# Project 2
# Dijkstra's Algorithm

Sydney Teo, Saniya Nangia, Namrah

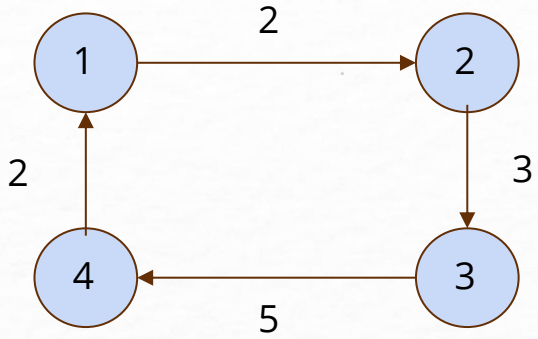
# Table of Contents

# Dijkstra's Algorithm

- Path finding algorithm
- Find the shortest path in a graph
- Only works for non-negative edge weights
- Can be implemented in 2 ways
    - Adjacency Matrix (with an Array Priority Queue)
    - Adjacency List (with a minimizing Heap Priority Queue)

# Dijkstra's Algorithm Visual Proof

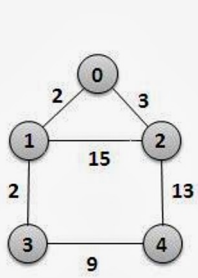
| Iteration | Vertex 1 | Vertex 2 | Vertex 3 | Vertex 4 |
|---|---|---|---|---|
| 1 | 0 | Inf | Inf | Inf |
| 2 | 0 | 2 | Inf | Inf |
| 3 | 0 | 2 | 2 + 3 = 5 | Inf |
| 4 | 0 | 2 | 5 | 5 + 5 = 10 |



# Algorithm 1:
# Adjacency Matrix & Priority Queue (Array)

# Adjacency Matrix

- Square matrix used to represent a finite V x V graph
- Rows and columns labelled by graph vertices
- $(v_i,v_j) = 0$ means no edge between vertices i and j, i.e. vertices are not adjacent
- $(v_i,v_j) = x$ means edge between vertices i and j with cost x, i.e. vertices are adjacent (i->j)




| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 2 | 3 | 0 | 0 |
| 1 | 2 | 0 | 15 | 2 | 0 |
| 2 | 3 | 15 | 0 | 0 | 13 |
| 3 | 0 | 2 | 0 | 0 | 9 |
| 4 | 0 | 0 | 13 | 9 | 0 |

**Adjacency Matrix Representation of Weighted Graph**

*Source: https://www.thecrazyprogrammer.com/2014/03/representation-of-graphs-adjacency-matrix-and-adjacency-list.html*

# Code Implementation

## Graph

- Used to represent a graph
- Inherited by GraphAdjMatrix and GraphAdjList classes

```
public class Graph {
  protected int noOfVertices;
  protected int noOfEdges;
  protected int source;

  Graph() {
  };

  Graph(int noOfVertices, int source) {
    this.noOfVertices = noOfVertices;
    this.source = source;
  }

  public void printGraph() {
  }
}
```

# Code Implementation

## GraphAdjMatrix

- Used for adjacency matrix representation of graph
- Inherits Graph class
- Stores graph in two-dimensional integer array

```
public class GraphAdjMatrix extends Graph {
  protected int[][] graph;

  GraphAdjMatrix(int[][] graph, int noOfVertices, int source) {
    super(noOfVertices, source);
    this.noOfEdges = setNoOfEdges(graph);
    this.graph = graph;
  }

  public int setNoOfEdges(int[][] graph) {
    int noOfEdges = 0;
    for (int i = 0; i < noOfVertices; i++) {
      for (int j = 0; j < noOfVertices; j++) {
        if (graph[i][j] > 0) {
          noOfEdges++;
        }
      }
    }
    return (noOfEdges);
  }
```

# Code Implementation

## GraphAdjMatrix

- Used for adjacency matrix representation of graph
- Inherits Graph class
- Stores graph in two-dimensional integer array

```
public void printGraph() {

  System.out.println("Source: " + source + ", Edges: " +
      noOfEdges + ", Vertices: " + noOfVertices);
  for (int i = 0; i < noOfVertices; i++) {
    for (int j = 0; j < noOfVertices; j++) {
      System.out.print(graph[i][j] + " ");
    }
    System.out.println();
  }
 }
}
```

# Code Implementation

## PriorityQueue1

- Used for array implementation of priority queue
- Sorts nodes in order of priority (cost)
- Removes and returns head of queue

```java
import java.util.*;

public class PriorityQueue1 {
  private List<Node> nodes;

  PriorityQueue1() {
    this.nodes = new ArrayList<Node>();
  }

  public void add(Node n) {
    nodes.add(n);
    sortNodes();
  }
```

# Code Implementation

### PriorityQueue1

- Used for array implementation of priority queue
- Sorts nodes in order of priority (cost)
- Removes and returns head of queue

```
public void sortNodes() {
  Collections.sort(nodes, new Node() {

    @Override
    public int compare(Node node1, Node node2) {

      if (node1.cost < node2.cost)
        return -1;

      if (node1.cost > node2.cost)
        return 1;

      return 0;
    }
  });
}
```

# Code Implementation

### PriorityQueue1

- Used for array implementation of priority queue
- Sorts nodes in order of priority (cost)
- Removes and returns head of queue

```
  public int remove() {
    int u = nodes.get(0).node;
    nodes.remove(0);
    return (u);
  }

  public boolean isEmpty() {
    return (nodes.isEmpty());
  }
}
```

# Code Implementation

## DijkstraAlgo1

- Implements Dijkstra's algorithm when graph is stored in an adjacency matrix and array is used for priority queue

```
public class DijkstraAlgo1 {
  private GraphAdjMatrix adj;
  private int distances[];
  private int traversed[];
  private PriorityQueue1 priorityQueue;

  public DijkstraAlgo1(GraphAdjMatrix adj) {
    this.adj = adj;
    distances = new int[adj.noOfVertices];
    traversed = new int[adj.noOfVertices];
    priorityQueue = new PriorityQueue1();
  }
```

# Code Implementation

### DijkstraAlgo1

- Implements Dijkstra's algorithm when graph is stored in an adjacency matrix and array is used for priority queue

```
public void shortestDistance() {

  for (int i = 0; i < adj.noOfVertices; i++) {
    distances[i] = Integer.MAX_VALUE;
    traversed[i] = 0;
  }

  priorityQueue.add(new Node(adj.source, 0));
  distances[adj.source] = 0;

  while (!priorityQueue.isEmpty()) {
    int u = priorityQueue.remove();
    if (traversed[u] == 0) {
      traversed[u] = 1;
      adjacentVertices(u);
    }
  }
}
```

# Code Implementation

## DijkstraAlgo1

- Implements Dijkstra's algorithm when graph is stored in an adjacency matrix and array is used for priority queue

```
    private void adjacentVertices(int u) {
      for (int v = 0; v < adj.noOfVertices; v++) {
        int vCost = adj.graph[u][v];

        if (traversed[v] == 0 && vCost != 0) {
          if (distances[u] + vCost < distances[v]) {
            distances[v] = distances[u] + vCost;
          }
          priorityQueue.add(new Node(v, distances[v]));
        }
      }
    }

    public void printResults() {
      System.out.println("Vertex \t\t Distance from Source");
      for (int i = 0; i < distances.length; i++)
        System.out.println(i + " \t\t " + distances[i]);
    }
}
```

# Theoretical Time Complexity

Time Complexity = |V| * Cost to Extract + |E| * Cost to Update

- Cost to Extract
  - Traverse the entire array to get minimum distance
  - **O(|V|)** ⇒ |V| vertices in entire array

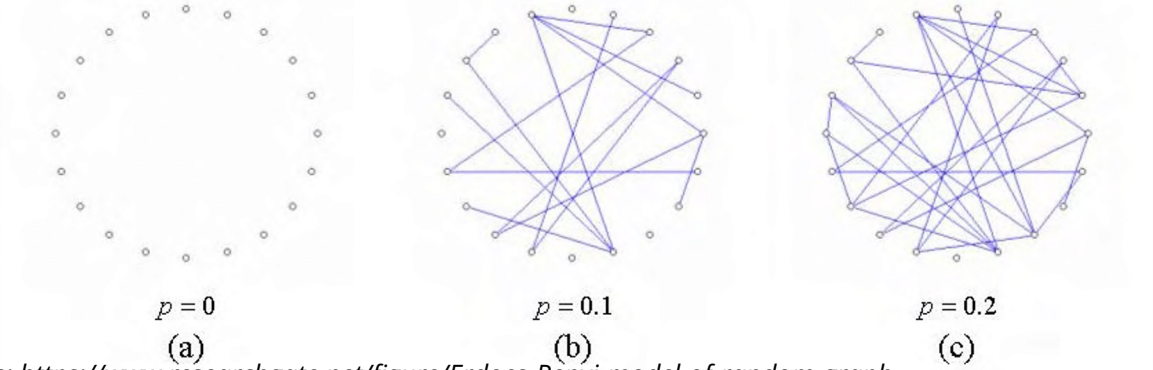
- Cost to Update
  - Array allows for direct access to update ‘d’ and ‘pi’
  - **O(1)** ⇒ Constant time
  - Every edge is processed at most once in the worst case

Overall Time Complexity = **O(|V|²)**

# Empirical Time Complexity

**Generation of graphs using Erdős–Rényi G(n,p) model:**

- Graph is constructed by connecting labeled nodes randomly
- p: probability of occurrence of each edge in graph, independently from every other edge
- n: number of nodes in graph
- Maximum number of edges = n(n-1)




*Source: https://www.researchgate.net/figure/Erdoes-Renyi-model-of-random-graph-evolution_fig10_313854183*

# Empirical Time Complexity

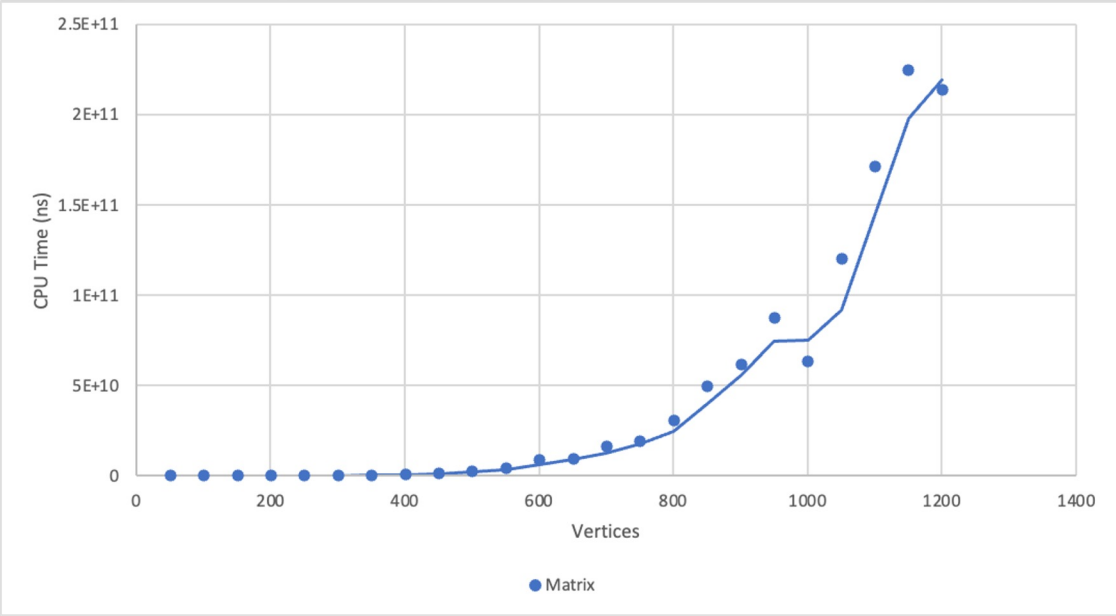
### 1. Fixed N

- Vary probability of occurrence of edges for a fixed number of vertices
- Plot run-time against edge density (number of edges/maximum number of edges)

### 2. Fixed p

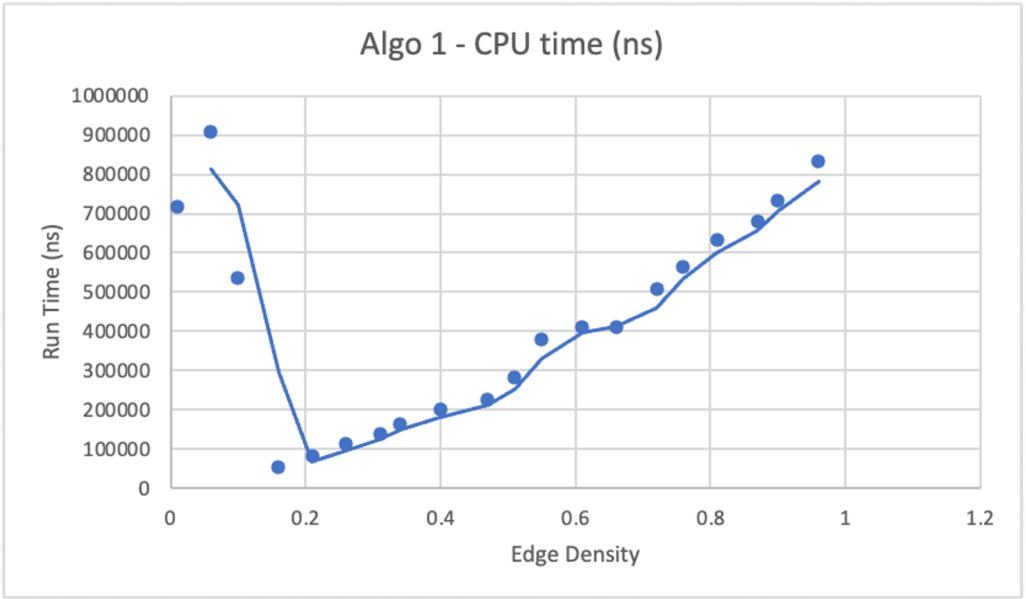
- Vary number of vertices for a fixed probability of occurrence of edges
- Plot run-time against number of vertices

# Empirical Time Complexity



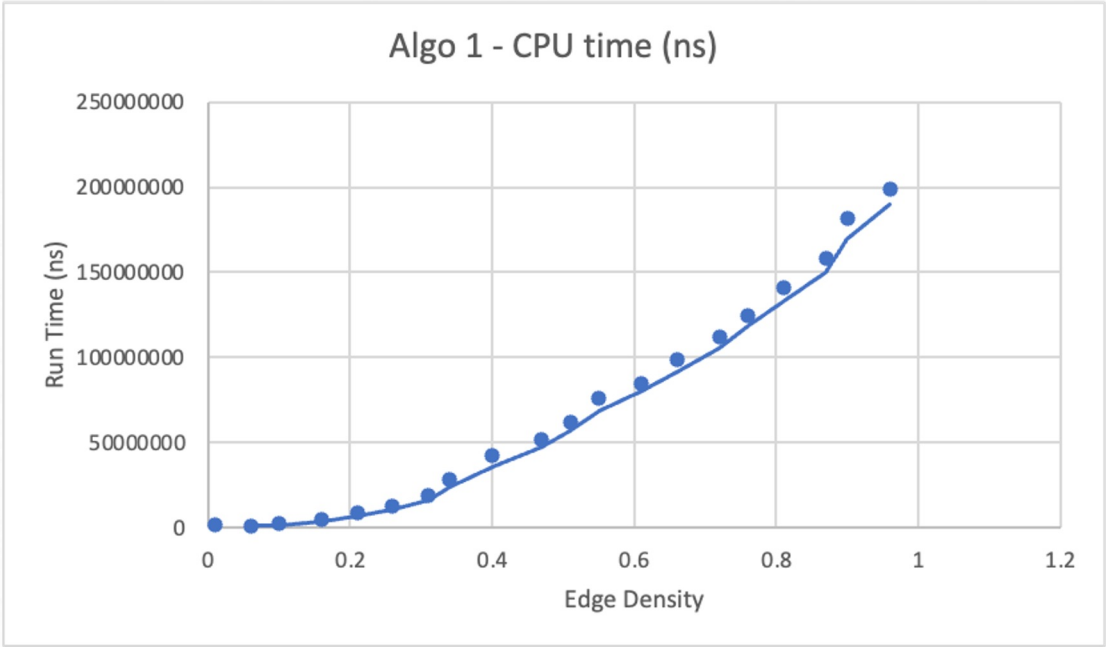

**Fixed p = 0.3**

# Empirical Time Complexity




## Fixed n = 50

# Empirical Time Complexity

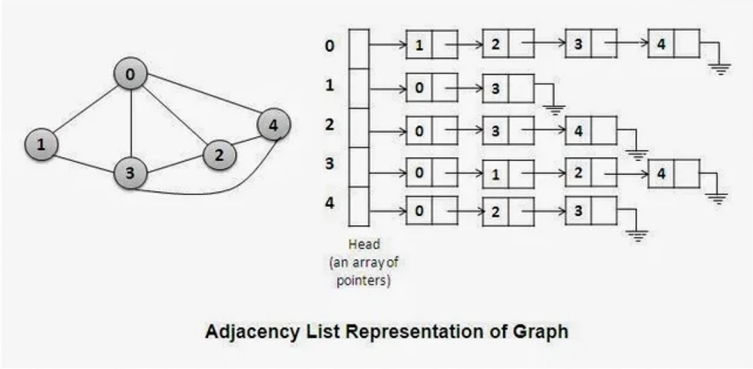



**Fixed n = 200**

# Algorithm 2:
# Adjacency List & Priority Queue
# (Minimizing Heap)

# Adjacency List

- A collection of unordered lists used to represent a finite graph
- For each vertex, a list of adjacent vertices is maintained using a linked list
- If vertex j is present in list of vertex j, vertices are adjacent (i->j)




Adjacency List Representation of Graph

*Source: https://www.thecrazyprogrammer.com/2014/03/representation-of-graphs-adjacency-matrix-and-adjacency-list.html*

# Code Implementation

## GraphAdjList

- Used for adjacency list representation of graph
- Inherits Graph class
- Stores graph in a nested list of nodes

```java
import java.util.List;

public class GraphAdjList extends Graph {
  protected List<List<Node>> graph;

  GraphAdjList(List<List<Node>> graph, int noOfVertices, int source) {
    super(noOfVertices, source);
    this.noOfEdges = setNoOfEdges(graph);
    this.graph = graph;
  }

  public int setNoOfEdges(List<List<Node>> graph) {
    int noOfEdges = 0;
    for (int i = 0; i < graph.size(); i++) {
      noOfEdges += graph.get(i).size();
    }
    return (noOfEdges);
  }
```

# Code Implementation

## GraphAdjList

- Used for adjacency list representation of graph
- Inherits Graph class
- Stores graph in a nested list of nodes

```
public void printGraph() {

  System.out.println("Source: " + source + ", Edges: " +
  noOfEdges + ", Vertices: " + noOfVertices);
  for (int i = 0; i < graph.size(); i++) {
    System.out.print(i);
    for (int j = 0; j < graph.get(i).size(); j++) {
      System.out.print(" -> " + graph.get(i).get(j).node);
    }
    System.out.println();
  }
 }
}
```

# Code Implementation

### PriorityQueue

- Class from java.util
- Implements unbounded priority queue based on a priority heap (minimizing heap by default)
- The head of this queue is the least element with respect to the specified ordering

java.util

## Class PriorityQueue<E>

java.lang.Object
  java.util.AbstractCollection<E>
    java.util.AbstractQueue<E>
      java.util.PriorityQueue<E>

**Type Parameters:**

E - the type of elements held in this collection

*Source: https://docs.oracle.com/javase/7/docs/api/java/util/PriorityQueue.html*

# Code Implementation

## DijkstraAlgo2

- Implements Dijkstra's algorithm when graph is stored in an adjacency list and minimizing heap is used for priority queue
- Uses Java's existing PriorityQueue class from java.util

```
import java.util.*;

public class DijkstraAlgo2 {
  private GraphAdjList adj;
  private int distances[];
  private int traversed[];
  private PriorityQueue<Node> priorityQueue;

  public DijkstraAlgo2(GraphAdjList adj) {
    this.adj = adj;
    distances = new int[adj.noOfVertices];
    traversed = new int[adj.noOfVertices];
    priorityQueue = new PriorityQueue<>(adj.noOfVertices, new Node());
  }
```

# Code Implementation

## DijkstraAlgo2

- Implements Dijkstra's algorithm when graph is stored in an adjacency list and minimizing heap is used for priority queue
- Uses Java's existing PriorityQueue class from java.util

```
public void shortestDistance() {

  for (int i = 0; i < adj.noOfVertices; i++) {
    distances[i] = Integer.MAX_VALUE;
    traversed[i] = 0;
  }

  priorityQueue.add(new Node(adj.source, 0));
  distances[adj.source] = 0;

  while (!priorityQueue.isEmpty()) {
    int u = priorityQueue.remove().node;
    if (traversed[u] == 0) {
      traversed[u] = 1;
      adjacentVertices(u);
    }
  }
}
```

# Code Implementation

## DijkstraAlgo2

- Implements Dijkstra's algorithm when graph is stored in an adjacency list and minimizing heap is used for priority queue
- Uses Java's existing PriorityQueue class from java.util

```
    private void adjacentVertices(int u) {
      for (int i = 0; i < adj.graph.get(u).size(); i++) {
        Node v = adj.graph.get(u).get(i);

        if (traversed[v.node] == 0 && v.cost != 0) {
          if (distances[u] + v.cost < distances[v.node]) {
            distances[v.node] = distances[u] + v.cost;
          }
          priorityQueue.add(new Node(v.node, distances[v.node]));
        }
      }
    }

    public void printResults() {
      System.out.println("Vertex \t\t Distance from Source");
      for (int i = 0; i < distances.length; i++)
        System.out.println(i + " \t\t " + distances[i]);
    }

}
```

# Theoretical Time Complexity

Time Complexity = |V| * Cost to Extract + |E| * Cost to Update

- Cost to Extract
  - fixHeap after extraction
  - **O(log|V|)**

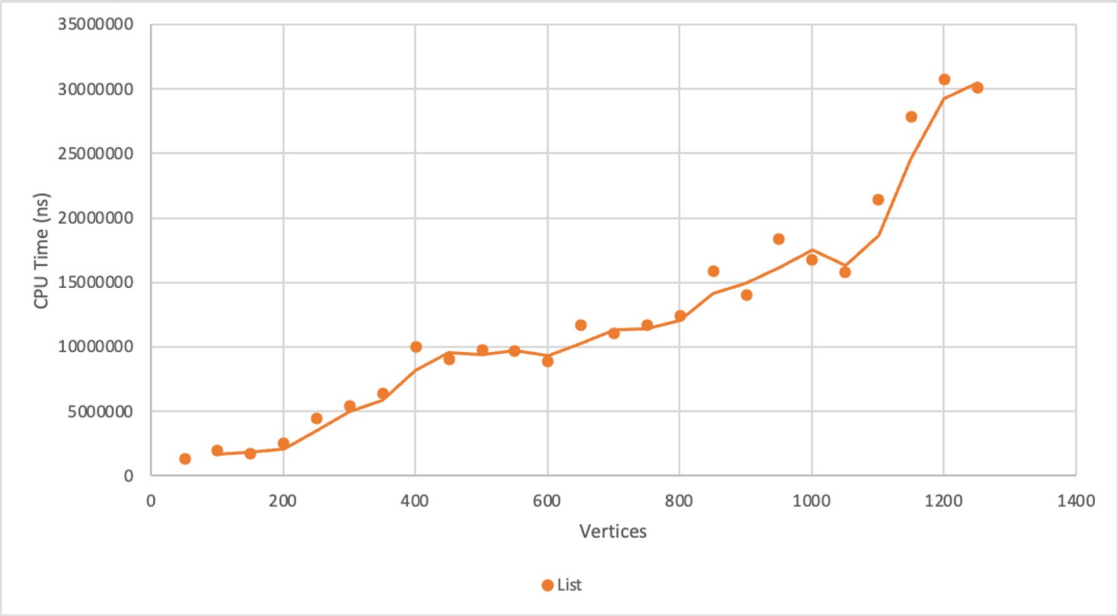
- Cost to Update
  - Remove and reinsert into heap ⇒ then fixHeap
  - **O(log|V|)**
  - Every edge is processed at most once in the worst case

Overall Time Complexity = O(|V|log|V|) + O(|E|log|V|)

= **O((|V| + |E|) log|V|)**

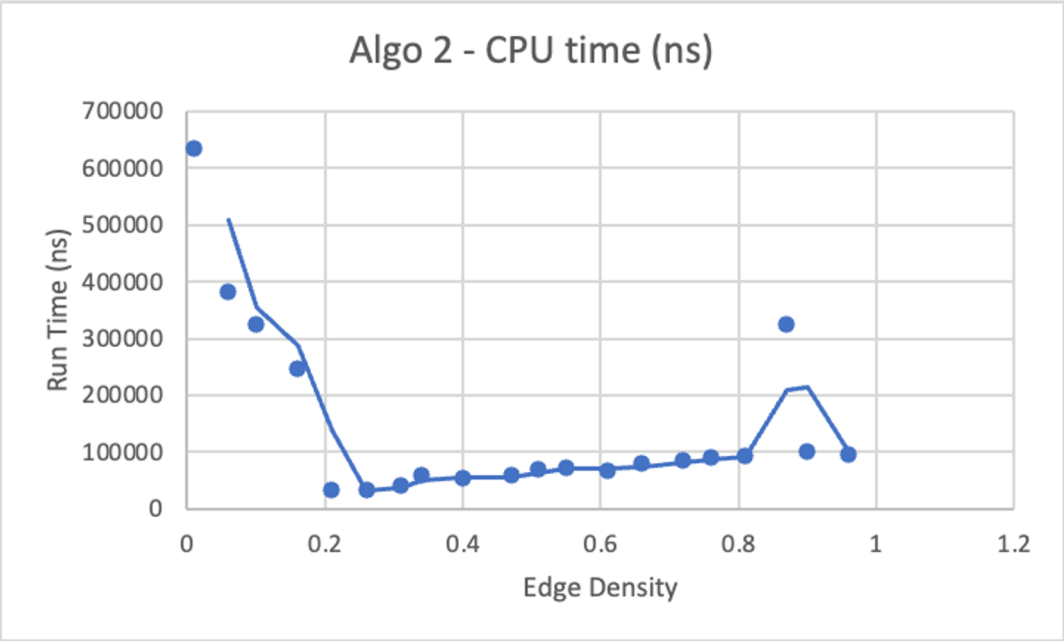
Note: If it is a connected graph, then |E| >= |V| - 1 ⇒ Overall Time Complexity = O(|E|log|V|)

# Empirical Time Complexity



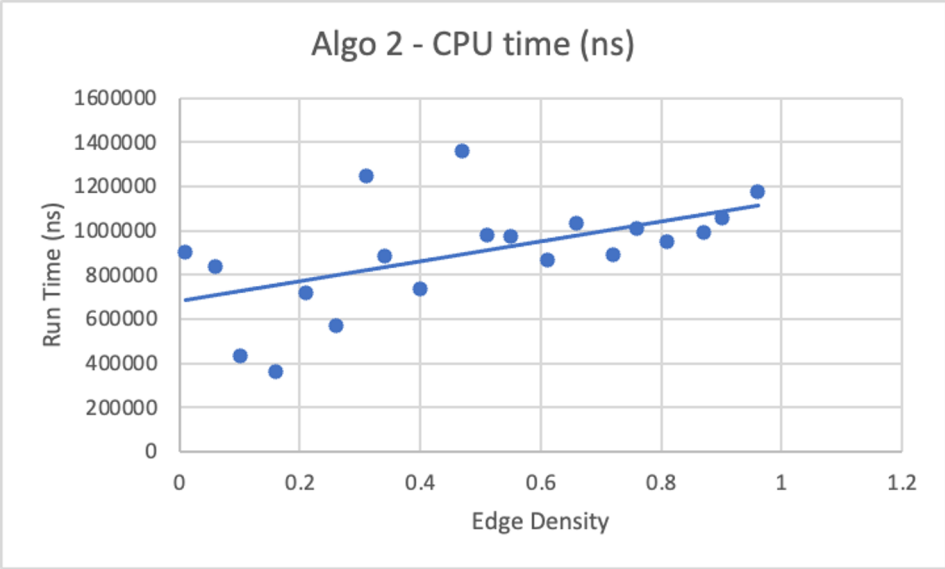

**Fixed p = 0.3**

# Empirical Time Complexity




**Fixed n = 50**

# Empirical Time Complexity




## Fixed n = 200

# Comparison of Methods

# Theoretical Comparison

$O((|V| + |E|) \log|V|) - O(|V|^2) < 0$

$(|V| + |E|) \log|V| < |V|^2$

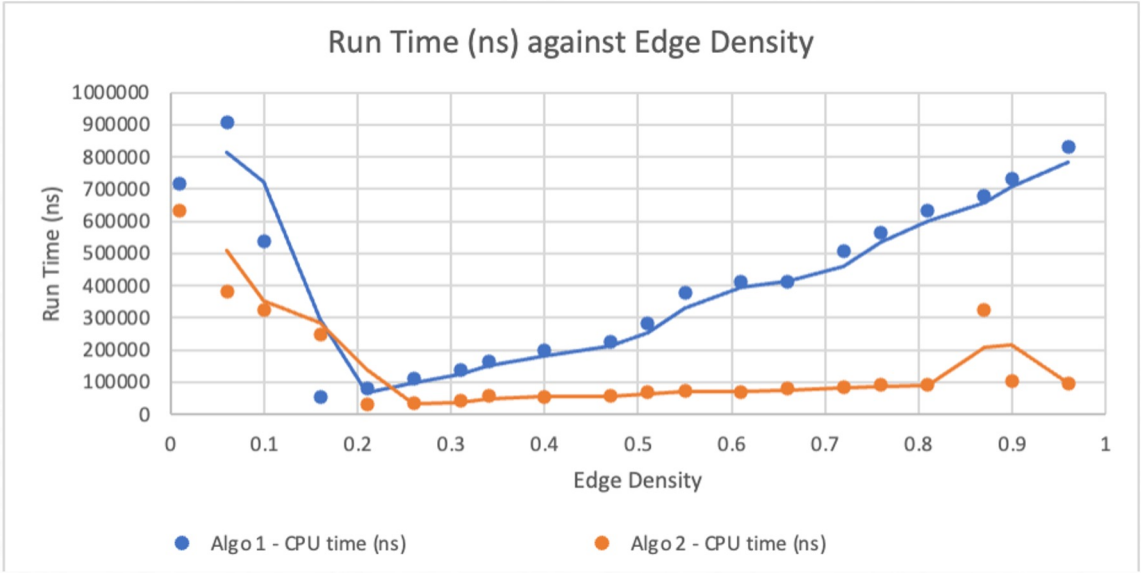
$|V| \log|V| + |E| \log|V| < |V|^2$

$|E| \log|V| < |V|^2 - |V| \log|V|$

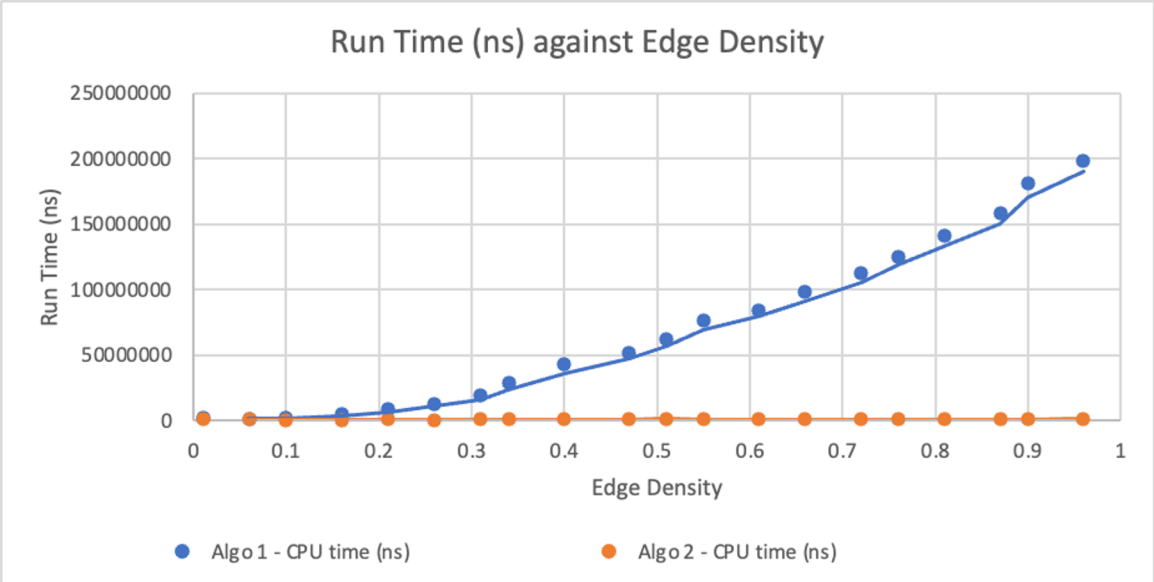
$|E| < (|V|^2 - |V| \log|V|) / \log|V|$

**$|E| < (|V|^2 / \log|V|) - |V|$**

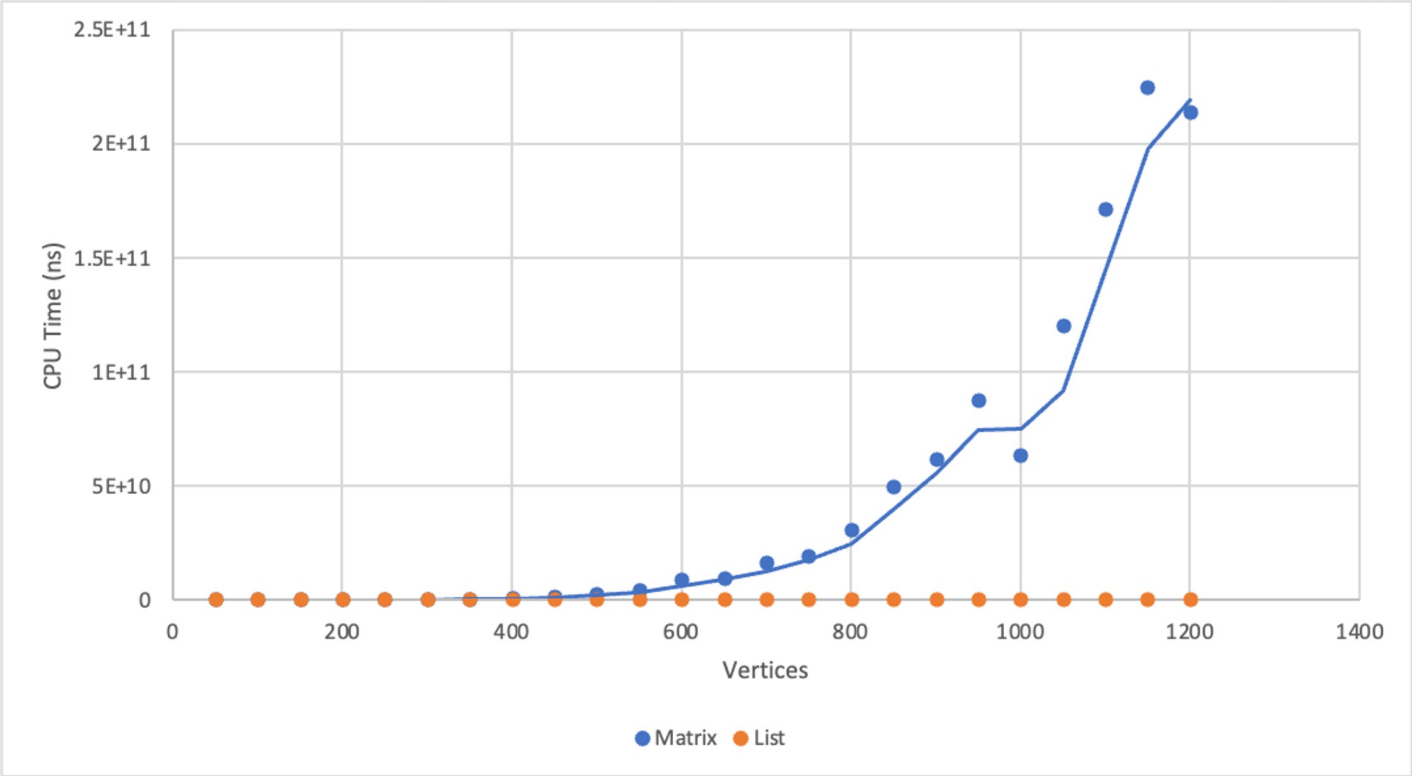
# Fixed n= 50


Run Time (ns) against Edge Density
Run Time (ns)
1000000
900000
800000
700000
600000
500000
400000
300000
200000
100000
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Edge Density
Algo 1 - CPU time (ns)
Algo 2 - CPU time (ns)

# Fixed n = 200


Run Time (ns) against Edge Density
250000000
200000000
150000000
100000000
50000000
0
Run Time (ns)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
Edge Density
Algo 1 - CPU time (ns)
Algo 2 - CPU time (ns)

# Fixed p = 0.3


2.5E+11
2E+11
1.5E+11
1E+11
5E+10
0
CPU Time (ns)
0
200
400
600
800
1000
1200
1400
Vertices
Matrix
List

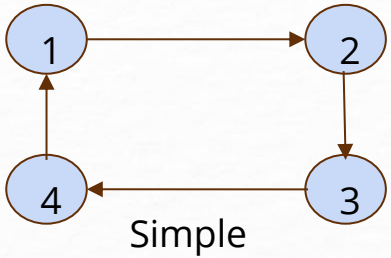
# Conclusion

# When To Use Each Algorithm

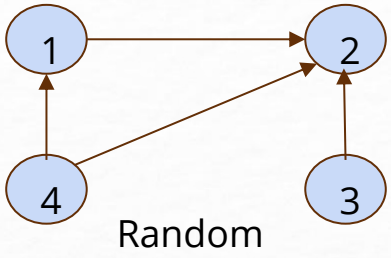
| Type of Graph | Determining Factor | Adjacency Matrix + Array Priority Queue | Adjacency List + Minimizing Heap Priority Queue |
|---|---|---|---|
| Sparse Graph<br>⇒ Fewer Edges | $V \Rightarrow C_{extract}$ | $C_{extract} = O(\|V\|)$ | ✅ ⇒ $C_{extract}$ is more efficient [$O(\log\|V\|)$] |
| Dense Graph<br>⇒ More Edges | $E \Rightarrow C_{update}$ | ✅ ⇒ $C_{update}$ is more efficient [$O(1)$] | $C_{update} = O(\log\|V\|)$ |
| Space Complexity | Adjacency Matrix VS Adjacency List | $O(\|V\|^2)$ | ✅ ⇒ $O(\|V\| + \|E\|)$<br>[except worst case: $O(\|V\|^2)$] |

# Limitations

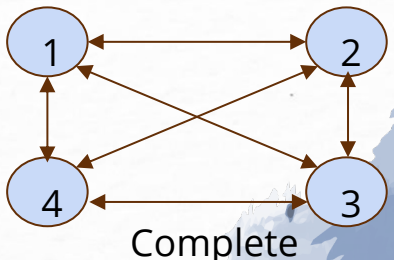
- There are 3 types of graphs that we can consider:
  - Simple graph
  - Randomly chosen edges graph
  - Complete graph
- Initial hypothesis: Array priority queue implementation would be better for dense graphs while the minimising heap implementation would be better for sparse graphs.
- Empirical results show otherwise though, since our analysis is for the average case (randomly chosen edges graphs) and thus may not be the most accurate for the best and worst cases



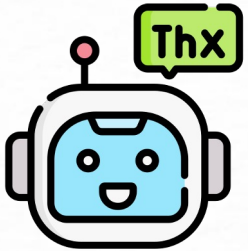
Simple



Random



Complete




# Thank you