Department of Informatics
Database Technology Group
University of Zurich Prof. Dr. Anton Dignös

Informatics II
Exercise 12

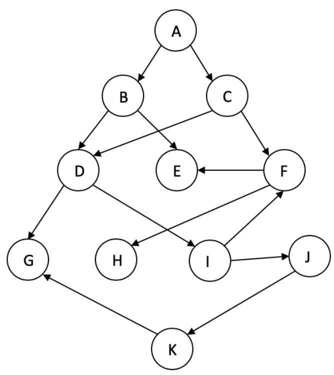
May 17, 2020

Goals:

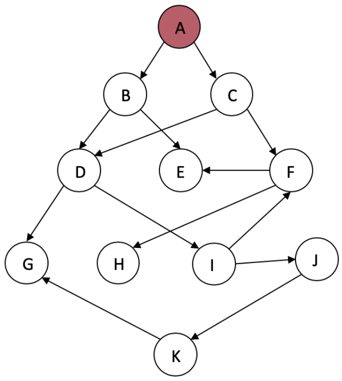
- Practice DFS and BFS
- Implementing a search algorithm in a 2-D array.
- Understand alternative graph representations and discuss them

Graphs(BFS, DFS)

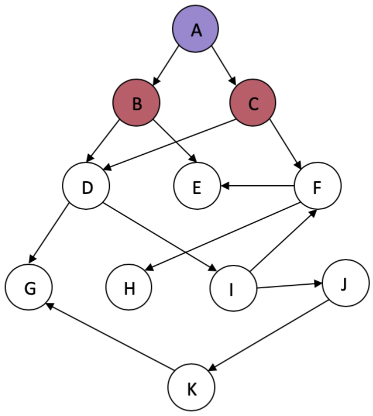
Task 1. In the given graph below, each vertex has an unique label. For example, we use vertex A to denote the vertex with label "A". Write a breadth first search (BFS) starts at vertex A using a queue. In this task, during the BFS search, neighbors of a vertex are visited in the alphabetical order of their labels. For example, in the BFS that starts at vertex A, vertex B is visited before vertex C. The first two steps of the solution are shown below.



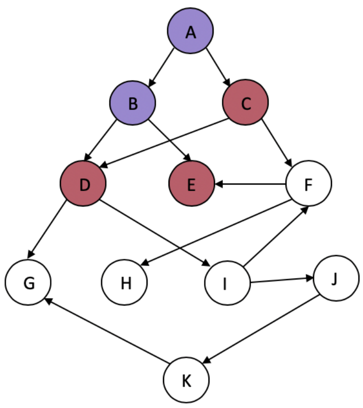
Solution: Vertices colored in blue are already visited by the BFS, and vertices colored in red are in queue and to be visitedby the BFS. Rememebr that the first-in-first-out principle in Queue.



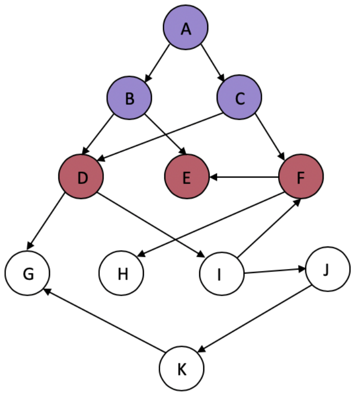
Queue: A



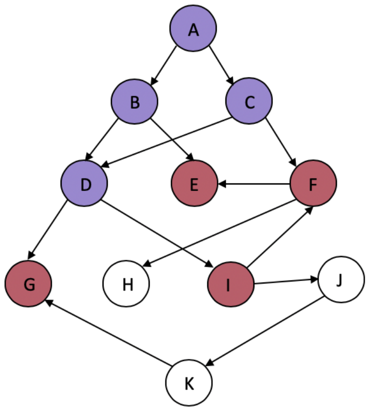
Queue: B, C



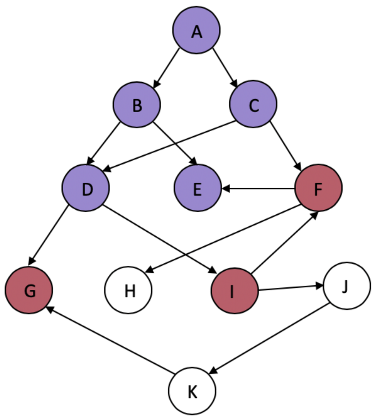
Queue: C, D, E



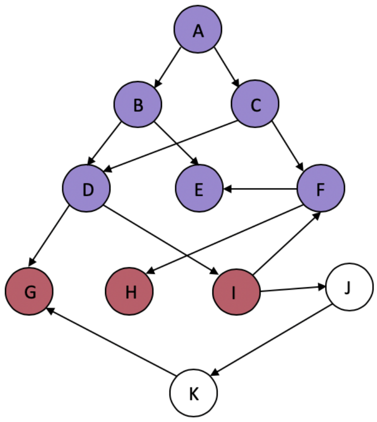
Queue: D, E, F



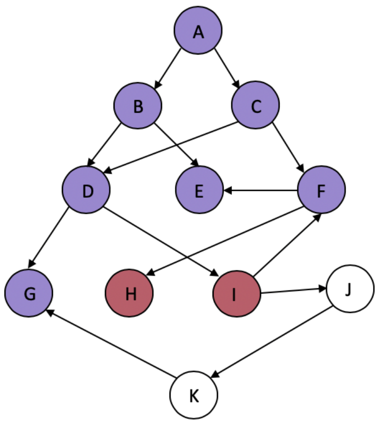
Queue: E, F, G, I



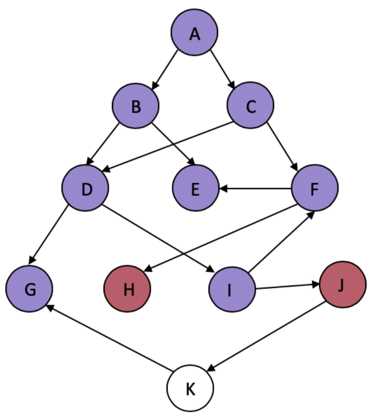
Queue: F, G, I



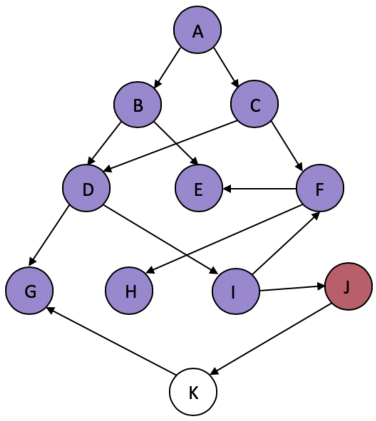
Queue: G, I, H



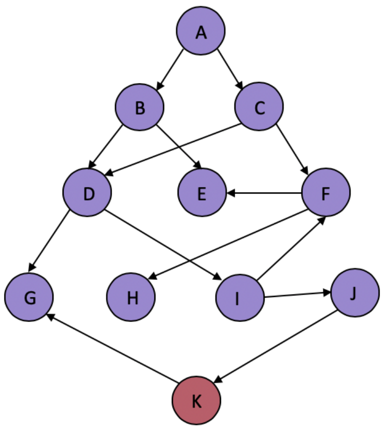
Queue: I, H



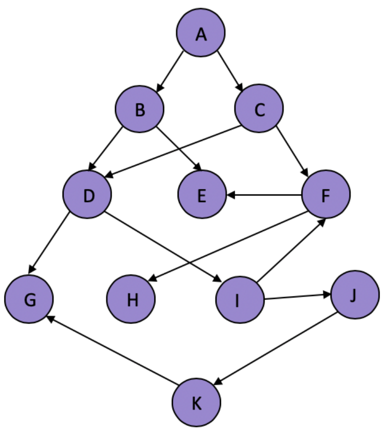
Queue: H, J



Queue: J



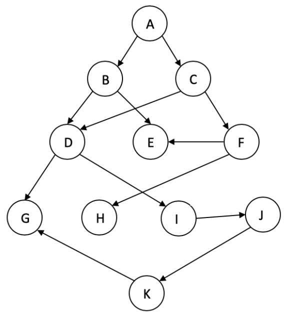
Queue: K



Queue:

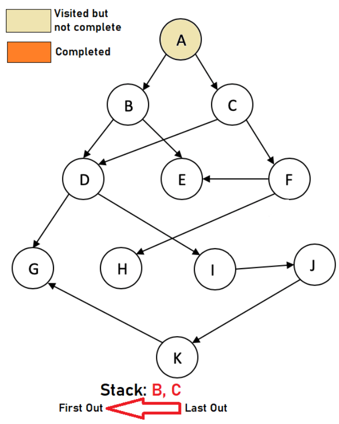
The BFS search starts at vertex A is A, B, C, D, E, F, G, I, H, J, K

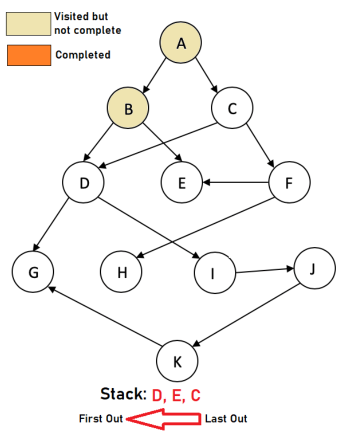
Task 2. Given a graph below, each vertex has an unique label. Write the depth first search (DFS) starts at vertex A using a stack. In this task, during the DFS search, neighbors of a vertex are traversed in the alphabetical order of their labels. For example, in the DFS that starts from vertex A, vertex B is visited before vertex C. Note that the recursive solution of DFS is different from the one using a stack.

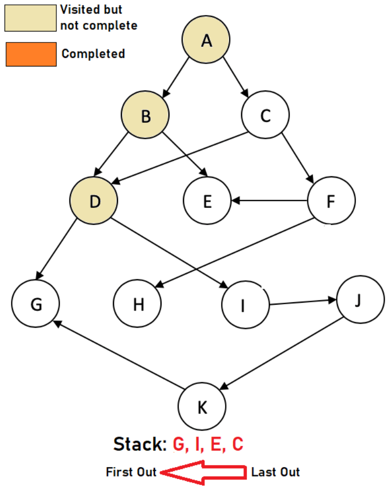


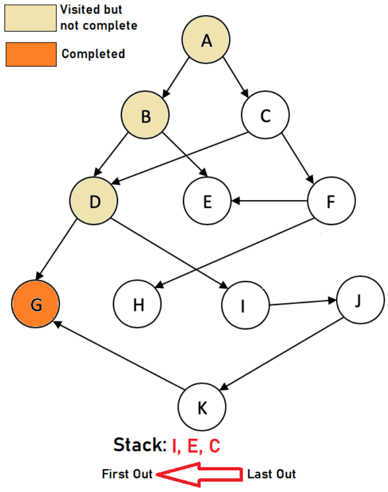
If all neighbors of a vertex are visted by the DFS, we highlight this vertex in orange (Completed). Vertices whose neighbors are not completely visited are colored in light green. Vertices without colors are not visited yet.

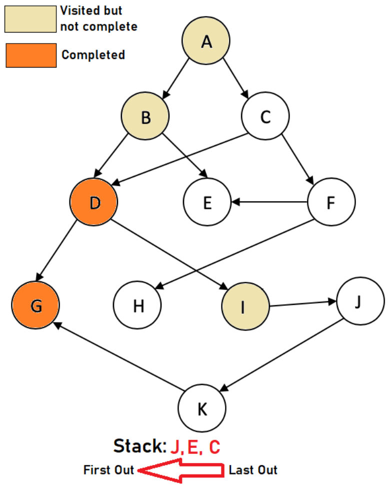
Solution

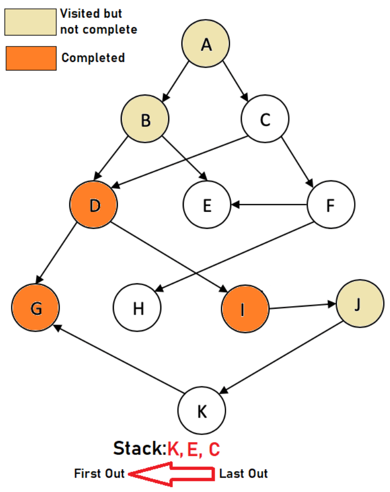


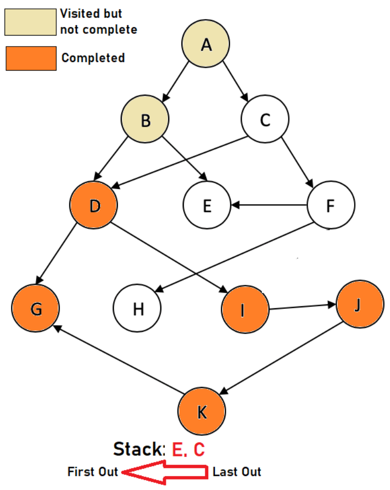


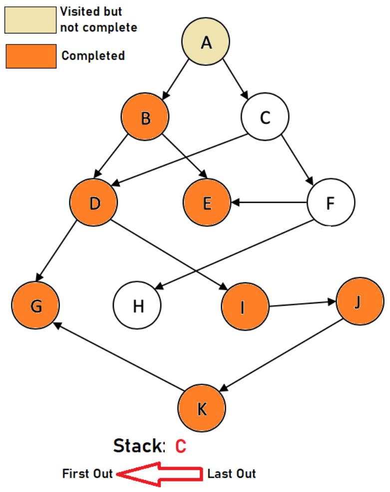

Visited but not complete
Completed
A
B
C
D
E
F
G
H
I
J
K
Stack: G, I, E, C
First Out
Last Out

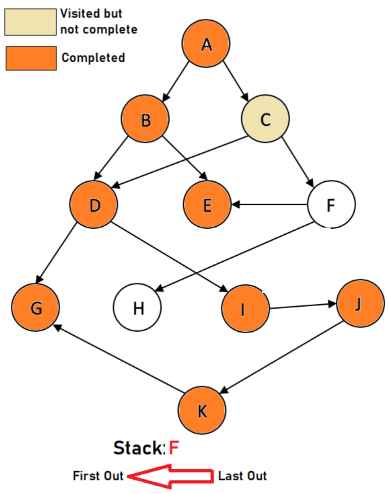

Visited but not complete
Completed
A
B
C
D
E
F
G
H
I
J
K
Stack: I, E, C
First Out
Last Out

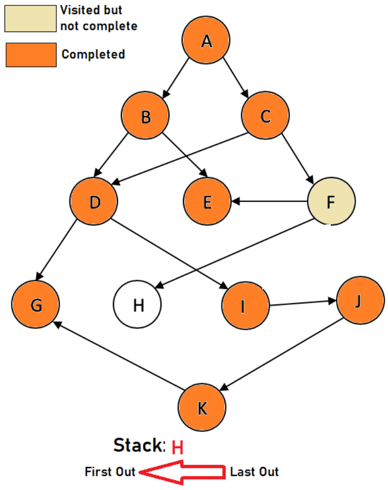

Visited but not complete
Completed
A
B
C
D
E
F
G
H
I
J
K
Stack: J, E, C
First Out
Last Out

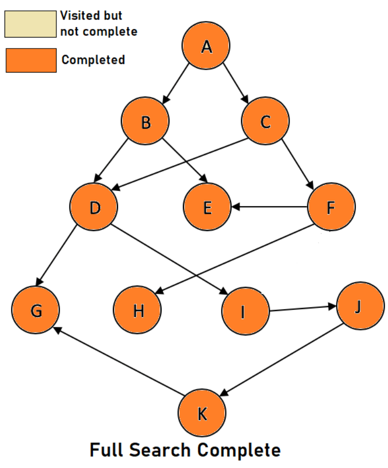

Visited but not complete
Completed
A
B
C
D
E
F
G
H
I
J
K
Stack:K, E, C
First Out
Last Out


Visited but not complete
Completed
A
B
C
D
E
F
G
H
I
J
K
Stack: E, C
First Out
Last Out


Visited but not complete
Completed
A
B
C
D
E
F
G
H
I
J
K
Stack: C
First Out
Last Out







The order in which nodes will be visited is: A, B, D, G, I, J, K, E, C, F, H

Task 3. This task is about implementing the depth-first search (DFS) on graphs. Graphs are represented by adjacency lists.

The following code snippet is everything you need for your graph:

```c
struct node {
  int vertex;
  struct node* next;
};

struct node* createNode(int v);

struct Graph {
  int numVertices;
  // We need int** to store a two dimensional array.
  // Similary, we need struct node** to store an array of Linked lists
  struct node** adjLists;
};


struct node* createNode(int v) {
  struct node* newNode = malloc(sizeof(struct node));
  newNode->vertex = v;
  newNode->next = NULL;
  return newNode;
}


struct Graph* createGraph(int vertices) {
  struct Graph* graph = malloc(sizeof(struct Graph));
  graph->numVertices = vertices;

  graph->adjLists = malloc(vertices * sizeof(struct node*));

  int i;
  for (i = 0; i < vertices; i++) {
    graph->adjLists[i] = NULL;
  }
  return graph;
}

void addEdge(struct Graph* graph, int src, int dest) {
  // Add edge from src to dest
  struct node* newNode = createNode(dest);
  newNode->next = graph->adjLists[src];
  graph->adjLists[src] = newNode;

  // Add edge from dest to src
  newNode = createNode(src);
  newNode->next = graph->adjLists[dest];
  graph->adjLists[dest] = newNode;
}
```

Implement the function `void DFS(struct node** graph, int start)` that prints a DFS on the Graph from start vertex. Hint: You can find Advanced Data Structure implementations on previous exercise sheets.

```c
int stack[STACK_SIZE];
int s_top = -1;

void stack_push(int value)
{
    if(s_top < STACK_SIZE-1)
    {
        if (s_top < 0)
        {
            stack[0] = value;
            s_top = 0;
        }
        else
        {
            stack[s_top+1] = value;
            s_top++;
        }
    }
    else
    {
        printf("Stackoverflow!!!!\n");
    }
}

int stack_isempty()
{
    return s_top < 0;
}

int stack_pop()
{
    if(!stack_isempty())
    {
        int n = stack[s_top];
        s_top--;
        return n;
    }
    else
    {
        printf("Error: the stack is empty!\n");
        return -99999;
    }
}

int stack_top()
{
    if (!stack_isempty())
    {
        return stack[s_top];
    }
    else
    {
        printf("Error: the stack is empty!\n");
        return -99999;
    }
}
```

```c
void DFS(struct Graph * graph, int start) {
  bool added[graph->numVertices];
  memset(added, false, sizeof added);
  stack_push(start);
  added[start] = true;
  while(!stack_isempty()){
    int current = stack_top();
    stack_pop();
    // mark current node as visited and print it.
    struct node* adjList = graph->adjLists[current];
    struct node* temp = adjList;
    printf("Visited %d \n", current);
    while (temp != NULL) { // while we still have neighbours
      int connectedVertex = temp->vertex;
      if (added[connectedVertex] == 0) { // if they are unvisited, visit them
          stack_push(connectedVertex);
          added[connectedVertex] = true;
      }
      temp = temp->next; //check next neighbour
    }
  }
}
```

Task 4. So far we have studied search in graphs. This task is about search in a 2D array (2D matrix). Imagine a 2D array, where each element shows a physical location in a maze. In this exercise, we use the following notation:

- **.** shows plain terrain
- # shows unpassable walls
- **S** and **E** shows start and end position respectively (also plain terrain)

The maze is a 7 by 7 matrix that is shown below. The possible movements for a position in the matrix are towards **west, south, east, north**

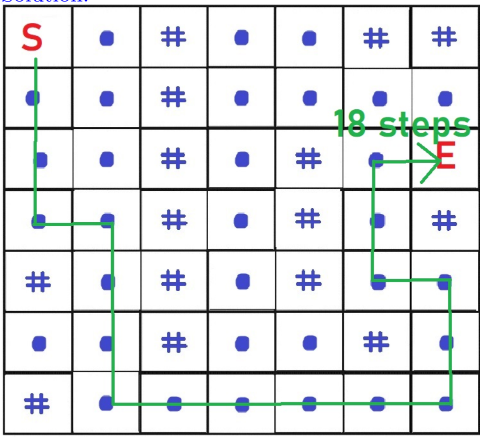
S	.	#	.	.	#	#
.	.	#	.	.	.	.
.	.	#	.	#	.	E
.	.	#	.	#	.	#
#	.	#	.	#	.	.
.	.	#	.	.	#	.
#	.	.	.	.	.	.

1. What graph search algorithm should be used to find the minimum amount of steps required to travel from start to end?

BFS algorithm should be used to find the minimum amount of steps required from start to end, with a terminating condition to stop when the end is found.

2. Run this algorithm by hand(no need for the progress of the algorithm, just show the path and total number of steps)

Solution:



3. Implement the algorithm that returns the shortest distance from the start position to the end position. You can use the following C codes.

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define GRIDSIZE 7
#define MAXQSIZE 49

struct Point{
    int x;
    int y;
};

struct queueNode
{
    struct Point pt; // The cordinates of a cell
    int dist; // cell's distance of from the source
};

bool isValid(int row, int col)
{
    // return true if row number and column number
    // is in range
    return (row >=0) && (row < GRIDSIZE) &&
           (col >=0) && (col < GRIDSIZE);
}
```

```
int sp_algo(int A[GRIDSIZE][GRIDSIZE], struct Point start, struct Point end)
```

```
int main() {
```

```
    int A[7][7] =
      {
            { '.', '.', '#', '.', '.', '#', '#'},
            { '.', '.', '#', '.', '.', '.', '.'},
            { '.', '.', '#', '.', '#', '.', '.'},
            { '.', '.', '#', '.', '#', '.', '#'},
            { '#', '.', '#', '.', '#', '.', '.'},
            { '.', '.', '#', '.', '.', '#', '.'},
            { '#', '.', '.', '.', '.', '.', '.'},
      };
    struct Point start = {0, 0};
    struct Point end = {2, 6};

    int spDist = sp_algo(A, start, end);
    printf("%d", spDist);
}
```

Hint: You can find Advanced Data Structure implementations in previous exercise sheets.

Solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

#define GRIDSIZE 7
#define MAXQSIZE 49

struct Point{
    int x;
    int y;
};

struct queueNode
{
    struct Point pt; // The cordinates of a cell
    int dist; // cell's distance of from the source
};

bool isValid(int row, int col)
{
    // return true if row number and column number
    // is in range
    return (row >=0) && (row < GRIDSIZE) &&
           (col >=0) && (col < GRIDSIZE);
}

struct queueNode queue[MAXQSIZE];

int front = -1;
int rear = -1;
int size = -1;

bool q_isempty()
{
    return size < 0;
```

```c
}

void q_enqueue(struct queueNode value)
{
    if(size<MAXQSIZE)
    {
        if(size<0)
        {
            queue[0] = value;
            front = rear = 0;
            size = 1;
        }
        else if(rear == MAXQSIZE-1)
        {
            queue[0] = value;
            rear = 0;
            size++;
        }
        else
        {
            queue[rear+1] = value;
            rear++;
            size++;
        }
    }
    else
    {
        printf("Queue is full\n");
    }
}

int q_dequeue()
{
    if(size<0)
    {
        printf("Queue is empty\n");
    }
    else
    {
        size--;
        front++;
    }
}

struct queueNode q_front()
{
    return queue[front];
}

int sp_algo(int A[GRIDSIZE][GRIDSIZE], struct Point start, struct Point end)
{
    // BFS
    // These arrays are used to get row and column
    // numbers of 4 neighbours of a given cell
    int rowNum[] = {-1, 0, 0, 1};
    int colNum[] = {0, -1, 1, 0};
```

```
    // initialise visited and add start to queue
    bool visited[GRIDSIZE][GRIDSIZE];
    memset(visited, false, sizeof visited);
    // Distance of source cell is 0
    struct queueNode start_node = {start, 0};
    q_enqueue(start_node);
    visited[start.x][start.y] = true;
    while(!q_isempty()){
        struct queueNode currentNode = q_front();

        struct Point pt = currentNode.pt;

        // If we have reached the destination cell, we are done
        if (pt.x == end.x && pt.y == end.y){
            return currentNode.dist;
        }
        // Dequeue the front
        q_dequeue();

        //enqueue its neighbours (efficiently)
        int i;
        for (i = 0; i < 4; i++)
        {
            int row = pt.x + rowNum[i];
            int col = pt.y + colNum[i];

            // if adjacent cell is valid, has path and
            // not visited yet, enqueue it.
            if (isValid(row, col) && A[row][col] == '.' &&
              !visited[row][col])
            {
                // mark cell as visited and enqueue it
                visited[row][col] = true;
                struct queueNode adjCell = { {row, col},
                                             currentNode.dist + 1 };
                q_enqueue(adjCell);
            }
        }
    }
}

int main() {
  int A[7][7] =
    {
        { '.', '.', '#', '.', '.', '#', '#'},
        { '.', '.', '#', '.', '.', '.', '.'},
        { '.', '.', '#', '.', '#', '.', '.'},
        { '.', '.', '#', '.', '#', '.', '#'},
        { '#', '.', '#', '.', '#', '.', '.'},
        { '.', '.', '#', '.', '.', '#', '.'},
        { '#', '.', '.', '.', '.', '.', '.'},
    };
  struct Point start = {0, 0};
  struct Point end = {2, 6};

  int spDist = sp_algo(A, start, end);
```

```
  printf("%d", spDist);
}

// Linux, Mac: gcc task12_4.c −o task12_4; ./task12_4
// Windows: gcc task12_4.c −o task12_4; task12_4
```

4. Is there a chance multiple shortest paths can be found? If so, show another path and explain what implementation detail determines which of the two is chosen?
Solution:
Yes, the BFS is not unique, and it can be the case that multiple shortest paths can exist, taking a different route but having the same total cost. This depends on the implementation of BFS and the order that it checks the neighbors.