Study the Properties of "Small World" and Compare Different Data Structures

張芫禎
資科三109102027

Dijkstra's algorithm using array

Source code : from GeeksForGeeks

```cpp
// C++ program for Dijkstra's single source shortest path algorithm.
// The program is for adjacency matrix representation of the graph.
#include <iostream>
using namespace std;
#include <limits.h>

// Number of vertices in the graph
#define V 1000

// A utility function to find the vertex with minimum distance value,
// from the set of vertices not yet included in shortest path tree.
int minDistance(int dist[], bool sptSet[]){
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}

// A utility function to print the constructed distance array.
void printSolution(int dist[]){
    cout << "Vertex \t Distance from Source" << endl;
    for (int i = 0; i < V; i++)
        cout << i << " \t\t\t\t" << dist[i] << endl;
}

// Function that implements Dijkstra's single source shortest path algorithm
// for a graph represented using adjacency matrix representation.
int dijkstra(int graph[V][V], int src, int des){
    int dist[V]; // The output array. dist[i] will hold the shortest distance from src to i.

    bool sptSet[V]; // sptSet[i] will be true if vertex i is included in
                    // shortest path tree or shortest distance from src to i is finalized.

    // Initialize all distances as INFINITE and stpSet[] as false.
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

    dist[src] = 0; // Distance of source vertex from itself is always 0.

    // Find shortest path for all vertices
    for (int count = 0; count < V - 1; count++) {
        // Pick the minimum distance vertex from the set of vertices not yet processed.
        // u is always equal to src in the first iteration.
        int u = minDistance(dist, sptSet);

        sptSet[u] = true; // Mark the picked vertex as processed.

        // Update dist value of the adjacent vertices of the picked vertex.
        for (int v = 0; v < V; v++)

            // Update dist[v] only if is not in sptSet, there is an edge from u to v,
            // and total weight of path from src to v through u is
            // smaller than current value of dist[v].
            if (!sptSet[v] && graph[u][v]
                && dist[u] != INT_MAX
                && dist[u] + graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }

    // Print the constructed distance array.
    // printSolution(dist);
    // I modify this part to return the distance to destination from source.
    return dist[des];
}
```

Implementation code :

```cpp
#include <iostream>
#include <random>
#include <fstream>
#include "array.cpp"

using namespace std;

#define X 100
#define Y 1
#define Z 1000

int main(){

    ofstream result("(100, 1, 1000)", ios::out);

    // a cycle with 1000 nodes
    int graph[V][V];
    for(int i=0; i<V; ++i){
        for(int j=0; j<V; ++j){
            if(i-j==1 || j-i==1) graph[i][j] = 1;
            else graph[i][j] = 0;
        }
    }
    graph[0][V-1] = 1;
    graph[V-1][0] = 1;

    random_device seed;
    default_random_engine generator(seed());
    uniform_int_distribution<int> vertex(0, V-1);

    // randomly add X=100 edges with the length of Y=1
    for(int i=0; i<X; ++i){
        int ver1 = vertex(generator);
        int ver2 = vertex(generator);
        if(ver1==ver2){
            --i;
            continue;
        }
        graph[ver1][ver2] = Y;
    }

    // randomly produce Z=1000 pairs of src and des to compute the shortest distance
    for(int i=0; i<Z; ++i){
        int src = vertex(generator);
        int des = vertex(generator);
        int dis = dijkstra(graph, src, des);
        result << dis << endl;
    }

    result.close();
    return 0;
}
```

Dijkstra's algorithm using binary heap

Source code : from GeeksForGeeks

```c
// C / C++ program for Dijkstra's
// shortest path algorithm for adjacency
// list representation of graph
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// A structure to represent anode in adjacency list
struct AdjListNode{
    int dest;
    int weight;
    struct AdjListNode* next;
};

// A structure to represent an adjacency list
struct AdjList{
    // Pointer to head node of list
    struct AdjListNode *head;
};

// A structure to represent a graph.
// A graph is an array of adjacency lists.
// Size of array will be V (number of vertices in graph)
struct Graph{
    int V;
    struct AdjList* array;
};

// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest, int weight){
    struct AdjListNode* newNode =
        (struct AdjListNode*) malloc(sizeof(struct AdjListNode));
    newNode->dest = dest;
    newNode->weight = weight;
    newNode->next = NULL;
    return newNode;
}

// A utility function that creates a graph of V vertices
struct Graph* createGraph(int V){
    struct Graph* graph = (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;

    // Create an array of adjacency lists.
    // Size of array will be V
    graph->array = (struct AdjList*) malloc(V * sizeof(struct AdjList));

    // Initialize each adjacency list as empty by making head as NULL
    for (int i = 0; i < V; ++i)
        graph->array[i].head = NULL;

    return graph;
}
```

```c
// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest, int weight){
    // Add an edge from src to dest.
    // A new node is added to the adjacency list of src.
    // The node is added at the beginning
    struct AdjListNode* newNode = newAdjListNode(dest, weight);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;

    // Since graph is undirected, add an edge from dest to src also
    newNode = newAdjListNode(src, weight);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}

// Structure to represent a min heap node
struct MinHeapNode{
    int v;
    int dist;
};

// Structure to represent a min heap
struct MinHeap{

    // Number of heap nodes present currently
    int size;

    // Capacity of min heap
    int capacity;

    // This is needed for decreaseKey()
    int *pos;
    struct MinHeapNode **array;
};

// A utility function to create a new Min Heap Node
struct MinHeapNode* newMinHeapNode(int v, int dist){
    struct MinHeapNode* minHeapNode =
        (struct MinHeapNode*)malloc(sizeof(struct MinHeapNode));
    minHeapNode->v = v;
    minHeapNode->dist = dist;
    return minHeapNode;
}

// A utility function to create a Min Heap
struct MinHeap* createMinHeap(int capacity){
    struct MinHeap* minHeap =
        (struct MinHeap*) malloc(sizeof(struct MinHeap));
    minHeap->pos = (int *) malloc(capacity * sizeof(int));
    minHeap->size = 0;
    minHeap->capacity = capacity;
    minHeap->array =
        (struct MinHeapNode**) malloc(capacity *sizeof(struct MinHeapNode*));
    return minHeap;
}
```

```c
// A utility function to swap two nodes of min heap.
// Needed for min heapify
void swapMinHeapNode(struct MinHeapNode** a, struct MinHeapNode** b){
    struct MinHeapNode* t = *a;
    *a = *b;
    *b = t;
}

// A standard function to heapify at given idx
// This function also updates position of nodes when they are swapped.
// Position is needed for decreaseKey()
void minHeapify(struct MinHeap* minHeap, int idx){
    int smallest, left, right;
    smallest = idx;
    left = 2 * idx + 1;
    right = 2 * idx + 2;

    if (left < minHeap->size &&
        minHeap->array[left]->dist < minHeap->array[smallest]->dist )
            smallest = left;

    if (right < minHeap->size &&
        minHeap->array[right]->dist < minHeap->array[smallest]->dist )
            smallest = right;

    if (smallest != idx){
        // The nodes to be swapped in min heap
        MinHeapNode *smallestNode = minHeap->array[smallest];
        MinHeapNode *idxNode = minHeap->array[idx];

        // Swap positions
        minHeap->pos[smallestNode->v] = idx;
        minHeap->pos[idxNode->v] = smallest;

        // Swap nodes
        swapMinHeapNode(&minHeap->array[smallest], &minHeap->array[idx]);

        minHeapify(minHeap, smallest);
    }
}

// A utility function to check if the given minHeap is empty or not
int isEmpty(struct MinHeap* minHeap){
    return minHeap->size == 0;
}
```

```c
// Standard function to extract minimum node from heap
struct MinHeapNode* extractMin(struct MinHeap* minHeap){
    if (isEmpty(minHeap))
        return NULL;

    // Store the root node
    struct MinHeapNode* root = minHeap->array[0];

    // Replace root node with last node
    struct MinHeapNode* lastNode = minHeap->array[minHeap->size - 1];
    minHeap->array[0] = lastNode;

    // Update position of last node
    minHeap->pos[root->v] = minHeap->size-1;
    minHeap->pos[lastNode->v] = 0;

    // Reduce heap size and heapify root
    --minHeap->size;
    minHeapify(minHeap, 0);

    return root;
}

// Function to decreasekey dist value of a given vertex v.
// This function uses pos[] of min heap to get the current index of node in min heap
void decreaseKey(struct MinHeap* minHeap, int v, int dist){

    // Get the index of v in heap array
    int i = minHeap->pos[v];

    // Get the node and update its dist value
    minHeap->array[i]->dist = dist;

    // Travel up while the complete tree is not heapified.
    // This is a O(Logn) loop
    while (i && minHeap->array[i]->dist < minHeap->array[(i - 1) / 2]->dist){
        // Swap this node with its parent
        minHeap->pos[minHeap->array[i]->v] = (i-1)/2;
        minHeap->pos[minHeap->array[(i-1)/2]->v] = i;
        swapMinHeapNode(&minHeap->array[i], &minHeap->array[(i - 1) / 2]);

        // move to parent index
        i = (i - 1) / 2;
    }
}

// A utility function to check if a given vertex 'v' is in min heap or not
bool isInMinHeap(struct MinHeap *minHeap, int v){
    if (minHeap->pos[v] < minHeap->size)
        return true;
    return false;
}

// A utility function used to print the solution
void printArr(int dist[], int n){
    printf("Vertex Distance from Source\n");
    for (int i = 0; i < n; ++i)
        printf("%d \t\t %d\n", i, dist[i]);
}
```

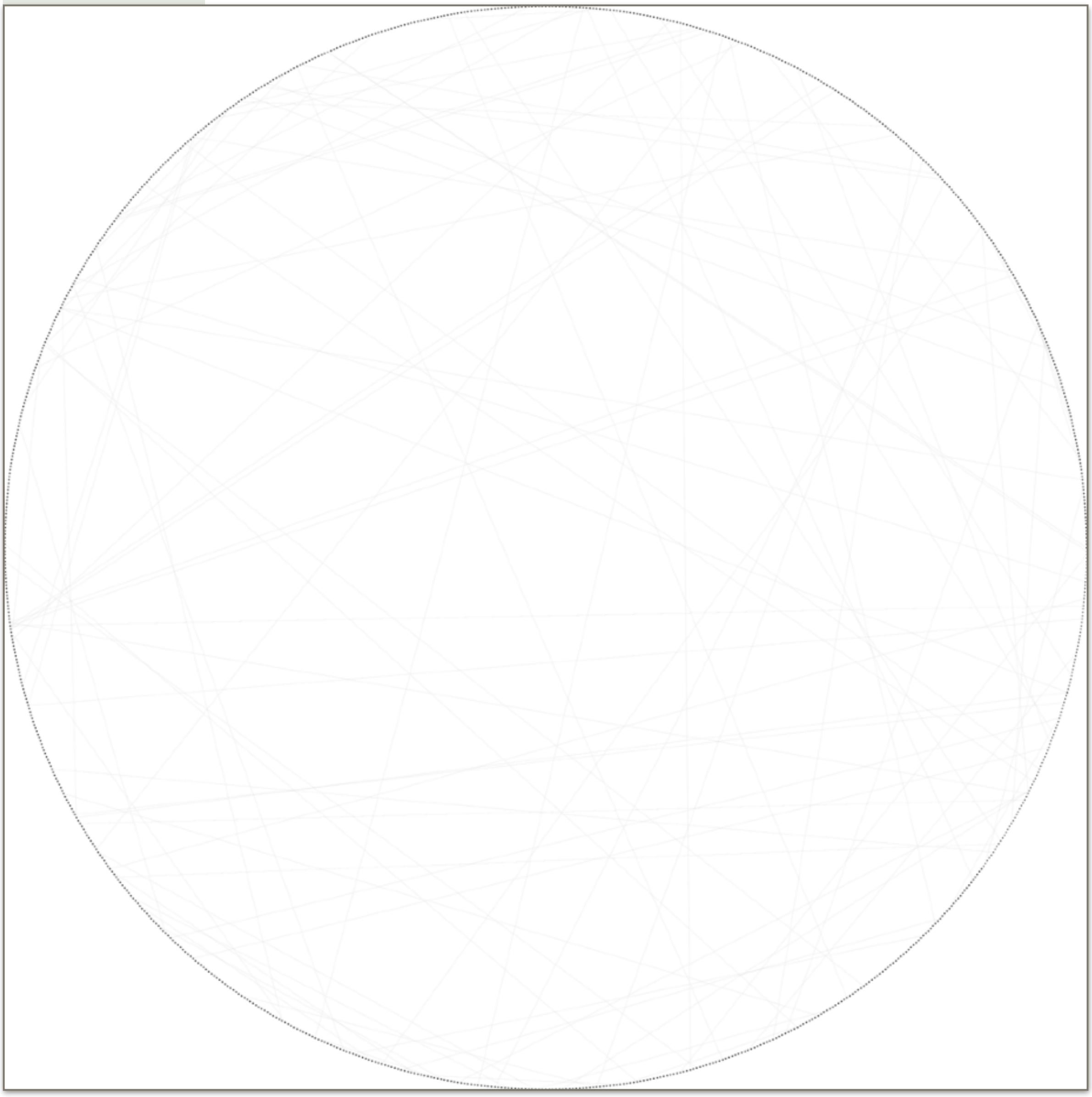
```c
// The main function that calculates distances of shortest paths from src to all vertices.
// It is a O(ELogV) function
int dijkstra(struct Graph* graph, int src, int des){

    // Get the number of vertices in graph
    int V = graph->V;

    // dist values used to pick minimum weight edge in cut
    int dist[V];

    // minHeap represents set E
    struct MinHeap* minHeap = createMinHeap(V);

    // Initialize min heap with all vertices.
    //  dist value of all vertices
    for (int v = 0; v < V; ++v){
        dist[v] = INT_MAX;
        minHeap->array[v] = newMinHeapNode(v, dist[v]);
        minHeap->pos[v] = v;
    }

    // Make dist value of src vertex as 0 so that it is extracted first
    minHeap->array[src] = newMinHeapNode(src, dist[src]);
    minHeap->pos[src] = src;
    dist[src] = 0;
    decreaseKey(minHeap, src, dist[src]);

    // Initially size of min heap is equal to V
    minHeap->size = V;

    // In the following loop, min heap contains all nodes
    // whose shortest distance is not yet finalized.
    while (!isEmpty(minHeap)){
        // Extract the vertex with minimum distance value
        struct MinHeapNode* minHeapNode = extractMin(minHeap);

        // Store the extracted vertex number
        int u = minHeapNode->v;

        // Traverse through all adjacent  vertices of u (the extracted vertex)
        // and update their distance values
        struct AdjListNode* pCrawl = graph->array[u].head;
        while (pCrawl != NULL){
            int v = pCrawl->dest;

            // If shortest distance to v is not finalized yet,
            // and distance to v through u is less than
            // its previously calculated distance
            if (isInMinHeap(minHeap, v) && dist[u] != INT_MAX &&
                pCrawl->weight + dist[u] < dist[v]){
                    dist[v] = dist[u] + pCrawl->weight;

                // update distance  value in min heap also
                decreaseKey(minHeap, v, dist[v]);
            }
            pCrawl = pCrawl->next;
        }
    }

    // print the calculated shortest distances
    // printArr(dist, V);
    // i moditied this part to return the distance to destination from source
    return dist[des];
}
```

Implementation code :

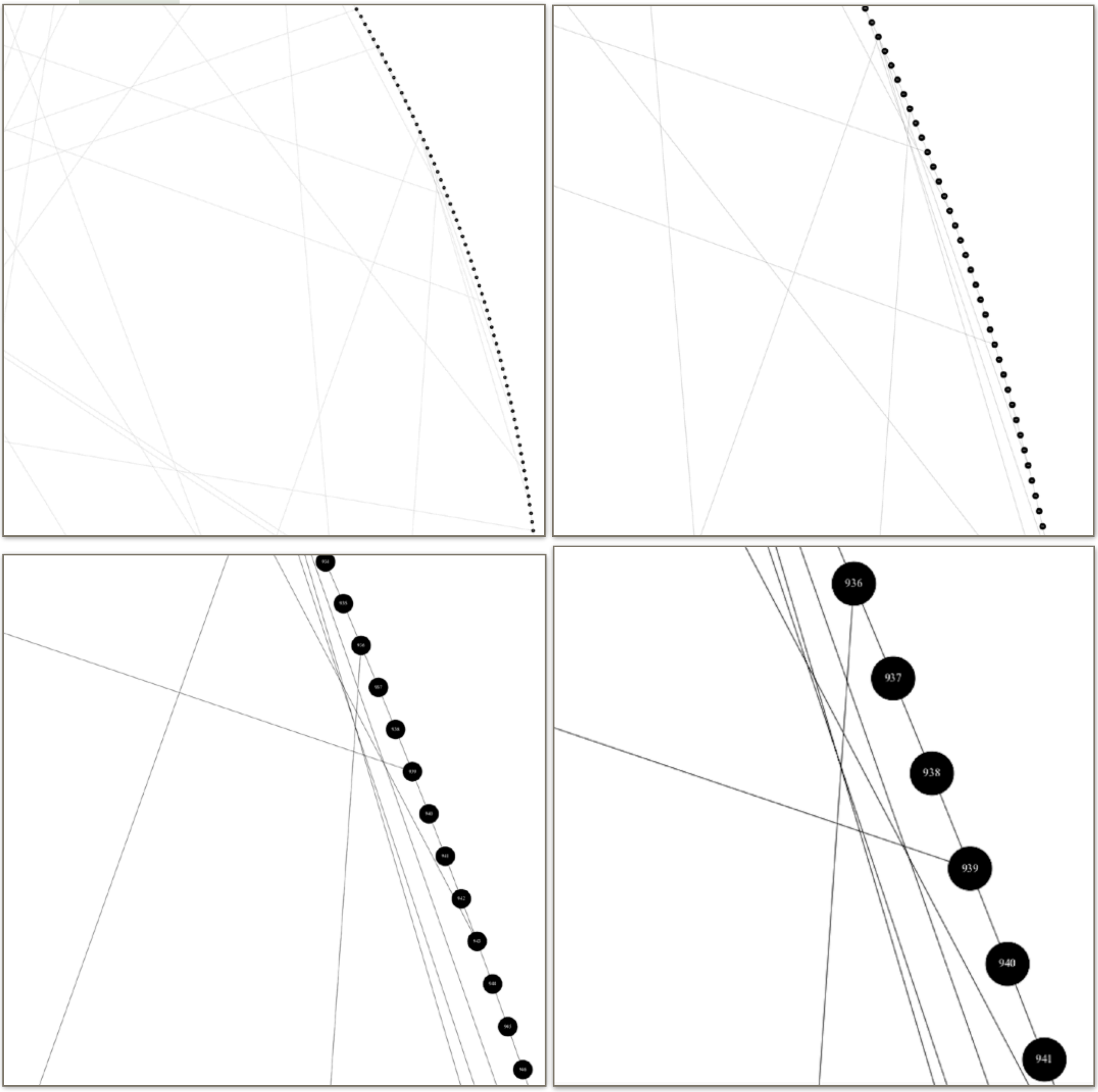
```cpp
#include <iostream>
#include <random>
#include <fstream>
#include "array.cpp"

using namespace std;

#define X 100
#define Y 1
#define Z 1000

int main(){

    ofstream result("(100, 1, 1000)", ios::out);

    // a cycle with 1000 nodes
    int graph[V][V];
    for(int i=0; i<V; ++i){
        for(int j=0; j<V; ++j){
            if(i-j==1 || j-i==1) graph[i][j] = 1;
            else graph[i][j] = 0;
        }
    }
    graph[0][V-1] = 1;
    graph[V-1][0] = 1;

    random_device seed;
    default_random_engine generator(seed());
    uniform_int_distribution<int> vertex(0, V-1);

    // randomly add X=100 edges with the length of Y=1
    for(int i=0; i<X; ++i){
        int ver1 = vertex(generator);
        int ver2 = vertex(generator);
        if(ver1==ver2){
            --i;
            continue;
        }
        graph[ver1][ver2] = Y;
    }

    // randomly produce Z=1000 pairs of src and des to compute the shortest distance
    for(int i=0; i<Z; ++i){
        int src = vertex(generator);
        int des = vertex(generator);
        int dis = dijkstra(graph, src, des);
        result << dis << endl;
    }

    result.close();
    return 0;
}
```

1. A picture of the graph with $x = 100$

Using Graphviz :



Zoom in :



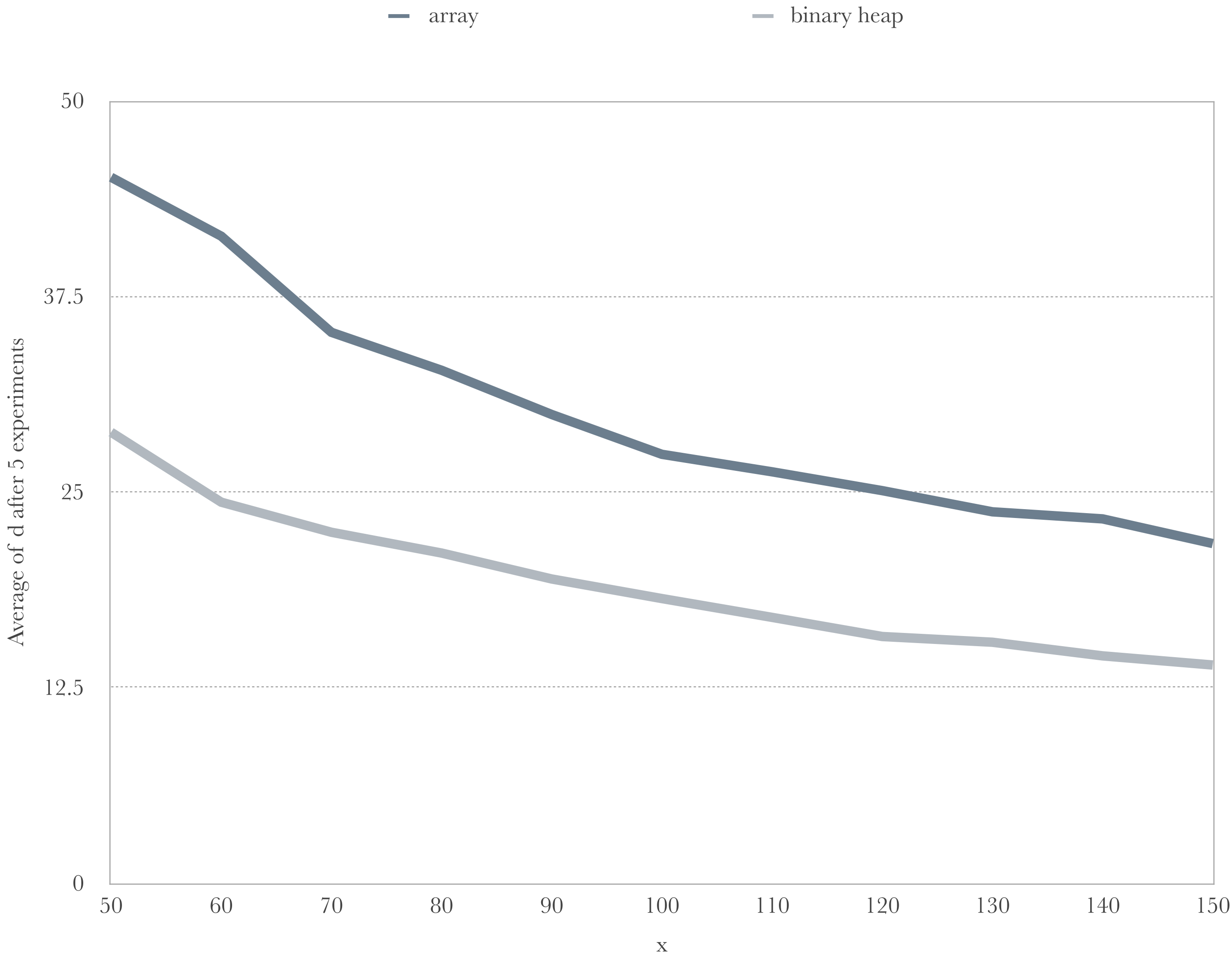
2. **Responses to the following questions:**

- **What is the relationship between x and d?**

  d is inversely proportional to x as the table and the graph below indicate.

  I execute the code 5 times for each of the x and calculate the average of results as shown below.

(X, Y, Z)		(50, 1, 1000)	(60, 1, 1000)	(70, 1, 1000)	(80, 1, 1000)	(90, 1, 1000)	(100, 1, 1000)	(110, 1, 1000)	(120, 1, 1000)	(130, 1, 1000)	(140, 1, 1000)	(150, 1, 1000)
Average of d	Using array	45.1146	41.3488	35.2056	32.782	29.955	27.4114	26.294	25.091	23.7378	23.2866	21.7082
	Using binary heap	28.8242	24.3526	22.4314	21.1038	19.4458	18.1948	16.9928	15.7704	15.405	14.534	13.9492

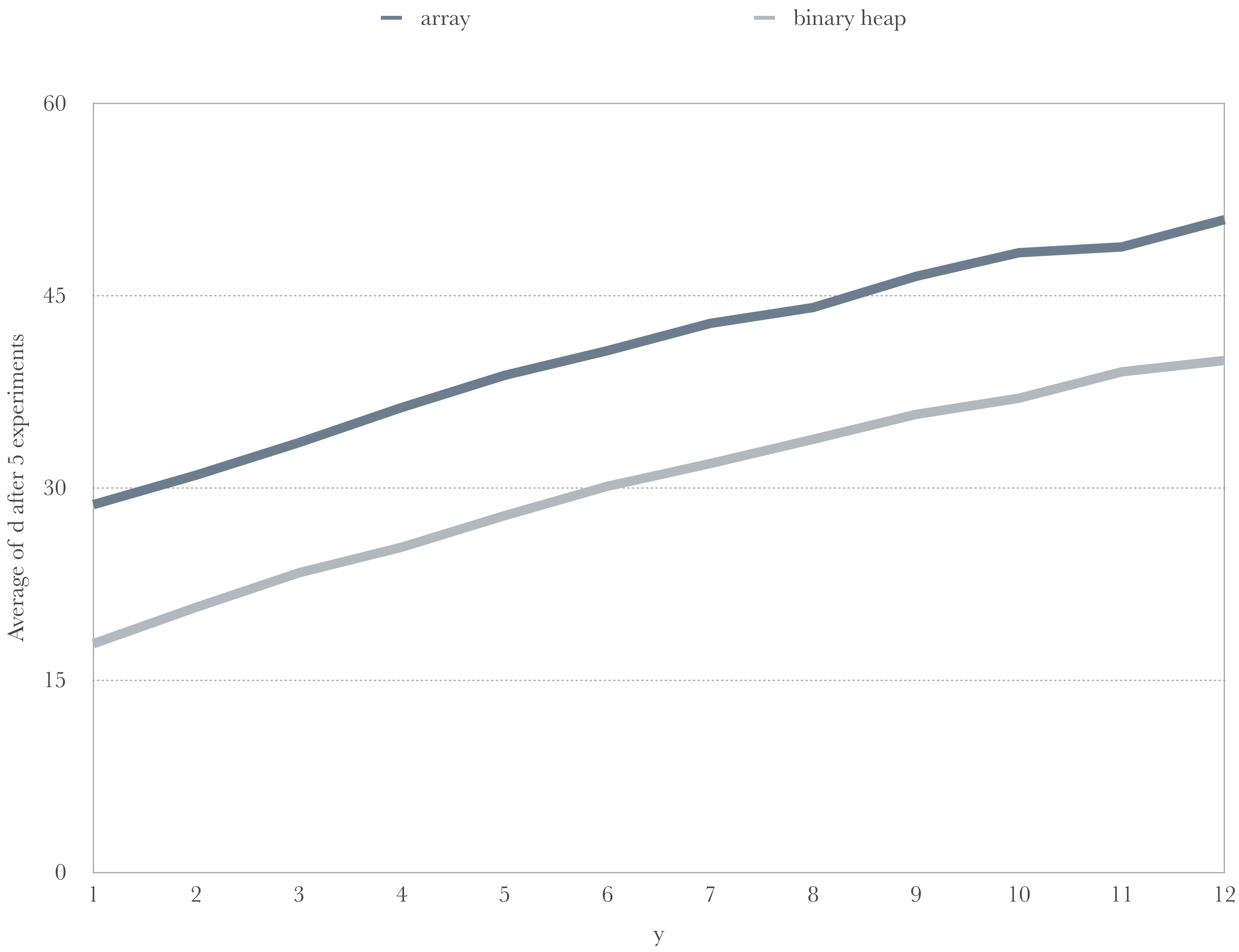


- **What is the relationship between y and d?**

  d is proportional to y as the table and the graph below indicate.

  I execute the code 5 times for each of the y and calculate the average of results as shown below.

(X, Y, Z)		(100, 1, 1000)	(00, 2, 1000)	(100, 3, 1000)	(100, 4, 1000)	(100, 5, 1000)	(100, 6, 1000)	(100, 7, 1000)	(100, 8, 1000)	(100, 9, 1000)	(100, 10, 1000)	(100, 11, 1000)	(100, 12, 1000)
Average of d	Using array	28.6866	30.9884	33.5266	36.2682	38.7766	40.702	42.8314	44.067	46.4982	48.343	48.795	50.9252
	Using binary heap	17.8338	20.674	23.3684	25.3668	27.8286	30.121	31.8932	33.7842	35.7268	36.9882	39.0484	39.9358



- **How to choose z to obtain a reasonable approximation of the true average shortest distance between all pairs of source and destination?**

  Under the condition of x=100 and y=1, I implement the experiment 4 times with z=10000, z=1000, and z=100 respectively and have the results as below. We can see that d roughly lies between 25 and 30. It's obvious that the statistics is quite consistent when z=1000 but moves more intensively when z=100. To exert this experiment effectively and precisely, in the course of this report, z is fixed to be 1000.

Choice of z

(X, Y, Z)	(100, 1, 10000)				(100, 1 1000)				(100, 1, 100)			
d	27.1468	28.8317	26.4294	28.1732	29.146	29.368	29.663	28.847	28.67	28.88	30.3	25.62
	37	23	29	41	42	41	35	29	22	18	21	21
	17	24	15	32	21	33	18	22	35	3	48	42
	24	32	33	36	20	32	29	25	33	33	30	23
	34	30	32	22	13	52	37	17	33	20	27	23
	11	23	31	24	30	24	24	9	12	23	17	21
	37	23	4	26	30	11	48	27	4	26	13	44
	33	26	38	25	31	44	4	32	26	21	37	23
	34	51	26	27	37	43	32	12	17	26	16	24
	36	35	26	26	43	8	11	25	15	16	23	7
	19	29	42	24	10	36	40	60	17	27	22	24
	19	20	21	33	31	46	32	36	46	10	28	18
	35	39	34	34	23	36	41	33	26	39	32	23
	25	18	32	22	31	35	31	31	56	53	20	24
	32	34	26	44	24	12	25	23	32	35	37	10
	17	35	16	24	26	30	39	25	31	27	22	39
	40	21	41	18	26	21	14	11	44	33	30	17
	23	12	33	34	34	24	27	40	33	26	12	18
	20	15	31	35	23	40	27	10	28	31	26	18
	22	36	30	33	26	47	20	25	46	23	35	26
	27	30	32	23	32	34	12	23	21	31	40	30
	31	15	22	21	18	27	43	30	36	30	31	41
	18	53	16	24	36	37	18	59	20	45	17	27
	31	35	24	24	24	30	13	19	26	24	19	25
	28	18	43	61	30	32	25	24	36	64	32	35

- **Which implementation of Dijkstra's Algorithm is faster?**

  The one using binary heap. I measure the execution time needed for the Dijkstra function and calculate the average time of the 1000 pairs of source-and-destination.

```
clock_t start, end;
    // randomly produce Z=1000 pairs of src and des to compute the shortest distance
    for(int i=0; i<Z; ++i){
        int src = vertex(generator);
        int des = vertex(generator);
        start = clock();
        int dis = dijkstra(graph, src, des);
        end = clock();
        result << (float)(end-start)/CLOCKS_PER_SEC << endl;
    }
```

(X, Y, Z)		Time complexity	(50, 1, 1000)	(60, 1, 1000)	(70, 1, 1000)	(80, 1, 1000)	(90, 1, 1000)	(100, 1, 1000)	(110, 1, 1000)	(120, 1, 1000)	(130, 1, 1000)
Average execution time	Using array	$O(V^2)$	0.00417718	0.00418497	0.00442315	0.00422114	0.00398796	0.00366231	0.00469442	0.00488219	0.00497297
	Using binary heap	$O(E \log V)$	0.00028459	0.00028147	0.00026019	0.00029081	0.00028061	0.00027095	0.00032536	0.00031041	0.00028983