Dijkstra’s Algorithm Application on the Pac-Man Game

Mahardiansyah Kartika (13509041)  
Program Studi Teknik Informatika  
Sekolah Teknik Elektro dan Informatika  
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia  
mahardiansyah.k@students.itb.ac.id

Abstract—This paper will explain about Dijkstra’s algorithm application to find the shortest path on the Pac-Man game. Finding the shortest path is needed because it can minimize the time of searching process. Many algorithm can be applied to find the shortest path, but the most used is Dijkstra’s algorithm. These algorithm find the shortest path in a number of steps and in each step we will choose the minimum side and put it into a set of solutions.

Index Terms—Dijkstra’s Algorithm, Pac-Man, Enemy, Shortest Path.

I. INTRODUCTION

Graph theory is an old subject but has a lot of applied until now. The first problem using graph is the problem of Konisberg bridge. Swiss mathematician, L.Euler was the first person who find the answer of that problem with simple proof. On that problem, graph used to illustrate the problems then that problem will be resolved.

There are many applications related to the graph. In that application, graph is used to represent or illustrate the problem. There are many applications related with path in graph, one of which is to determine the shortest path.

Finding shortest path in graph is an optimization problem. Graph which used to find the shortest path is weighted graph. Weighted graph is graph that each side is given a weight or value. Many algorithms can be applied to find the the shortest path, but the most used is Dijkstra’s algorithm. This algorithm uses the principle of Greedy. Greedy principle on Dijkstra’s algorithm states that at each step we choose the side of minimum weight and put it into a set of solutions. [1]

One example of Dijkstra’s algorithm is used for artificial intelligence in Pac-Man game to find the shortest path.

II. PAC-MAN GAME

The player controls Pac-Man through a maze, eating pac-dots. When all dots are eaten, Pac-Man is taken to the next stage, between some stages one of three intermission animations plays. Four enemies (Blinky, Pinky, Inky and Clyde) roam the maze, trying to catch Pac-Man. If an enemy touches Pac-Man, a life is lost. When all lives have been lost, the game ends. Pac-Man is awarded a single bonus life at 10,000 points by default—DIP switches inside the machine can change the required points or disable the bonus life altogether. Near the corners of the maze are four larger, flashing dots known as power pellets that provide Pac-Man with the temporary ability to eat the enemies. The enemies turn deep blue, reverse direction and usually move more slowly. When an enemy is eaten, its eyes remain and return to the center box where it is regenerated in its normal color. Blue enemies flash white before they become dangerous again and the amount of time the enemies remain vulnerable varies from one stage to the next, but the time period generally becomes shorter as the game progresses. In later stages, the enemies do not change colors at all, but still reverse direction when a power pellet is eaten.

In addition to dots and power pellets, bonus items, usually in the form of fruit appear near the center of the maze. These items score extra bonus points when eaten. The items change and bonus values increase throughout the game. [2]
The enemies in Pac-Man are known variously as "ghosts" and "monsters". Despite the seemingly random nature of the enemies, their movements are strictly deterministic, which players have used to their advantage. In an interview, creator Toru Iwatani stated that he had designed each enemy with its own distinct personality in order to keep the game from becoming impossibly difficult or boring to play. The behaviors of each enemy have been exactly determined by reverse-engineering the game. [2]

Pac-Man supposedly has no ending—as long as the player keeps at least one life, he or she should be able to continue playing indefinitely. However, this is rendered impossible by a bug. Normally, no more than seven fruits are displayed on the side of the screen at any one time, but when the internal level counter (stored in a single byte) reaches 255, the subroutine erroneously causes this value to "roll over" to zero before drawing the fruit. This causes the routine to attempt to draw 256 fruits, which corrupts the bottom of the screen and the whole right half of the maze with seemingly random symbols, making the level unwinnable. However, through additional analysis, it has been revealed what happens should the 256th level be cleared - the fruits and intermissions would restart from level 1 conditions, but the enemies would retain their higher speed and invulnerability to power pellets from the higher stages. [2]

A perfect Pac-Man game occurs when the player achieves the maximum possible score on the first 255 levels (by eating every possible dot, power pellet, fruit, and enemy) without losing a single life, and then scoring as many points as possible in the last level. As verified by the Twin Galaxies International Scoreboard on July 3, 1999, the first person to achieve this maximum possible score (3,333,360 points) was Billy Mitchell of Hollywood, Florida, who performed the feat in about six hours.

In September 2009, David Race of Beavercreek, Ohio, became the sixth person to achieve a perfect score. His time of 3 hours, 41 minutes, and 22 seconds set a new record for the fastest time to obtain a perfect score.

In December 1982, an 8-year-old boy, Jeffrey R. Yee, supposedly received a letter from U.S. President Ronald Reagan congratulating him on a worldwide record of 6,131,940 points, a score only possible if he had passed the Split-Screen Level. Whether or not this event happened as described has remained in heated debate among video-game circles since its supposed occurrence.

III. DIJKSTRA'S ALGORITHM

Dijkstra's algorithm, conceived by Dutch computer scientist Edsger Dijkstra in 1956 and published in 1959, is a graph search algorithm that solves the single-source shortest path problem for a graph with nonnegative edge path costs, producing a shortest path tree. This algorithm is often used in routing. An equivalent algorithm was developed by Edward F. Moore in 1957. [3]

For a given source vertex (node) in the graph, the algorithm finds the path with lowest cost (i.e. the shortest path) between that vertex and every other vertex. It can also be used for finding costs of shortest paths from a...
Suppose you want to find the shortest path between two intersections on a city map, a starting point and a destination. The order is conceptually simple: to start, mark the distance to every intersection on the map with infinity. This is done not to imply there is an infinite distance, but to note that that intersection has not yet been visited. (Some variants of this method simply leave the intersection unlabeled.) Now, at each iteration, select a current intersection. For the first iteration the current intersection will be the starting point and the distance to it (the intersection's label) will be zero. For subsequent iterations (after the first) the current intersection will be the closest unvisited intersection to the starting point—this will be easy to find.

From the current intersection, update the distance to every unvisited intersection that is directly connected to it. This is done by determining the sum of the distance between an unvisited intersection and the value of the current intersection, and relabeling the unvisited intersection with this value if it is less than its current value. In effect, the intersection is relabeled if the path to it through the current intersection is shorter than the previously known paths. To facilitate shortest path identification, in pencil, mark the road with an arrow pointing to the relabeled intersection if you label/relabel it, and erase all others pointing to it. After you have updated the distances to each neighboring intersection, mark the current intersection as visited and select the unvisited intersection with lowest distance (from the starting point) -- or lowest label— as the current intersection. Nodes marked as visited are labeled with the shortest path from the starting point to it and will not be revisited or returned to.

Continue this process of updating the neighboring intersections with the shortest distances, then marking the current intersection as visited and moving onto the closest unvisited intersection until you have marked the destination as visited. Once you have marked the destination as visited (as is the case with any visited intersection) you have determined the shortest path to it, from the starting point, and can trace your way back following the arrows in reverse.

In the accompanying animated graphic, the starting and destination intersections are colored in light pink and blue and labelled a and b respectively. The visited intersections are colored in red, and the current intersection in a pale blue. [3]
6 dist[source] := 0 ;
// Distance from source to source
7 Q := the set of all nodes in Graph ;
// All nodes in the graph are unoptimized – thus are in Q
8 while Q is not empty:
// The main loop
9 u := vertex in Q with smallest dist[] ;
10 if dist[u] = infinity:
11 break ;
// all remaining vertices are inaccessible from source
12 fi ;
13 remove u from Q ;
14 for each neighbor v of u:
// where v has not yet been removed from Q.
15 alt := dist[u] + dist_between(u, v) ;
16 if alt < dist[v]:
// Relax (u,v,a)
17 dist[v] := alt ;
18 previous[v] := u ;
19 fi ;
20 end for ;
21 end while ;
22 return dist[] ;
23 end Dijkstra.

Figure 5. Dijkstra's Pseudocode

If we are only interested in a shortest path between vertices source and target, we can terminate the search at line 13 if u = target. Now we can read the shortest path from source to target by iteration:

1 S := empty sequence
2 u := target
3 while previous[u] is defined:
4 insert u at the beginning of S
5 u := previous[u]

Figure 6. Iteration

Now sequence S is the list of vertices constituting one of the shortest paths from target to source, or the empty sequence if no path exists.

A more general problem would be to find all the shortest paths between source and target (there might be several different ones of the same length). Then instead of storing only a single node in each entry of previous[] we would store all nodes satisfying the relaxation condition. For example, if both r and source connect to target and both of them lie on different shortest paths through target (because the edge cost is the same in both cases), then we would add both r and source to previous[target]. When the algorithm completes, previous[] data structure will actually describe a graph that is a subset of the original graph with some edges removed. Its key property will be that if the algorithm was run with some starting node, then every path from that node to any other node in the graph is the same as the shortest path between those nodes in the original graph, and all paths of that length from the original graph will be present in the new graph. Then to actually find all these short paths between two given nodes we would use a path finding algorithm on the new graph, such as depth-first search. [3]

B. Running Time

An upper bound of the running time of Dijkstra's algorithm on a graph with edges E and vertices V can be expressed as a function of | E | and | V | using the Big-O notation.

For any implementation of set Q the running time is , where dQ and emQ are times needed to perform decrease key and extract minimum operations in set Q, respectively.

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 running time is O( | V | 2 + | E | ) = O( | V | 2).

For sparse graphs, that is, graphs with far fewer than O( | V | 2) edges, Dijkstra's algorithm can be implemented more efficiently by storing the graph in the form of adjacency lists and using a binary heap, pairing heap, or Fibonacci heap as a priority queue to implement extracting minimum efficiently. With a binary heap, the algorithm requires O(( | E | + | V | ) log | V | ) time (which is dominated by O( | E | log | V | ), assuming the graph is connected), and the Fibonacci heap improves this to O( | E | + | V | log | V | ).

Note that for Directed acyclic graphs, it is possible to find shortest paths from a given starting vertex in linear time, by processing the vertices in a topological order, and calculating the path length for each vertex to be the minimum or maximum length obtained via any of its incoming edges. [3]

IV. IMPLEMENTATION

First step to implementation Dijkstra’s algorithm on the Pac-Man game is visualize the maze as a graph which has edges and vertices. Dijkstra’s algorithm works first by finding the shortest path from source vertice (which inhabited the enemy) to target vertice (which inhabited by Pac-Man) by moving one step down each immediate reach the edge. Then, on that vertice the enemy do the same and doing that continously until catch the Pac-Man.

There are many optimization that can be dony by the enemy to catch Pac-Man like consider where the position
of Pac-Man previously and where Pac-Man will move so the enemy can get the shortest path to catch Pac-Man. Another way is to try to find all the immediately vertex reachable from Pac-Man and the enemy try to cover as many of these vertices by enemy. So, the possibility of Pac-Man to escape become smaller.

V. CONCLUSION

Dijkstra’s algorithm is the shortest path search algorithm which is used in the directed graph as well as for the not directed graph. This algorithm can be implemented very well in Pac-Man game to determine the shortest path to be taken by enemy to catch Pac-Man that moves continuously. First, modeled the system into a weighted graph, then execute Dijkstra’s algorithm to determine the shortest path with the minimum running time.

VI. ACKNOWLEDGMENT

Thanks to Allah for His guidance and mercy for giving me a lot of ability to finish this “Struktur Diskrit” paper and then I would like to thanks for my parents and teacher who supporting me in knowledge and material. Grateful to all of my friends who always helped me when I have difficulty in doing this paper.

REFERENCES


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, 15 Desember 2010

Mahardiansyah Kartika (13509041)