Optimizing Grid-Based Pathfinding with Visibility Graph and Floyd-Warshall Algorithm

Felix Limanta 13515065
Program Studi Teknik Informatika
Institut Teknologi Bandung
Bandung, Indonesia
felixlimanta@gmail,com, 13515065@std.stei.itb.ac.id

***Abstract*—This paper discusses improvements to the pathfinding process on uniform-cost grid maps. Instead of modifying the pathfinding algorithm, the graph generation is discussed to minimize the number of nodes in the pathfinding graph and frontload some of the process for faster dynamic pathfinding. This paper first discusses the concept of visibility graph and pathfinding, then attempts to implement them in an algorithm. Then, a simple experiment is held to compare the performances of pathfinding without a visibility graph, with a visibility graph, and with precomputed paths. Finally, the results are compared and discussed.**

***Keywords*—pathfinding; visibility graph; A*; Floyd-Warshall**

I. Introduction

Pathfinding is a fundamental part of many important applications in the fields of, for examples, GPS, video games, robotics, and logistics. Pathfinding has been and can be implemented in static, dynamic, and real-time environments. Although recent developments have improved the accuracy and efficiency of pathfinding techniques over the past few decades, the problem still attracts research. Generally, pathfinding consists of two main steps: graph generation and a pathfinding algorithm.

Graph generation refers to the process of modelling actual environments to a graph useable by a pathfinding algorithm. A popular representation of pathfinding algorithms is the uniform-cost grid maps, which is widely used in areas such as robotics, artificial intelligence, and video games.

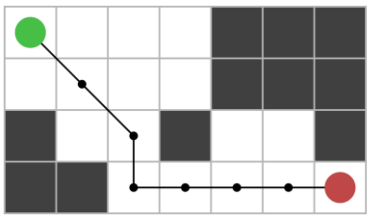

Fig. 1. Grid-based pathfinding example
Source: www.growingwiththeweb.com

The second step in the pathfinding process is the pathfinding algorithm itself. Here, the problem is to return the shortest (optimal) path from a defined origin point to a destination point in an efficient manner. There are a variety of pathfinding algorithms for various situations. The most notable algorithm for finding the shortest path between two vertices is A*, which uses a heuristic function to travel a search graph in a best-first manner until a goal node is found.

However, basic pathfinding algorithms applied wholesale to the entire grid maps can be prohibitively expensive. Furthermore, basic pathfinding algorithms do not cope well with any changes to the map, such as changes to the origin and destination points, which invalidates the whole pathfinding process.

The pathfinding discussed in this paper is dynamic pathfinding, where the destination point moves dynamically. Consequently, paths must be recalculated every time the destination point moves.

This paper discusses improvements on basic pathfinding done on uniform-cost grid maps. Instead of discussing changes to the pathfinding algorithm, this paper discusses better graph generation for a more efficient pathfinding. Specifically, this paper discusses skeletonizing a grid map to a visibility graph, then precomputing the shortest path between all pairs of vertices in said visibility graph.

The paper is organized as follows. Section II reviews the basic concepts relevant to this paper, which includes visibility graphs and pathfinding algorithms (A* and Floyd-Warshall). Section III how to implement them in a computer program. Section IV presents a sample experiment to test the performance between pathfinding without a visibility graph, pathfinding with a visibility graph, and pathfinding with precomputed paths. Section V then analyzes the results from the experiment and suggests further improvements. Finally, section VI concludes this paper.

II. Basic Concepts

A. *Visibility Graph*

A visibility graph is a graph of intervisible locations. In this case, the visibility graph of a map is a graph composed of each junction in the map. A junction is any corner or intersection in the map.

Visibility graphs serve to reduce the total number of vertices to check. Without a visibility graph, a pathfinding algorithm must check every tile in a map, which is very costly in terms of computational time and space. Using a visibility graph, a pathfinding algorithm finds the shortest path using only the tiles associated with the vertices in the graph, with two additional points: the origin and the destination, which is constantly updated in real-time pathfinding.

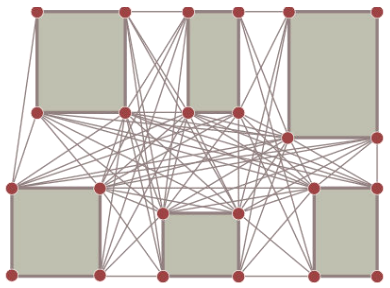

Fig. 2. Visibility graph for polygonal map representation
Source: [4]

B. A*

A* is an algorithm for finding the shortest path between a pair of vertices (single-pair shortest-path). A* is based on Djikstra's Algorithm, which is intended to find the shortest path from a vertex to any other vertex (single source shortest path).

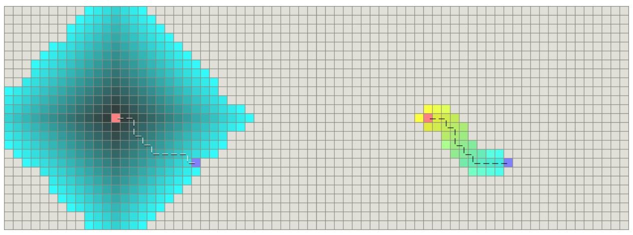

Fig. 3. Tiles checked by Djikstra's (left) and A* (right)
Source: [4]

A* is an informed search algorithm. It finds a solution by searching among all possible paths to the destination for the least expensive. Among all available paths, it first considers the ones that appear to lead faster to the solution. A* is applied on a weighted graph: starting from a specific vertex, it constructs a tree of paths starting from that vertex while expanding paths one vertex at a time, until one of its paths ends at the destination vertex.

A* selects a path which minimizes

$$f(n) = g(n) + h(n)$$

where *g(n)* is cost of going from the origin to the last vertex on the path and *h(n)* is the estimated cost from said vertex to the destination. The heuristic is problem-specific. For the algorithm to find the actual shortest path, the heuristic must be admissible, which means it won't overestimate the actual cost to the destination.

For grid-based pathfinding with four movement possibilities, an admissible heuristic is the Manhattan Distance, which is the distance between points strictly using horizontal and vertical movement. This paper uses Manhattan Distance as its heuristic.

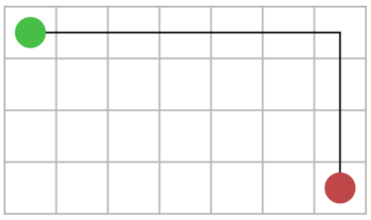

Fig. 4. Manhattan Distance
Source: www.growingwiththeweb.com

A* has a worst-case time complexity of O(|E|) and a worst-case space complexity of O(|V|).

C. Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is an algorithm for finding the shortest paths for all pairs of vertices (all-pair shortest-path). The algorithm is considered applications of dynamic programming.

Floyd-Warshall uses the adjacency matrix D^0 as its input. If there is an edge between vertices *i* and *j*, D^0_{ij} contains its length; otherwise, D^0_{ij} is set to positive infinity.

In each iteration, this matrix is recalculated such that it contains the cost of paths among all pairs of nodes using a gradually enlarging set of intermediate nodes. The matrix D^1 contains costs among all nodes using at most 1 intermediate node. D^n contains costs using at most n intermediate nodes.

This transformation can be described using the following recurrence relation.

$$D^n_{ij} = \max_{i,j}\{D^{n-1}_{ij}, D^{n-1}_{ik} + D^{n-1}_{kj}\}$$

Because this transformation never rewrites elements which are used to calculate the new matrix, the same matrix can be used for both D^n and D^{n+1}.

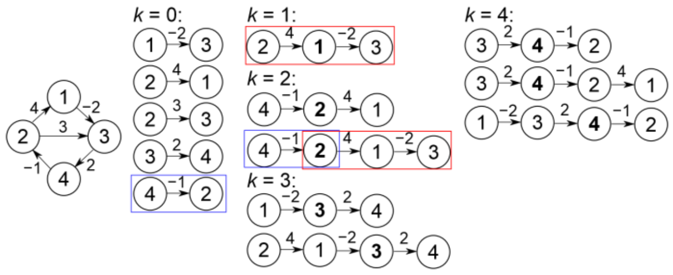

Fig. 5. Floyd-Warshall algorithm execution example illustration
Source: Wikimedia Commons

The basic Floyd-Warshall algorithm finds the total cost from all pairs of vertices; details of the path can be easily reconstructed with simple modifications to the algorithm. Besides storing the cost in the matrix D, an additional matrix N can be used to store which vertex is next on a path for a pair of vertices. The matrix is also continually updated along with the cost matrix.

Floyd-Warshall has a worst-case time complexity of $\Theta(|V|^3)$ and a worst-case space complexity of $\Theta(|V|^2)$.

III. Implementation

A. *Visibility Graph Generation*

The visibility graph of a map can be generated with any graph-traversal algorithm. Breadth-first search is used here to improve spatial locality.

Graph generation begins at the first junction in map, which is added to the graph as its first vertex. Every subsequent junction found is then queued for processing.

Each cardinal direction in the junction currently explored is then explored until either an inaccessible tile, which means that going to that direction from that junction leads to a dead end, or another junction. If another junction is found, the junction is added to the graph and queued for inspection, then a directed edge with information regarding direction and distance from the parent junction to the found junction is added to the graph.

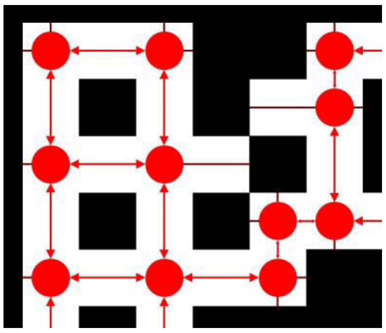
Fig. 6. Visibility graph generation illustration
Source: author

Pseudocode for visibility graph generation with BFS follows:

```
function generateGraph() → Graph
  Graph g
  Set visited
  Queue toVisit

  Point curr_point ← first junction in map
  g.addNode(curr_point)
  toVisit.enqueue(curr_point)

  while (toVisit is not empty)
    currPoint ← toVisit.dequeue()
    if not(visited.contains(currPoint))
      visited.add(currPoint)

      for (each cardinal direction)
        Point checked_point ← curr_point
        while (isAccessible(checked_point) and
               not(isJunction(checked_point)))
          checked_point ← move(checked_point, direction)

        if (isAccessible(checked_point))
          if (Node checked_point does not exist in g)
            g.addNode(curr_point)
            toVisit.enqueue(curr_point)
            g.addEdge(curr_point, checked_point,
                      direction, distance between points)
  → g
```

B. *Floyd-Warshall Algorithm*

The Floyd-Warshall is used here to further frontload cost calculation. Given that a visibility graph is usually static, the cost between all pairs of junctions can be precomputed to further drive down pathfinding computational cost at real-time.

The pseudocode for general Floyd-Warshall implementation is as follows. `FloydWarshall()` is used to find minimum cost between all pairs, while `ReconstructPath(i, j)` is used to find the shortest path between node i and node j.

```
int [|V|][|V|] dist
int [|V|][|V|] next

procedure FloydWarshall()
  for (i from 1 to |V|)
    for (j from 1 to |V|)
      if (edge(i,j) exists)
        dist[i][j] ← cost(i,j)
        next[i][j] ← j
      else
        dist[i][j] ← infinity
        next[i][j] ← null

  for (k from 1 to |V|)
    for (i from 1 to |V|)
      for (j from 1 to |V|)
        if (dist[i][j] > dist[i][k] + dist[k][j])
          dist[i][j] ← dist[i][k] + dist[k][j]
          next[i][j] ← next[i][k]

function ReconstructPath(int i, int j) → List<int>
  if (next[i][j] = null)
    return null
  List<int> path = {u}
  while (u ≠ v)
    u ← next[u][v]
    path.add(u)
  return path
```

C. *A**

A* is implemented here using a general node data structure.

```
class Node
  public List<Point> path
  public Point curr_point
  public int cost
  public int heuristic
  public int total

  public Node()
    curr_point ← origin
    path ← {curr_point}
    cost ← 0
    heuristic ← ManhattanDistance(origin, dest)
    total ← cost + heuristic

  public Node(NaiveNode parent, Point next_point,
              int add_cost)
    curr_point ← next_point
    path ← parent.path + {curr_point}
    cost ← parent.cost + add_cost
    heuristic ← ManhattanDistance(path.last, dest)
    total ← cost + heuristic
```

1) *A* Without Visibility Graph*

Without a visibility graph, A* finds a path using all tiles in the map as nodes. Each pair of adjacent tile is considered an edge with weight 1.

```
function aStar() → direction
  PriorityQueue<Node> alive_nodes
  curr_node ← new Node()
  alive_nodes.enqueue(curr_node)

  while (curr_node.curr_point ≠ dest)
    for (each cardinal direction)
      Point adj_point ← curr_point moved 1 tile
      if (isAccessible(adj_point)
        if not(curr_node.path.contains(adj_point))
          alive_nodes.enqueue(new Node(curr_node,
                                       adj_point, 1)
      curr_node ← alive_nodes.dequeue()

  → direction from origin to curr_node.path[1]
```

The above algorithm returns a direction to be used by a controller moving a unit.

While simple, checking every tile in a map is costly both in terms of computation and memory. For example, pathfinding in a 50 * 25 map means that there are up to (50 * 25) = 1250 vertices and (50 * 24) + (49 * 25) = 2425 edges to be explored.

2) A With Visibility Graph*

The visibility graph only contains junctions, while either or both origin and destination points can be located arbitrarily. Before pathfinding with visibility graph, both origin and destination points must be added to the visibility graph.

```
procedure addPointToGraph(input/output Graph g,
                          input Point p)
  if (g.hasNode(p))
    // do nothing
  else
    for (each cardinal direction)
      Point q ← p
      do
        q ← q moved 1 tile
      while (isAccessible(q) and not(g.containsKey(q)))
      if (isAccessible(q))
        g.addEdge(p, q, direction, distance(p, q))
        g.addEdge(q, p, reversed direction,
                  distance(p, q))
```

With a visibility graph, A* finds a path only using the junctions in the map and the origin and destination points.

```
function aStarWithVisibilityGraph(Graph g) → direction
  PriorityQueue<Node> alive_nodes
  curr_node ← new Node()
  alive_nodes.enqueue(curr_node)

  while (curr_node.curr_point ≠ dest)
    Point u ← curr_node.curr_point
    for (each neighbor v of u)
      if not(curr_node.path.contains(v))
        alive_nodes.enqueue(new Node(curr_node, v,
                                     distance(u, v))
    curr_node ← alive_nodes.dequeue()

  → direction from origin to curr_node.path[1]
```

Pathfinding with a visibility graph drastically reduces the number of vertices to check. As the number of junctions in a map is a fraction of the total number of tiles in a map, pathfinding with a visibility graph is noticeably faster than pathfinding without a visibility graph.

Optionally, after every few path calculations, non-junction previous origin and destination nodes should be removed to prevent bloat on the visibility graph.

3) A With Visibility Graph And Precomputed Paths*

Using the Floyd-Warshall algorithm, the number of vertices to check can be reduced to only at most six vertices: the origin (0), two vertices in the visibility graph closest to the origin (1, 2), two vertices in the visibility graph closest to the destination (3, 4), and the destination (5).

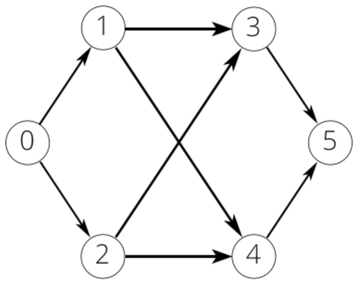

Fig. 7. Graph illustration for pathfinding with visibility graph and precomputed paths
Source: author

If the origin is located on a junction, vertices 0, 1, and 2 can be collapsed into one vertex representing the origin. Likewise, if the destination is located on a junction, vertices 3, 4, and 5 can be collapsed into one vertex representing the destination. Therefore, the pathfinding graph has at least two vertices and one edge and at most has six vertices and eight edges.

Because the above graph is simple, instead of using the visibility graph for pathfinding, a new graph composed of the aforementioned vertices can be generated every path calculation. The weight of edges between vertices 1, 2, 3, and 4 are obtained from the minimum cost calculated previously with Floyd-Warshall, while the weight of edges from 0 and to 5 are simply the horizontal or vertical distance to the nearest junctions.

```
Graph vg  // Visibility Graph

function minDist(Point u, Point v) → Point
  → minimum distance from u to v using Floyd-Warshall

function dir(Point u, Point v) → direction
  → direction from u to reconstructPath(u, v)[1]

function generateGraph() → Graph
  Graph g
  g.addNode(origin)
  g.addNode(dest)

  if not(vg.hasNode(origin))
    for (each neighbor u of origin)
      g.addNode(u)
  if not(vg.hasNode(dest))
    for (each neighbor v of dest)
```

```
      g.addNode(v)

  for (point u in each node in g)
    for (point v in each node in g)
      if (vg.hasNode(u) and vg.hasNode(v))
        g.addEdge(u, v, minDist(u, v), dir(u, v))
      else
        if (u = origin and v is neighbor of origin)
          g.addEdge(u, v, distance(u, v),
            direction from u to v
        else if (u is neighbor of dest and v = dest)
          g.addEdge(u, v, distance(u, v),
            direction from u to v
  → g
```

The path can then be calculated using the generated graph.

```
function aStarWithFloydWarshall(Graph g) → direction
  PriorityQueue<Node> alive_nodes
  curr_node ← new Node()
  alive_nodes.enqueue(curr_node)

  while (curr_node.curr_point ≠ dest)
    for (each neighbor v of curr_node)
      if not(curr_node.path.contains(v.curr_point))
        alive_nodes.enqueue(new Node(curr_node,
                    v.curr_point, g.edge(u, v).distance)
    curr_node ← alive_nodes.dequeue

  → direction from origin to curr_node.path[1]
```

Pathfinding with both visibility graph and precomputed paths is significantly faster than pathfinding with only visibility graph, as there are at most 6 nodes and 8 edges to go through.

IV. Experiment

A. *Setup*

The above algorithms are implemented in C# for testing purposes.

Testing is done on the following 50 * 25 map. Dots ('.') represent accessible tiles, while pounds ('#') represent inaccessible tiles. Initial origin point is marked by S in point (1, 48), while initial destination point is marked by E in point (48, 1).

```
50 25
##################################################
#..........###..........##..........###.........E#
#.###.####.###.####.###....###.####.###.####.###.#
#.......................##.......................#
#.###.##.###.###.##.###.##.###.##.###.###.##.###.#
#.....##.###.###.##.....##.....##.###.###.##.....#
#####.##.........##.##########.##.........##.#####
#####.##.###.###.##.##########.##.###.###.##.#####
#........#.....#........##........#.....#........#
#####.##.#######.##.##########.##.#######.##.#####
#................................................#
#.###.####.#.#.####.###.##.###.####.#.#.####.###.#
#...#...............#...##...#...............#...#
###.#.##.###.###.##.#.######.#.##.###.###.##.#.###
###.#.##.###.###.##.#.######.#.##.###.###.##.#.###
#.....##...#.#...##............##...#.#...##.....#
#.###.####.#.#.####.###.##.###.####.#.#.####.###.#
#.###.####.....####.###.##.###.####.....####.###.#
#..........###..........##..........###..........#
#.###.####.###.####.###.##.###.####.###.####.###.#
#.......................##.......................#
#.###.##.###.###.##.###.##.###.##.###.###.##.###.#
#.###.##.###.###.##.###....###.##.###.###.##.###.#
#S......................##.......................#
##################################################
```

Paths are recalculated every time the origin point moves 1 tile. To simulate moving destinations, the destination point is moved back and forth between points (48, 1) and (47, 1).

The pathfinding tests measure the time needed for the origin and the destination to have the same coordinates. Visibility graph generation and all-pair shortest-path computation also have their execution times measured.

The order in which algorithm testing is done on is as follows: visibility graph generation, all-pair shortest-path computation, pathfinding without visibility graph, pathfinding with visibility graph, and pathfinding with precomputed shortest path. Each of the five algorithms are tested 100 times to retrieve the average running time.

B. *Results*

The generated visibility graph is as follows. Blank spaces represent inaccessible tiles, dots represent accessible tiles, and pounds represent junctions associated with each vertex in the visibility graph.

```
#...#....#   #....#...#  #...#....#   #....#...#
.   .    .   .    .   #..#   .    .   .    .   .
#...#..#.#.#.#.#..#...#  #...#..#.#.#.#.#..#...#
.   .  .   .   .  .   .  .   .  .   .   .  .   .
#...#  .   .   .  #...#  #...#  .   .   .  #...#
    .  #...#...#  .          .  #...#...#  .
    .  .   .   .  .          .  .   .   .  .
....#..# ..#.. #..#....  ....#..# ..#.. #..#....
    .  .       .  .          .  .       .  .
#...#..#.#.#.#.#..#...#..#...#..#.#.#.#.#..#...#
.   .    . . .    .   .  .   .    . . .    .   .
#.# #..#.#.#.#.#..# #.#  #.# #..#.#.#.#.#..# #.#
  . .  .   .   .  . .      . .  .   .   .  . .
  . .  .   .   .  . .      . .  .   .   .  . .
#.#.#  #.# . #.#  #.#.#..#.#.#  #.# . #.#  #.#.#
.   .    . . .    .   .  .   .    . . .    .   .
.   .    #.#.#    .   .  .   .    #.#.#    .   .
#...#....#   #....#...#  #...#....#   #....#...#
.   .    .   .    .   .  .   .    .   .    .   .
#...#..#.#.#.#.#..#...#  #...#..#.#.#.#.#..#...#
.   .  .   .   .  .   .  .   .  .   .   .  .   .
.   .  .   .   .  .   #..#   .  .   .   .  .   .
#...#..#...#...#..#...#  #...#..#...#...#..#...#
```

The statistics of each test are described in the following table. All units are in milliseconds.

TABLE I. STATISTICS FOR EACH TEST

Test	Mean	Stdev	Range	Min	Max
Visibility Graph	16.08	1.606049171	7	14	21
Floyd-Warshall	88.32	2.352217952	15	86	101
A* w/o VG	3201.32	108.7941138	742	3113	3855
A* w/ VG	328.45	6.609413244	45	310	355
A* w/ precomputed	15.18	3.833346509	17	13	30

V. ANALYSIS

A. *Analysis*

As seen above, pathfinding without a visibility graph takes an average of 3 seconds to complete. Pathfinding with a visibility graph takes an average of 328ms to complete with an overhead of 16ms for visibility graph generation. Pathfinding with precomputed paths takes an average of 15ms, with an overhead of 104ms for both visibility graph generation and shortest-path precomputation.

Pathfinding with precomputed shortest paths are more than 200 times faster than pathfinding without a visibility graph and 20 times faster than pathfinding with only a visibility graph. The overhead of 104ms is even smaller than the average time taken for pathfinding with only a visibility graph.

Because there is no possibility of suboptimal paths produced if the algorithm is implemented correctly, there is no drawbacks on using pathfinding with precomputed paths beside negligibly longer loading times.

Note that the pathfinding results here are made from repeated calculations every step taken. If there is no need for repeated pathfinding (i.e. static destination), a single session pathfinding without a visibility graph and all-pairs shortest-path computation is faster.

B. *Possible Improvements*

The map used in the experiment here does not contain any rooms. This is because the visibility graph generation algorithm described above only works with 1-tile wide corridors and would treat each tile in a room as a junction, which is unnecessarily expensive, especially for large open spaces. A room detection algorithm can be added to the basic algorithm above for a more general performance.

For maps with large open spaces instead of narrow corridors, a more suitable pathfinding algorithm would be Rectangular Symmetry Reduction and Jump Point Search, which cuts down symmetrical paths found on basic A*.

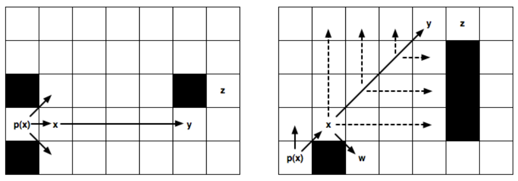

Fig. 8. Jump Point Search
Source: Rune de Groot

The pathfinding algorithms implemented here recalculate its path every time the origin or destination moves, which wastefully discards most of the path information produced by the algorithms. Furthermore, because most of the information required for pathfinding is precomputed, both the visibility graph and the precomputed shortest paths may be invalidated with minor changes to the map.

Variants of A*, such as Dynamic A* (D*) and Lifelong Planning A* (LPA*). D* is intended to for A* without complete information; if A* makes mistakes, D* can correct A* without taking too much time. LPA* is intended for changing costs; while A* is invalidated when the map changes, LPA* can use previous A* computations to generate a new path. However, both D* and LPA* requires large amounts of memory, making it unsuitable if there are a lot of moving units.

VI. CONCLUSION

To improve efficiency in uniform-cost grid-based pathfinding, the map can be preprocessed to produce both a visibility graph and precomputed shortest paths from all pairs of vertices. This frontloads the cost of repeated pathfinding, making the actual pathfinding much faster. In addition, both visibility graph generation and shortest path precomputation is relatively cheap in terms of computation.

Other algorithms to improve pathfinding exists, most notably by changing the pathfinding algorithm itself. Combined with the techniques discussed in this paper, the cost of the pathfinding process can be made even cheaper as to be viable for real-time pathfinding.

ACKNOWLEDGMENT

The author would like to express his deepest gratitude and appreciation to Drs. Nur Ulfa Maulidevi, Masayu Leylia Khodra, ST., MT., and Dr. Ir. Rinaldi Munir, MT. as the lectures of IF2211 Strategi Algoritma. Appreciation is also given to his parents, who have given him moral and material support for this paper, as well as classmates in IF2211 K2 who gave moral support and contributed ideas for this project.

REFERENCES

[1] R. Munir, *Diktat Kuliah Strategi Algoritma*, 2nd ed. Bandung: Institut Teknologi Bandung, 2009.

[2] T. Cormen, C. Leiserson, R. Rivest and C. Stein, *Introduction to Algorithms*, 1st ed. Cambridge, Massachusetts: The MIT Press, 2014, pp. 684-707.

[3] A. Levitin, *Introduction to the Design & Analysis of Algorithms*, 3rd ed. Boston, Massachusetts: Pearson, 2012, pp. 304-310, 333-337.

[4] A. Patel, "Amit's A* Pages", *theory.stanford.edu*, 2017. [Online]. Available: http://theory.stanford.edu/~amitp/GameProgramming/index.html. [Accessed: 14- May- 2017].

[5] "Floyd-Warshall algorithm", *Programming-algorithms.net*, 2017. [Online]. Available: http://www.programming-algorithms.net/article/45708/Floyd-Warshall-algorithm. [Accessed: 15- May- 2017].

[6] Z. Abd Algfoor, M. Sunar and H. Kolivand, "A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games", *International Journal of Computer Games Technology*, vol. 2015, pp. 1-11, 2015.

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 Mei 2017

[illegible]

Felix Limanta 13515065