CS512 Fibonacci Heap - Fall 2019

Aditya Larka
Rutgers University
Piscataway, NJ, USA
al1247@scarletmail.rutgers.edu

Dhruv Mundhra
Rutgers University
Piscataway, NJ, USA
dm1396@scarletmail.rutgers.edu

Sai Krishna Chaitanya
Rutgers University
Piscataway, NJ, USA
sn650@scarletmail.rutgers.edu

***Abstract*— Graph algorithm has a wide variety of applications like finding routes in geographical maps (google maps), IP routing, telephone networks, finding optimal road networks between different cities etc. This huge range of application calls for the need to optimize graph algorithms heavily. In this project we aim to improve the running time of Dijkstra's Algorithm and Prim's Algorithm by improving the underlying priority queue implementation used by both algorithms. Fibonacci heap has constant amortized running time for most of its operations over basic Binomial heap implementation of priority queue.**

I. Project Description

Dijkstra's algorithm finds the shortest distance from a given source vertex to every other vertex in a weighted graph. This algorithm is used in telephonic networks, IP networks and social network graphs (ex: Facebook and LinkedIn) to find the shortest distances. Prim's algorithm finds the minimum spanning tree connecting different vertices in a graph, which is very useful in minimizing overall sum of edge weights in a minimally connected sub graph (which is tree). This helps in finding optimal road networks to connect different cities. With the increasing size and complexity of the above graphs we need to make sure that every component of the algorithm is optimized. This project aims at improving the overall running time of both of the above algorithms using an advanced data structure for priority queue called Fibonacci heap which has the following improved amortized running time.

Operation	Average running time [1]
Insert	$\Theta(1)$
Find-min	$\Theta(1)$
Delete-min	$O(\log n)$
Decrease-key	$\Theta(1)$
Merge	$\Theta(1)$

The project belongs to the category of Deterministic Algorithm and Algorithm snippets.

Following are the stumbling blocks of the project:

- Understanding the Fibonacci Heap
- Implementing the Fibonacci Heap followed by its integration with the Dijkstra's algorithm and Prim's algorithm
- Collecting real life graph data on which we would run the algorithms
- Comparing and analysing the above implementation with the basic implementation of priority queue using binary heap
- Visualisation of the heap at intermediate stages for better understanding

The project has four stages: Gathering, Design, Infrastructure Implementation, and User Interface.

A. Stage1 - The Requirement Gathering Stage.

In this project we deliver an improved Dijkstra's algorithm and Prim's algorithm to calculate the shortest path from a source to all the possible destinations and minimum spanning tree respectively. We plan to improve the running time of both the algorithms by using the Fibonacci Heap data structure. We also make a comparison between the above mentioned algorithm and the naive implementations using Binomial Heap. We also visualize the intermediate states of the heap and draw comparisons on different large graphs data sets.

1) System Description - Dijkstra's algorithm:

- Input: Graph in the form of adjacency list/list of edges/adjacency matrix, weights of edges, and the source vertex from which the distances to all other nodes present in the given graph need to be calculated
- Data type: (List/2D array: containing the graph), (list of positive real numbers: containing the weights of corresponding edges), (integer: indicating the source vertex number)
- Output: Distance of all the nodes from the given source node
- Output Type: Positive real numbers representing the distances of each node from the source vertex

2) System Description - Prim's algorithm:

- Input: Graph in the form of adjacency list/list of edges/adjacency matrix, weights of edges
- Data type: (List/2D array: containing the graph), (list of positive real numbers: containing the weights of corresponding edges)
- Output: Minimum spanning tree of the given graph
- Output Type: Tree in the form of adjacency list

3) Constraints:

- Dijkstra's algorithm is valid only for graphs with non-negative edge weights, weights of the edges should be always ≥ 0.
- Prim's algorithm does not have any such constraints.

TABLE I
TIME COMPLEXITIES [1]

Operation	Binary heap (worst case)	Fibonacci Heap (amortized)
Make-heap	$\Theta(n)$	$\Theta(1)$
Insert	$\Theta(\log(n))$	$\Theta(1)$
Minimum	$\Theta(1)$	$\Theta(1)$
Extract-min	$\Theta(\log(n))$	$\mathcal{O}(\log(n))$
Decrease-key	$\Theta(\log(n))$	$\Theta(1)$

4) Real World scenarios:

- Dijkstra's algorithm can be used to find Erdős number in co-author network of research papers (input), where the source node from which shortest distance to be calculated is Paul Erdős. Output would be the Erdős numbers (positive integers) of all the authors present in the network.
- LinkedIn job search platform to tell how many connections further (output - positive integer) one user is from another in the LinkedIn connections network (input).
- Improve running time of algorithm to find the optimal road networks, connecting different cities. Input will be a fully connected graph with weights of edges representing the euclidean distances between each of the corresponding cities. Output will be a minimum spanning tree spanning all the cities present in the input.

5) Project Timeline:

- Nov 6: Project proposal (gathering requirements)
- Nov 24: First Report (Data collection and implementation of Fibonacci heap)
- Nov 27: Second Report (Integration of Fibonacci heap with Dijkstra's algorithm and Prim's algorithm followed by comparison, analysis and visualization)

B. Stage2 - The Design Stage.

1) Dijkstra's and Prim's Algorithm: Following are the Pseudo code and flow diagram for Dijkstra's and Prim's Algorithms. These algorithms are very similar and their time complexities heavily depend on the time complexity of the heap operations which in-turn depend on their implementations.

In terms of heap operations, the time complexities of these algorithms are: $(|V| * extractMin) + (|V| * insert) + (|E| * decreaseKey)$. The time complexities of Dijkstra's algorithm with Binary heap and Fibonacci heap implementations are $\mathcal{O}((|V| + |E|)\log(|V|))$ and $\mathcal{O}((|V|\log(|V|) + |E|))$ respectively [1]. Refer Table I for the time complexity of different operations in each of the implementations.

2) Fibonacci Heap pseudo code and time analysis: Since there is an improvement in time complexity of Dijkstra's and Prim's algorithms using Fibonacci heap, we implement it and use it to reduce run-time. We use amortized analysis for analysing the time complexity of operations on Fibonacci heap. The time complexities given in Table I are amortized time complexities.

Algorithm 1: Dijkstra's Algorithm on $G(V, E)$ [1]

```
foreach vertex v∈V do
    v.dist = ∞
    v.parent = NULL
end
s.dist = 0
make-queue(V)
while Q ≠ φ do
    u = extract − min(Q)
    foreach (u, v)∈E do
        if v.dist > u.dist + w(u,v) then
            v.dist = u.dist + w(u,v)
            v.parent = u
            decrease-key(Q, v, v.dist)
        end
    end
end
```

Algorithm 2: Prim's Algorithm on $G(V, E)$ [1]

```
foreach vertex v∈V do
    v.dist = ∞
    v.parent = NULL
end
s.dist = 0
make-queue(V)
while Q ≠ φ do
    u = extract − min(Q)
    foreach (u,v) ∈E do
        if v.dist > w(u,v) then
            v.dist = w(u,v)
            v.parent = u
            decrease-key(Q, v, v.dist)
        end
    end
end
MST = ∪_{v∈V−{s}}{v.prev, v}
```

Best way to analyse amortized time complexity is to use the potential function approach [1]. It is a mathematical way of modeling the extra work done in an operation and storing it as "potential" energy and using it for other operations, thus achieving lower run-time on other operations.

$\therefore AmortizedTime = \Delta(\phi) + ActualTime$ where ϕ is the potential of Fibonacci heap.

Potential function of Fibonacci heap is $\phi(H) = t(H) + 2 * m(H)$ where $t(H)$ is the number of nodes in the root list of Fibonacci heap and $m(H)$ is number of marked nodes in the Fibonacci heap.

C. Stage3 - The Implementation Stage.

We have used C++ for the implementation of the algorithms and have used Python for pre-processing of the data and visualization of the time complexity of all the algorithms.

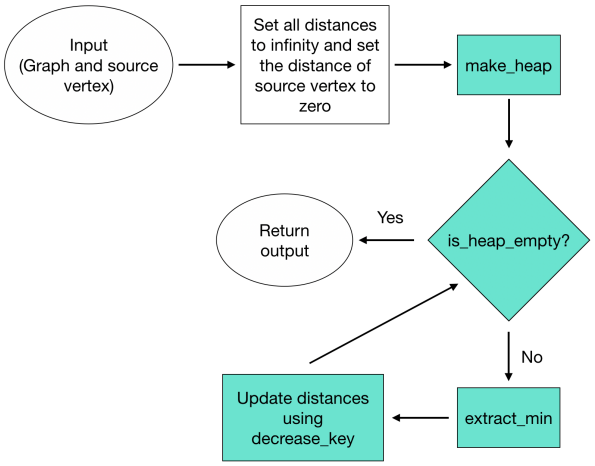


Fig. 1. Flow diagram for Dijkstra's and Prim's algorithm. For all the blue cells, we will be use both, binary heap and Fibonacci heap and make a comparison between them.

Algorithm 3: FIBONACCI HEAP INSERT [1]

$x.degree = 0$
$x.parent = NULL$
$x.child = NULL$
$x.mark = False$
if $H.min == NULL$ **then**
  Create root list for H containing just x
  $H.min = x$
else
  insert x into H's root list
  if $x.key < H.min.key$ **then**
    $H.min = x$
  end
end
$H.n = H.n + 1$

There is an increase of one new node in the root list.
Time Analysis:

$$\begin{aligned}\phi(H_{before}) &= t(H_{before}) + 2 * m(H_{before}) \\ \phi(H_{after}) &= (t(H_{before}) + 1) + 2 * m(H_{before}) \\ Amortized &= \Delta(\phi) + ActualTime \\ &= \mathcal{O}(1) + 1 \\ &= \mathcal{O}(1)\end{aligned}$$

Input Representation of Graph: The first number in first line of input represents number of nodes in the graph, followed by a line for each and every node which starts with label of the node followed by it's degree. Remaining numbers in the line represent the list of labels followed by edge weight.

5
0 3 1 1 2 1 3 4
1 3 0 1 2 3 4 6
2 4 0 1 1 3 3 2 4 1
3 3 0 4 2 2 4 1
4 3 1 6 2 1 3 1

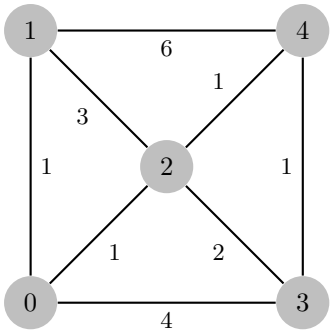
Algorithm 4: FIBONACCI HEAP EXTRACT MIN [1]

$z = H.min$
if $z \neq NULL$ **then**
  foreach *child x of z* **do**
    add x to the root list of H
    $x.p = NULL$
  end
  Remove z from root list of H
  if *z == z.right* **then**
    H.min = NULL
  else
    H.min = z.right
    Consolidate(H)
  end
  H.n = H.n - 1
end
return z

Since $\log(n)$ is the maximum degree of a node in root list of Fibonacci heap, removing the minimum key node from heap will add at-most $\log(n)$ nodes to the root list. Since Consolidate takes $\mathcal{O}$(number of nodes in root list) $= \mathcal{O}(\log(n) + t(H_{before}))$, actual time of extract min operation is $\mathcal{O}(\log(n) + t(H_{before}))$. After consolidation root list contains at most $\log(n) + 1$ nodes having unique degrees, there will be no change in the count of marked nodes.

Time Analysis:

$$\begin{aligned}\phi(H_{before}) &= t(H_{before}) + 2 * m(H_{before}) \\ \phi(H_{after}) &= (\log(n) + 1) + 2 * m(H_{before}) \\ Amortized &= \Delta(\phi) + ActualTime \\ &= \mathcal{O}(\log(n) + 1 - t(H_{before})) \\ +\mathcal{O}(\log(n) + t(H_{before})) & \\ &= \mathcal{O}(\log(n))\end{aligned}$$



Following is the graph represented by the input.

Output of Dijkstra's Algorithm: In the first part of output, each line represents the label of the node and the distance from 0^{th} node. Second part of output represents the previous node in the shortest path tree given by Dijkstra's. Figure: 2 represents the shortest path tree for the output given.

Algorithm 5: FIBONACCI DECREASE KEY [1]

if $k > x.key$ **then**
  error "new key is greater than current key"
end
x.key = k
y = x.p
if $y \neq NULL \wedge x.key < y.key$ **then**
  CUT(H, x, y)
  CASCADING-CUT(H, y)
end
if *x.key* $<$ *H.min.key* **then**
  H.min = *x*
end

Let us say the call to cascading cut has c recursive iterations. Initial cut and all cascading cut calls (except the last cascading cut call) add a node to the root list. Hence, the final root list will contain c extra nodes. Since c calls to cascading cut unmarks c-1 nodes, the initial cut may not have unmarked a marked node, and the last call to cascading cut will mark a node, the count of marked nodes is decreased by c-1 (differed by a constant). Since each of the cascading cut and cut takes constant time, actual time taken is $\mathcal{O}(c)$.

Time Analysis:

$$\begin{aligned} Amortized &= \Delta(\phi) + ActualTime \\ &= \Delta(t(H)) + 2 * \Delta(m(H)) \\ &+ ActualTime \\ &= c + 2(-c+1) + \mathcal{O}(c) \\ &= \mathcal{O}(1) \end{aligned}$$

0, 0
1, 1
2, 1
3, 3
4, 2

4, 2
3, 2
2, 0
1, 0

Output of Prim's Algorithm: Output represents the previous node in the minimum spanning tree given by the algorithm. Figure: 3 represents the minimum spanning tree of the output given.

4, 2
3, 4
2, 0
1, 0

Data-sets and results: We have run Dijkstra's and Prim's algorithms with Binary and Fibonacci heaps on 10 different data-sets[2] whose number of nodes, edges and their run time with each heap are given below.

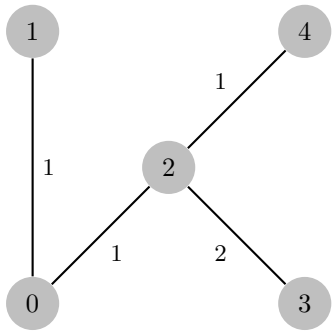


Fig. 2. Output of Dijkstra's

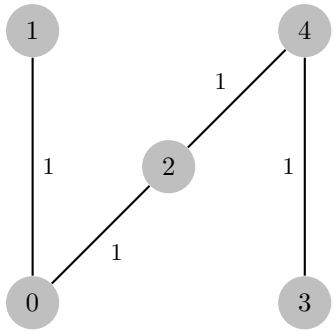


Fig. 3. Output of Prim's

Nodes	Edges	Binary(μs)	Fibonacci(μs)
450	9803	3261	2711
450	17425	4382	5064
7125	35323	46170	30232
62586	147892	498340	247882
22469	171001	157065	104571
37700	289003	287011	184422
77360	828161	691575	440828
82168	870161	734771	473706
10001	990000	168574	139537
1971281	5533214	13221199	8613626

Comparison of run time for Dijkstra's using Binary and Fibonacci heaps.

Nodes	Edges	Binary	Fibonacci
450	9803	7355	4522
450	17425	9854	7344
7125	35323	55493	34866
62586	147892	523045	316467
22469	171001	228654	142114
37700	289003	378351	236834
77360	828161	1066159	651082
82168	870161	1004193	640392
10001	990000	418010	340492
1971281	5533214	23338031	8283441

Comparison of run time for Prim's using Binary and Fibonacci heaps.

Conclusion: Based on above results we see that using Fibonacci Heap instead of Binary heap for

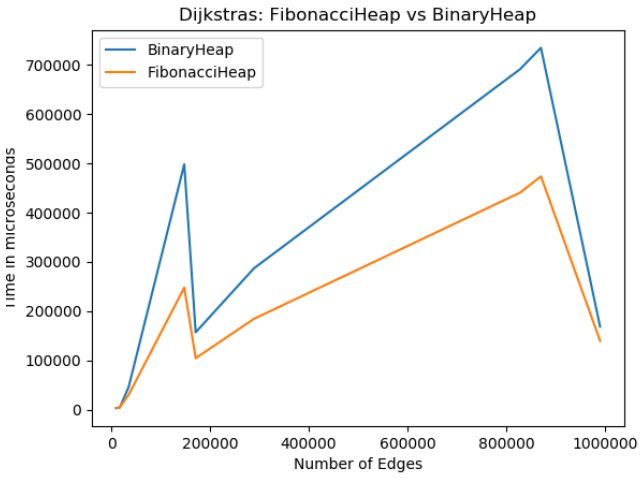


Fig. 4. Comparison of run time for Dijkstra's using Binary and Fibonacci heaps.

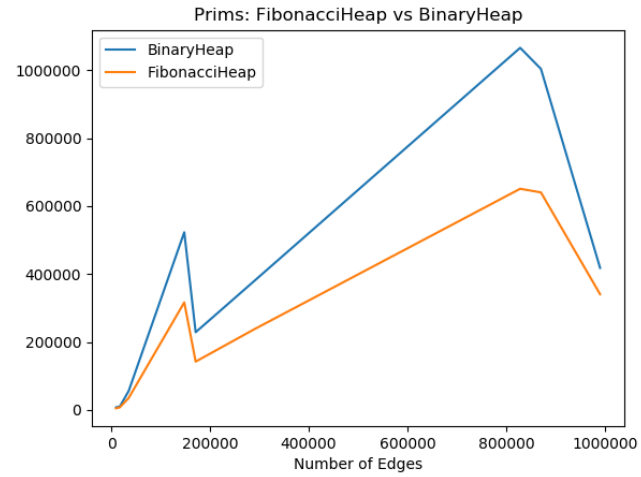


Fig. 5. Comparison of run time for Prim's using Binary and Fibonacci heaps.

Dijkstra's and Prim's algorithms will improve run time considerably for large data sets. Run time of these algorithms in terms of priority queue operations is $|V|*Extract-Min+(|V|+|E|)*(Insert/Decrease-Key)$. Since, Fibonacci heap has better running time (amortized) for Insert and Decrease-Key operations, $\mathcal{O}(1)$ it performs better.
Run time with Binary Heap = $\mathcal{O}((|V|+|E|)\log(|V|))$
Run time with Fibonacci Heap = $\mathcal{O}(|V|\log(|V|)+|E|)$

REFERENCES

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. "Introduction to Algorithms." Third Edition
[2] Jure Leskovec and Andrej Krevl. SNAP Datasets, Stanford: Large Network Dataset Collection https://snap.stanford.edu/data/