# Dijkstra's Algorithm

Run Qiang (Johnny) Mei and Mohammed J Roshid

# What is Dijkstra's Algorithm

Is a greedy algorithm that finds a single source shortest path

- Greedy Property
- Visits all vertices
- Finds shortest path between source and all nodes
- Fails with negative weights

Real World Applications

GIS systems (ex. Google Maps?)


http://web.stanford.edu/class/archive/cs/cs106b/cs106b.1182/assn/Trailblazer.html

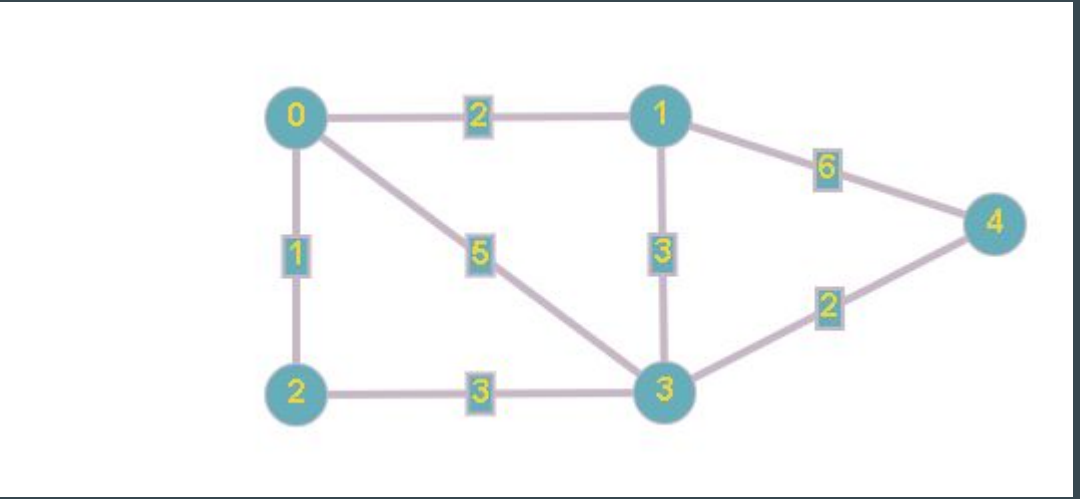
# Pseudo Code

```
1:    function Dijkstra(Graph, source):
2:        for each vertex v in Graph:                 // Initialization
3:            dist[v] := infinity                     // initial distance from source to vertex v is set to infinite
4:            previous[v] := undefined                // Previous node in optimal path from source
5:        dist[source] := 0                           // Distance from source to source
6:        Q := the set of all nodes in Graph          // all nodes in the graph are unoptimized - thus are in Q
7:        while Q is not empty:                       // main loop
8:            u := node in Q with smallest dist[]
9:            remove u from Q
10:           for each neighbor v of u:               // where v has not yet been removed from Q.
11:               alt := dist[u] + dist_between(u, v)
12:               if alt < dist[v]                    // Relax (u,v)
13:                   dist[v] := alt
14:                   previous[v] := u
15:       return previous[]
```

http://www.gitta.info/Accessibiliti/en/html/Dijkstra_learningObject1.html

# Example: 0 -> 4

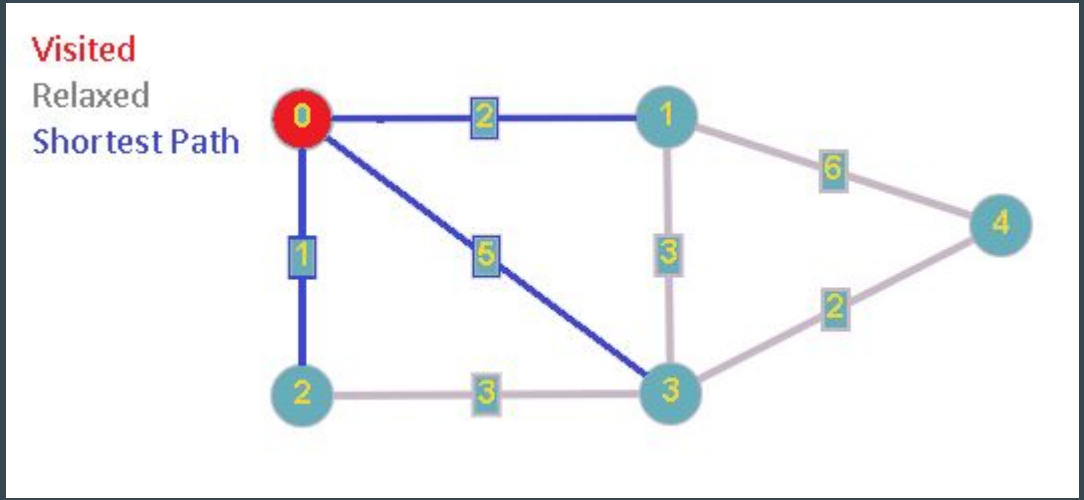
| Node | Dist | Prev |
|---|---|---|
| 0 | 0 | NULL |
| 1 | ∞ | NULL |
| 2 | ∞ | NULL |
| 3 | ∞ | NULL |
| 4 | ∞ | NULL |




Priority Queue {0}

# Example Continued: 0 -> 4

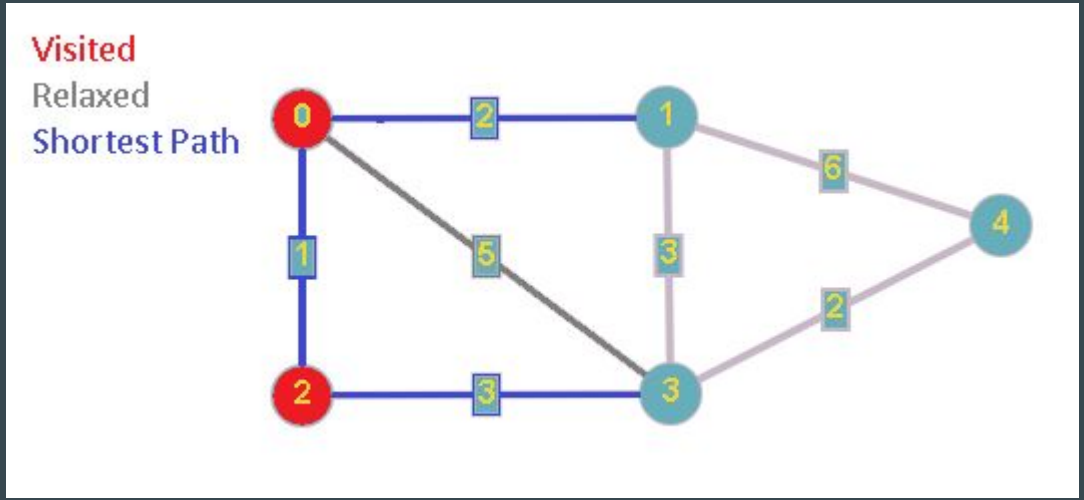
| Node | Dist | Prev |
|---|---|---|
| 0 | 0 | NULL |
| 1 | 2 | 0 |
| 2 | 1 | 0 |
| 3 | 5 | 0 |
| 4 | ∞ | NULL |




Priority Queue {~~0~~ 2,1,3}

# Example Continued: 0 -> 4

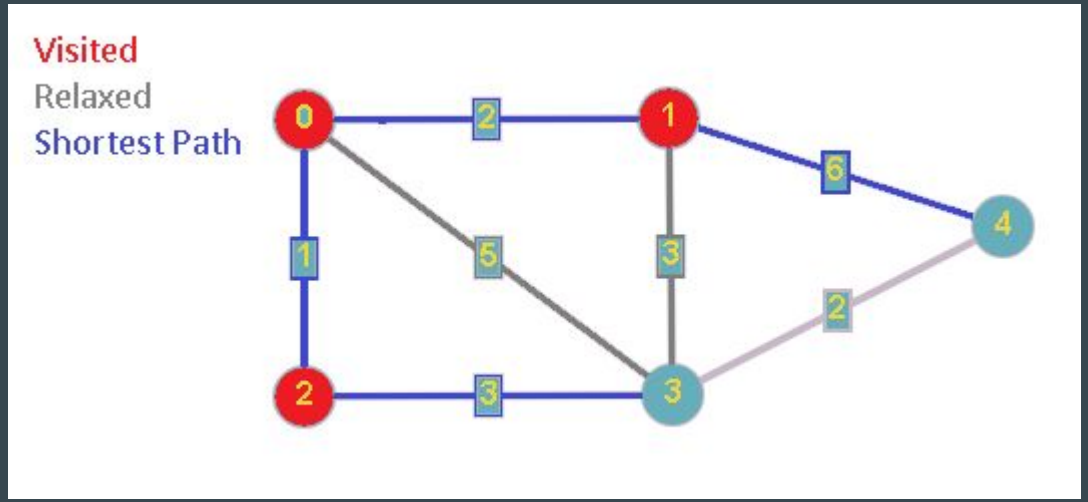
| Node | Dist | Prev |
|---|---|---|
| 0 | 0 | NULL |
| 1 | 2 | 0 |
| 2 | 1 | 0 |
| 3 | ~~5~~ 4 | ~~0~~ 2 |
| 4 | ∞ | NULL |




Priority Queue {~~0 2~~ 1,3}

# Example Continued: 0 -> 4

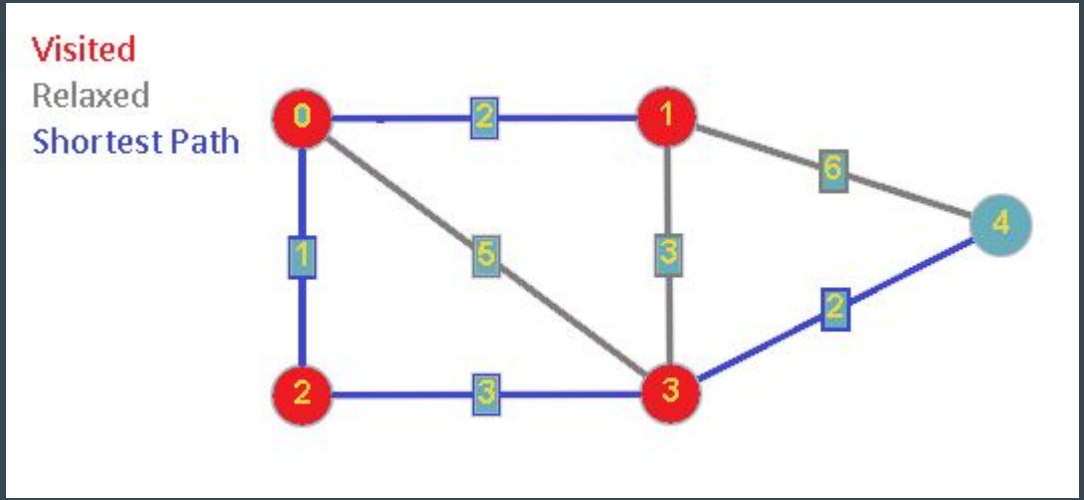
| Node | Dist | Prev |
|---|---|---|
| 0 | 0 | NULL |
| 1 | 2 | 0 |
| 2 | 1 | 0 |
| 3 | ~~5~~ 4 | ~~0~~ 2 |
| 4 | 8 | 1 |




Priority Queue {~~0 2 1~~ 3,4}

# Example Continued: 0 -> 4

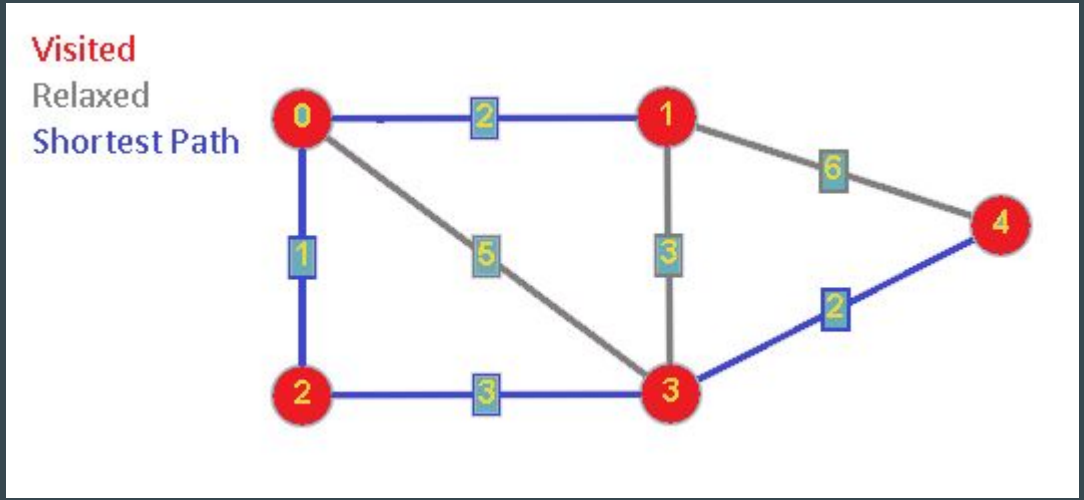
| Node | Dist | Prev |
|---|---|---|
| 0 | 0 | NULL |
| 1 | 2 | 0 |
| 2 | 1 | 0 |
| 3 | ~~5~~ 4 | ~~0~~ 2 |
| 4 | ~~8~~ 6 | ~~1~~ 3 |




Priority Queue {~~0 2 1 3~~ 4}

# Example Continued: 0 -> 4

| Node | Dist | Prev |
|---|---|---|
| 0 | 0 | NULL |
| 1 | 2 | 0 |
| 2 | 1 | 0 |
| 3 | ~~5~~ 4 | ~~0~~ 2 |
| 4 | ~~8~~ 6 | ~~1~~ 3 |




Priority Queue {~~0 2 1 3 4~~}

# Algorithm Analysis

- Interesting Features
  - Uses edge relaxation => important part to the greedy property which is selecting the smallest-weighted edge at the current node
  - Finds the shortest path from the source to all other nodes
- Dijkstra's Algorithm Time Complexity using Heap: O(Elog(V))
  - Priority Queue
  - Array
- Dijkstra's Algorithm Time Complexity using Linear Search: O(V^2)
  - List
  - Array

# Implementation

- C++ and Java
- Graph Generator (C++)
- Dijkstra's Algorithm (Java)
  - GraphList.java
  - HeapNode.java
  - Node.java
  - ListNode.java
  - DjikstraHeap.java
  - DjikstraList.java
  - DjikstraDriver.java

# Code for Dijkstra's using Priority Queue

```
public void pathDistance(GraphList g, int start, int dest) {
    if(start == dest) {                    //no self loops
        minPathSum = 0;
        return;
    }

    //Minimum Priority Queue
    PriorityQueue<HeapNode> minH = new PriorityQueue<HeapNode>((Comparator<? super HeapNode>) new Comparator<HeapNode>() {
        @Override
        public int compare(HeapNode o1, HeapNode o2) {
            return o1.getWeight() - o2.getWeight();
        }
    });

    minH.add(new HeapNode(g.getNode(start), 0));    //first element is the starting node
    HashMap<Node,Integer> m = g.getList(start);     //adj list of starting node

    for(int i = 0; i < size; i++) {
        distance[i] = Integer.MAX_VALUE;  //set distances to "infinity"
        visited[i] = false;               //set all nodes to unvisited
        prev[i] = -1;                     //set all previous nodes to "NULL"
    }

    distance[start] = 0;                  //starting vertex set to 0

    HeapNode current;
    while(!minH.isEmpty()) {
        current = minH.remove();          //remove priority
        m = current.getNode().getList();
        for(Node k:m.keySet()) {          //relax edges
            if(!visited[k.getInd()] && m.get(k) + distance[current.getNode().getInd()] < distance[k.getInd()]) {
                distance[k.getInd()] = m.get(k) + distance[current.getNode().getInd()];
                minH.add(new HeapNode(k,m.get(k)));
                prev[k.getInd()] = current.getNode().getInd();
            }
        }
        visited[current.getNode().getInd()] = true;    //each iteration of the while loop visits a node

    }
    minPathSum = distance[dest];
}
```

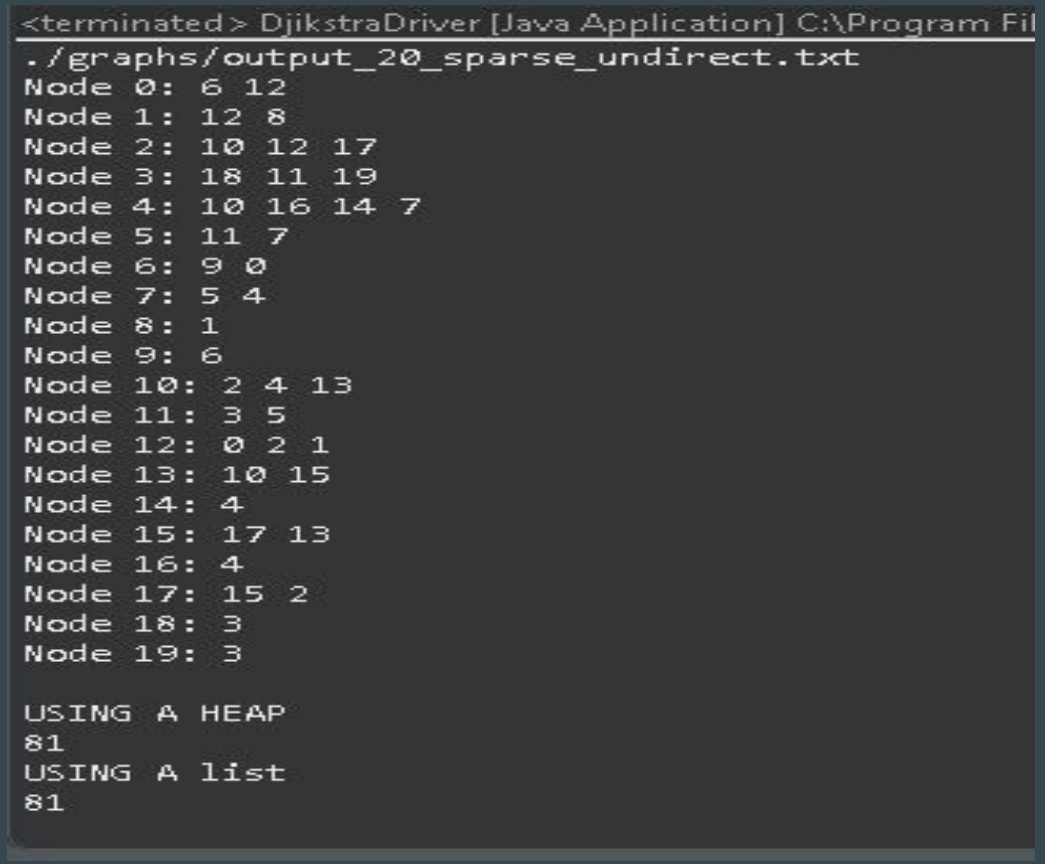
# Code for Dijkstra's using a List

```
public void pathDistance(GraphList g, int start, int dest) {
    if(start == dest) {                    //no self loops
        minPathSum = 0;
        return;
    }

    HashMap<Node,Integer> m = g.getList(start);        //adjacency list of starting node

    List<ListNode> l = new ArrayList<ListNode>();      //list to hold edges
    l.add(new ListNode(g.getNode(start), 0));

    for(int i = 0; i < size; i++) {
        distance[i] = Integer.MAX_VALUE;   //set distances to "infinity"
        visited[i] = false;                //set all nodes to unvisited
        prev[i] = -1;
    }

    distance[start] = 0;                   //starting vertex set to 0

    ListNode current;
    while(!l.isEmpty()) {
        current = l.get(0);
        int ind = 0;
        for(int i = 1; i < l.size(); i++) {        //get minimum weight node
            if(l.get(i).getWeight() < current.getWeight()) {
                current = l.get(i);
                ind = i;
            }
        }
        l.remove(ind);
        m = current.getNode().getList();
        for(Node k:m.keySet()) {               //relax edges
            if(!visited[k.getInd()] && m.get(k) + distance[current.getNode().getInd()] < distance[k.getInd()]) {
                distance[k.getInd()] = m.get(k) + distance[current.getNode().getInd()];
                l.add(new ListNode(k,m.get(k)));
                prev[k.getInd()] = current.getNode().getInd();
            }
        }
        visited[current.getNode().getInd()] = true;        //each iteration of the while loop visits a node
    }
    minPathSum = distance[dest];
}
```

# Problem Statement

Given a connected sparse graph, when does a Dijkstra's algorithm implemented with a priority queue perform better than a list?

# Demo


<terminated> DjikstraDriver [Java Application] C:\Program Fil
./graphs/output_20_sparse_undirect.txt
Node 0: 6 12
Node 1: 12 8
Node 2: 10 12 17
Node 3: 18 11 19
Node 4: 10 16 14 7
Node 5: 11 7
Node 6: 9 0
Node 7: 5 4
Node 8: 1
Node 9: 6
Node 10: 2 4 13
Node 11: 3 5
Node 12: 0 2 1
Node 13: 10 15
Node 14: 4
Node 15: 17 13
Node 16: 4
Node 17: 15 2
Node 18: 3
Node 19: 3
USING A HEAP
81
USING A list
81

# Verification

- Smaller Graphs = Manual Verification
- Proof of Correctness
  - Every vertex in V has shortest distance d{v}
  - New vertex u has path d{u}
  - Assume to the Contrary that there is a path Q less than d{u}
  - Q is split into two parts; path Qi, the path within V and d(xy), the first edge to leave V
    - Qi + d{xy} <= Q
  - Update d{y} so that d{y} <= d{x} + d{xy}
  - D{u} must be smallest since u is picked by Dijkstra's so d{u} <= d{y}
  - Combining above inequalities gives contradiction d{x} < d{x}
  - Thus no shorter path Q exists.

https://web.engr.oregonstate.edu/~glencora/wiki/uploads/dijkstra-proof.pdf

# Experimental Plan

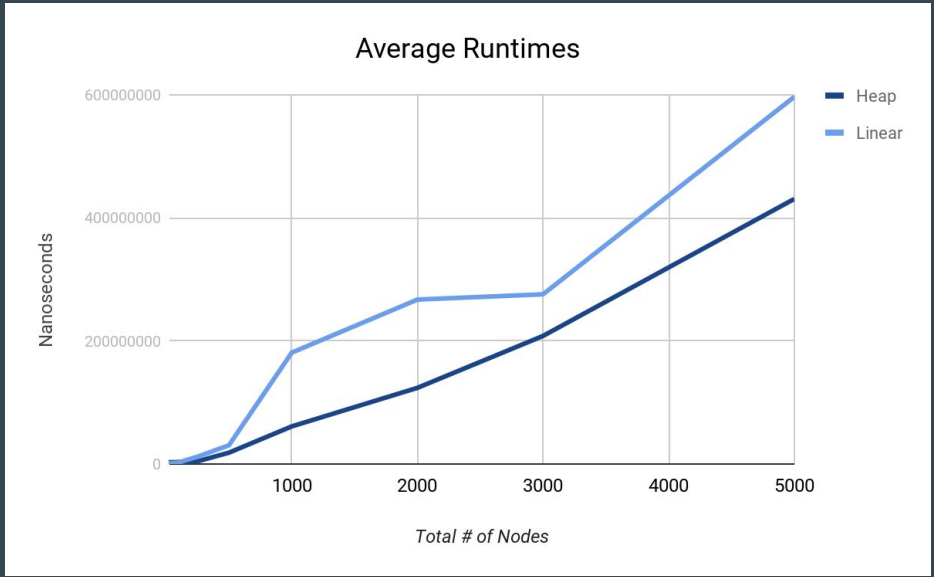
- Randomly generated graphs of varying nodes and edges
- Generated with program using adjacency matrices
- Consider only Sparse graphs (10% of total nodes)
- All graphs generated were made to be fully connected
- Varied in the number of nodes
- Total Nodes = 20, 100, 250,
- 500, 1000, 2000, 3000

```
rmei6@remote04:~/cs375/Djikstras-Presentation$ make
g++ -Wall -DBSD -DNDEBUG -c graphgen.cpp
g++ -Wall -DBSD -DNDEBUG -o graphgen graphgen.o
rmei6@remote04:~/cs375/Djikstras-Presentation$ ./graphgen incomplete undirected 20 2 10 output.txt
rmei6@remote04:~/cs375/Djikstras-Presentation$ cat output.txt
20
3,1 4,5 5,10 7,5 18,5 19,7
9,9 15,3
5,6 6,10 7,2 10,8
0,1
0,5 12,4
0,10 2,6 8,6
2,10 11,5 18,5
0,5 2,2
5,6 15,2
1,9 12,2 14,4
2,8 16,3
6,5 17,9
4,4 9,2
14,2
9,4 13,2
1,3 8,2
10,3
11,9
0,5 6,5 19,5
0,7 18,5
```

# Results

**Sparse Graph Runtimes (in nanoseconds)**

| | # of Nodes | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 20 | 100 | 250 | 500 | 1000 | 2000 | 3000 | 5000 |
| Heap Averages | 2694233 | 3153133 | 4413966 | 18386533 | 61113000 | 123739466.7 | 208125966.7 | 430800033.3 |
| List Averages | 846200 | 2441600 | 11905366 | 30425333 | 180979766.7 | 267215666.7 | 275802466.7 | 596757500 |




- Initially, the runtime of heap was longer than that of linear search when the nodes were 20-100.
- As the number of nodes increase, the runtime of the heap becomes smaller than the runtime of the linear search.

# Limitations and Future Work

- Limitations
  - Dijkstra's is only implemented with adjacency list
  - Only did test cases with undirected sparse graphs
- Additional Tests
  - Run tests with directed graphs
  - Run tests uses dense and complete graphs
  - Perform all the testing on an adjacency matrix implementation

# Conclusion

- Roles
  - Graph Generator: Run Qiang Mei
  - Dijkstra's Algorithm: Mohammed J Roshid
  - Presentation: Both of Us