Pathfinding Algorithms and Implementations on Grid Map

Steven Andrew / 13509061
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
andy165@students.itb.ac.id

***Abstract*—This paper explains some pathfinding algorithms such as: depth-first search, breadth-first search, Floyd–Warshall algorithm, Dijkstra's algorithm, best-first search, and A* algorithm, and also their implementations on a grid map that are commonly used in real-time strategy and role-playing games. Experiments show that A* algorithm results in the best path and is efficient in performance.**

***Index Terms* —A* algorithm, graph, grid map, heuristic, pathfinding.**

I. Introduction

In most kind of games, such as real-time strategy and role-playing games, there are many problems dealing with a game character. A problem we're trying to solve is to get a game object from the starting point to a goal. Depending on distance and obstacles, there are many ways to find a path. But, the best path we want to find is the shortest path.

Pathfinding is a process to find the object the shortest path. It is complex [3]. Consider the situation as shown in Fig. 1.

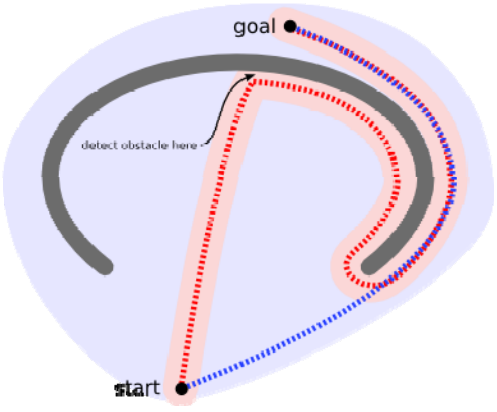


Fig. 1. Path from start towards goal, passing a concave obstacle

The unit at start point wants to get to the goal. Naively, it prefers to move straight towards the goal. When it detects an obstacle, it turns and follow the red path. In contrast, a pathfinder would have scanned a larger area (shown in light blue) and found a shorter path (blue).

Pathfinders let us look ahead and make plans rather than seeking the goal directly. Planning with pathfinders is slower but gives better results [3].

There are many algorithms we can use to do pathfinding. We will discuss about some pathfinding algorithms and do experiments with them to see results. We will also compare experiment results and determine which pathfinding algorithm is the best.

II. Grid Map Representation and Data Structure

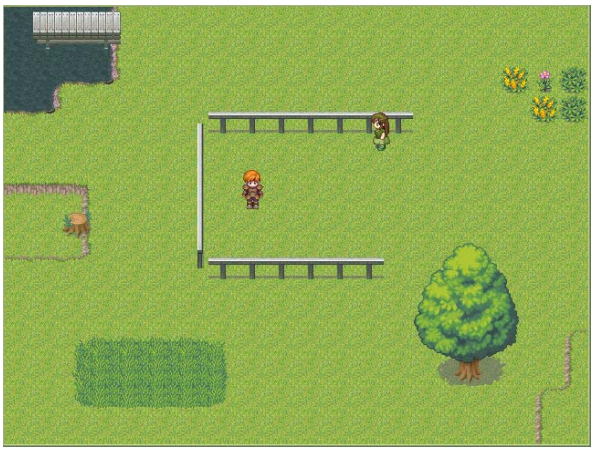


Fig. 2. A sample map on a role-playing game

There are some map representations we can apply, one of them is grid map. A grid map, as in Fig. 2, uses a uniform subdivision of the world into small regular shapes sometimes called "tiles" [4]. Common grids in use are square, triangular, and hexagonal, but in this paper we focus on square grids.

Data structure used on grid maps are commonly two-dimensional array (or table). Array elements represent properties of tiles (such as tile types and passages). For example, we can use boolean elements to represent passage of tiles, 1 means passable and 0 means impassable by the object. We may call it a passage table, as shown in Fig. 3.

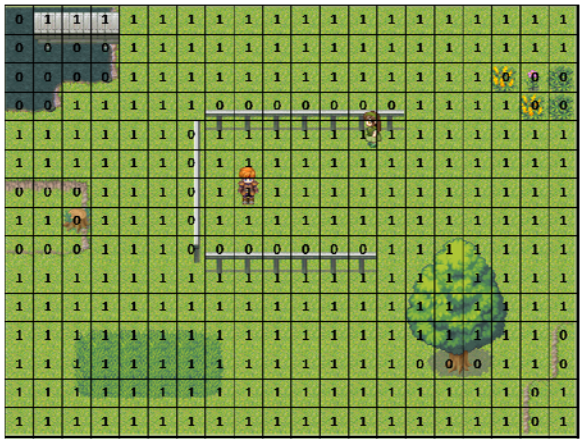
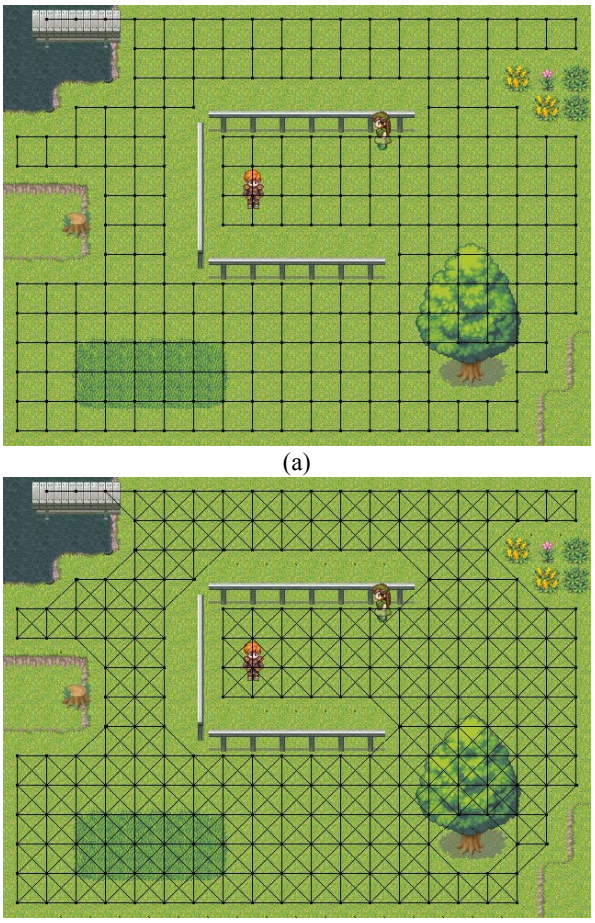


Fig. 3. Passage table of a sample map

Or we can use a directional passage table that uses more than one boolean variables in one element, 1 means passable and 0 means impassable by the object for each one direction.

Maps using this data structure have common properties: width and height, both are map dimensions. Cells have properties x-position (or column index) and y-position (or row index), i.e., two-dimensional array indices.

Graph implementation on grid map. A graph can be established from a grid map, with map cells as vertices/nodes. Edges are made based on cell passages, as shown in Fig. 4a (4-directional movement) and Fig. 4b (8-directional movement, with diagonal movements).



(a)

(b)

Fig. 4. Graph established from a grid map for (a) 4-directional movement and (b) 8-directional movement

III. PATHFINDING ALGORITHMS

There are many ways to find a path.

A. Depth-first Search

Imagine that we want to examine all rooms in a big building by visiting them as many as possible. Consider that we can always remember rooms already visited so we will never visit them again. Then we will backtrack (go back to previously visited room) if unvisited neighbor rooms aren't found. This is how depth-first search (DFS), a graph search algorithm works. The result of the search can be a spanning tree.

The algorithm is described as the following steps [8]:

1. Set initial node as current node.
2. Mark current node as visited.
3. If the current node has any neighbors which have not been visited, select an unvisited neighbor and set it as current node, and then go to step 2; otherwise backtrack the current node record and back to step 2.

Algorithm performance. For explicit graphs traversed without repetition, time performance of depth-first search

is $O(|V| + |E|)$.

Power and weakness in "pathfinding". Depth-first search requires less memory, only previous node records. It doesn't use sets of nodes and doesn't calculate distances like other pathfinding algorithms. However, depth-first search seems not to be a pathfinder since it doesn't make shortest path. Although it can be optimized by heuristic method, it still doesn't work as well as the others.

B. Breadth-first Search

Unlike depth-first search, breadth-first search explores all neighboring nodes. Then for each of those nearest nodes, it explores their unexplored neighbor nodes, and so on, until it finds the goal. The result of the search can be also a spanning tree.

The algorithm is described as the following steps [7]:

1. Set initial node as current node and enqueue it to a queue.
2. Dequeue a node from the queue and set it as current node. Terminate the search if it is goal node.
3. Enqueue all unvisited neighbors of current node and mark them as visited.
4. If queue is not empty, back to step 2.

Algorithm performance. Given a branching factor b and graph depth d, time performance of breadth-first search is $1 + b + b^2 + ... + b^d = O(b^d)$.

Power and weakness in pathfinding. Breadth-first search produces a shortest path for a non-weighted graph, but not always for a weighted graph. It is slow since it works very hard to do expansion from starting node.

C. Floyd–Warshall Algorithm

Floyd–Warshall algorithm is a graph analysis algorithm for finding shortest paths in a weighted graph (with positive or negative edge weights). A single execution of the algorithm will find the lengths (summed weights) of the shortest paths between all pairs of vertices though it does not return details of the paths themselves [10].

Consider a graph G with vertices V, each numbered 1 through N. Consider also a function shortestPath(i, j, k) that returns the shortest possible path from i to j using vertices only from the set $\{1,2,...,k\}$ as intermediate points along the way. Now, given this function, our goal is to find the shortest path from each i to each j using only vertices 1 to $k + 1$.

shortestPath(i, j, k) can be defined in terms of the following recursive formula [10]:

$$\text{shortestPath}(i, j, k) = \min(\text{shortestPath}(i, j, k-1), \text{shortestPath}(i, k, k-1) - \text{shortestPath}(k, j, k-1)) \quad (1)$$

$$\text{shortestPath}(i, j, 0) = \text{cost of edge between } i \text{ and } j \quad (2)$$

Algorithm performance. Time performance of Floyd–Warshall algorithm is $O(|V|^3)$ and it is fixed.

Power and weakness in pathfinding. Floyd–Warshall algorithm is guaranteed to find a shortest path for all cases. However it is very slow, much slower than the others do. Furthermore, for example a 20×15 grid map, which has 300 cells/nodes, takes about $300^3 = 27$ million path calculations.

D. Dijkstra's Algorithm

Dijkstra's algorithm, conceived and published by Edsger W. Dijkstra, a computer scientist, in 1959, is a graph search algorithm that finds shortest path for vertices in a graph, producing a shortest path tree [9].

The algorithm is described as the following steps:

1. Assign to every node a distance value. Set it to 0 for initial node and to Infinity for all other nodes.
2. Create empty open set and add initial node to it.
3. Find a node in open set whose distance value is smallest, set it as current node and remove it from open set.
4. For current node, consider all its neighbors and calculate their *tentative* distance (from the initial node) with formula:

$$\text{Tentative distance} = \text{Current node distance} + \text{Length of edge connecting them to current node} \quad (3)$$

   If this distance is less than the previously recorded distance (infinity in the beginning, zero for the initial node) or they aren't in open set, overwrite the distance, add them to open set, and set their previous node record as current node.
5. If open set is empty, finish. Otherwise, back to step 3.

Usage of Dijkstra's algorithm on other purposes. For graph coloring (i.e., for bitmaps), Dijkstra's algorithm can be used to color all nodes having the same color with target source node. Neighbors to select are adjacent nodes that have same color with current node. In this purpose, we call the algorithm *flood-fill*.

Dijkstra's algorithm can be used to find and construct a minimum spanning tree, where starting point is selected to be a node of minimum-cost edge of the graph. In this purpose, we call it *Prim's algorithm*.

Algorithm performance. Time performance of Dijkstra's algorithm depends on implementation of open set. Let $|E|$ number of edges and $|V|$ number of vertices/nodes. The simplest implementation of the Dijkstra's algorithm stores vertices of set Q in an ordinary linked list or array, and extract minimum from Q is simply a linear search through all vertices in Q. In this case, the worst-case time performance is $O(|V|^2 + |E|) = O(|V|^2)$. If we use implementation of open set more efficiently, worst-case time performance can be down to $O(|E| + |V| \log |V|)$.

Power and weakness in pathfinding. Dijkstra's

algorithm is guaranteed to find a shortest path for all cases. However, it works as hard as breadth-first search does to do expansion.

E. Best-first Search

Best-first search (BFS) is a graph search algorithm which explores and follows neighboring nodes that is closer to the goal, using heuristic function. We may call this a greedy process.

Judea Pearl described best-first search as estimating the promise of node *n* by a "heuristic evaluation function *f*(*n*) which, in general, may depend on the description of *n*, the description of the goal, the information gathered by the search up to that point, and most important, on any extra knowledge about the problem domain." [6]

Heuristic distance. Heuristic distance from a node to the goal is an estimated distance that can be used to predict how close the end of a path is to a solution. For a grid map, we can calculate heuristic distance using either following formulas:

$$\text{Euclidean distance heuristic} = \sqrt{(\Delta x)^2 + (\Delta y)^2} \quad (4)$$

$$\text{Manhattan distance heuristic} = \Delta x + \Delta y \quad (5)$$

where Δx and Δy are absolute value of x-position and y-position of the node and the goal, respectively.

The algorithm is described as the following steps:

1. Create empty open set and add initial node to it.
2. Create empty closed set.
3. Remove the best node from open set, call it *n*, add it to closed set.
4. If *n* is goal node terminate the search and build a path.
5. For each successor do:
   a. If it is not in closed set: evaluate it, add it to open set, and record its parent.
   b. Change recorded parent if this new path is better than previous one.
6. If queue is not empty, back to step 3.

Power and weakness in pathfinding. Guided by heuristic distance, BFS explores towards the goal and thus it does less exploration. However, in case of concave obstacles, because of greedy behavior, BFS results in a worse path. How good resulting path is depends on what heuristic function is used.

F. A Algorithm*

A* is a graph search algorithm that can potentially search a huge area of the map. It is like Dijkstra's algorithm in that it can be used to find a shortest path. It also behaves greedy like best-first search that uses heuristic function to get closer to the goal [3].

To combine behavior of Dijkstra's and BFS, A* uses distance function with formula:

$$f(x) = g(x) + h(x) \quad (6)$$

where $g(x)$ is path cost from initial node to node x as calculated by Dijkstra's algorithm and $h(x)$ is heuristic cost of node x to goal node.

The algorithm is described as the following pseudocode [5]:

```
function A*(start,goal)
    closedset := the empty set                 // The set of nodes already evaluated.
    openset := set containing the initial node // The set of tentative nodes to be evaluated.
    came_from := the empty map                 // The map of navigated nodes.
    g_score[start] := 0                        // Distance from start along optimal path.
    h_score[start] := heuristic_estimate_of_distance(start, goal)
    f_score[start] := h_score[start]           // Estimated total distance from start to goal through y.
    while openset is not empty
        x := the node in openset having the lowest f_score[] value
        if x = goal
            return reconstruct_path(came_from, came_from[goal])
        remove x from openset
        add x to closedset
        foreach y in neighbor_nodes(x)
            if y in closedset
                continue
            tentative_g_score := g_score[x] + dist_between(x,y)

            if y not in openset
                add y to openset
                tentative_is_better := true
            elseif tentative_g_score < g_score[y]
                tentative_is_better := true
            else
                tentative_is_better := false
            if tentative_is_better = true
                came_from[y] := x

                g_score[y] := tentative_g_score
                h_score[y] := heuristic_estimate_of_distance(y, goal)
                f_score[y] := g_score[y] + h_score[y]
    return failure

function reconstruct_path(came_from, current_node)
    if came_from[current_node] is set
        p := reconstruct_path(came_from, came_from[current_node])
        return (p + current_node)
    else
        return current_node
```

Algorithm performance. Time performance of A* depends on the used heuristic. It also varies for some cases.

Advantages in pathfinding. Behaving like Dijkstra's and BFS, A* produces shortest path and works efficiently. However, it may depend on the heuristic used on it.

IV. Experiments and Results

I apply pathfinding algorithms as discussed in the previous section (except Floyd–Warshall algorithm) on the grid map by programming using Game Maker, a game engine.

The test map is a 20×15 map with concave obstacle, starting position and goal position as shown in Fig. 5–10. Explored nodes will be highlighted with gray color and green highlight indicates the path.

Experiments show the path results for 4-directional movement and 8-directional movement in each two figures.

A. Depth-first Search

On a grid map, depth-first search scans nodes (cells) by selecting each one neighbor node. In experiment, DFS selects a neighbor node based on priority: west, east, north, and south neighbor, respectively. This results in path as shown in Fig. 5.

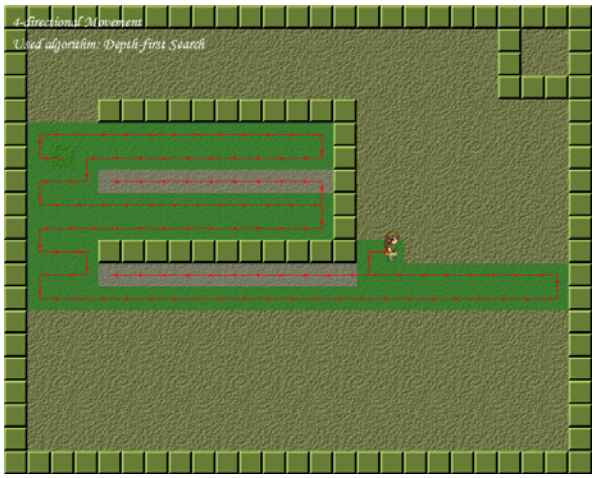
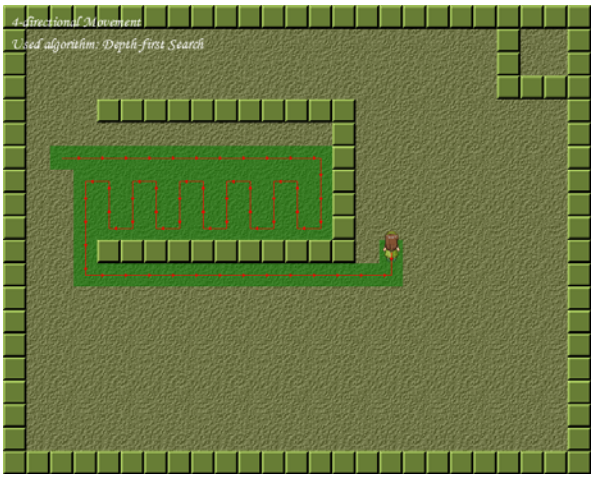
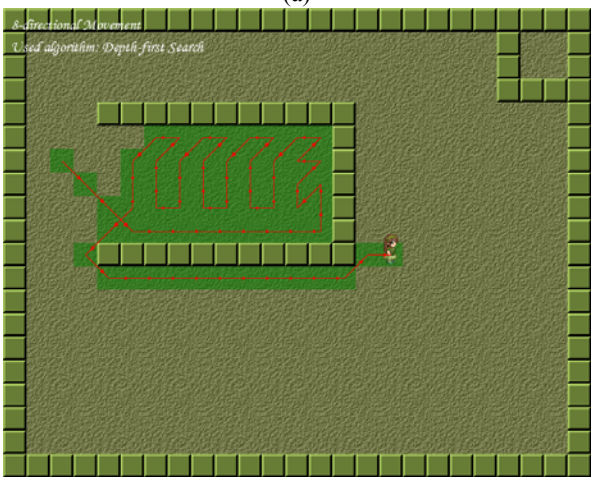


Fig. 5. Path result using depth-first search

The path in Fig. 5 seems too long. To make the path shorter, I use heuristic method (estimate distance from node to the goal) that prioritizes neighbor node with smallest heuristic distance. This leads the path to be as shown in Fig. 6a (for 4-direction movement) and Fig. 6b (for 8-direction movement).



(a)

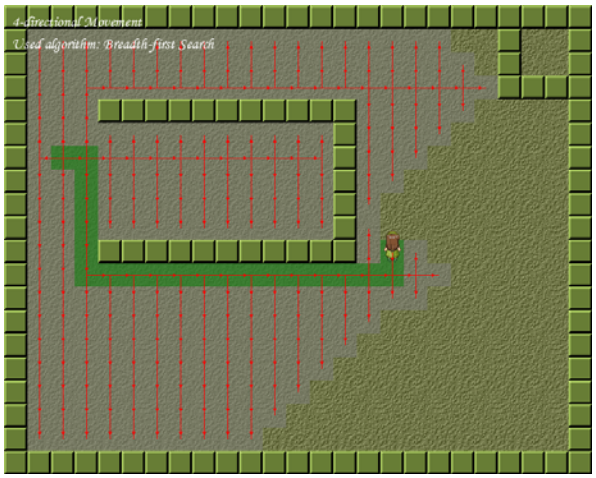


(b)

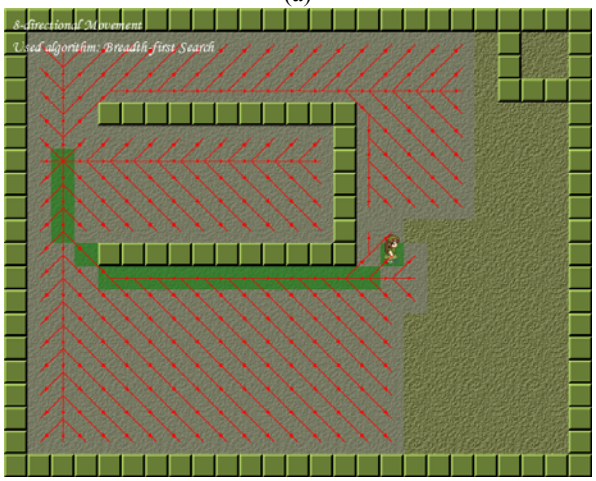
Fig. 6 Path result using depth-first search, improved with heuristic method, for (a) 4-directional movement and (b) 8-directional movement

B. Breadth-first Search

As in Fig. 7a and Fig. 7b, experiment shows that breadth-first search works very hard but results in a short(est) path.

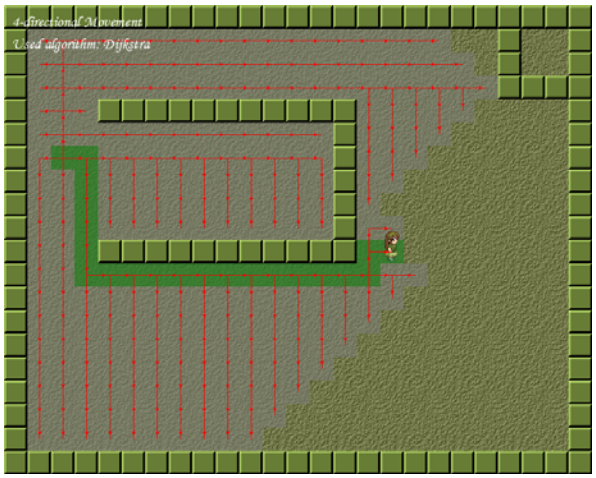


(a)

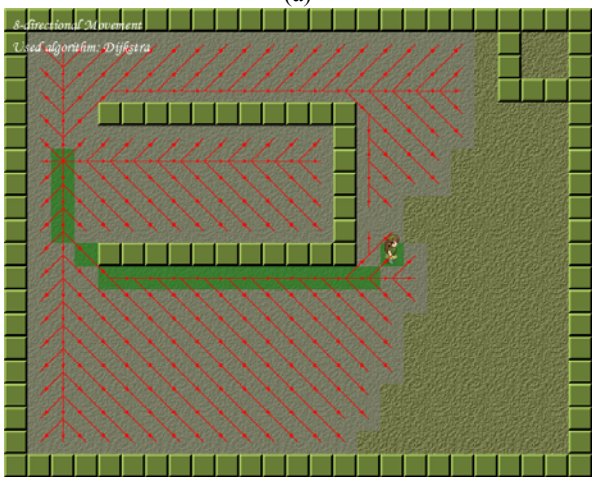


(b)

Fig. 7. Path result using breadth-first search for (a) 4-directional movement and (b) 8-directional movement



(a)



(b)

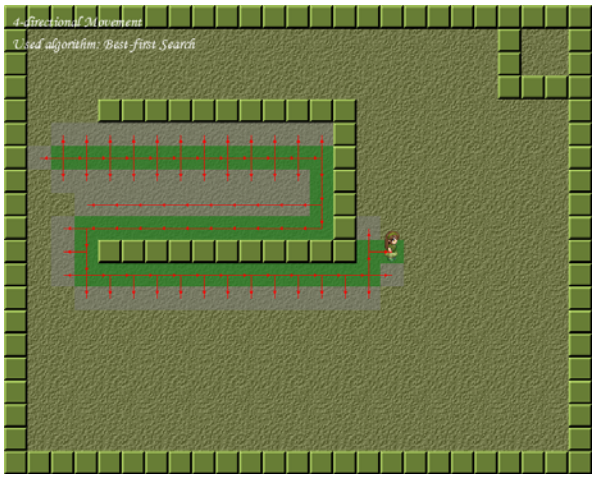
Fig. 8. Path result using Dijkstra's algorithm for (a) 4-directional movement and (b) 8-directional movement

C. Dijkstra's Algorithm

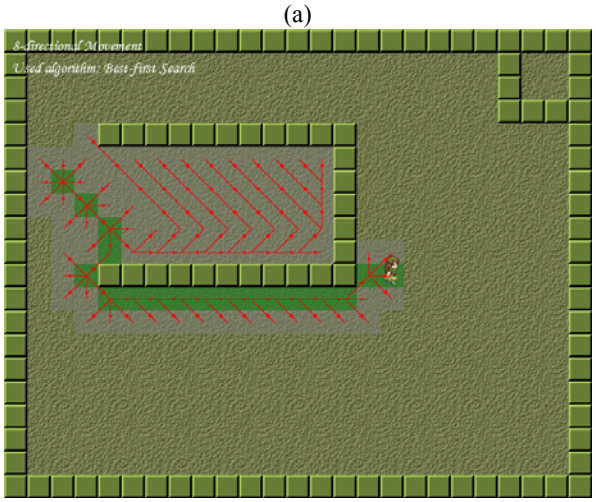
Experiment shows that Dijkstra's algorithm works as hard as breadth-first search to produce shortest path, as shown in Fig. 8a and Fig. 8b.

D. Best-first Search

Experiment shows that BFS prefers to explore towards the goal, as shown in Fig. 9a and Fig. 9b, because of its greedy behavior. Here BFS uses Euclidean distance heuristic.

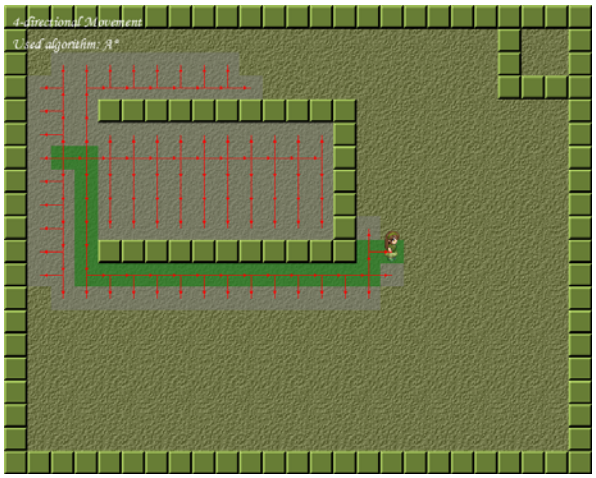


(a)

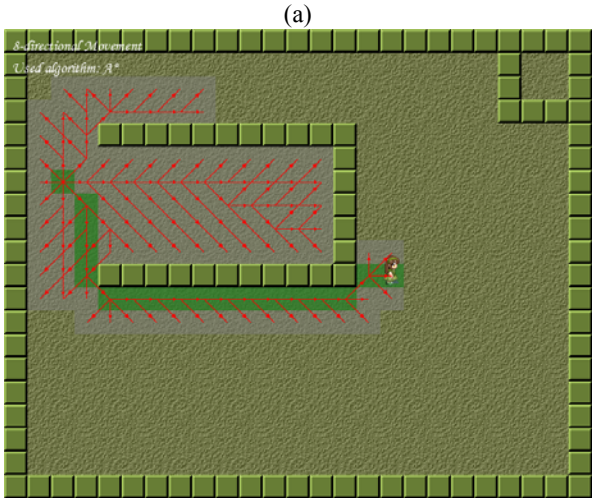


(b)

Fig. 9. Path result using best-first search for (a) 4-directional movement and (b) 8-directional movement



(a)



(b)

Fig. 10 Path result using A algorithm (8-directional movement)*

F. A Algorithm*

Experiment shows that A* results in shortest path but it doesn't work too hard, as shown in Fig. 10a and Fig. 10b. A* also uses Euclidean distance heuristic here.

V. CONCLUSION

From experiments, breadth-first search and Dijkstra's algorithm search the goal by doing expansion from starting position and produce shortest path. A* algorithm also does expansion and produce shortest path but it is guided with heuristic, so it tends to move towards the goal and thus it does less exploration than breadth-first search and Dijkstra's do. The result of experiments have confirmed that A* is the best pathfinding algorithm.

REFERENCES

[1] David M. Bourg, Glenn Seeman, *AI for Game Developers*. Gravenstein Highway North, Sebastopol: O'Reilly, 2004, ch. 7.

[2] Jobe Makar, *Macromedia® Flash™ MX Game Design Demystified: The Official Guide to Creating Games with Flash.* Berkeley, CA: Peachpit Press, 2002, ch. 9.

[3] *http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html*, accessed in 11/30/2010.

[4] *http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html*, accessed in 11/30/2010.

[5] *http://en.wikipedia.org/wiki/A*_search_algorithm*, accessed in 11/30/2010.

[6] *http://en.wikipedia.org/wiki/Best-first_search*, accessed in 11/30/2010.
[7] *http://en.wikipedia.org/wiki/Breadth-first_search*, accessed in 11/30/2010.
[8] *http://en.wikipedia.org/wiki/Depth-first_search*, accessed in 11/30/2010.
[9] *http://en.wikipedia.org/wiki/Dijkstra's_algorithm*, accessed in 11/30/2010.
[10] *http://en.wikipedia.org/wiki/Floyd–Warshall_algorithm*, accessed in 11/30/2010.

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 17 Desember 2010

Steven Andrew / 13509061