Pathfinding Solution in 2D and 3D Tile-based Space with Dynamic Programming

Kevin Erdiza Yogatama, 13515016
Informatics Engineering
School of Electrical Engineering and Informatics, ITB
Bandung, Indonesia
13515016@std.stei.itb.ac.id

***Abstract*—Pathfinding problems are a common problem in computer science. One ways to solve pathfinding problem are using the dynamic programming approach. In this paper, we are interested in extending the basic dynamic programming approach that is used for pathfinding on a given weighted graph to be applicable on 2-dimensional and 3-dimensional tile-based space.**

Keywords—height; cost; dynamic programming; 2-dimensional tile-based; 3-dimensional tile-based;

I. Introduction

Humans are mobile beings. We always want to move from one point to another. Whether when we want to get some foods from the fridge or when we have to go to work as a responsible worker, moving from one place to another place is a must. However, before we move away from our original location to another location, we have to know which path to take. And most of the time, the further our original location is to the destination, the harder it is for us to choose which path to take. In that case, several things have to be considered, such as the distance traveled, the accessibility of the path, and many other issues. That very problem is what we call pathfinding.

Pathfinding problem is a problem asking which path one must take on a given space from one point to another desired point that follows a given criteria. The criteria is not limited to finding the shortest path from two point on the space, it can also include finding the longest path possible, or finding a path that spans a specific distance. Measuring a path is also not limited to calculating distance, we can also determine another variable such as travel cost or difficulty so traveling from one point to another does not only depend on the distance alone, knowing that shorter path does not mean easier path.

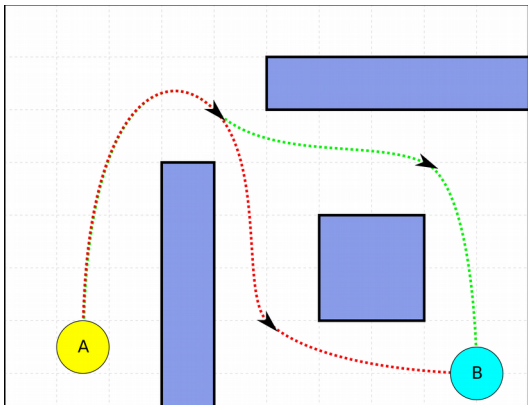

Fig. 1. *There can be more than one path from one point to another. Choosing between the possible paths is the core of pathfinding problem*

In computer science, there are several ways to solve this problem, and one of them is the dynamic programming approach. In this paper, we only focus on solving pathfinding problem that asks which possible path is the shortest in the term of distance and cost to traverse height using dynamic programming approach. We are also interested in extending the basic dynamic programming approach that is used to solved pathfinding problem on a given cost graph to become applicable in solving pathfinding problem on a tile-based space.

II. Dynamic Programming

A. What is Dynamic Programming Approach?

Dynamic programming approach is an approach that treats a problem as a structure that is made up of smaller subproblems and storing their values in case of the same subproblem occurs. The storing part is what is called “memoization”. In dynamic programming, the subproblems that made the whole problem is also considered as stages or steps in such a way that the main problem can be seen as a sequence of interrelated decision [1].

In the term of seeing the problem as a sequence of decision, dynamic programming approach is similar to greedy approach. One major difference is that in greedy approach, every decision made is not necessarily related to each and each decision is only a local optimum meanwhile in dynamic programming

approach, every decision is made with consideration of previously made related decision that is assumed to be optimal for the previous subproblem.

B. Problems with optimal sub-structure

Problem that is solved by dynamic programming approach is the problem that has an optimal sub-structure. A problem has an optimal sub-structure if the optimum solution to the problem contains the optimum solution to its smaller subproblem [2].

For example, the pathfinding problem searching for the shortest path has an optimal sub-structure. Suppose we have $s \rightarrow u \rightarrow v$ as the optimum solution of shortest path problem from s to v, if shortest path problem truly has an optimal sub-structure, then it means $s \rightarrow u$ is the optimum solution shortest path problem from s to u, being a part of the optimum solution of the main problem. This holds true because it can be proven by a contradiction. If $s \rightarrow u$ is not the optimum solution shortest path problem from s to u, then there will be a shorter path in $s \rightarrow u \rightarrow v$ as the actual optimum solution. However, since it is assumed that $s \rightarrow u \rightarrow v$ is the shortest path, then we have a contradiction [2].

C. Memoization

Many problem that is solved by dynamic programming exhibits a property called overlapping subproblems. The problem exhibits the overlapping subproblems is the problem that there are two or more of its subproblem have a smaller same subproblem.

For example, the problem of finding any Fibonacci number has the property of overlapping subproblems. the formula to find any nth Fibonacci number f is as follows:

$$f = \mathrm{F(n)} = \mathrm{F(n-1)} + F(\mathrm{n-2}) \text{ , if } n > 2$$

$$f = 1 \text{ , if } n = 2$$

$$f = 1 \text{ , if } n = 1$$

From the formula, we can describe that the problem of finding the nth Fibonacci number is consist of two smaller subproblems that are finding the (n-1)th Fibonacci number and finding (n-2)th Fibonacci number. The subproblems of (n-1)th Fibonacci number can then be broken down to smaller subproblems which are the (n-2)th Fibonacci number and the (n-3)th Fibonacci number. From there, we can describe that the subproblem of he nth Fibonacci number and the he (n-1)th Fibonacci number are overlapping because both involves the same subproblem of he (n-2)th Fibonacci number.

To avoid recalculating the solution of the same subproblem, in dynamic programming approach, "memoization" technique is used, which is to store the solution of the subproblems that can be accessed later if the same subproblem occurs again so there will be no same calculations done more than once.

III. Pathfinding on a Tile-Based Space

A. The problem

In this paper, we focus to pathfinding problem searching the shortest path on a given tile-based space. The shortest path is defined as the path with the lowest cost to traverse.

We define a 2-dimensional tile-based space as a collection of points that have integer Cartesian coordinates of the axis x and axis y and each point represents a tile. Any object on that space can only move to 4 directions at a time: north, south, 1.

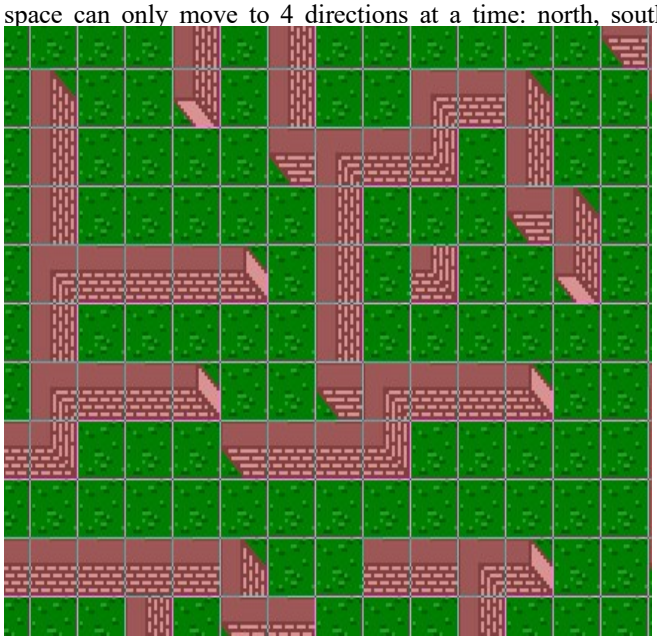

Fig. 2. *2-dimensional tile-based space model are commonly used in computer games*

In the given space, there might be tiles that can't be passed so the adjacent tiles will have one less option for movement and there will be no path that will consist of impassable tiles.

Extending the 2-dimensional tile-based space, we define 3-dimensional tile-based space as a collection of points that not only have integer Cartesian coordinates but also have an integer value representing the tile height. The cost for moving between tiles with different heights is defined as follows: it will be either 0 or the height difference minus 1, whichever the smaller. We treat movement cost of going to a tile with higher height the same as going to a tile with lower height.

B. The solution

To demonstrate the solution with dynamic programming approach. We made a program to calculate and visualize the solution. The program is made in Java and can be found here https://github.com/keychera/Pathfinding-with-Dynamic-Programming.

Here are some of the examples of the input and its visualized solution:

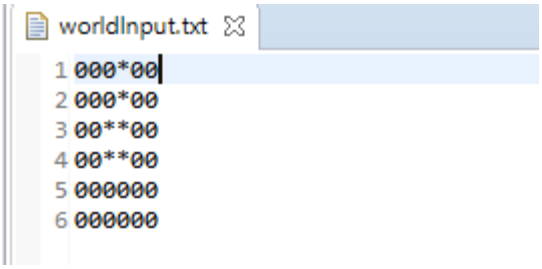

Fig. 3. *Example of input for 2-dimensional tile-based space. A number '0' represents a tile, the character '*' represent impassable tiles.*

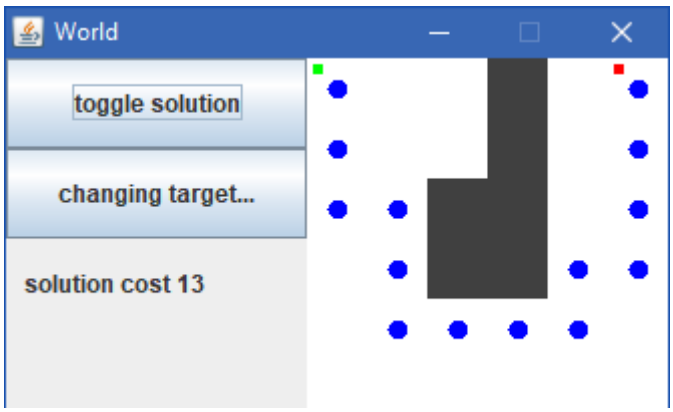

Fig. 4. Visualized solution for pathfinding problem on a 2-dimensional tile-based space. The dark tiles are the impassable tiles.

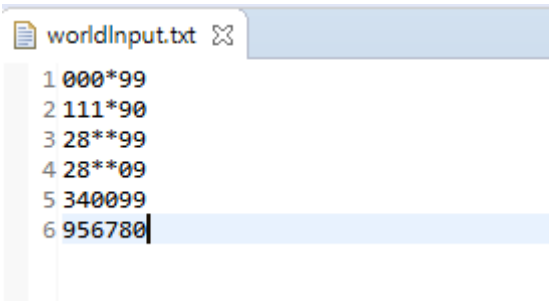

Fig. 5. *Example of input for 3-dimensional tile-based space. A number represents a tile and its height, the character '*' represent impassable tiles.*

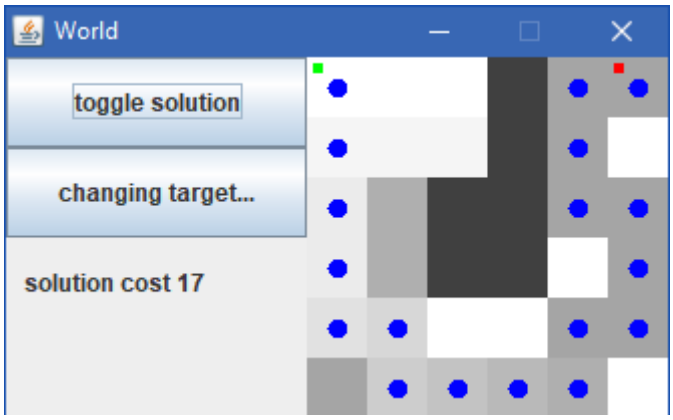

Fig. 6. *Visualized solution for pathfinding problem on a 3-dimensional tile-based space. The darker the gray is, the higher its height. The very dark ones are the impassable tiles.*

IV. Solving the Problem with Dynamic Programming Approach

A. 2D Tile-based space

1) Constructing the solution

Suppose for the shortest pathfinding problem, we are given an m x *n* tile-based space and point (i, j) as the starting point and point (k, l) as the destination point where $0 \leq$ i, k < *m* and $0 \leq$ j, l < *n*. the cost for each movement is the integer value of 1. the solution is defined as a sequence of point in the space taken that make up the path. The data structure for the points is done with the following pseudo java code:

```
class Point {
  int row;
  int column;
}
```

Code 1. *data structure of Point*

and the information of world is defined simply with arrays as follows:

```
class World {
  char[][] tiles;
  char getInfo(Point p) { ... }
}
```

Code 2. *data structure of World*

We can define the problem to have the following structure: the shortest path from point (i, j) to point (k, l) is made up of two type of smaller subproblems, the first type is the shortest pathfinding problem from point (i, j) to either one of the 4 points adjacent to point (k, l) and the second type involves finding which direction taken and its cost from each of the adjacent points to the point (k, l). the first subproblem is the same shortest pathfinding problem and can also be broken down to two smaller subproblems as above. The second subproblem's solution is which point to take to reach the point (k, l) and along with its cost, they will be stored in a memo.

The dynamic programming approach treats the first subproblems as the stage of the decision-making and treats the second subproblem's solution as the possible current optimal solution to take if the path ever goes there. Breaking down the first subproblem will create a new stage and a new solution and because the problem exhibits the overlapping subproblem property, it is possible to encounter a first subproblem that breaks down to an existing second subproblem that is already saved in the memo, and have a new solution.

For the memo, we define the data stored with the pseudo java code as follows:

```
class Memo {
    List<Point> directions;
    int cost;
}
```

Code 3. *data structure of Memo*

Every point on the space will have its own memo containing the decisions to take and its cost and each point will have initially infinite cost stored in the memo representing that there are no calculation done for that specific point. A point can have two different decision that have the same cost hence it is saved as a list of value instead. Every time the second subproblem is calculated, the new result is compared to current memo and the memo is updated accordingly if the new one has better cost, removing the previous one.

The problem is solved iteratively using queue with the pseudo java code as follows:

```
void findPath(Point target, Point start) {
   queue.add(target);
   memos[target].addNewMemo();
   boolean isStartFound = (target == start);
   while(!queue.isEmpty() && !isStartFound) {
     Point checking = queue.remove();
     isStartFound = (checking == start);
     if (!isStartFound) {
       evaluate(checking);
     }
   }
}
```

Code 4. *main iteration code for finding shortest path*

The evaluate method part will evaluate a given point as the target point for a shortest pathfinding problem and break it into maximum of 4 new points to check which are the adjacent points of the given point and 4 corresponding second type subproblem's solution. The next removed head queue will be evaluated as the next subproblem of the first type: shortest pathfinding problem.

The second subproblem is handled with the pseudo java code as follows:

```
void addNewNote(Point adj, Point checking) {
   int newCost = memos[checking].cost + 1;
   if (newCost <= memos[adj].cost) {
     if (newCost < memos[adj].cost) {
       directions.clear();
       memos[adj].cost = newCost;
     }
     directions.add(checking);
     return true;
   } else {
     return false;
   }
}
```

Code 5. *the method of calculating new cost and adding it to the corresponding memo*

This code calculate the new cost for the adjacent point by adding the checking point cost by 1 since every step taken cost 1 integer value. Then it is checked whether the newly calculated cost is better than the one stored in the memo. If it's better, the memo is updated accordingly, but if it's not, the past calculation is considered. Since the initial cost int the memo for each point is effectively infinite, the memo will always be updated for its first calculation.

Notably, The code 5 also returns a boolean value based on whether the new cost is higher than the stored one or not. The value will be used in evaluation whether the new tile that have the new cost will be added to the main queue or not, since having the new cost higher than the stored one means the past calculation on that point is more optimum than the new one, hence not using the new one.

In the code 4, it is defined that the loop will run the calculation until the queue is empty or until the start point is found. In 2-dimensional tile-based space, it is given that every movement cost between any adjacent point is the same which is an integer value of 1. hence the cost is the same as the number of step and it also means the first path that found the start point is the shortest path since the calculation is done step by step.

After the calculation is done, the solution is constructed by reading the directions stored in the memo from the starting point until reaching the target. The code that will construct the solution is as follows (the solution is stored as the modified world model, marking the path with the character '^'):

```
World constructSolution() {
   World solution = new World(world);
   Point traverse = start;
   solution.setTile(traverse, '^');
   while (traverse != target)) {
     traverse = memos[traverse].directions.get(0);
     solution.setTile(traverse, '^');
   }
   return solution;
}
```

Code 6. *main iteration code for finding shortest path*

The following pictures will show several examples of input and its solution.

worldInput.txt

```
1 000*0
2 0*000
3 0*000
```

Fig. 7. *Input A Example*

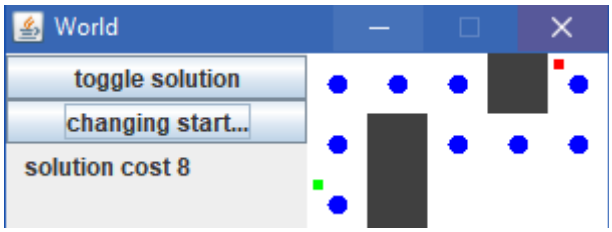

Fig. 8. *The solution for input A with the starting point denoted with green square and destination point denoted with red square*

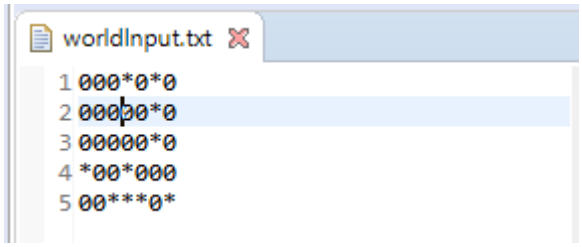

Fig. 9. *Input B Example*

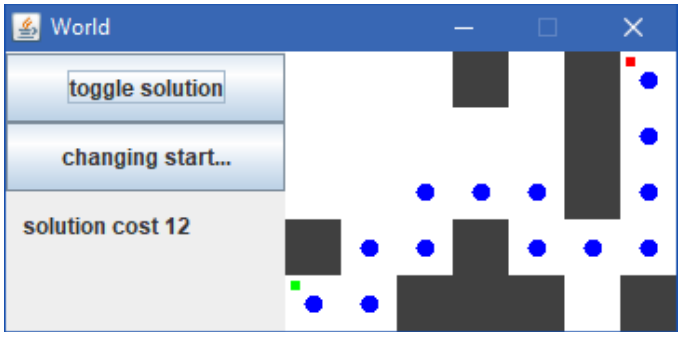

Fig. 10. *The solution for input B with the starting point denoted with green square and destination point denoted with red square*

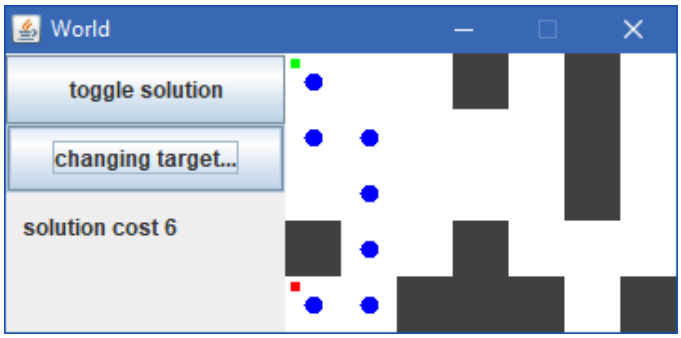

Fig. 11. *The solution for input B with different starting and destination point*

B. 3D Tile-based space

To find the solution for shortest pathfinding problem on a 3-dimensional tile-based space. The previous solution for solving the problem on 2-dimensional space just need some slight modification as the 3D model is a slightly modified from the 2D model. The modification needed in 3D model is the calculation of the cost each movement since movement between adjacent tiles is no longer the same. In the 3D model, the cost between two adjacent cell is calculated from the height difference. If two adjacent points are of the same height, the cost is 1 but they have different heights, the cost between tile p and tile q, as defined in the previous section, is calculated with the following formula:

$$\text{cost}(p,q) = 1 + \min(0, |p.\text{heght} - q.\text{height}| - 1)$$

Using that formula, it means that moving between tiles that have height difference of 1 cost the same as moving between tiles of the same height. It also means that moving to higher tile will cost the same as moving to lower tile for the same height difference. This formula is used so the program will avoid path with steep changes of height in its course and prefer a gradual increase in heights, reflecting on the real life examples that it is safer for the path to be not steep.

Hence the implementation on the pseudo java code is as follows:

```
void addNewNote(Point adj, Point checking) {
   int heightCost = 1 + Math.min(0, Math.abs(p.heght - q.height) - 1);
   int newCost = memos[checking].cost + 1 + heightCost;
   if (newCost <= memos[adj].cost) {
     if (newCost < memos[adj].cost) {
        directions.clear();
        memos[adj].cost = newCost;
     }
     directions.add(checking);
     return true;
   } else {
     return false;
   }
}
```

Code 7. *modified method of calculating cost for shortest pathfinding problem on a 3-dimensional tile-based space.*

Furthermore, the code 4 also needs a slight modification for the solution to work on 3-dimensional tile-based space shortest pathfinding problem. For 2-dimensional space, it is assumed that the shortest path is the same as the path with the least amount of steps since the cost for each movement is the same for all cases. But this does not held true for 3-dimensional space since the cost can vary depends on the tile's height difference hence the code modification is as follows:

```
void findPath(Point target, Point start) {
   queue.add(target);
   memos[target].addNewMemo();
   while(!queue.isEmpty()) {
     Point checking = queue.remove();
     evaluate(checking);
   }
}
```

Code 8. *modified method of main iteration code for finding shortest path om 3-dimensional space .*

The above code means that the loop will now exhaustively search every possible move which is until there is no addition

to the queue(every memo on each point has already the most optimum possible solution).

After the calculation is done, the construction solution part is the same as the 2-dimensional one.

The following pictures will show several examples of input used in the 2D section with height modification and its solution for comparison.

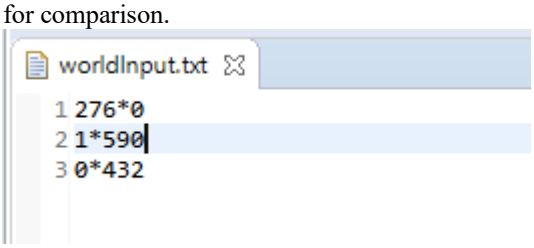

Fig. 12. *Input A Example with height modification represented by the integer from 0-9*

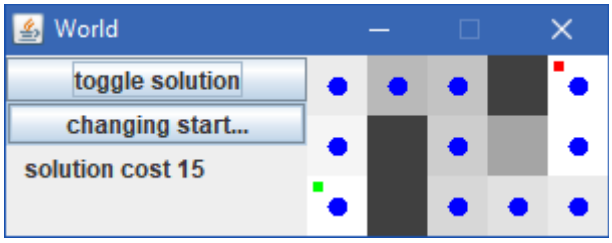

Fig. 13. *The solution for input A with the starting point denoted with green square and destination point denoted with red square*

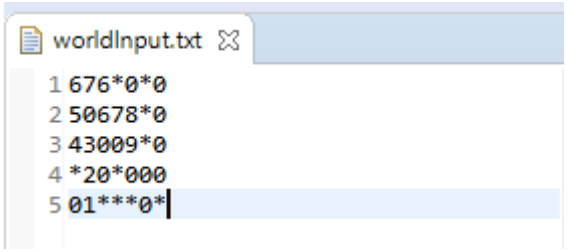

Fig. 14. *Input B Example with height modification represented by the integer from 0-9*

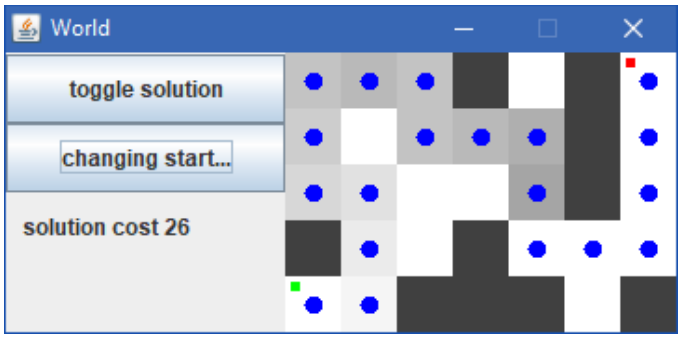

Fig. 15. *The solution for input B with the starting point denoted with green square and destination point denoted with red square. We can observe that the path take different route considering the height information compare to the 2-dimensional conterpart*

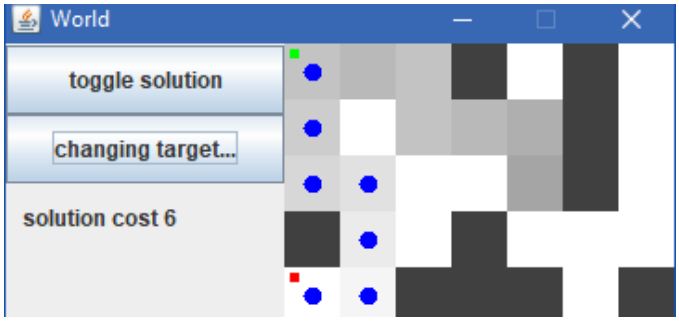

Fig. 16. *The solution for input B with different starting and destination point*

V. Conclusions

In this paper, we have shown you the extension of the dynamic programming approach used in pathfinding on a given weighted graph into pathfinding on a 2-dimensional and 3-dimensional tile-based space. We have shown that the three problems above share the similar solution and they differs on how the costs are calculated.

Acknowledgment

Firstly, the author thanks to God who have given me strength and perseverance for finishing this task. The author thanks to Mr. Rinaldi Munir, Mrs. Nur Ulfa Maulidevi, and Mrs. Masayu Leylia Khodra for their teaching and support along the course of my study of course of IF 2211 Strategi Algoritmik. The author also thanks to all his friends and fellow students of Informatics Engineering for their assistance and support all this time.

[1] Munir, Rinaldi (2005), Diktat Kuliah IF 2211 Strategi Algoritmik

[2] https://courses.csail.mit.edu/6.006/fall11/rec/rec19.pdf Accessed on 16 May 2017

Statement

I hereby declare that the paper I wrote is my own work. It is not a copy nor a translation of someone else's paper, and not a plagiarism.

Bandung, 29 April 2017

signature

Kevin Erdiza Yogatama , 13515016