjacent vertices to *gray*2. After first scan is complete, scan for *gray*2 vertices and add all *white* adjacent vertices to *gray*1. This procedure continues until there is no vertex added to *gray*1 or *gray*2. Since we are using three colors, by using Lemma 3, we can explore BFS using $n \lg 3 + O(\lg^2 n)$ bits and in $O(mn)$ time.

Theorem 3 *[3] Given a directed or undirected graph G, its vertices can be output in a BFS order starting at a vertex using* $n \lg 3 + o(n)$ *bits of space and in* $O(m \lg^2 n)$ *time.*

To improve runtime of Theorem 2, we maintain two queues $Q0$ and $Q1$ with size of $n/\lg^2 n$. Whenever we change a *white* vertex to *gray*1 or *gray*2, we push those vertex into queue $Q0$ or $Q1$. When queue successfully maintains every vertices, we pop each vertex in the queue. However, if queue happens to be overflow, empty the queue and simply perform Theorem 2 instead. Overflow occurs when there are least $n/\lg^2 n$ vertices in the level, and this cannot occur more than $\lg^2 n$ times. Hence, it takes $O(m \lg^2 n)$ time. Color array takes $n \lg 3 + O(\lg^2 n)$ and queue takes $O(n/\lg n)$ bits. Overall the space requirement is $n \lg 3 + o(n)$ bits.

Theorem 4 *[3] Given a directed or undirected graph G, its vertices can be output in a BFS order starting at a vertex using* $n \lg 3 + O(n/f(n))$ *bits of space* *and in $O(mf(n)\lg n)$ time where f(n) is any extremely slow-growing function of n.*

We now adjust two queues' size from Theorem 3 to be $n/f(n)\lg n$, where function $f(n)$ is any slow growing function, then the space requirement of queues is $O(n/f(n))$ bits while running time is $O(mf(n)\lg n)$.

3.2 Rotate model

Theorem 5 *[6] Given a directed or undirected graph G with depth of the BFS tree starting at the source vertex s be l, then in rotate model, its vertices can be output in a BFS order starting at s using $n+O(\lg n)$ bits and $O(m+nl^2)$ time.*

By the property of BFS, if a vertex i is located in level *dist* in BFS tree, we know that distance from starting vertex to i is *dist*. With this property, we can backtrack visited vertices' depth level. First, we maintain bit vector of length n and maintain variable *dist* initialized to 0. Mark starting vertex and all adjacent vertices visited and increment *dist* by 1. When we set the vertices visited, we rotate their adjacency lists such that parent vertex becomes first index of adjacency list. Now, we scan *visited* bit vector, and if a vertex is marked as visited, we check if that vertex is located in targeted level. This is checked by traveling first index of adjacency list. If we reach root starting from that vertex at exactly *dist* steps, we add all unvisited adjacent vertices of that vertex to *visited*. After each scan of *visited* bit vector, increment *dist*. We stop algorithm after there is no vertex added to *visited*. Time spent on level *dist* can by analyzed to be $n\,dist + \sum_{i\in V(d)} d_i$ where $V(d)$ is the set of vertices in level *dist*. The runtime of overall level is analyzed to be $O(m+nl^2)$ where l is the depth of the BFS tree.

Theorem 6 *[6] Given a directed or undirected graph G with depth of the BFS tree starting at the source vertex s be l, and s can reach all other vertices, then* *in rotate model, its vertices can be output in a BFS order starting at s using $O(\lg n)$ bits and $O(ml + nl^2)$ time.*

Space usage can be further reduced to $O(\lg n)$ bits by not containing *visited* bit vector and backtrack to check if vertex is visited. This scarifies runtime but algorithm achieves BFS without *visited* bit vector. By the property of BFS, given *dist* depth explored BFS, we know that there is no such unvisited vertex that can reach starting vertex s within *dist* steps of travel. With this property, we can check if vertex is visited with $O(dist)$. The total time spent at level *dist* is $O(n\,dist + dist \sum_{i \in V(d)} deg(i))$, and overall runtime is $O(ml + nl^2)$ time.

3.3 Implicit model

Theorem 7 *[6] Given a directed or undirected graph G with source vertex that can reach all other vertices by a distance of at most l, then in implicit model, its vertices can be output in a BFS order using $O(\lg n)$ bits and $O(m + nl^2)$ time.*

In implicit model, we can simulate Theorem 5, but without visited bit vector. A vertex with degree higher than two can be checked if visited in $O(1)$ by swapping visited vertex's second and third element of adjacency list. For degree-1 vertex, there exist only a single adjacent vertex (a parent) that it can not be visited twice. Therefore, degree-1 vertex does not need to be encoded for visited. For degree-2 vertex, vertex can be encoded using first and second element of adjacency list. Degree-2 vertex does not need to store parent vertex information. If both of the adjacency vertices are already visited, vertex will be a leaf node and it will not travel any further from the vertex, such that there is no need to travel back to starting vertex to count the depth level. If only one of the adjacent vertices is visited, BFS will branch through unvisited adjacency vertex. When child of degree-2 vertex tries to travel back to the starting vertex, we know the child vertex of the degree-2 vertex, which means we also know the parent vertex. This results BFS using $O(\lg n)$ bits and $O(m + nl^2)$ time.

Theorem 8 *[6] Given a directed or undirected graph G with source vertex that can reach all other vertices by a distance of at most l and if there are no degree 2 vertices, then in implicit model, its vertices can be output in a BFS order using $O(\lg n)$ bits and $O(m + nl)$ time.*

For graph with no such vertex with degree-2, further improvement can be achieved by implementing Theorem 1. We can implement 4 colors ($white$, $gray1$, $gray2$, and $black$) with permutation of three element of adjacency list. For degree-1 vertex, we need not encode anything since it will be visited only once from adjacency vertex. Since we do not maintain any queue, we need to scan whole vertex list for l many times, where l is the height of the BFS tree. This results BFS using $O(\lg n)$ bits and $O(m + nl)$ time.

Chapter 4

Depth First Search

Depth first Search (DFS) is another graph searching algorithm. With DFS, all the cut vertices and bridges in a graph can be found [3]. Other known applications of DFS are maze searching[14], web-crawling and AI.

4.1 ROM model

Theorem 9 *[3] Given a graph G, its vertices can be output in a DFS order starting at a vertex in $O(m+n)$ time using $2m + (\lg 3 + 2)n + o(m+n)$ bits for directed and $4m + (\lg 3 + 2)n + o(m+n)$ bits for undirected.*

Given a graph G, we can construct a bit sequence O of length $m+n$ bits such that it consists of 0-bit to represent a vertex i and followed by d_i many 1-bit representing number of degree at a vertex i. We also construct a bit sequences E of length $m+n$ bits which initially set to be a sequence of 0-bit as illustrated in Figure 4.1. During a DFS, we can use *select* operation from Lemma 2 to find a position of a vertex and mark on a bit sequence E with a bit 1 at corresponding edge that we have traveled. We also construct a color array C with three colors: *white*, *gray* and *black*. Given a starting vertex s, we perform DFS starting with adding vertex s to color *gray*. Given a current vertex i, DFS searches any *white* adjacent vertex. If any such vertex, say j, is found, we set that j to color *gray* and mark corresponding bit of e_{ij} on E to 1. DFS continues to travel until no *white* adjacent vertex is found. When such event occurs, mark current vertex to *black* and backtrack to the parent. We can find the parent by searching the adjacency vertex with an edge between the two vertices on E. DFS continues until no *white* adjacent vertex is found. The space of both bit sequences E and O takes $2m + 2n$ bits for directed and $4m + 2n$ bits for undirected. We can represent color array using Lemma 3 in $n \lg 3 + o(n)$. The bit sequence O supports *select* operation, taking $o(m+n)$ bits. Overall we need total $2m+(\lg 3+2)n+o(m+n)$ bits for directed and $4m+(\lg 3+2)n+o(m+n)$ bits for undirected.

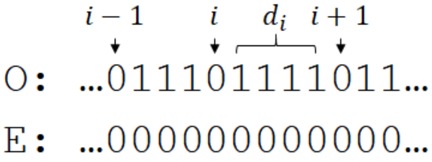

Figure 4.1 Bit sequences of E and O

Theorem 10 *[3] Given a graph G, its vertices can be output in a DFS order starting at a vertex in $O(m+n)$ time using $2m+3n+o(m+n)$ bits for directed and $4m + 3n + o(m + n)$ bits for undirected.*

Observed from Theorem 9, DFS requires to know if vertex is visited or unvisited, however, it does not require if vertex is currently exploring or finished. We can combine *gray* and *black* color together and use bit vector of n bits instead of color array. This reduces space usage to $2m + 3n + o(m + n)$ bits for directed and $4m + 3n + o(m + n)$ bits for undirected.

Theorem 11 *[3] Given a directed or undirected graph G, its vertices can be output in a DFS order starting at a vertex using* $O(n \lg(m/n))$ *bits in* $O(m+n)$ *time.*

The space usage of previous DFS theorem can be further reduced to $O(n \lg(m/n))$ bits. We first construct a bit sequence B such that B consists the sequence of $0^{\lceil \lg d_i \rceil - 1}1$ for vertices $1 \leq i \leq n$ and supports *select* operation. The length of bit sequence B analyzed to be $\sum_{i=1}^{n} \lceil \lg d_i \rceil$, which is bound by $O(n \lg(m/n))$ bits. We also construct a bit sequence P with same length as B with initially a sequence of 0-bit. When DFS takes path from vertex v_i to vertex v_j, we perform *select* operation of v_j on bitvector B to find the corresponding location, and we can save e_{ij} using $\lceil \lg d_j \rceil$ bits. This DFS theorem takes $O(n \lg(m/n))$ bits in $O(m+n)$ time.

4.2 Rotate model

Theorem 12 *[7] Given a directed or undirected graph G, its vertices can be output in a lex−DFS order starting at vertex using* $n \lg 3 + O(\lg^2 n)$ *bits in* $O(m+n)$ *time.*

We make use of color array containing three colors: *white*, *gray* and *black*. We start with adding a starting vertex s to color *gray* and travel DFS order. At a current vertex i, we scan adjacency list for *white* adjacent vertex. When we encounter first *white* vertex j, we rotate adjacency list of i such that vertex j becomes the head of adjacency list and set j's color to *gray* as we travel to vertex j. If we did not encounter any *white* vertex, we change vertex i's color to *black*, and we scan adjacency list for vertex that is *gray* and head of adjacency list is i. The only space usage is color array, which costs $n \lg 3 + O(\lg^2 n)$ bits, and total time of cost is $O(m+n)$ time.

Theorem 13 *[7] Given a directed or undirected graph G, its vertices can be output in a DFS order starting at vertex using* $n + O(\lg n)$ *bits in* $O(m + n)$ *time.*

The space can be further improved by maintaining *visited* bit vector instead of color array. Following Theorem 12, we can backtrack to parent by scanning adjacency list for vertex marked with *visited* and head of adjacency list is i.

Theorem 14 *[7] Given a G, its vertices can be output in a DFS order starting at vertex with capability to reach all other vertices using* $O(\lg n)$ *bits in* $O(m^2/n + ml)$ *time for undirected graph and* $O(m(n + l^2))$ *time for directed graph, where* l *is the maximum depth of the DFS tree.*

Undirected graph

To further improve space usage to $O(\lg n)$, we do not maintain any color nor *visited* bit array. We maintain two variables, current depth level *depth* and maximum depth level *max*. *depth* maintains current depth level, increments by 1 whenever we travel to *white* vertex and decrements by 1 whenever we backtrack to its parent. *max* maintains maximum depth *depth* has achieve. We start with starting vertex s with both *depth* and *max* to be 1. Whenever we travel to *white* vertex j from current vertex i, we rotate i's head to be j, and we also rotate j's head to be i. When we backtrack to i from j, we rotate j's head to be i again. With this design, as shown in Figure 4.2, any adjacency vertex after the parent vertex in adjacency list may be considered as visited vertices. We can recover the parent vertex by traveling *depth* steps from the root. Any vertex between head and parent vertex is candidate for *white* vertex but requires test to verify.

white vertex can be identified by checking if a vertex is neither *gray* nor *black*. A vertex is a *gray* if we can travel within *depth* steps from root following head in the adjacency list. To identify if a vertex is *black*, we first check whether a path exist such that we can travel from vertex j to vertex i at most $(max - depth)$ steps by following head of each vertex. If there exists such a path, let z be the vertex before i in path, then vertex j is *black* if z appears after parent vertex.

Identifying *gray* vertex takes at most *depth* steps. Identifying *black* vertex takes at most $(max - depth)$ for path, and d_i steps to check if z appears after parent vertex. Together, we spend $max + d_i$ at vertex i. Overall runtime is $\sum_{v \in V} d_i\,(d_i + l) = O(m^2/n + ml)$ where l is maximum depth.

Directed graph

Undirected graph's adjacency list uses parent vertex to split processed vertices and pending vertices. Since directed out-adjacency list does not guarantee to have a parent vertex, we spend $O(m)$ time during preprocessing step to rotate the out-adjacency list to bring minimum vertex front for every vertex. The minimum adjacent vertex will be used to split processed vertices and pending vertices among adjacency list. We maintain two variables *depth* and *max* as before. At vertex i, when we travel to *white* vertex j, vertex i's out-adjacency list rotates such that j becomes head, and vertex j's in-adjacency list rotates such that i becomes head.

To determine vertex j as *white* vertex, we need to test possible scenarios as shown in Figure 4.3. Vertex j is *gray* if we can travel to j from starting vertex following head of the out-adjacency list in *depth* steps. If there is a path from j to i traveling head of the in-adjacency list in $max - depth$ steps, then let vertex

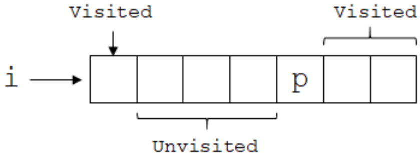

Figure 4.2 Adjacency list property of rotate model

c be the vertex appearing in path before i. If c appears after minimum vertex in i's out-adjacency list, then j is a *black* vertex. If there is a path from j to starting vertex in max steps, then let z be the first *gray* vertex appearing in path and c be the vertex before z. If c appears after minimum vertex in z's out-adjacency list, then j is a *black* vertex. If vertex j is neither *gray* or *black*, then vertex j is *white*.

Identifying *gray* vertex takes at most *depth* steps. Identifying *black* vertex takes at most *max* steps for finding path, and at each vertex in the path, we take *depth* steps to identify whether the vertex is *gray*. Once a *gray* vertex is reached, we spend d_z to determine whether c appears after or before the minimum vertex. At vertex i, we spend $depth + depth \cdot max + d_z$ time. Since z can have at most degree of n, overall runtime of algorithm is $\sum_{v \in V} d_v(depth + depth \cdot l + n) = O(m(n + (1 + l)l) = O(m(n + l^2))$.

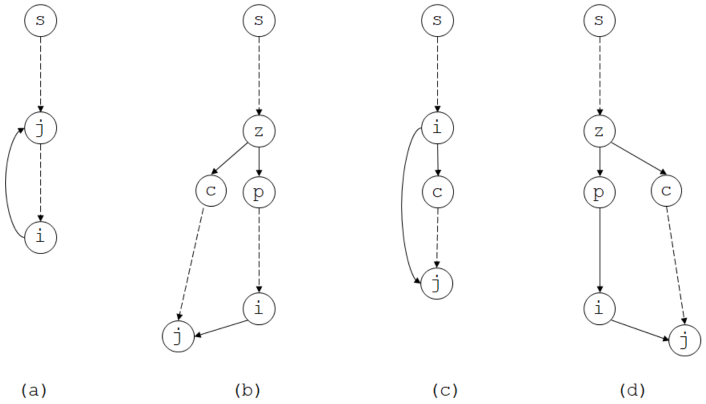

Figure 4.3 Possible scenarios to consider when vertex i checking vertex j. (a) j is gray (b) j is black and path does not contain i (c) j is black path contains i (d) j is white and path does not contains i

4.3 Implicit model

Theorem 15 *[7] Given a graph G, its vertices can be output in a lex−DFS order using $O(\lg n)$ bits in $O(m^3/n^2+lm^2/n)$ time if G is given in an adjacency linked list and $O(m^2 \lg n/n)$ time if G is given in adjacency array for undirected graph, where l is the maximum depth of the DFS tree. For directed graph, algorithm takes $O(m^2(n+l^2)/n)$ time if G is given in an adjacency linked list and $O(m \lg n(n+l^2))$ time if G is given in an adjacency array.*

lex−DFS order using $O(\lg n)$ bits can be achieved by implementing Theorem 14 with slight modification. Whenever we need to bring parent to front, we simulate rotation without changing the order of the adjacency list. By Lemma 1, this takes $O(\sum_{v \in V} d_v \lg d_v + n) = O(m^3/n^2 + lm^2/n)$ time for adjacency linked list and $O(\sum_{v \in V} d_v(d_v + l) \cdot \lg d_v) = O(m^2(\lg n)/n + ml \lg n))$ time for adjacency array of undirected graph. Directed also follows similar manner, resulting time bound showed in Theorem 15.

Theorem 16 *[7] Given a directed or undirected graph G, its vertices can output in a general-DFS order using $O(\lg n)$ bits in $O(m^2/n)$ time if G is given in an adjacency list and in $O(m^2(\lg n)/n+ml \lg n))$ time if G is given in an adjacency array.*

DFS traversal can achieve more optimized run time while maintaining $O(\lg n)$. Instead of explicitly storing *visited* bit vector, we can encode visited information by switching second and third vertices in adjacency list such that second vertex value is greater than third vertex. This requires a preprocessing to guarantee that unvisited vertex has lesser value of second vertex than third vertex initially. We reserve first vertex of adjacency list to be the parent vertex such that whenever we have finished visiting current vertex, we can backtrack to the parent with $O(1)$ time. This covers only vertex with degree of 3 or higher. Degree-1 vertex has only one vertex that is capable of being parent and such vertex will not be visited again. Thus, degree-1 vertex does not require to be encoded for visited information. For degree-2 case, by the property of DFS, such vertex always have 1 parent and 1 child. When we encounter such case, we travel through such vertex until we encounter non-degree-2 vertex. If we encountered a degree-1 vertex, we simply output vertex and return to the parent of the first degree-2 vertex. If we encounter a degree-3 or higher vertex, we set that vertex as visited and continue to DFS from that vertex. If we need to backtrack, we know which vertex is a child vertex such that other vertex must be a parent. The total runtime of this algorithm is bounded by $\sum_{v \in V} d_v^2 = O(m^2/n)$. We can improve further by implementing DFS algorithm in adjacency array and initially sorted. Since adjacency list is sorted, we can use binary search. The total runtime would be $\sum_{v \in V} d_v \lg d_v = O(m \lg m + n)$.

Chapter 5

Experimental Results

The algorithms have been implemented in the C++ programming language and compiled with g++ 8.2.1. The environment in which the tests were executed features a Intel Core i7-7700k 4.20GHz CPU, 64GB DDR4 RAM. The SDSL [17] library was used to aid our implementation with bitvector and its related operations.

Throughout the experiments, both synthetic and real data sets were considered. Synthetic graphs consist of $n = 10000$, 20000 and 50000 with $m = n - 1$, $3n$, $5n$, $10n$, $20n$ and complete graph. Directed graphs have average degrees equal to $\frac{m}{n}$. For undirected graph, there is no difference between in-edge and out-edge that average degrees equal to $\frac{2m}{n}$.

For real world graphs, data sets were obtained from [23] and [29] with attributes shown in Table 5.1. `Facebook` and `FacebookA` are undirected graphs containing circles from Facebook. `Brightkite` is an undirected graph derived from a geo-location based social networking service. `EmailEU` is a directed graph generated from the e-mail network in a large European research institution, `Slashdot` is a directed graph of a network containing links between users of a forum and `Flickr` is a directed graph generated from a crawl process of Flickr social network service. Those graphs are modified such that root can reach every vertices.

	name	n	m	d_{max}	d_{avg}	d_{stdev}
Undirected	`Facebook`[23]	4039	88234	1045	43.7	0.546
	`Brightkite`[23]	58228	214624	1134	7.37	0.0264
	`FacebookA`[29]	3097165	23667394	4915	15.3	0.00812
Directed	`EmailEU`[23]	1005	24969	333	24.8	0.784
	`Slashdot`[23]	77360	834623	2507	10.8	0.0316
	`Flickr`[29]	820878	10109620	272410	12.3	0.0125

Table 5.1 Attributes of real world data sets.

For following experiment, time is measured in milliseconds, and space is measured in kilobytes.

5.1 BFS

In this section, we refer Theorems 1, 2, 3, and 4 as *ROM1*, *ROM2*, *ROM3* and *ROM4*, Theorems 5 and 6 as *ROT1* and *ROT2*, and Theorems 7 and 8 as *IMP1* and *IMP2*. For the following BFS experiments, *ROM4* has been implemented with queue size to be $\frac{n}{\lg^2 n}$ elements. Because *IMP2* has requirement for degree of every vertex, experiments of *IMP2* performed with only some graphs that satisfy the requirement.

Undirected BFS

Tables 5.2 to 5.4 show runtime result of undirected BFS for synthetic graphs. Among the algorithms in ROM model, *ROM2* shows the slowest runtime when number of edges is low. This is due to scanning m time for each level of the BFS tree. As number of edges increases, queues used in *ROM3* and *ROM4* would overflow, resulting similar runtime to that of *ROM2*.

	9999	30000	50000	100000	200000	Complete
STD	1.9	2.4	2	1.6	3	652
ROM1	1.9	1.8	1.8	1.9	2.1	104
ROM2	22.5	11.4	12.7	13.4	24.7	4300
ROM3	13.7	8	8.6	12.8	20.9	4322
ROM4	13.7	7.9	8.5	12.6	20.8	4302
ROT1	47.5	2.34	3.62	7.05	21.4	0.766
ROT2	53.3	3.14	5.45	8.8	23.6	0.746
IMP1	96	4.18	3.33	3.92	5.95	6375
IMP2				3.31	4.58	5772

Table 5.2 Undirected BFS time with synthetic graph ($n = 10000$).

	19999	60000	100000	200000	400000	Complete
STD	1.6	1.4	2.5	3	6.1	2607
ROM1	3.7	3.6	3.7	4	4.6	414.2
ROM2	4.83	25.1	23.4	35.9	50.4	18537
ROM3	30.1	16.4	21	26.6	49.1	18623
ROM4	30	16.2	20.9	26.4	48.5	18554
ROT1	132	9.3	9.45	19	41.4	1.5
ROT2	143	10.4	14.5	26	53	1.5
IMP1	469	12.8	9	10.4	15.8	38054
IMP2				8.4	12.5	26731

Table 5.3 Undirected BFS time with synthetic graph ($n = 20000$).

	49999	100000	2500000	500000	1000000	Complete
STD	2.1	11	7	9.7	17	15894
ROM1	9.4	9.4	9.9	11.3	13.3	2571
ROM2	129	66.2	75.8	83.8	123.6	115776
ROM3	83	47.6	49.3	80.6	120.4	116217
ROM4	82.5	47.3	48.8	79.3	119.1	115865
ROT1	463	29.2	34.2	54.5	95	3.9
ROT2	457	36.1	48.1	79	122	4
IMP1	773	62	48.2	47	73.5	358145
IMP2				43.8	64.5	313489

Table 5.4 Undirected BFS time with synthetic graph ($n = 50000$).

Figure 5.1 shows relative time ratio of the undirected BFS algorithms on $n = 10000$. We set the datum point as $m = n - 1$. Algorithms in the in-place model depend on the depth of the result BFS tree. When $m = n - 1$, depth of tree tends to be very deep, giving high l.

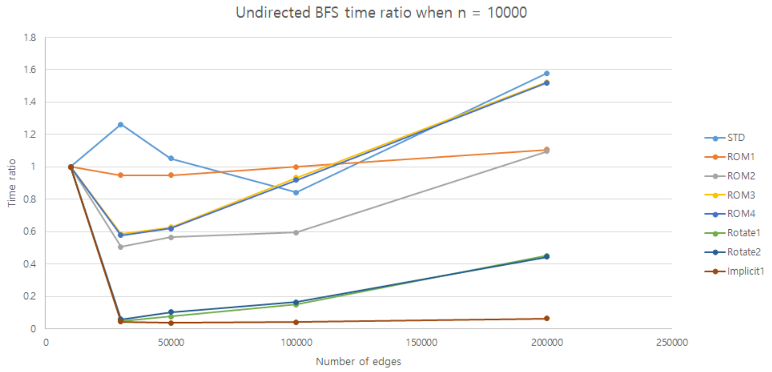

Figure 5.1 Time ratio of undirected BFS with synthetic graph ($n = 10000$).

Tables 5.5 to 5.7 show result of time and space cost of undirected BFS in the real data graphs. For runtime, standard BFS and *ROM1* show the fastest runtime. For `Facebook`, it has average degree of 43.7 while queue size for both

ROM3 and *ROM4* is 29. Having higher average degree than $\frac{n}{\lg^2 n}$ results similar runtime compared to *ROM2*. For space, *ROM1* takes the most space, followed by standard BFS. Except *ROM1*, algorithms in ROM model take much lesser space than standard BFS. *ROT1* takes significantly less space and other in-place model algorithms do not take any extra space except some variables.

	Time(ms)	Space(KB)
STD	1.17	23
ROM1	0.824	37
ROM2	19.6	7.9
ROM3	19.8	8.4
ROM4	19.7	8.4
ROT1	20.2	0.5
ROT2	20.4	0
IMP1	9.8	0

Table 5.5 Undirected BFS result of `Facebook`.

	Time(ms)	Space(KB)
STD	550.6	15332.6
ROM1	789.1	25569.4
ROM2	12930.5	3456.7
ROM3	8661.1	3560.8
ROM4	8600.9	3560.8
ROT1	9300	378.1
ROT2	6059	0
IMP1	3645	0

Table 5.6 Undirected BFS result of `FacebookA`.

	Time(ms)	Space(KB)
STD	3.85	235.3
ROM1	10.1	492.3
ROM2	133.9	91
ROM3	78.1	94.6
ROM4	76.2	94.6
ROT1	55.5	7.11
ROT2	21.8	0
IMP1	10.4	0

Table 5.7 Undirected BFS result of `Brightkite`.

Directed BFS

Tables 5.8 to 5.10 show the runtime result of directed BFS for synthetic graphs. The results show arguably similar result to those of undirected BFS. Nevertheless, for $m = n - 1$, *IMP1* performs much worse than the undirected version, though this phenomenon is not derived from algorithmic perspective. This is because vertices of the original graph have a single path to the source vertex, giving very deep BFS tree (i.e., large l). Note that the minimum spanning tree, constructed while generating synthetic undirected graphs, allows multiple neighbors in a vertex, different from the only possible case for directed graphs.

	9999	30000	50000	100000	200000	Complete
STD	0.883	0.499	0.532	1.4	1.5	650
ROM1	2.4	1.8	1.8	1.9	2	105
ROM2	5273	1.3	10.6	12.6	13.4	4305
ROM3	1.8	8.5	7.6	8.7	12.8	4326
ROM4	1.9	8.5	7.6	8.6	12.6	4308
ROT1	262	3.6	1.9	2.8	4.5	0.67
ROT2	220	3.2	2.6	4.4	6.6	0.65
IMP1	7883300	11.4	4.6	4.3	4.6	6083
IMP2					4	5308

Table 5.8 Directed BFS time with synthetic graph ($n = 10000$).

	19999	60000	100000	200000	400000	Complete
STD	1.7	1.7	1.7	1.8	3.6	2486
ROM1	4.8	3.9	3.7	3.8	4.3	415
ROM2	22718	31.6	28.7	27.3	37.1	18525
ROM3	3.9	20.6	16.3	18.7	26.9	18610
ROM4	3.9	20.4	16.1	18.5	26.7	18537
ROT1	661	10.4	8	7.5	11.8	1.38
ROT1	579	9.3	8.5	11.4	21.4	1.34
IMP1	-	30.2	13.8	13	14	33096
IMP2					12.6	61150

Table 5.9 Directed BFS time with synthetic graph ($n = 20000$).

	49999	100000	2500000	500000	1000000	Complete
STD	0.703	4.9	4.9	65.9	10.5	15874
ROM1	12.1	9.6	9.8	11	12.3	514
ROM2	14198	76	64.2	59.8	87.2	23153
ROM3	9.7	47.4	46.9	53.3	83.2	23245
ROM4	9.8	47.1	46.4	53	82.5	23174
ROT1	2278	38	22.4	19.7	31.1	3.57
ROT2	1886	32.6	27.2	35.1	57	3.52
IMP1	-	145	61.5	50.5	56.5	340179
IMP2					52	557214

Table 5.10 Directed BFS time with synthetic graph ($n = 50000$).

Space usage for space-efficient algorithms does not depend on number of edges but only on number of vertices. Table 5.11 shows space usage for undirected and directed standard BFS, ROM model algorithms and *ROT1*. *ROM1* shows the highest space usage among all, but since *ROM1* does not depend on number of edges, *ROM1* is more space efficient than standard BFS as number of edges increases.

	Number of vertices		
	10000	20000	50000
$uSTD(n-1)$	11.9	24.4	54.5
$uSTD(20n)$	70.4	140.6	350.7
$dSTD(n-1)$	1.2	2.5	127.4
$dSTD(20n)$	63.9	107.3	318.5
ROM1	87.4	172.3	423.2
ROM2	17.4	31.3	78.1
ROM3	18.3	32.8	81.4
ROM4	18.3	32.8	81.4
ROT1	1.22	2.44	6.1

Table 5.11 BFS space usage for synthetic graphs.

Tables 5.12 to 5.14 show result of time and space cost of directed BFS in the real life graphs. The result follows simliar tendency to the discussion in undirected BFS.

	Time(ms)	Space(KB)
STD	8.1	392.2
ROM1	15.6	653
ROM2	1796	109
ROM3	122	114
ROM4	122	114
ROT1	196	0.49
ROT2	512	0
IMP2	234	0

Table 5.12 Directed BFS result of `Slashdot`.

	Time(ms)	Space(KB)
STD	0.18	5.91
ROM1	0.21	9.99
ROM2	2.19	2.27
ROM3	2.25	2.43
ROM4	2.16	2.43
ROT1	0.37	0.12
ROT2	0.39	0
IMP1	0.13	0

Table 5.13 Directed BFS result of `EmailEU`.

	Time(ms)	Space(KB)
STD	110.3	2462.7
ROM1	205.6	6813.5
ROM2	3283.2	986.7
ROM3	1777.2	1019.9
ROM4	1924.8	1019.9
ROT1	1252	7.11
ROT2	906	0
IMP1	588	0

Table 5.14 Directed BFS result of `Flickr`.

5.2 DFS

In this section, *ROM1*, *ROM2* and *ROM3* denote Theorems 9, 10 and 11, respectively. Also, *ROT1*, *ROT2* and *ROT3* denote Theorems 12, 13 and 14, respectively. Lastly, *IMP1* and *IMP2* refer to Theorems 15 and 16, respectively.

Undirected DFS

Tables 5.15 to 5.17 show runtime result of undirected DFS for synthetic graphs. As number of edges increases, runtime of *ROT3* and *IMP1* increased dramatically. This is due to the original theoretical time bound, which contains m^2. Thus, those algorithms could not terminate when m gets significantly large. *ROM1* and *ROM2* occupy lesser space compared with standard DFS in small edges, but occupy more space as number of edges increases. Since space usage in *ROM3* is proportional to $\lg m$ instead of m, it always takes less space. Space used in *ROT1* and *ROT2* is proportional to number of vertices so that they take significantly lesser space. Other in-place model algorithms take only constant number of variables.

	9999		30000		50000		100000		200000		Complete	
	time	space	time	space	time	space	time	space	time	space	time	space
STD	0.893	79.4	0.937	58.5	3.5	66.9	3.3	73.7	4.3	76.4	865	79.4
ROM1	7.6	24.7	12.4	34.5	16.8	44.3	30.9	68.7	51.1	117	169717	24432
ROM2	1.9	8.5	3.8	18.3	5.5	28.1	11	52.5	13.2	101	138071	24415
ROM3	1.1	8.1	1.9	13	2	15.1	2.2	17.8	2.7	20.5	165	40
ROT1	5.5	17.3	8	17.4	10.6	17.4	17.6	17.4	31.7	17.4	6763	17.4
ROT2	0.492	1.2	0.703	1.2	0.794	1.2	0.963	1.2	1.4	1.2	139.9	1.2
ROT3	1	0	4468	0	8449	0	17530	0	-	-	-	-
IMP1	1.1	0	4079	0	7761	0	16396	0	-	-	-	-
IMP2	0.202	0	0.72	0	0.905	0	1.6	0	2.6	0	812	0

Table 5.15 Undirected DFS result with synthetic graph ($n = 10000$).

	19999		60000		100000		200000		400000		Complete	
	time	space	time	space	time	space	time	space	time	space	time	space
STD	1.1	158.7	3.2	115.9	3.6	133.5	4.4	146.1	8.7	152.6	3463	158
ROM1	16	46	25.2	65.5	36.1	85	67	133	116	231	1282247	97688
ROM2	3.8	17.1	7.7	36.7	11.5	56.2	23.3	104	32.8	202	1048344	97659
ROM3	2.3	16.2	4.1	35.7	3.9	30.1	4.6	35.5	6	41	643	85.5
ROT1	11.8	31.3	16.9	31.3	22.8	31.3	38.2	31	69.5	31.3	29336.5	31.3
ROT2	1.1	2.5	1.5	2.5	1.7	2.5	2.2	2.5	3.7	2.5	555	2.5
ROT3	2.8	0	20200	0	38489	0	76894	0	-	-	-	-
IMP1	2.4	0	18849	0	35624	0	73665	0	-	-	-	-
IMP2	0.43	0	1.5	0	2	0	3.3	0	7	0	4273	0

Table 5.16 Undirected DFS result with synthetic graph ($n = 20000$).

	49999		150000		250000		500000		1000000		Complete	
	time	space	time	space	time	space	time	space	time	space	time	space
STD	4.3	396.7	5.5	292	10.4	334.2	14.5	365	23.7	380.9	21283	396.7
ROM1	40	114	70.5	163	100	212	190	334	398	578	-	-
ROM2	10	42	22	91	35	140	81.3	262	190	506	-	-
ROM3	6.2	40.5	11.9	64.8	12.5	75.3	15.7	89	18.7	102	4013.5	225.9
ROT1	29.9	78.2	48.4	78.7	62.5	78.2	103	78.2	180	78.2	188432	78.2
ROT2	2.7	6.1	4.8	6.1	6.1	6.1	8.7	6.1	11.3	6.1	3737	6.1
ROT3	10.9	0	259966	0	622744	0	684010	0	-	-	-	-
IMP1	13.7	0	126081	0	252075	0	582581	0	-	-	-	-
IMP2	1.3	0	5.6	0	8	0	14.6	0	25.6	0	47185	0

Table 5.17 Undirected DFS result with synthetic graph ($n = 50000$).

Figure 5.2 shows relative time ratio of the undirected DFS algorithms on $n = 10000$. As in the BFS, we set the datum point as $m = n - 1$. Since growth of *ROT3* and *IMP1* increases to units of thousands, they could not be included into this figure. We can observe that *ROM3* has faster runtime compared to those for *ROM1* and *ROM2*. Since we are storing edge information on the parent for *ROM1* and *ROM2*, number of scans is relevant to degree of the visited neighbors. This gets reduced to logarithmic in *ROM3* by storing edge information on its child. Another observation we can make is time difference between *ROT1* and *ROT2*. This arises from the internal data structures, which are color array and `visited` bit vector. *ROT2* simply reads and writes on a plain bit vector during operations. On the other hand, *ROT1* uses Lemma 3, where read and write operations involve multiplication, division and modulus operations.

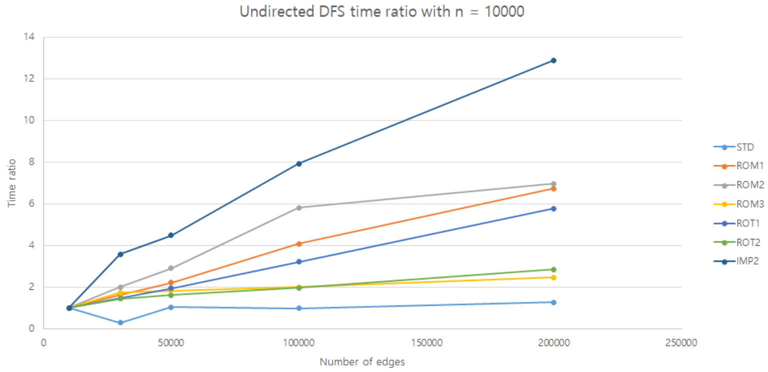

Figure 5.2 Time ratio of undirected DFS with synthetic graph ($n = 10000$.)

Figure 5.2 shows time spent ratio for undirected DFS when number of vertices increases while m is fixed to $10n$. In this plot, we can observe *ROT3* and *IMP1* have the highest time growth. Other than those algorithms, we observe steady slow growth of runtime.

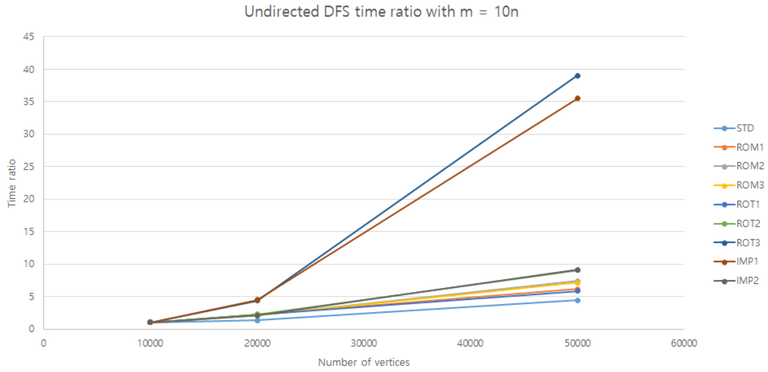

Figure 5.3 Time ratio of undirected DFS with synthetic graph ($m = 10n$).

Tables 5.18 to 5.20 show result of time and space cost of undirected DFS for real graphs. For runtime, *ROT2* and *IMP2* show best results, followed by standard DFS and *ROM3*. *ROT3* and *IMP3* did not terminate while running on `FacebookA`. For space usage, *ROM3* occupies lesser space than standard DFS. *ROT3*, *IMP1* and *IMP2* have the best space usage since they use only constant number of variables.

	Time(ms)	Space(KB)
STD	1.73	15
ROM1	22	540
ROM2	6.4	44.6
ROM3	1	7.4
ROT1	13.5	7.9
ROT2	0.496	0.49
ROT3	300	0
IMP1	2719	0
IMP2	0.754	0

Table 5.18 Undirected DFS result of `Facebook`.

	Time(ms)	Space(KB)
STD	2926	777.7
ROM1	38643.7	15769.2
ROM2	8016.4	12690.6
ROM3	1278.9	3239.5
ROT1	30229.24	3456.7
ROT2	867.1	378.1
ROT3	-	-
IMP1	-	-
IMP2	1017.6	0

Table 5.19 Undirected DFS result of `FacebookA`.

	Time(ms)	Space(KB)
STD	13.2	102.1
ROM1	148.7	210
ROM2	26.6	126.1
ROM3	10.8	61
ROT1	118.1	91
ROT2	5.31	7.1
ROT3	57151.6	0
IMP1	53167.6	0
IMP2	3.43	0

Table 5.20 Undirected DFS result of `Brightkite`.

Directed DFS

Tables 5.21 to 5.23 show runtime and space usage result of directed DFS for synthetic graphs. The results show arguably similar result to that of undirected DFS, except directed *ROT3* and *IMP1*. Those consume significantly more time than the undirected version. As mentioned in the discussion of directed BFS, directed graphs have lesser average degree, resulting larger depth. *ROM3* and *IMP1* have theoretical time bound containing l^2 that they are much slower on directed graphs.

	9999		30000		50000		100000		200000		Complete	
	time	space	time	space	time	space	time	space	time	space	time	space
STD	1.6	79.4	2.4	47.8	1.8	57	1.7	66.9	2.1	72.7	858	79.4
ROM1	5.1	22.3	8.3	27.2	10.7	32.1	17.2	44.3	32.6	68.7	169274	24431
ROM2	1	6.1	2.4	11	3.4	15.9	5.7	28.1	12.4	52.5	136324	24415
ROM3	0.622	6.1	1.7	10.1	2.1	12.3	2.2	15.125	2.7	17.8	165	40.3
ROT1	3.5	17.4	5.7	17.4	7.4	17.4	10.7	17.4	18.6	17.4	6817.6	17.4
ROT2	0.261	1.2	0.662	1.2	0.8	1.2	1	1.2	1.5	1.2	144	1.2
ROT3	497	0	168324	0	200196	0	160147	0	-	-	-	-
IMP1	188	0	144318	0	181223	0	151696	0	-	-	-	-
IMP2	0.053	0	0.793	0	0.938	0	1.5	0	2.5	0	971	0

Table 5.21 Directed DFS result with synthetic graph ($n = 10000$).

	19999		60000		100000		200000		400000		Complete	
	time	space	time	space	time	space	time	space	time	space	time	space
STD	2.2	158.7	3.2	94.1	2.8	112.8	2.9	133.7	6.7	146	3453.7	158
ROM1	10.8	41.1	19.8	50.8	23.4	60.6	38.4	85	71.4	133.8	1300978	97687
ROM2	2.1	12.2	5.4	22	7	31.8	12.8	56.2	27.4	105	1094831	97658
ROM3	1.4	12.2	3.5	20.1	4.2	24.5	5	30.2	7.2	35.6	656.8	85.5
ROT1	7.5	31.3	13.6	31.3	15.8	31.3	23.9	31.3	40.3	31.3	29399	31.3
ROT2	0.54	2.4	1.8	2.5	1.7	2.5	2.6	2.5	4.7	2.5	568	2.5
ROT3	2010	0	1545320	0	1928539	0	1285334	0	-	-	-	-
IMP1	641	0	1415216	0	1789548	0	1180314	0	-	-	-	-
IMP2	0.104	0	1.7	0	2.2	0	3.9	0	7.6	0	5024	0

Table 5.22 Directed DFS result with synthetic graph ($n = 20000$).

	49999		150000		250000		500000		1000000		Complete	
	time	space	time	space	time	space	time	space	time	space	time	space
STD	3.1	396.7	5.4	239.7	6	280	9.7	334	15.3	365.1	21285553	396.7
ROM1	27	102.6	50.9	127	68.7	151	113	212	203.2	334	-	-
ROM2	5.1	30.5	16.2	55	24.7	79.4	47.9	140	92.5	262	-	-
ROM3	3.2	30.5	11.8	50.1	15.7	61.2	20.8	75.4	24	89	4176	225
ROT1	18.6	78.1	37	78	46.3	78.2	68.5	78.2	109	78.2	188432	78.2
ROT2	1.4	6.1	5.7	6.1	8.6	6.1	13.3	6.1	17.3	6.1	3790	6.1
ROT3	12832	0	26689962	0	-	-	-	-	-	-	-	-
IMP1	4061150	0	25342641	0	-	-	-	-	-	-	-	-
IMP2	0.331	0	7.9		11.4	0	18.7	0	28.6	0	53739	0

Table 5.23 Directed DFS result with synthetic graph ($n = 50000$).

Tables 5.24 to 5.26 show the result of time and space cost of directed DFS in the real life data sets. The result resembles the result we have discussed in undirected DFS.

	Time(ms)	Space(KB)
STD	40	130640
ROM1	431	332
ROM2	73.7	232
ROM3	26.6	83.7
ROT1	360.7	109
ROT2	18.1	17.6
ROT3	3577231	0
IMP1	3301660	0
IMP2	16198	0

Table 5.24 Directed DFS result of `Slashdot`.

	Time(ms)	Space(KB)
STD	0.33	4.34
ROM1	4.27	8.63
ROM2	1.51	6.49
ROM3	0.23	1.62
ROT1	2.43	2.27
ROT2	0.11	0.13
ROT3	50.1	0
IMP1	47.1	0
IMP2	0.168	0

Table 5.25 Directed DFS result of `EmailEU`.

	Time(ms)	Space(KB)
STD	210135.8	777.7
ROM1	2143571	3655.3
ROM2	43497.9	2768.8
ROM3	33915.2	751.9
ROT1	2180105	986.7
ROT2	27528.1	100.2
ROT3	-	-
IMP1	-	-
IMP2	215.1	0

Table 5.26 Directed DFS result of `Flickr`.

Chapter 6

Conclusion

Since study of space efficient graph algorithms has been initiated, numerous approaches have tackled the problem to break space bound barrier. In this thesis, we have summarized various space efficient graph algorithms based on ROM model and in-place model, and evaluated their performance on both synthetic and real graphs. In our knowledge, this thesis is the first to implement various space efficient graph algorithms suggested by Banerjee et al. [3] and Chakraborty et al. [7].

As discussed in Chapter 5, we showed that our implementations match the expected theoretic bound of time and space, both for undirected and directed graphs.

Comparing runtime with standard BFS, ROM model and in-place model algorithms give slower runtime. However, if number of edges increases as many as a graph becomes complete, Theorems 5 (BFS *ROT1*) and 6 (BFS *ROT2*) perform the fastest. When compared for space efficiency of the BFS algorithms, the ROM model algorithms do not depend on number of edges but only on number of vertices, whereas standard BFS may increase space usage proportional to number of edges. In Chapter 5, we observed Theorem 1 (BFS *ROM1*) used more space than standard BFS, but for case when edge density becomes larger, we observe BFS *ROM1* has lesser space usage than standard BFS. For other ROM model algorithms, standard BFS has better space usage only when $m = n - 1$, which is the most optimal situation for standard BFS. Other than this case, those ROM model algorithms occupy much smaller space. For in-place model, BFS *ROT1* occupies n bits regardless of number of edges that it shows much better space efficiency compared with the standard BFS and ROM model algorithms. Other in-place model algorithms use only constant number of variables that they have the best space usage.

For DFS, Theorem 11 (DFS *ROM3*) has superior runtime and space efficiency compared with standard DFS, and runtime and space gaps between the two algorithms increase as the number of edges increases. However, standard DFS quickly outperforms other ROM model algorithms than *ROM3* when number of edges increases. Although most in-place model DFS algorithms have poor time performance compared to that of standard DFS, Theorem 16 (DFS *IMP2*) gives similar runtime performance while using only constant number of variables. Moreover, Theorem 13 (DFS *ROT2*) even dominates runtime versus all other DFS algorithms while maintaining only a bit vector `visited`.

Studies of space efficient graph algorithms continue as study of model is actively ongoing, such as restore model and catalytic-space model. Chakraborty et al. [7] pointed some future directions of in-place model, such as improving the running time further, defining in-place model algorithms for adjacency matrix representation and implementing parallelism of the algorithms. As a future work, implementing and evaluating performance of algorithms in various models will explorer future researches.

Bibliography

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and system sciences*, 58(1):137–147, 1999.

[2] T. Asano, T. Izumi, M. Kiyomi, M. Konagaya, H. Ono, Y. Otachi, P. Schweitzer, J. Tarui, and R. Uehara. Depth-First Search Using O(n) bits. In *International Symposium on Algorithms and Computation*, pages 553–564. Springer, 2014.

[3] N. Banerjee, S. Chakraborty, V. Raman, and S. R. Satti. Space efficient linear time algorithms for bfs, dfs and applications. *Theory of Computing Systems*, pages 1–27, 2018.

[4] H. Buhrman, R. Cleve, M. Kouckỳ, B. Loff, and F. Speelman. Computing with a full memory: catalytic space. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 857–866. ACM, 2014.

[5] H. Buhrman, M. Kouckỳ, B. Loff, and F. Speelman. Catalytic space: nondeterminism and hierarchy. *Theory of Computing Systems*, 62(1):116–135, 2018.

[6] S. Chakraborty, A. Mukherjee, V. Raman, and S. R. Satti. Frameworks for designing in-place graph algorithms. *arXiv preprint arXiv:1711.09859*, 2017.

[7] S. Chakraborty, A. Mukherjee, V. Raman, and S. R. Satti. A Framework for In-place Graph Algorithms. In Y. Azar, H. Bast, and G. Herman, editors, *26th Annual European Symposium on Algorithms (ESA 2018)*, volume 112 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 13:1–13:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[8] T. M. Chan, J. I. Munro, and V. Raman. Selection and sorting in the restore model. In *Proceedings of the twenty-fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 995–1004. Society for Industrial and Applied Mathematics, 2014.

[9] D. Clark. Compact pat trees. 1997.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms.* MIT press, 2009.

[11] C. Demetrescu, I. Finocchi, and A. Ribichini. Trading off space for passes in graph streaming problems. *ACM Trans. Algorithms*, 6(1):6:1–6:17, Dec. 2009.

[12] Y. Dodis, M. Patrascu, and M. Thorup. Changing base without losing space. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 593–602. ACM, 2010.

[13] A. Elmasry, T. Hagerup, and F. Kammer. Space-efficient basic graph algorithms. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 30. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[14] S. Even and G. Even. *Graph Algorithms, Second Edition.* Cambridge University Press, 2012.

[15] J. Feigenbaum, S. Kannan, A. McGregor, S. Suri, and J. Zhang. On graph problems in a semi-streaming model. *Theoretical Computer Science*, 348(2-3):207–216, 2005.

[16] G. N. Frederickson. Upper bounds for time-space trade-offs in sorting and selection. *Journal of Computer and System Sciences*, 34(1):19–26, 1987.

[17] S. Gog, T. Beller, A. Moffat, and M. Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[18] A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. A framework for dynamizing succinct data structures. In *International Colloquium on Automata, Languages, and Programming*, pages 521–532. Springer, 2007.

[19] T. Hagerup and F. Kammer. Succinct choice dictionaries. *arXiv preprint arXiv:1604.06058*, 2016.

[20] W.-K. Hon, K. Sadakane, and W.-K. Sung. Succinct data structures for searchable partial sums with optimal worst-case performance. *Theoretical Computer Science*, 412(39):5176–5186, 2011.

[21] F. Kammer and A. Sajenko. Linear-time in-place DFS and BFS in the restore model. *CoRR*, abs/1803.04282, 2018.

[22] A. Labrinidis and H. V. Jagadish. Challenges and opportunities with big data. *Proceedings of the VLDB Endowment*, 5(12):2032–2033, 2012.

[23] J. Leskovec and A. Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, June 2014.

[24] J. Mervis. Agencies rally to tackle big data, 2012.

[25] C. W. Mortensen, R. Pagh, and M. Pătraçcu. On dynamic range reporting in one dimension. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 104–111. ACM, 2005.

[26] J. I. Munro. Tables. In *International Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 37–42. Springer, 1996.

[27] J. I. Munro and M. S. Paterson. Selection and sorting with limited storage. In *Foundations of Computer Science, 1978., 19th Annual Symposium on*, pages 253–258. IEEE, 1978.

[28] R. Raman, V. Raman, and S. S. Rao. Succinct dynamic data structures. In *Workshop on Algorithms and Data Structures*, pages 426–437. Springer, 2001.

[29] R. A. Rossi and N. K. Ahmed. The network data repository with interactive graph analytics and visualization. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[30] D. E. Willard. Log-logarithmic worst-case range queries are possible in space θ (n). *Information Processing Letters*, 17(2):81–84, 1983.

[31] X. Wu, X. Zhu, G.-Q. Wu, and W. Ding. Data mining with big data. *IEEE transactions on knowledge and data engineering*, 26(1):97–107, 2014.

요약

소셜 네트워크나 빅 데이터로부터 생성된 다양한 그래프들은 방대한 양의 데이터를 포함하고 있다. 이러한 그래프를 탐색하기 위해서는 그래프의 크기에 비례하여 필요한 메모리의 용량이 늘어난다. Asano 등(ISAAC (2014))은 공간 효율적 그래프 알고리즘 연구를 개시했다. 이 연구를 통해 선형적 시간보다 약간 더 걸리는 대신 저선형적 공간을 사용하는 DFS 알고리즘과 활용 방안들이 제안됐다. Banerjee 등 (ToCS **62**(8), 1736-1762 (2018))은 ROM 모델을 기반으로 하는 공간 효율적인 그래프 알고리즘들을 제안했다. 그래프 G의 n개의 정점과 m개의 간선이 주어졌을 때, $O(m+n)$의 시간과 $2n+o(n)$ 의 비트를 사용하는 BFS가 제안됐고, $f(n)$을 n에 비례해서 매우 느리게 커지는 함수라고 했을 때, $O(m \lg n f(n))$의 시간과 $n \lg 3 + o(n)$의 비트를 사용하는 알고리즘이 제안됐다. DFS의 경우, $O(m+n)$의 시간과 $O(m \lg \frac{m}{n})$의 비트를 사용하는 알고리즘이 제안됐다. Chakraborty 등(ESA (2018))은 ROM 모델이 가지고 있는 한계점을 넘기 위해 ROM 모델의 제한점을 완화시키는 in-place 모델을 소개했다. In-place 모델을 기반으로 한 알고리즘들은 $n + O(\lg n)$의 비트를 사용하여 BFS와 DFS를 수행할 수 있고, 추가적으로 더 긴 시간을 소요하여 $O(\lg n)$ 비트의 공간만으로 알고리즘을 수행할 수 있다. 이 논문에서는 ROM 모델과 in-place 모델에서 제안된 다양한 알고리즘들을 연구 및 구현하고 실험을 통하여 이들 알고리즘의 수행 결과를 평가한다.

주요어: DFS, BFS, 공간 효율적인 그래프 알고리즘

학번: 2015-22905

Acknowledgements

Firstly, I express my sincere gratitude to my advisor, professor Srinivasa Rao Satti. I managed to progress my Master's degree with his continues support, deep patients and guidance.

I would also give thanks to the other thesis defense committee, professor Sun Kim and professor Berhard Egger, for their effort to give advice.

I give my thank to members in Computer Theory and Algorithm Engineering Lab: Wonil Jeong, Wooyoung Park, Seyoung Kim, Seungeun Lee, Seungwoo Kim and MohammadSadegh Najafi. I could not ask for any better colleague than you. I also give my thanks to our lab alumni members: Seungbum Jo, Jeongsoo Shin, Edman Paes dos Anjos and Hagos Alema. It was their legacy that guided me. Among all the colleagues, I would like to give special thanks to Junhee Lee. Without his immeasurable support, I would never have finished my Master.

My sincere thanks also goes to professor Sam Chung. Without his advice, I would have gave up Computer Science. He fixed my depressive character, told me to look up and forced me to have a dream. He is my mentor for both life and religious.

Finally, I would like to give my thanks to my family who showed endless support. My sister, Nari Yoo, gave me countless advice and shared 'home' where I can have at least one family member. Last but not least, my father, Uksang

Yoo, and my mother, Poongja Yoon, sacrificed everything for my sister and me.