Shortest Path Problem

Dijkstra's vs Network Scaling Algorithm

Team Xi

Project Goals

1. Optimizing Dijkstra's with D-ary heap in place of Binary heap
2. Implementation and Comparison of Dijkstra with Network Scaling algorithms on different graph types
3. Experimenting with varying graph densities and edge weights

Algorithm Introduction

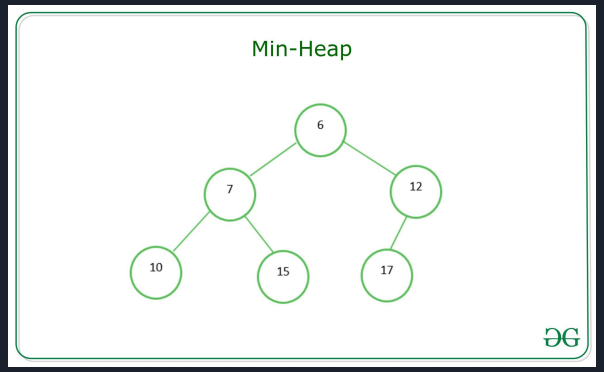
Dijkstra's Shortest path

- It is a greedy algorithm for finding the shortest paths between node in a graph from a single source.
- Uses a priority queue to make the greedy choice
- This is asymptotically the fastest known single-source shortest-path algorithm for arbitrary directed graphs with unbounded non-negative weights

Data Structures for efficient use of Dijkstra's

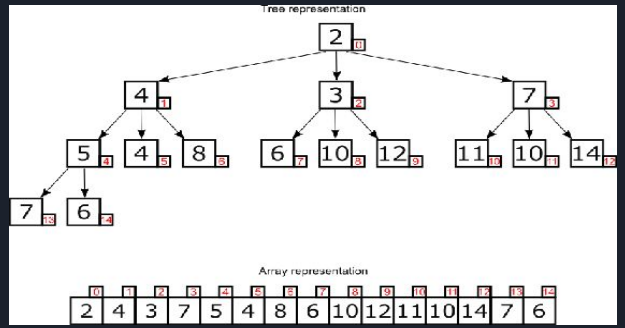
Binary Heap

- A binary min-heap is a heap data structure where each node have 2 children and the parent node is always equal to or less than its children
- The Extract Min and Decrease Key functionality allow it to be used as a priority queue



D-ary Heap

- The *d*-ary heap is a priority queue data structure, where nodes have *d* children instead of 2
- This data structure allows decrease priority operations to be performed more quickly than binary heaps at the expense of slower delete minimum operations which leads to better runtime of Dijkstra's



Why D-ary Heap?

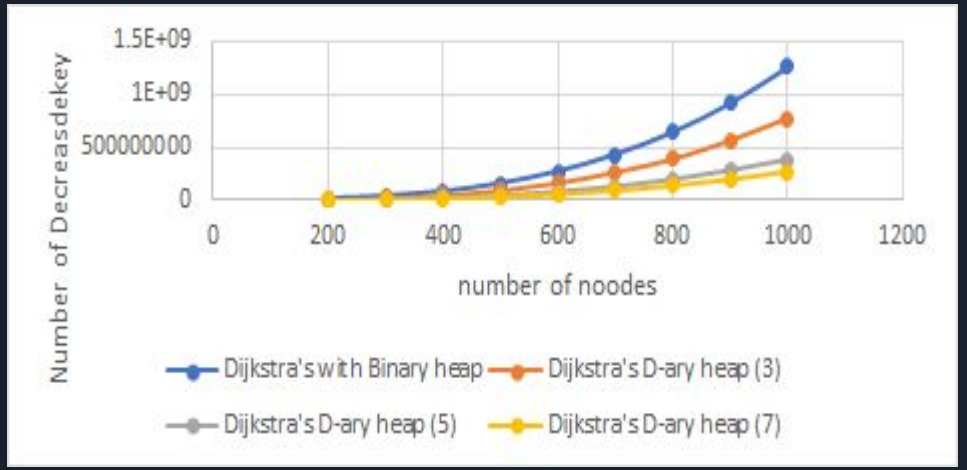
- Less efficient extract min operation is called once per node where as the now more efficient decrease key operation is called for each edge adjacent edge.
- Dense graphs with lots of connections between nodes will see increases in efficiency because of the optimization of the decrease key operation

What we expect to see?

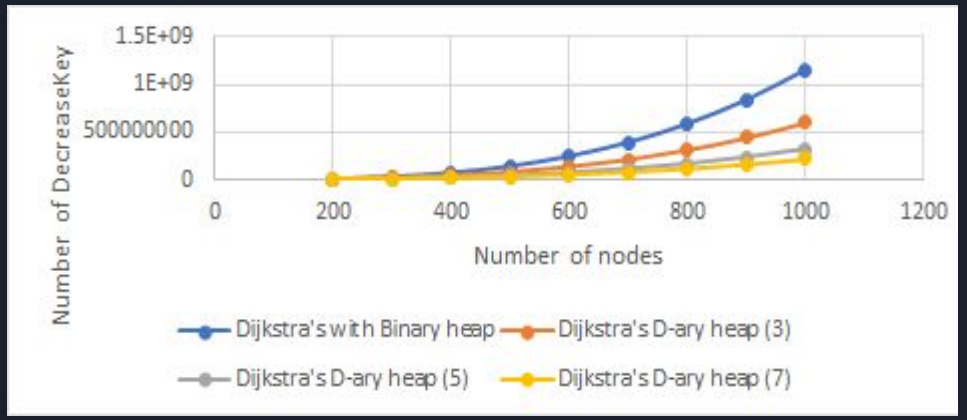
1. D-Ary Heap reduces the number of operations needed to decrease the Key
2. D-Aray Heap implementation will perform better on Dense Graphs
3. D-Ary Heap will be Faster than a binary heap

Total operations of decrease key on Dense and Sparse Graphs

Edge Density 0.2

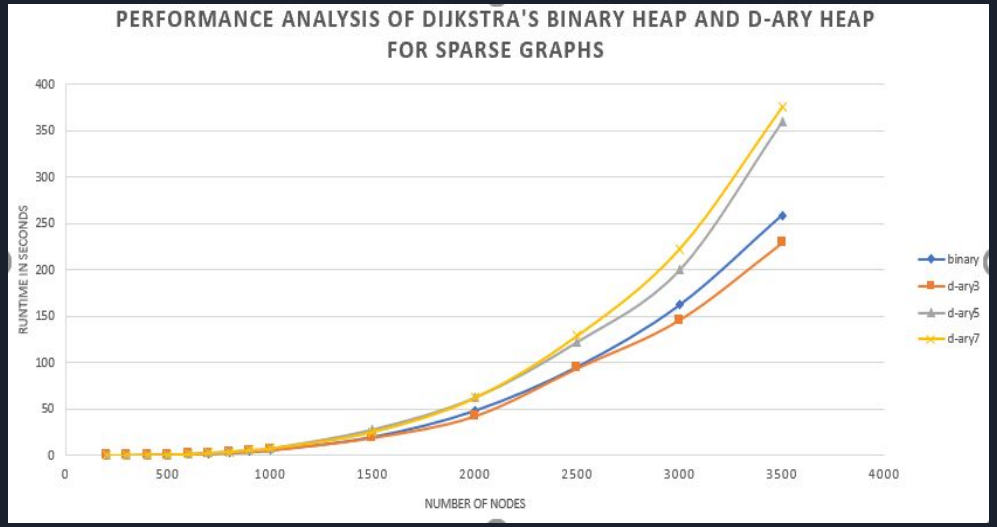


Edge Density 0.8

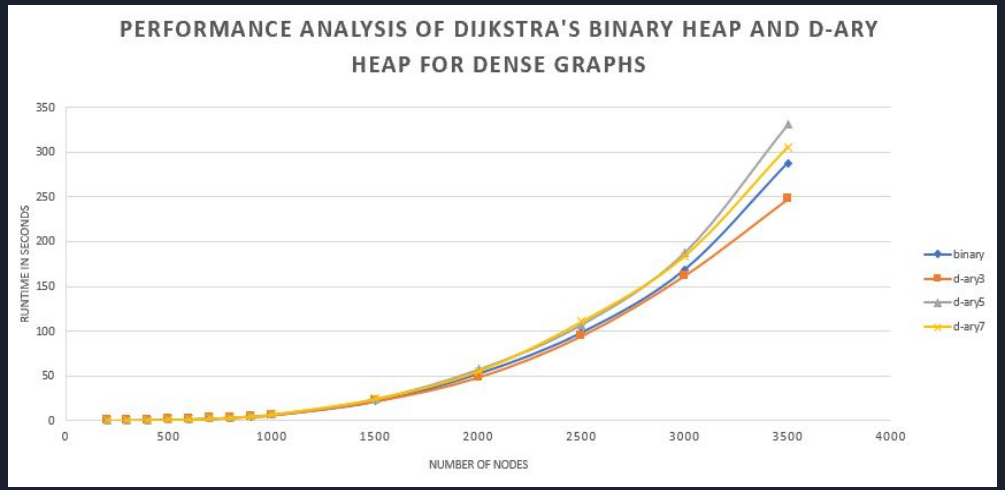


Total Runtime

Edge density 0.4



Edge density 0.8



Results on improving dijkstra's Algorithm

- Slight improvements are seen when using D heaps on graphs with an advantage
- Denser graphs see more of an advantage of D heaps
- In practice runtimes do not improve dramatically with implementation of a D heap

Using Scaling to Improve Network Problems

- Scaling can be an alternative to a Priority Queue
- Breaks down the problem into more manageable sub problems
  - Start with a “Near Optimum Solution”
  - Scale the problem up to an optimal solution

- Can be used to produce a single source shortest path algorithm that runs in $O(m \log_{2+m/n} N)$ time

Scaling Single Source Shortest Path

- “near-optimum” definition
  - $d_s = 0$
  - d_i “dominates” any edge ij where the $d_i + l_{ij}$ is >= d_j
  - Each vertex has a path stemming from s with a length between d_i and d_i + the number of edges(m)

“The iterative version of scaling is as follows. View each number as its k-bit binary expansion $b_1, \ldots b_k$, where k = floor(lg N) + 1. First solve the problem on the network where all parameters are 0. Then for s =0 , … , k - 1, transform the solution for parameters $b_1, \ldots b_s$, to a solution for parameters $b_1, \ldots b_s$,.”

Scaling Shortest Path Implementation

```
GABOW(G, w, v_0)
 1  for each vertex u ∈ V(G)
 2      u.dist = ∞
 3  i = lg(max{range(w)})
 4  v_0.dist = 0
 5  Q = ∅
 6  INSERT(Q, v_0, 0)
 7  while Q ≠ ∅
 8      u = EXTRACT-MIN(Q)
 9      for each vertex v ∈ V(G) such that (u, v) ∈ E(G)
10          if v.dist > u.dist + w_i(u, v)
11              v.dist = u.dist + w_i(u, v)
12              INSERT(Q, v, v.dist)
13  while i > 0
14      i--
15      for each vertex u ∈ V(G)
16          u.extra-dist = ∞
17      v_0.extra-dist = 0
18      Q = ∅
19      INSERT(Q, v_0, 0)
20      while Q ≠ ∅
21          u = EXTRACT-MIN(Q)
22          for each vertex v ∈ V(G) such that (u, v) ∈ E(G)
23              l_(u,v) = w_i(u, v) + 2(u.dist − v.dist)
24              if v.extra-dist > u.extra-dist + l_(u,v)
25                  v.extra-dist = u.extra-dist + l_(u,v)
26                  INSERT(Q, v, v.extra-dist)
27      for each vertex u ∈ V(G)
28          if u.dist < ∞
29              u.dist = 2 × u.dist + u.extra-dist
```

Why this works

- The conditions of a near optimal solution allows for an array to be used as the queue and can be to be converted in O(m)
- The repeated Single Source Shortest Path Subproblems are bound in a way that makes them cheaper to solve than an unbounded iteration of dijkastra's
- $O(m \log_{2 + m/n} N)$

Questions?