

Tutorial 9: Greedy on Graph

Algorithms - I

By Deepank Agrawal

Table of Contents

- ① Important Greedy Algorithms
- ② Previous Year's Lab Assignment
Problem
Hints and Solution
Implementations
- ③ Dijkstra's Algorithm Demo
- ④ Extra Problems for Better Understanding

Table of Contents

- 1 Important Greedy Algorithms
- 2 Previous Year's Lab Assignment
Problem
Hints and Solution
Implementations
- 3 Dijkstra's Algorithm Demo
- 4 Extra Problems for Better Understanding

Greedy Graph Algorithms

Following are some important greedy algorithms -

- Dijkstra's Shortest Path Search ✓
- Bellman Ford Algorithm for Shortest Path
- Prim's Minimum Spanning Tree Algorithm ✓
- Kruskal's Minimum Spanning Tree Algorithm ✓

Negative weight cycle



Greedy Graph Algorithms

Following are some important greedy algorithms -

- Dijkstra's Shortest Path Search
- Bellman Ford Algorithm for Shortest Path
- Prim's Minimum Spanning Tree Algorithm
- Kruskal's Minimum Spanning Tree Algorithm



Table of Contents

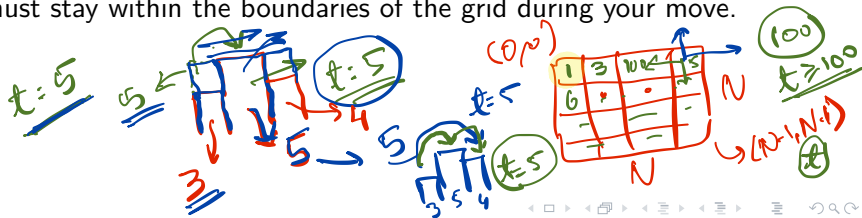
- ① Important Greedy Algorithms
- ② Previous Year's Lab Assignment
Problem
Hints and Solution
Implementations
- ③ Dijkstra's Algorithm Demo
- ④ Extra Problems for Better Understanding

Table of Contents

- ① Important Greedy Algorithms
- ② Previous Year's Lab Assignment
Problem
Hints and Solution
Implementations
- ③ Dijkstra's Algorithm Demo
- ④ Extra Problems for Better Understanding

Problem Statement

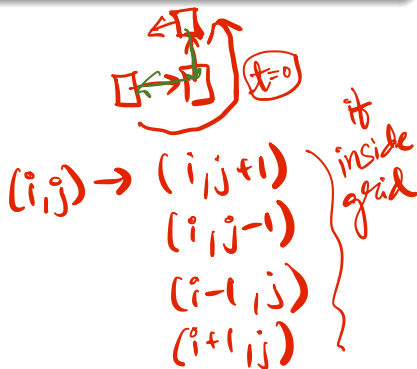
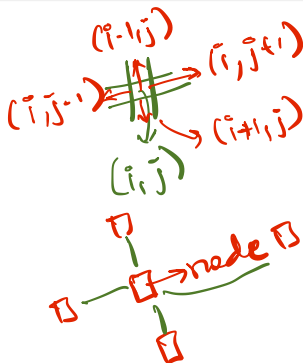
Consider a $N \times N$ grid of squares (or blocks) with coordinates ranging from $(0, 0)$ to $(N - 1, N - 1)$. Each value of the grid ($grid[i][j]$) represents the elevation at that particular block (i, j) . When you start to move at time $t = 0$, the blocks rise such that at time t the elevation of (i, j) th block would be $\max(grid[i][j], t)$. You can only move to your adjacent blocks (top, bottom, left or right) if and only if the elevation of both the blocks individually are at most t . You can move infinite distance in zero time, but you must stay within the boundaries of the grid during your move.



Understand the Plot

Idea

Consider the blocks as nodes and edges run to the four neighboring blocks (top, bottom, right and left).



You are asked to...

Initially you are standing at (S_x, S_y) th block.

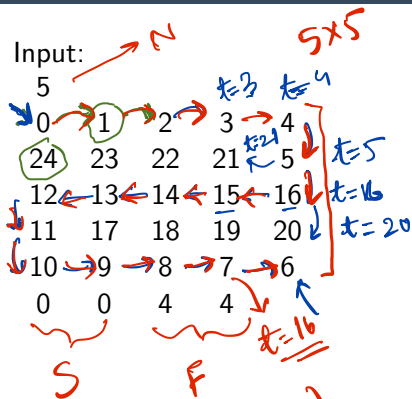
- What is the **least time** in which you can reach the (F_x, F_y) th block?
- Print the path travelled to reach (F_x, F_y) th block.
- Also print the number of blocks traversed to reach (F_x, F_y) th block.

Note

There are many ways to solve this problem. But we will use Dijkstra's Shortest Path Algorithm.

$(S_x, S_y) \rightsquigarrow (F_x, F_y)$
Path

A Working Example



$$t=0 \quad (0,0) \rightarrow (0,1) \text{ or } (1,0)$$

$$t=1 \quad (0,0) \rightarrow (0,1)$$

$$t=24 \quad (0,0) \rightarrow (1,0)$$

$$(5) + (16 - 5) \Rightarrow 16$$

$$(16) + (6 - 7) = 9$$

$$\boxed{t=16}$$

17 blocks

$$t_v + \underline{ele(u)} - \underline{ele(v)} \quad \checkmark$$

t_v

$t=0$

$grid[u]$
 $grid[v]$

$$t_u = \max(t_v, \underline{grid(u)})$$

if you
are moving $v \rightarrow u$

Table of Contents

- ① Important Greedy Algorithms
- ② Previous Year's Lab Assignment
 - Problem
 - Hints and Solution**
 - Implementations
- ③ Dijkstra's Algorithm Demo
- ④ Extra Problems for Better Understanding

Can you think of an approach?

Have 2 – 3 minutes to think. //

Approach Hints

- How is it analogous to Dijkstra's Shortest Path Search case?

Approach Hints

- How is it analogous to Dijkstra's Shortest Path Search case?
 - In both cases, we want to reach a node F from a node S with some minimal cost.




Approach Hints

- How is it analogous to Dijkstra's Shortest Path Search case?
 - In both cases, we want to reach a node F from a node S with some minimal cost.
- Suppose you know the time taken to reach a neighbour v of some node u . How will you use this information to reach node u via v ?

Approach Hints

- How is it analogous to Dijkstra's Shortest Path Search case?
 - In both cases, we want to reach a node F from a node S with some minimal cost.
- Suppose you know the time taken to reach a neighbour v of some node u . How will you use this information to reach node u via v ?
 - Time taken to reach u via $v \leftarrow$ elevation of u or time taken to reach v ; whichever is maximum.

Approach Hints

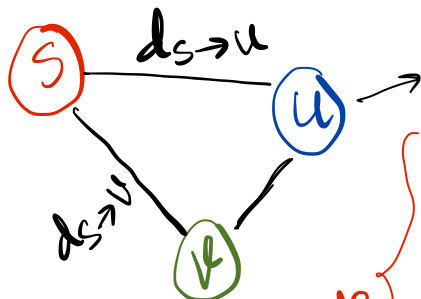
- How is it analogous to Dijkstra's Shortest Path Search case?
 - In both cases, we want to reach a node F from a node S with some minimal cost.
- Suppose you know the time taken to reach a neighbour v of some node u . How will you use this information to reach node u via v ?
 - Time taken to reach u via $v \leftarrow$ elevation of u or time taken to reach v ; whichever is maximum. 
- Consider a source node S , current node u and its un-visited neighbour node v . Under what condition will you update time taken to reach v in this step of Dijkstra's Algorithm?  

Approach Hints

- How is it analogous to Dijkstra's Shortest Path Search case?
 - In both cases, we want to reach a node F from a node S with some minimal cost.
- Suppose you know the time taken to reach a neighbour v of some node u . How will you use this information to reach node u via v ?
 - Time taken to reach u via $v \leftarrow$ elevation of u or time taken to reach v ; whichever is maximum.
- Consider a source node S , current node u and its un-visited neighbour node v . Under what condition will you update time taken to reach v in this step of Dijkstra's Algorithm?
 - $d_{S \rightarrow v} > \max(d_{S \rightarrow u}, t_v)$

$$d_{S \rightarrow v} = \max(d_{S \rightarrow u}, \text{elevation}(v))$$

Solution Approach



① When we "visit" a node u in Dijkstra we have the optimal path to u .

② Study proof of Dijkstra

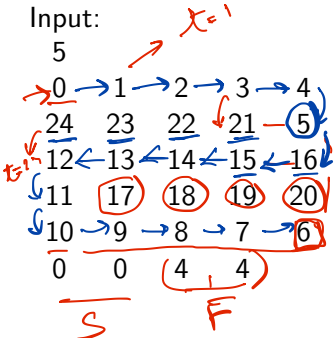
$$d_s \rightarrow u \rightarrow v = \max(d_s \rightarrow u, \overset{16}{\text{quad}} \overset{15}{v})$$

if $d_s \rightarrow v > d_s \rightarrow u \rightarrow v$:
update $d_s \rightarrow v$
parent[v] = u

Let's Rework The Example

Input:

5



Set $\Rightarrow \{ (1,4) \}$

$(1,4) \rightarrow (1,3)$
 \downarrow
 21

Set = $\{ (0,0) \}$ $d_0 \rightarrow 0 \rightarrow 0$
 ∞

$t=0$ Set = $\{ (0,0) \}$

(16)

$t=21$

max(16,15)

"Visit"

$\rightarrow (0,1)$

$(1,0)$

$d_{01} \rightarrow 1$

$d_{10} \rightarrow 24$

$t=1$ Set $\{ (0,0), (0,1) \}$

$(0,1) \rightarrow (0,2)$ $(1,1)$

\downarrow
2

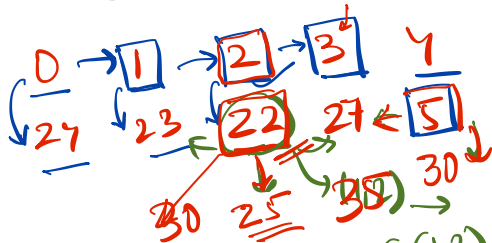
\downarrow
23

$(2,4)$
 \downarrow
16

Continued...

distance set d

$\Rightarrow \{0, 1, 2, 3, 4, 24, 23, 22, 21, 5\}$



parent[22] = 2

parent[25] = 22

update

$\begin{cases} (1,3) \\ (1,1) \\ (2,3) \end{cases}$

$25 \rightarrow 22 \rightarrow 2 \rightarrow 1 \rightarrow 0$

$(0,2) \rightarrow \text{visited}$

unvisited set ∞
 $\rightarrow (22, 23, 24, 27, 30, \dots)$
visited set
 $\rightarrow (0, 1, 2, 3, 4, 5)$


Table of Contents

- ① Important Greedy Algorithms
- ② Previous Year's Lab Assignment
 - Problem
 - Hints and Solution
 - Implementations**
- ③ Dijkstra's Algorithm Demo
- ④ Extra Problems for Better Understanding

C Code - Helper Methods

- First, let's define a struct to denote a cell.

```
typedef struct {  
    int row, col;  
} Cell;
```



C Code - Helper Methods

- First, let's define a struct to denote a cell.

```
typedef struct {  
    int row, col;  
} Cell;
```



- A helper function to check if next cell is within the grid.

```
bool check_cell_validity(Cell c, int nrows, int ncols) {  
    return !(c.row < 0 || c.row >= nrows  
            || c.col < 0 || c.col >= ncols);  
}
```



C Code - Helper Methods

- First, let's define a struct to denote a cell.

```
typedef struct {  
    int row, col;  
} Cell;
```



4 if
conditions

- A helper function to check if next cell is within the grid.

```
bool check_cell_validity(Cell c, int nrows, int ncols) {  
    return !(c.row < 0 || c.row >= nrows  
            || c.col < 0 || c.col >= ncols);  
}
```

- Declare a static array to navigate neighbours.

```
const Cell dir[] = {{-1, 0}, {0, -1}, {1, 0}, {0, 1}};
```

$(i, j) \rightarrow (i+1, j) \quad (i-1, j) \quad (i, j-1) \quad (i, j+1)$

C Code - Main Loop

Cell c = {-1, -1}, v;

// find out the next candidate node

```
for(k = 0; k < n; k++) {  
    for(l = 0; l < n; l++) {  
        if(visited[k][l] == 1) continue;  
        if(c.row == -1 || time[c.row][c.col] > time[k][l]) {  
            c.row = k, c.col = l;  
        }  
    }  
}
```

visited[c.row][c.col] = 1;

// update visit time for it's neighbors

```
for(k = 0; k < 4; k++) {  
    v.row = c.row + dir[k].row;  
    v.col = c.col + dir[k].col;  
    if(check_cell_validity(v, n, n) == false) continue;  
    if(visited[v.row][v.col]) continue;  
  
    int time2v = mat[v.row][v.col];  
    if(time2v < time[c.row][c.col]) time2v = time[c.row][c.col];  
  
    if(time[v.row][v.col] > time2v){  
        time[v.row][v.col] = time2v;  
        parent[v.row][v.col] = c;  
    }  
}
```

→ loop(i → 1-n) loop(j → 1-n)

time

next
nearest
node
to put
in the
visited
set

→ c is "visited" node

K → 0, 1, 2, 3

→ grid[v]

} max(t_c , grid[v])

$t_v > \max(t_c, \text{grid}[v])$

1D Array based Efficient Approach ✓

→ Cache hit/miss

Basic Idea

The size of grid is $N \times N$ i.e., there are N^2 cells. We can represent the (i, j) th cell as $(i * N + j)$ th number on the number line. And, a number x will represent the $(x/N, x \% N)$ th cell in the grid.

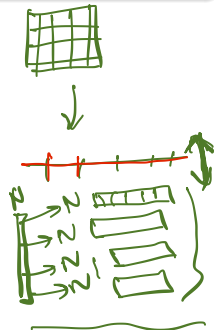
$N \times N \rightarrow N^2$ cells $N=7$

$(3, 5)$ $\rightarrow 21 + 5 \Rightarrow$ (26)

$(0, 0) \rightarrow 0$

vector

26 $\rightarrow (\underline{26/7}, 26\%7)$ Old
 $\Rightarrow (3, 5)$



1D Array based Efficient Approach

The transformed main loop will look like -

```
int c = -1, v;  
  
for(j = 0; j < n * n; j++) {  
    if(visited[j] == 1) continue;  
    if(c == -1 || time[c] > time[j]) {  
        c = j;  
    }  
}  
  
visited[c] = 1;  
for(j = 0; j < 4; j++) {  
    int vrow = c / n + dir[j] / n;  
    int vcol = c % n + dir[j] % n;  
    if(check_cell_validity(vrow, vcol, n) == false) continue;  
    v = n * vrow + vcol;  
    if(visited[v]) continue;  
  
    int time2v = (mat[v] > time[c]) ? mat[v] : time[c];  
    if(time[v] > time2v){  
        time[v] = time2v;  
        parent[v] = c;  
    }  
}
```

Handwritten notes:

- $\{ (i, j) \}$ next to the inner loop.
- Red arrow pointing to `check_cell_validity` with the note `max(gf(v), t)`.
- Red underline under `parent[v] = c;`.

Solution Code Links

Please find the full implementation of above explained approaches at these Ideone links -

- A simple C implementation
- An efficient C implementation using 1D arrays (for interested folks)
- A C++ implementation using STL (bonus)

Table of Contents

- ① Important Greedy Algorithms
- ② Previous Year's Lab Assignment
Problem
Hints and Solution
Implementations
- ③ Dijkstra's Algorithm Demo
- ④ Extra Problems for Better Understanding

Real Life Application: Path Planning

Let's have a look at a small 30-seconds [video clip](#)!

Table of Contents

- ① Important Greedy Algorithms
- ② Previous Year's Lab Assignment
Problem
Hints and Solution
Implementations
- ③ Dijkstra's Algorithm Demo
- ④ Extra Problems for Better Understanding

Problem 1

Question

A directed graph with N vertices and M edges is given. What is the minimum number of edges you need to reverse in order to have at least one path from vertex 1 to vertex N ?

Problem 1

Question

A directed graph with N vertices and M edges is given. What is the minimum number of edges you need to reverse in order to have at least one path from vertex 1 to vertex N ?

Introduce reverse edges with weight 1. The existing edges have weight 0. Use any shortest path algorithm from vertex 1.

If shortest path is p , it means we used p reverse edges in the path. The shortest path algorithm will always try to use as less reverse paths as possible because they have higher weight than original edges.

Approach

Use 0 – 1 BFS to solve the problem. Dijkstra's Algorithm is an overkill here as the edge weights are constrained to 0 or 1 only. Read [here](#).

Link to the Question: [Chef and Reversing](#)

Problem 2

Question

You are given a weighted undirected graph. The vertices are enumerated from 1 to n . Your task is to find the shortest path between the vertex 1 and the vertex n . It is possible that the graph has loops and multiple edges between pair of vertices.

Problem 2

Question

You are given a weighted undirected graph. The vertices are enumerated from 1 to n . Your task is to find the shortest path between the vertex 1 and the vertex n . It is possible that the graph has loops and multiple edges between pair of vertices.

Approach

Simple Problem based on Dijkstra's Shortest Path Algorithm.

Link to the Question: [Problem - 20C - Codeforces](#)

Problem 3

Question

There are n cities and m roads between them. The condition of the roads is so poor that they cannot be used. Your task is to repair some of the roads so that there will be a decent route between any two cities.

For each road, you know its reparation cost, and you should find a solution where the total cost is as **small** as possible.

Every road is between two different cities, and there is at most one road between two cities.

Problem 3

Question

There are n cities and m roads between them. The condition of the roads is so poor that they cannot be used. Your task is to repair some of the roads so that there will be a decent route between any two cities. For each road, you know its reparation cost, and you should find a solution where the total cost is as **small** as possible. Every road is between two different cities, and there is at most one road between two cities.

Approach

Simple Problem based on Minimum Spanning Tree Algorithm.

Link to the Question: [CSES - Road Reparation](#)

More Problems

Solve these problems on your own (in increasing order of difficulty) -

- [Jack goes to Rapture - HackerRank](#) (Similar to last year assignment)
- [Even Tree - HackerRank](#)
- [59E - Shortest Path - Codeforces](#)

Thank You!

Any Further Questions?