Movement Search Algorithm in Strategy RPG Video Games

Mahessa Ramadhana - 13511077
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
mahessaramadhana@itb,ac,id

***Abstract*—A strategy RPG (SRPG) is complex video game that benefits from having proper algorithm strategies. The game usually requires the player to pick a square in the map in order to move units. This movement is, however, limited, so the game needs to provide a way to check which squares each units can move to, and also to find a path from current unit's square to its destination square. This function is crucial and will be used very often, so a robust, reliable, and efficient algorithm will be very helpful. This paper will study an algorithm that can solve this problem efficiently.**

***Index Terms*—SRPG, path, search, movement.**

I. Introduction

A video game is an application which focuses more on the fun aspect. Usually, video games have appealing, interactive UI, and a lot of different, complex algorithm to provide the player with some mechanics to play with. The very first video games are very simple, due to the limits of the hardware's processing power, however as hardware becomes more powerful, more complex games have emerged. There are a lot of different kinds of video games, one of them is a strategy RPG (SRPG) game. This type of game have many complex algorithm inside, hidden from the players that enable them to play the game properly.

An SRPG is a video game where we have several units on a map. Some of those units belong to the player, while the others belong to the enemy. Each unit waits for their turn, if a player's unit gets a turn, player need to input a command to have the unit act, and if enemy's unit gets a turn, the computer will decide how the unit should act. This action may include move, attack, and many other actions depending on the video game's mechanic. The map is usually divided into squares, where each square may be occupied by one unit. In more complex games, certain squares may be more difficult or easier for certain units to move. For example, water squares are harder for normal units to move in, but aquatic units move better in water than on the ground. Some squares can't be reached by a unit no matter what, but may be accessible by other units. For example, a flying unit may go beyond a cliff and stay afloat, while ground units can't. Or a fortress that no unit can go through and thus have to find a way around. Or squares containing enemy unit where the player cannot pass.

Fig. 1 Fire Emblem, one of the very first SRPG.[1]

Some of these complex algorithm requires an algorithm strategy in order to solve it in the most efficient manner. Despite hardwares getting more powerful, a complex video game can still be tasked with a lot of computing-heavy tasks, and some video games will need to run on a portable platform with relatively limited hardware powercompared to desktops, so using an efficient algorithm is required. One of those algorithm in an SRPG is the search algorithm to find which squares each units can move to and the path to get to said square.

In an SRPG, player doesn't move each unit per step; instead, player moves each unit by selecting a square for the unit to move to. Every unit has its own move points, which indicates how far he can move. However, as mentioned earlier, certain squares have different difficulty for units to move in. A high movement ground unit still can't move very far in water, for example. This means that the game needs to do a search for squares that the unit can actually reach, based on its movement, its terrain proficiency, and the surrounding terrain that it's going to move in. Additionally, since games need to be animated,

we also need fo find the path from the unit's current square to its destination in order to animate the unit actually moving to the destination instead of just teleporting away. These problem require proper algorithm to make sure every solution is correct, and that every path is the most optimal path.

Fig. 2 Movement range in Front Mission 4[2]

In Fig. 2, we can see an example of what the game needs to show the player. The blue squares indicate squares that the unit can reach. The unit can't go to squares that are not blue.

To sum it up, the game needs an algorithm to find all reachable squares and all optimal paths to the reachable squares based on these criteria:

1. Each unit has its own movement points.
2. Each square has a cost. Visiting the square require the unit to have enough movement points.
3. Cost may be dependent on the unit's terrain proficiency.
4. If a square's total cost from the unit's current position if higher than the unit's movement points, the unit can't reach that square.

Fig. 3 Some squares cannot be reached due to obstacles.[3]

II. THEORIES

For this purpose, we will base our algorithm on Djikstra's algorithm, modified to fit the four criteria above and the game's mechanic.

Djikstra's algorithm is an algorithm that can be used to find the shortest-path and the distance from a node to every other node in a graph. To accomplish this, Djikstra's algorithm uses several terms and steps to describe and solve the problem.

A node has a distance which represents an estimation of its distance from the source through an estimated shortest-path. It is only "estimated" because it may not always hold the shortest-distance throughout the algorithm, but it will end up holding the shortest-distance when the algorithm ends.

A node also has a status label. A node's status begins as unvisited. As the algorithm runs, each node that the algorithm expands is marked as visited. A visited node longer be checked.

A node also has a predecessor. This represents, in the shortest-path including said node, the node before the said node. For example, if the shortest path is A-B-C-F, then F's predecessor would be C, and C's predecessor would be C.

Current node is the node which is currently expanded. We will cover more about expanding a node in the steps explanation below.

With these terms, Djikstra algorithm work in these steps:

1. Assign the source node's distance to zero.
2. Assign all the other node's distance to infinity.
3. Mark all nodes unvisited.
4. Set source node as the current node.
5. Expand current node by checking all of its unvisited neighbors and calculate their distance. For example, if current node is A with a distance of 3, and the edge connecting to B has a weight of 4, then B's distance through A is 7.
6. For each neighbor, if the distance calculated is lower than its current distance, update its distance, and set its predecessor as the current node.
7. After all neighbors has been calculated, mark the current node as visited.
8. Select an unvisited node with the smallest distance.
9. If all nodes has been visited, or if the selected node's distance is infinity, then stop the algorithm. Otherwise, set it as current node. Repeat step 5.

At the end of the algorithm, all nodes will have their distance set as the shortest distance to source, their predecessor set as the nodes that come before them in the shortest-path, and all nodes unreachable from source node will still hold infinity as their distance.

An example of how Djikstra's algorithm works:

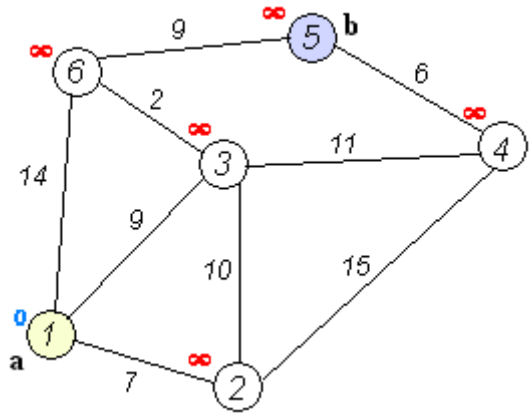

Fig 3. Graph for Djikstra example [4]

Using this graph, first we set all node's distance to infinite, but set the source to zero. Also, set all nodes as unvisited. Set source node as the current node. Therefore we have this table (* next to a node indicates it's the current node):

Node	Distance	Visited	Predecessor
1*	0	F	-
2	INF	F	-
3	INF	F	-
4	INF	F	-
5	INF	F	-
6	INF	F	-

Table 1a. Djikstra table.

Next, we check all of current nodes' unvisited neighbors. In this case, we check nodes 2, 3, and 6, calculate their distance and update the table if necessary.

From the calculation, we found that node 2, 3, and 6's distances are 7, 9, and 16 respectively. These are lower than infinity, so update node 2, 3, and 6's distance in the table and set their predecessor to current node. Mark current node as visited. Then, check all unvisited node, pick one with the smallest distance, and set it as current node.

Node	Distance	Visited	Predecessor
1	0	T	-
2*	7	F	1
3	9	F	1
4	INF	F	-
5	INF	F	-
6	14	F	1

Table 1b. Djikstra table.

Repeat the above step; this time, we check node 2's unvisited neighbors 3 and 4 (node 1 is a visited neighbor s we do not check it). We calculate that through node 2, node 3 and 4's distances are 17 and 22 respectively. 17 is larger than 9, so we do not update node 3, but 22 is smaller than INF, so we update node 4's distance to 22 and set its predecessor to 2. Mark current node as visited, pick one unvisited node with the smallest distance, and set it as current node.

Node	Distance	Visited	Predecessor
1	0	T	-
2	7	T	1
3*	9	F	1
4	22	F	2
5	INF	F	-
6	14	F	1

Table 1c. Djikstra table.

Now we expand node 3. Node 3's unvisited neighbors are node 4 and 6. Their distances through 3 are 20 and 11 respectively. Both of these values are smaller than their counterpart in the table, so update the table accordingly.

Node	Distance	Visited	Predecessor
1	0	T	-
2	7	T	1
3	9	T	1
4	20	F	3
5	INF	F	-
6*	11	F	3

Table 1d. Djikstra table.

Repeat the steps until done.

Node	Distance	Visited	Predecessor
1	0	T	-
2	7	T	1
3	9	T	1
4*	20	F	3
5	20	F	6
6	11	T	3

Table 1e. Djikstra table.

Node	Distance	Visited	Predecessor
1	0	T	-
2	7	T	1
3	9	T	1
4	20	T	3
5*	20	F	6
6	11	T	3

Table 1f. Djikstra table.

Node	Distance	Visited	Predecessor
1	0	T	-
2	7	T	1
3	9	T	1
4	20	T	3
5	20	T	6
6	11	T	3

Table 1g. Djikstra table.

With this, we now have the distance from node 1 to every other node, and we can also determine the shortest path. For example, to get to 5, we can see that 6 comes before 5, 3 comes before 6, and 1 comes before 3, so the path is 1-3-6-5.

While Djikstra's algorithm certainly works on such a graph, if we were to use it on an SRPG, we have to modify it in order fit the criteria described above. While the basic principle is the same, there are a lot of difference is an SRPG mechanic with a graph.

III. Movement Search Algorithm

We have covered the basics of movement in an SRPG and how Djikstra's algorithm in general works. However, we now need to modify Djikstra's algorithm in order to get the result we want.

First of all, we assume each square is a node in a graph. Very obvious, since each square acts exactly like a node in a graph. We can go from square to square just like we go from node to node.

Since we do not have adjacency matrix, we set the weight of an edge between two nodes as the cost of entering the destination node.

10	10	15
20	20	25
10	15	INF

Table 2 Squares on a map with their costs.

0	10	–	20	–	–	–	–	–
10	0	15	–	20	–	–	–	–
–	10	0	–	–	25	–	–	–
10	–	–	0	20	–	10	–	–
–	10	–	20	0	25	–	15	–
–	–	15	–	20	0	–	–	–
–	–	–	20	–	–	0	15	–
–	–	–	–	20	–	10	0	–
–	–	–	–	–	–	–	–	–

Table 3 Adjacency matrix from Table 2

For example, if squares on a map and their costs are represented in Table 2, we can make an adjacency matrix like the one in table 3. In practice though, we do not need to make such adjacency matrix. All we need at the neighbors distance calculating step is to find each neighbors cost and use it to calculate the distance. It should be noted that, as mentioned above, the cost of each square may be dependent upon the unit's terrain proficiency. So unit A and unit B may have different costs for each square.

Next modification is to change the table approach to a list approach. Instead of having a table containing all squares, their distances, their status, and their predecessor, we make a list instead. Our step becomes like this:

1. Make a new list of squares. Each entry in the list contains the square, the distance, status, and predecessor.
2. Push the unit's square into the list, set the distance to zero, status unvisited.
3. Set current square to unit's square.
4. Expand current square by checking all of its neighboring squares. Calculate the distance to each neighbors.
5. For every unvisited neighbors, first check if the distance is smaller or equal to the unit's movement points, then check if it is already in the list. If it is, update distance and predecessor if the calculated distance is smaller than the one in the list. If not, add the square to the list with its distance and predecessor, set it to unvisited.
6. After all neighbors has been calculated, mark the current node as visited.
7. From the list, look for a square that is unvisited and has the lowest distance.
8. If all squares are already visited, then stop. Otherwise, set it as current square. Repeat step 4.

Using this steps, we will stop the search once all reachable squares has been visited, and skip expanding squares which is not yet determined to be reachable. Otherwise, the algorithm will search for all squares on the map, which is very time consuming. Most of the time, the reachable area is just a fraction of the whole map, so using this limit will speed up the calculation considerably.

Another modification we can make is at step 7. If we know the minimum cost of a square, we can ignore squares whose distance + minimum square is larger than the unit's movement points. This modification is not applicable if we do not know the minimum cost of square.

10	21	20	25	20	15	10
15	20	20	25	25	20	20
20	15	INF	20	10	20	15
25	20	10	S	10	10	20
15	25	INF	INF	10	15	25
20	10	15	10	10	15	10
25	15	15	20	25	10	15

Table 4 Example squares

We test the algorithm above on the map above. Suppose the unit's movement points is 50. The minimum cost of a

square is 10. First, we have a list containing only the S square (3,3) and its information

Square	Distance	Visited	Predecessor
3,3*	0	F	-

Table 5a Movement search table

Using the steps defined above, we get:

Square	Distance	Visited	Predecessor
3,3	0	T	-
3,2	20	F	3,3
2,3*	10	F	3,3
4,3	10	F	3,3

Table 5b Movement search table

Keep repeating the steps until the algorithm is done.

Square	Distance	Visited	Predecessor
3,3	0	T	-
3,2	20	F	3,3
2,3	10	T	3,3
4,3*	10	F	3,3
1,3	30	F	2,3

Table 5c Movement search table

Square	Distance	Visited	Predecessor
3,3	0	T	-
3,2*	20	F	3,3
2,3	10	T	3,3
4,3	10	T	3,3
1,3	30	F	2,3
4,2	20	F	4,3
5,3	20	F	4,3
4,4	20	F	4,3

Table 5d Movement search table

Square	Distance	Visited	Predecessor
3,3	0	T	-
3,2	20	T	3,3
2,3	10	T	3,3
4,3	10	T	3,3
1,3	30	F	2,3
4,2*	20	F	4,3
5,3	20	F	4,3
4,4	20	F	4,3
3,1	45	F	3,2

Table 5e Movement search table

Square	Distance	Visited	Predecessor
3,3	0	T	-
3,2	20	T	3,3
2,3	10	T	3,3
4,3	10	T	3,3
1,3	30	F	2,3
4,2	20	T	4,3
5,3*	20	F	4,3
4,4	20	F	4,3
3,1	45	F	3,2
4,1	45	F	4,2
5,2	40	F	4,2

Table 5f Movement search table

Square	Distance	Visited	Predecessor
3,3	0	T	-
3,2	20	T	3,3
2,3	10	T	3,3
4,3	10	T	3,3
1,3	30	F	2,3
4,2	20	T	4,3
5,3	20	T	4,3
4,4*	20	F	4,3
3,1	45	F	3,2
4,1	45	F	4,2
5,2	40	F	4,2
6,3	40	F	5,3
5,4	35	F	5,3

Table 5g Movement search table

Square	Distance	Visited	Predecessor
3,3	0	T	-
3,2	20	T	3,3
2,3	10	T	3,3
4,3	10	T	3,3
1,3*	30	F	2,3
4,2	20	T	4,3
5,3	20	T	4,3
4,4	20	T	4,3
3,1	45	F	3,2
4,1	45	F	4,2
5,2	40	F	4,2
6,3	30	F	5,3
5,4	35	F	5,3
4,5	30	F	4,4

Table 5h Movement search table

Square	Distance	Visited	Predecessor
3,3	0	T	-
3,2	20	T	3,3
2,3	10	T	3,3
4,3	10	T	3,3
1,3	30	T	2,3
4,2	20	T	4,3
5,3	20	T	4,3
4,4	20	T	4,3
3,1	45	F	3,2
4,1	45	F	4,2
5,2	40	F	4,2
6,3	40	F	5,3
5,4	35	F	5,3
4,5*	30	F	4,4
1,2	45	F	1,3

Table 5i Movement search table

Square	Distance	Visited	Predecessor
3,3	0	T	-
3,2	20	T	3,3
2,3	10	T	3,3
4,3	10	T	3,3
1,3	30	T	2,3
4,2	20	T	4,3
5,3	20	T	4,3
4,4	20	T	4,3
3,1	45	F	3,2
4,1	45	F	4,2
5,2	40	F	4,2
6,3	40	F	5,3
5,4*	35	F	5,3
4,5	30	T	4,4
1,2	45	F	1,3
3,5	40	F	4,5
5,5	45	F	4,5

Table 5j Movement search table

Square	Distance	Visited	Predecessor
3,3	0	T	-
3,2	20	T	3,3
2,3	10	T	3,3
4,3	10	T	3,3
1,3	30	T	2,3
4,2	20	T	4,3
5,3	20	T	4,3
4,4	20	T	4,3
3,1	45	F	3,2
4,1	45	F	4,2
5,2*	40	F	4,2
6,3	40	F	5,3
5,4	35	T	5,3
4,5	30	T	4,4
1,2	45	F	1,3
3,5	40	F	4,5
5,5	45	F	4,5

Table 5k Movement search table

Square	Distance	Visited	Predecessor
3,3	0	T	-
3,2	20	T	3,3
2,3	10	T	3,3
4,3	10	T	3,3
1,3	30	T	2,3
4,2	20	T	4,3
5,3	20	T	4,3
4,4	20	T	4,3
3,1	45	F	3,2
4,1	45	F	4,2
5,2	40	T	4,2
6,3*	40	F	5,3
5,4	35	T	5,3
4,5	30	T	4,4
1,2	45	F	1,3
3,5	40	F	4,5
5,5	45	F	4,5

Table 5l Movement search table

Square	Distance	Visited	Predecessor
3,3	0	T	-
3,2	20	T	3,3
2,3	10	T	3,3
4,3	10	T	3,3
1,3	30	T	2,3
4,2	20	T	4,3
5,3	20	T	4,3
4,4	20	T	4,3
3,1	45	F	3,2
4,1	45	F	4,2
5,2	40	T	4,2
6,3	40	T	5,3
5,4	35	T	5,3
4,5	30	T	4,4
1,2	45	F	1,3
3,5*	40	F	4,5
5,5	45	F	4,5

Table 5m Movement search table

Square	Distance	Visited	Predecessor
3,3	0	T	-
3,2	20	T	3,3
2,3	10	T	3,3
4,3	10	T	3,3
1,3	30	T	2,3
4,2	20	T	4,3
5,3	20	T	4,3
4,4	20	T	4,3
3,1	45	F	3,2
4,1	45	F	4,2
5,2	40	T	4,2
6,3	40	T	5,3
5,4	35	T	5,3
4,5	30	T	4,4
1,2	45	F	1,3
3,5	40	T	4,5
5,5	45	F	4,5

Table 5n Movement search table

We end the algorithm at this stage since the remaining unvisited nodes' distances plus the minimum cost of a square is larger than the unit's movement points. Thus, the reachable squares are:

10	21	20	25	20	15	10
15	20	20	25	25	20	20
20	15	INF	20	10	20	15
25	20	10	S	10	10	20
15	25	INF	INF	10	15	25
20	10	15	10	10	15	10
25	15	15	20	25	10	15

Table 6 Reachable squares

We can also determine the path that the unit should take to get to a certain square. For example, to get to (3,5), we see that (4,5) comes before (3,5), (4,4) comes before (4,5), (4,3) comes before (4,4), and (3,3) comes before (4,3). Thus, the path to get to (3,5) is

(3,3) – (4,3) – (4,4) – (4,5) – (4,5).

This is also used during the enemy turn, for the computer to determine where can its units move to. For the computer side, we also need an algorithm to determine the best square to go to out of all the reachable squares, however, it is beyond the scope of this paper.

IV. Conclusion

An SRPG requires an algorithm to search for squares that units can reach. This search algorithm can be derived from Djikstra's shortest-path algorithm with some modifications to fit the problem better. The resulting search is accurate and efficient..

References

[1] http://www.gamefaqs.com/nes/562649-fire-emblem-ankoku-ryu-to-hikari-no-tsurugi/images/screen-7

[2] http://www.gamefaqs.com/ps2/918883-front-mission-4/images/screen-4

[3] http://vglounge.com/wp-content/uploads/2012/04/Hoshigami-3.jpg

[4] http://en.wikipedia.org/wiki/File:Dijkstra_Animation.gif

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, 20 Desember 2013

ttd

Mahessa Ramadhana 13511077