UE17CS311 - Advanced Algorithms Project Performance comparison of Dijkstra's Algorithm using different Heaps

1st Mehul Thakral
PES1201701122
B-Tech, CSE
PES University
Bangalore, India
mehul.thakral@gmail.com

2nd G R Dheemanth
PES1201700229
B-Tech, CSE
PES University
Bangalore, India
dheemanthgr@gmail.com

***Abstract*—Shortest path algorithms are used in several real-life applications like geographical Maps, IP routing to find Open shortest Path First, telephone network ,etc. to find single source shortest path to all the destinations in the network. Dijkstra's algorithm which is one such algorithm is implemented for both directed and undirected graph and compared based on the heap employed namely fibonacci, binomial and binary to find the one giving the best performance in terms of space and time.**

***Index Terms*—dijkstra's algorithm, fibonacci heap, binary heap, binomial heap, shortest path finding algorithms**

I. Introduction

Dijkstra's algorithm which is even used in most modern map systems like google maps takes $O(V^2)$ where V indicates number of vertices in the graph, when implemented using adjacency matrix which can be brought down to $O((V + E)log(V))$ where V indicates number of edges in the graph, when implemented using adjacency list and typical heap like binary heap and can be further reduced to $O(V * log(V) + E)$ when implemented using fibonacci heap. This substantial decrease is useful and can be harnessed in applications.

A. Input

Input is taken as list of neighbors of each vertex with the weight of the edge between them.

B. Dataset

Dijkstra's algorithm was run on 2 datasets:

1) Trivial dataset: It consisted of 9 nodes and 14 edges and results are summarized in table II.

2) Real-world dataset: It is the dataset used by 9th DIMACS Implementation Challenge - Shortest Paths [1]. We used New York city's road network data which consisted of 264346 nodes and 733846 edges and was run on binary heap implementation of dijkstra's algorithm and results are summarized in table III.

II. Methodology

First the graph is taken as input in main.cpp and is stored in the form of an adjacency list. Then the graph along with the source vertex taken from user is fed into dijkstra's algorithm which computes the shortest distance of all the vertices from the source using a particular type of heap which is suffixed in the function name and returns the distances. Execution of the algorithm is timed and can later be used to compare the efficiency of the heaps.

III. Algorithms employed

A. Dijkstra's shortest path algorithm

Steps followed are given in Figure 1. Here set is implemented using selected heap.

```
1) Initialize distances of all vertices as infinite.

2) Create an empty set.  Every item of set is a pair
  (weight, vertex). Weight (or distance) is used used
  as first item  of pair as first item is by default
  used to compare two pairs.

3) Insert source vertex into the set and make its
   distance as 0.

4) While Set doesn't become empty, do following
    a) Extract minimum distance vertex from Set.
       Let the extracted vertex be u.
    b) Loop through all adjacent of u and do
       following for every vertex v.

          // If there is a shorter path to v
          // through u.
          If dist[v] > dist[u] + weight(u, v)

              (i) Update distance of v, i.e., do
                    dist[v] = dist[u] + weight(u, v)
              (i) If v is in set, update its distance
                  in set by removing it first, then
                  inserting with new distance
              (ii) If v is not in set, then insert
                   it in set with new distance

5) Print distance array dist[] to print all shortest
   paths.
```

Fig. 1. Dijkstra's shortest path algorithm

B. Fibonacci heap

Operations used are:

- Insert : Shown in figure 2.

1. Create a new node 'x'.
2. Check whether heap H is empty or not.
3. If H is empty then:
   - Make x as the only node in the root list.
   - Set H(min) pointer to x.
4. Else:
   - Insert x into root list and update H(min).

Fig. 2. Fibonacci heap insert

- Erase : Shown in figure 3.

1. Decrease the value of the node to be deleted 'x' to minimum by Decrease_key() function.
2. By using min heap property, heapify the heap containing 'x', bringing 'x' to the root list.
3. Apply Extract_min() algorithm to the Fibonacci heap.

Fig. 3. Fibonacci heap erase

C. Binomial heap

Operations used are:

- Union: This operation takes two different binomial heaps as it's input and merges them based on the algorithm in figure 4

BINOMIAL-HEAP-UNION (H1, H2)

Merge the root lists of binomial heaps H1 and H2 into a single linked linked list H that is sorted by degree into monotonically increasing order.

Links roots of equal degree until at most one root remains of each degree.

Binomial-Link (y,z)

p[y] = z
sibling[y] = child[z]
child[z] = y
degree [z] = degree[z] + 1

Fig. 4. Union of binomial heaps

- Insert: This operation first creates a Binomial Heap with single key and then we call Union function on the existing heap and the new Binomial heap.
- Delete: This operation removes a particular node and create a new Binomial Heap by connecting all subtrees of the removed node and then call the Union on the newly create binomial tree and the existing binomial tree

D. Binary heap

Operations used are:

- Heapify: This operation takes an array as input and then converts it into a binary heap using the algorithm in figure 5

```
Max-Heapify (A, i):
    left ← 2xi     // ← means "assignment"
    right ← 2xi + 1
    largest ← i

    if left ≤ heap_length[A] and A[left] > A[largest] then:
        largest ← left

    if right ≤ heap_length[A] and A[right] > A[largest] then:
        largest ← right

    if largest ≠ i then:
        swap A[i] and A[largest]
        Max-Heapify(A, largest)
```

Fig. 5. Binary heap formation

- Insert: This operation adds a new key at the end of the heap and then calls the Heapify function to restore the violated heap property.
- Delete: This operation swaps the node to be deleted with the last node and then deletes the last node. After which we call the Heapify function to restore the violated heap property.

IV. Results

Theoretic amortized time complexities of heaps' operations are shown in the table.

Operations	Fibonacci	Binomial	Binary
Insert	(1)	(1)	$(Logn)$
Erase	$O(Logn)$	$(Logn)$	$(Logn)$

TABLE I
TIME COMPLEXITIES OF HEAPS' OPERATIONS

Time complexities obtained are summarized in the table.

Operations	Fibonacci	Binomial	Binary
Insert	0.00118	0.0046	0.0009
Erase	0.00227	0.00409	0.0008
Overall	0.126	0.145	0.055

TABLE II
TIME COMPLEXITIES OBTAINED IN MILLI-SECONDS.

Time complexities obtained on New York city's road network using binary heap are:

Operations	Insert	Erase	Overall
Binary	0.000637991	0.00122643	1006.82

TABLE III
TIME COMPLEXITIES OBTAINED IN SECONDS ON NEW YORK CITY'S ROAD NETWORK.

V. Achievements

We are able to show that for a small input graph dijkstra's algorithm has provided performance boost when implemented using binary heap as compared to binomial and fibonacci heaps.

References

[1] 9th DIMACS Implementation Challenge: Shortest Paths, users.diag.uniroma1.it/challenge9/download.shtml.