Summer Research Report 1:

Inclusion-Exclusion Algorithm for Hamiltonian Path Problem With Parallelized Approach

Ziyue Wang
Project Supervisor: Dr. Ben Li

Computer Science
University of Manitoba
September 2017

Contents

1 Introduction

This report is for the summary of first half of the research project under Dr. Ben Lis supervision in summer 2017. In this report, we will use inclusion-exclusion technique to solve hamiltonian path problem. The algorithm is originally from the book: Exact Exponential Algorithms, by Fedor V. Fomin and Dieter Kratsch [1]. In section 4 of the book [1], they introduce an inclusion-exclusion algorithm for hamiltonian path problem. We will start with this algorithm and make some changes to the algorithm. We will introduce the hamiltonian path problem as well as the algorithm, improve the algorithm, and convert the algorithm to a parallel version.

1.1 Inclusion-Exclusion Algorithm

Inclusion-exclusion algorithms are used to solve complex counting problems. By using some techniques, we could also use the algorithms to solve problems more than count problem. In this report, we will use the reduction rule introduced in section 4 of [1] to find the hamiltonian path. The original idea of inclusion-exclusion algorithm is from mathematical formula.

$$|A \cup B| = |A| + |B| - |A \cap B| \tag{1}$$

This formula could be extended to solve more complex problems (i.e. sets have more than 3 items)

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C| \tag{2}$$

Furthermore this could be extended to be a formula that applies to sets has any size that greater than 2.

"**Theorem 4.2**. *Given a collection of N combinatorial objects and properties $Q(1), Q(2), ..., Q(n)$ such that each of the objects has a subset of those properties. For any subset $W \subseteq \{1, 2, ..., n\}$, let $N(W)$ be the number of objects having none of the properties $Q(w)$ with $w \in W$ (but possibly some of the others). Let X be the number of objects having all properties $Q(1), Q(2), ..., Q(n)$. Then*

$$X = \sum_{W \subseteq \{1,2,...,n\}} (-1)^{|W|} N(W) \tag{3}$$

" [1]

Therefore we could apply this formula to solve counting problems.

1.2 Hamiltonian Path

Hamiltonian path problem is a graph problem and is considered to be NP-complete complexity. The problem is to determine whether there is a path from one point to another and visit all points exactly once in a graph. An extension of the problem is to find the path if it exists. There are multiple algorithms can solve the problem in $O(2^n)$ time, and the inclusion-exclusion is one of them.

2 Inclusion-Exclusion Algorithm for Hamiltonian Path

2.1 Introduction

In this section we will briefly introduce the algorithm from the book [1], all details could be found in the book [1]. Furthermore, we will introduce a parallel approach of the algorithm.

2.2 Algorithm Details

[1] section 4.2.2 introduced an inclusion-exclusion algorithm to solve hamiltonian path problem in $O(2^n)$ time. The problem asks whether there is a path of length $n + 1$ from vertex s to vertex t and visits all vertices exactly once. By using inclusion-exclusion algorithm, we count the number of walks from s to t of length $n+1$ without considering how many vertices have the walks passed for all possible subsets of the graph and apply the formula 3 to get the final result.

$$X = \sum_{W \subseteq \{1,2,\ldots,n\}} (-1)^{|W|} N(W)$$

W would be all possible subsets of the graph, $N(W)$ is the number of paths from s to t of length $n + 1$ without visiting any vertex in W. The final result X is the number of hamiltonian paths. To generate all subsets of a set, we use binary technique. We do a FOR loop from 0 to $n - 1$, for each iteration i, we convert i into binary. If the tth digit of i is 1, we add tth edge to the set, otherwise, do nothing. There are 2^n subsets therefore the dominative complexity is $O(2^n)$.

The book [1] introduces 2 methods to count $N(W)$, one is using the adjacency matrix, the other is using dynamic programming. In our case, we are going to use the adjacency matrix since it requires less space. In one sentence, the kth power of an adjacency matrix A on the intersection of column s and row t, is the number of walks from s to t of length $k + 1$. Therefore we could easily calculate $N(W)$ by matrix multiplications. Instead of using the usual way to calculate the power of matrix, there is a faster technique called Exponentiation by Squaring. Rather than directly multiply the current matrix with the original matrix one by one, we take multiply the current matrix with itself, therefore we could get a higher exponent in each multiplication, and in each multiplication, there will be more extra calculations. As the result of this, we could calculate the matrix multiplication faster, and it could be done in polynomial time and the complexity of the entire algorithm still $\in O(2^n)$.

The book [1] also introduces a self-reduction method that could find the hamiltonian path based on previous algorithm. The basic idea is for each edge e in the graph g, we remove e from the g, if the number of hamiltonian paths is decreased to be 0, we add e back to the graph since there is a hamiltonian path goes through e. However, we are not going to use this technique since there is a faster way to do this which is also introduced in [1]. Instead of handle with each e, we look at each vertex v. We start from s, try to remove each edge e_s from s. If there number of hamiltonian paths is decreased, we can conclude that there is at least one hamiltonian path goes through e_s, and we remove all other edges from s, and recursively do same thing for the other end of e_s. At the end, the remaining path is the hamiltonian path we are looking for.

2.3 Pseudocode

This section provides pseudo codes for the algorithms we talked in previous section.

Algorithm 1: Find the Number of Hamiltonian Path $(NumPath(s, e, n, g))$

```
Input: Start Vertex s, Terminate Vertex e, Length of the Path n, Input
       Graph g
Output: The Number of Hamiltonian Path paths from s to e of length n
        in input graph g
int paths = 0;
foreach Vertex Subset w of g do
    Compute the adjacency matrix A of w;
    $\mathtt{w} = \mathtt{w}^{\mathtt{n}}$;
    $\mathtt{paths}+ = (-1)^{|\mathtt{w}|} * \mathtt{w[s][e]}$;
end
```

Algorithm 2: Find the Actual Hamiltonian Path

```
Input: Start Vertex s, Terminate Vertex e, Length of the Path n, Input
       Graph g
Output: The Hamiltonian Path from s to e of length n in input graph g
if $NumPath(s, e, n, g) > 0$ then
    remove one edge $d$ from $g$;
    if $NumPath(s, e, n, g)$ == 0 then
        Add $d$ back to the graph;
        Add $d$ to the solution set;
    end
end
```

Algorithm 3: The Faster Algorithm For Finding the Actual Hamiltonian Path

```
Input: Start Vertex s, Terminate Vertex e, Length of the Path n, Input
       Graph g
Output: The Hamiltonian Path from s to e of length n in input graph g
int count1 = $NumPath(s, e, n, g)$;
while $n > 0$ do
    remove one edge $d$ start from $s$ from $g$;
    int count2 = $NumPath(s, e, n, g)$;
    if $count1 > count2$ then
        Add $d$ to the solution set;
        Remove all edges start from $s$;
        $s$ = the end vertex of $d$;
        n − −;
        count1 = $NumPath(s, e, n, g)$;
    end
end
```

3 Enhanced Hamiltonian Path Algorithm

The original algorithm is well designed and has acceptable performance, nevertheless, there is some way to improve the performance by using more space. In this section, we are going to introduce a faster hamiltonian path algorithm with tradeoff of space usage.

3.1 Introduction

In the previous section, we have introduced the algorithm from [1]. The original algorithm has very low space requirement, however, if we have very large memory and disk, we could improve the performance by using more space. Recall from the original algorithm, we generate all possible subsets of W. There are 2^n possible subsets thus we have to calculate $N(W) * 2^n$ times. Each time we calculate $N(W)$ costs a lot of matrix multiplications, which is quite expensive. We want to reduce the number of subsets have to be computed, therefore we could reduce the number of matrix multiplications and achieve better performance.

The idea of reduce the number of subsets is, if there is no hamiltonian path from s to t of length k existing in subset $u(i.e.N(u) = 0)$, any superset of u

contains no hamiltonian path from s to t of length k. In other words, in a graph g, if we remove one vertex from g, the number of paths will never be increased since the possibility of g is decreased. From this observation, we can conclude that if we found a subset u such that $N(u) = 0$, we dont need to compute any superset of u since all of them are 0. To make the best use of this observation, we have to generate the subset sequence by number of vertices in the graph. One technique could be used here is the combination algorithm. Start with generate combinations of W, from number of elements 0 to n. This could help us to generate subset sequence from sub to super.

3.1.1 Enhanced With All Supersets Hash Table (ASHT)

The first approach is to store all u such that $N(u) = 0$ in a hash table h. A hash table offer constant expected time insertion and constant worst time access. Each time before we actually compute the $N(W)$, we search the current subset u in the hash table h. If u does not exist in h, we compute $N(u)$, otherwise, $N(u) = 0$. If after compute $N(u)$, we get $N(u) = 0$, we add all possible supersets of u into h. This is the simplest way to reduce the number of subsets have to be computed, however, it requires exponential space to hold the entire hash table h. Usually the hash table is not able to be fit in memory, thus we have to create a database of complexity $O(2^n)$ to store the hash table.

In terms of complexity, in each iteration, we reduce the number of W to be computed, and add a new factor 2^n which is the insertions of hash table. Assume there are a subsets such that $N(u) = 0$. We have $2^n * a + (2^n - a) * N(W)$, which is still $\in O(2^n)$ in worst case of speed. However, with larger a, we could achieve better performance. In terms of storage, in worst case we have to storage almost all subsets in the hash table, therefore the complexity of storage will be $O(2^n)$ as well.

The tradeoff we have is speed $vs.$ space. The bigger a is, the more speedup we could achieve. Vice versa. One drawback of the algorithm is that we may need to set the flags duplicately since the current subset might be the superset of multiply sets.

The pseudocode is provided blow:

Algorithm 4: Find the Number of Hamiltonian Path $(NumPath(s, e, n, g))$

```
Input: Start Vertex s, Terminate Vertex e, Length of the Path n, Input
       Graph g
Output: The Number of Hamiltonian Path paths from s to e of length n
        in input graph g
int paths = 0;
hash table h.init{};
foreach Vertex Subset w of g do
    if w is NOT in h then
        Compute the adjacency matrix A of w;
        $w = w^n$;
        if $w[s][e] == 0$ then
            Add w and all its supersets to h;
        else
            $paths += (-1)^{|w|} * w[s][e]$;
        end
    end
end
```

3.1.2 Enhanced With Minimum Subsets Hash Table (MSHT)

To store all supersets requires too much space than we have. It is inefficient in terms of memory and disk usage. An alternative way to do this is only store the minimum subsets. In each iteration, we check whether the current subset is the superset of any set in hash table h. If it is, $N(u) = 0$, otherwise, we compute $N(u)$. If after compute $N(u)$, we found $N(u) = 0$, then we store u into h. In worst case we have hash table size of $O(nchoosen/2)$ in worst case. In terms of time complexity, we still have $2^n - a$ subsets have to be computed. On the other hand, in each iteration, we have to check the entire hash table to see whether the current subset is a superset of any set in the hash table. We assume that by applying a certain hash rule, we could still achieve the complexity of $O(2^n)$.

The pseudocode is provided blow:

Algorithm 5: Find the Number of Hamiltonian Path $(NumPath(s, e, n, g))$

```
Input: Start Vertex s, Terminate Vertex e, Length of the Path n, Input
       Graph g
Output: The Number of Hamiltonian Path paths from s to e of length n
        in input graph g
int paths = 0;
int i = 0;
hash table h.init{};
for i ≤ number of vertices in g do
    i++;
    foreach Vertex Subset w which |w| = i from g do
        if w is NOT in h AND w is NOT a superset of any set in h then
            Compute the adjacency matrix A of w;
            $w = w^n$;
            if $w[s][e] == 0$ then
                Add w to h;
            else
                $paths += (-1)^{|w|} * w[s][e]$;
            end
        end
    end
end
```

3.2 Enhanced Performance Evaluations

Since the only difference between all three algorithm are the way they counting the number of hamiltonian paths, thus we only test counting the number of hamiltonian paths. Compared with the original algorithm, the actual performance of both ASHT and MSHT are highly depend on the input graph. Although all algorithms have complexity of $O(2^n)$, the actual performance will be different. The number of sets u such that $N(u) = 0, a$, plays an important role in the algorithm. The actual number of reduced matrix multiplication is depend on a. The original algorithm requires constant memory, ASHT requires $O(2^n)$ space, and MSHT requires $O(nchoosen/2)$ space.

3.2.1 Benchmark Sets

The machine we are using is Linux machines at University of Manitoba (hawk.cs.umanitoba.ca). The machines are equipped with 4X Intel(R) Core(TM) i5-4590 CPU @ 3.30GHz and 16GB Memory. The datasets we are testing have various of numbers of vertices and edges. We will test the performance of all three algorithms on those various datasets.

3.2.2 Result Analyze

number of vertices	number of edges	original algorithm	ASHT	MSHT	number of cycles
10	15	1.0X	1.0X	2.0X	0
10	15	1.0X	1.0X	2.0X	8
12	19	1.0X	1.15X	5.0X	1
13	14	1.0X	6.33X	57X	1
15	30	1.0X	0.8X	2.7X	320

Based on the result, we can see the performances are various depend on the graph. MSHT always offer better performance than the original algorithm, however, ASHT does not always better, sometimes even worse, thus we could conclude that ASHT is not a good way to go. The reason behind this might be the memory cost and the number of subsets has been pruned. The smaller edge/vertex ratio we have, the better performance we could achieve by using ASHT and MSHT. We can also predict that the less number of edges, the less paths are there, the more subsets we could prune, the better performance we could achieve. For some instance with large number of paths existing in the graph, we usually could not gain much performance by using hash tables. In this report we do not have enough time and datasets to test large data size. If we could test on more large dataset that will be helpful.

4 Parallelized Inclusion-Exclusion Algorithm for Hamiltonian Path

In this section, we will introduce parallelized implementations for original algorithm and MSHT we talked above. Although the parallelized algorithm is not the focus of this summer project, we would like to include this section as a side note.

4.1 Introduction

Since the loops are dominating the main part of the algorithm, therefore parallelize the matrix multiplication does not help a lot, thus we mainly parallelize the loops. Huge independent loops and matrix multiplications can be easily speed-upped by parallelized implementation.

4.2 Parallel Approach For Original Algorithm

In the original algorithm, we can see there is a giant loop that sums the result of all 2^n subsets. In parallelized approach, we could parallel the giant loop and gain performance from here. In particular, we will use OpenMP to parallelize the algorithm. The algorithm itself is simple, we add a parallel-for before the for loop begins, and reduction sum the path count, the only critical section will be the path count.

4.3 Parallel Approach For Enhanced With Minimum Subsets (MSHT)

On the other hand, MSHT generates subsets in a different way. It generates subsets by combinations on-fly. The loops are not independent anymore since we have hash table to check whether the current subset has 0 path. The good approach could be have a master thread that generate subsets, and dispatch subsets to each thread. At the end, take all results and do a reduction. In this approach, the critical section will be the hash table. However, most threads are only going to read the hash table but not modify it, thus we could use a smart implementation to solve the problem. A set a could not be a superset of a set b such that a and b has same size, a does not equal to b. Therefore we could let only master thread add sets to the hash table and all other threads could become independent. An alternative way to do this is let master thread to check the hash table. Whenever the master thread generates a new subset, check whether the subset is a superset of any element in the hash table. If it isnt, then we could dispatch the subset to other threads. Unfortunately, we do not have enough time to implement the parallelized algorithms, I will include them in the further once I get the opportunity.

5 Conclusion And Future Works

In conclusion, we implement the inclusion-exclusion algorithm for hamiltonian path from [1], section 4, and try to improve the performance of the algorithm by using more space. Although the performance of the improved algorithm is highly depend on the input graph, and the worst case complexity are same, we could still get better performance. We introduced two algorithms, ASHT and MSHT, to improve the performance. The ASHT uses exponential space while does not always offer better performance due to the memory latency, therefore we conclude that ASHT is not a good approach. The MSHT always offer better performance under our tests and uses less space. However the tests are not good enough for get the conclusion, and there are still some possible ways to improve the algorithm. First of all, if we could find a better hash rule or hash structure of the hash table, we may reduce the access and modify time of the hash table. Secondly, we think the way we generate the subset (find all combinations) is not the best way to generate subsets by the number of elements in the set. There might be some data structure could solve the problem and we could find all subsets by the number of elements efficiently. Finally, the tests are not enough. Most test graphs are hard coded by ourselves, and they are too small. We need more various graphs to test. As a side note, the hamiltonian path problem is a NP-complete problem, large graph will require a lot of time to get the result. Once we get the opportunity, these three points could be our next steps. Furthermore, both algorithms are highly parallelizable, we introduced the parallel approaches for the algorithm. Unfortunately, we do not have time to complete the implementation of the parallel version, we will complete this part in the future and compare the performance of the sequential algorithms and the parallel algorithms.

6 Acknowledges

With the great appreciate to Dr. Ben Li and Dr. Parimala Thulasiraman, they give us a lot of valuable advices and we could not achieve such result without their helpfulness. Dr. Ben Li is the supervisor and the founder of the project, he comes with the idea of the project and helped us a lot with developing the algorithm, without him the project will not exist. Dr. Parimala Thulasiraman gives many advices of the parallel approach and the article structure, she helped us to complete the report and gives many useful ideas to develop the parallel algorithm. We really appreciate to Dr. Ben Li and Dr. Parimala Thulasiraman, they are indispensable

in this project.

References

[1] Fedor V Fomin and Dieter Kratsch. *Exact exponential algorithms*. Springer Science & Business Media, 2010.