6-1-2009

Comparison of Efficiency in Pathfinding Algorithms in Game Development

Nikhil Krishnaswamy
DePaul University

A Comparison of Efficiency in Pathfinding Algorithms in Game Development

A Thesis

Submitted to
The College of Computing and Digital Media
DePaul University

In fulfillment of the requirement of
The Honors Program Senior Thesis

by
Nikhil Krishnaswamy

June, 2009

Thesis Advisors:
Prof. Joseph Linhoff
Prof. Noriko Tomuro

A Comparison of Efficiency in Pathfinding Algorithms in Game Development

Nikhil Krishnaswamy
Honors Senior Thesis
DePaul University
nkrishna@depaul.edu

Abstract

This paper is the summary of a study done to assess the efficiency of three different pathfinding algorithms in a game-like environment. "Efficiency" is, in this case, defined as finding the shortest path possible in the least amount of time possible, and is tracked using a number of metrics, including the number of visitations made by a particular algorithm to any node in the graph tree, and the physical length of the traversable path in the game world. The three algorithms were tested using randomly generated sets of nodes in three different navigable environments in order to assess if any patterns appeared based on the operation of a particular pathfinding algorithm in a particular kind of environmental layout.

1. Introduction

Artificial Intelligence (AI) in game development refers to the various logical constructs programmers use to simulate the behaviors of non-player-driven characters in a game environment. As with everything in a game, AI subroutines are expected to be optimized for speed and efficiency to produce the best and most realistic results possible.

One of the most basic elements of game AI is pathfinding, which enables an agent to calculate a path from a starting point to a target around any number of obstacles. This project analyzes a few of the most common pathfinding algorithms to determine if one is the best or if different algorithms are better suited to different environmental layouts or situations.

This analysis gathers data on and compares the efficiency and speed of different pathfinding algorithms. The three algorithms examined are A*, Dijkstra's, and D*.

2. System Descriptions

A C++/OpenGL-based application is used to run these test. The application renders everything in three dimensions, although the simulations are concerned only with two dimensions: the x-axis and the z-axis.

A scripting system was used to ensure a consistent setup for the tests. The script allows placement of spawn points, targets, obstacles, and other elements. For these experiments, three script files were used: one for each environment. The first environment is a grid pattern, the second environment is a symmetrical object layout, and the third environment is a random, non-symmetrical layout of objects. These layouts cover frequently occurring environments in.

Environment 1 contains 15 obstacles of uniform size, Environment 2 contains 16, with a slight variance in size, and Environment 3 contains 20, with a higher variance in size. The different number and placement of obstacles in each environment give each one a varying level of restrictiveness. Restrictiveness here is defined as being inversely proportional to the amount of the open space on the environment's "floor" unblocked by obstacles – i.e. the less space obstacles take up in an environment, the less restrictive it is considered to be. This allows the data gathered to reflect if a particular algorithm is better suited to a more open environment or a more restrictive one.

2.1. Screenshots of environments

In the screenshots below, the agent (1) is represented, for purposes of differentiation, by a model of a penguin in a Roman centurion's helmet, and the target (2) is represented by the white box with the bull's-eye texture. The entire area represented is about 200 units wide, along the x-axis, and 130 units deep, along the z-axis. The agent is approximately 5 units wide by 5 units deep. The x-axis runs from left the right, right being the positive direction, and the z-axis runs from top to bottom, bottom being the positive direction. The positive y-axis comes out of the screen, toward the viewer.

The units of measurement are based on the units used in the modeling program, in this case, MAXON Cinema 4D (C4D). C4D uses meters as its base unit. When a meter is rendered, from the camera angles shown below, on a screen with a resolution of 1440 pixels by 900 pixels, it is equivalent to approximately 7 pixels. However, the application scales the window size to the fill the screen of the machine it is running on, so these equivalency values can vary from computer to computer.

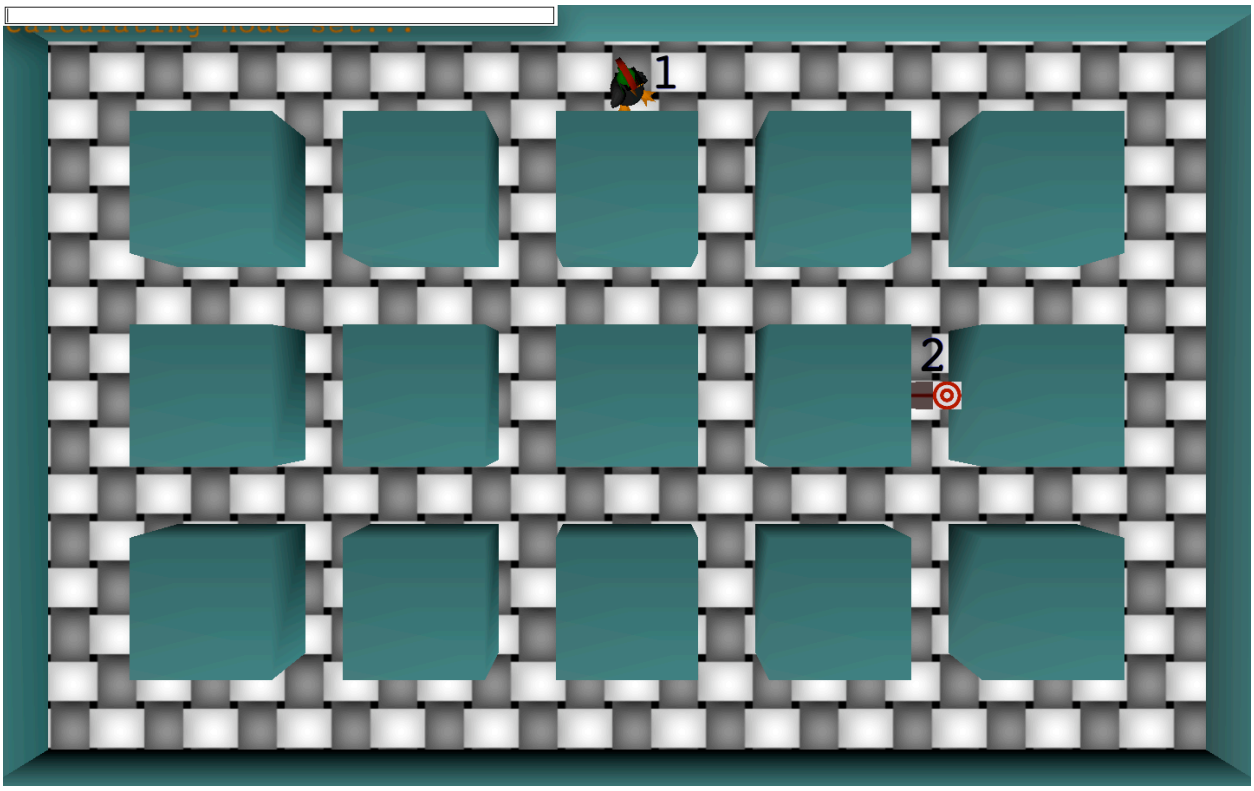


Figure 1. Environment 1: Obstacles in a grid pattern

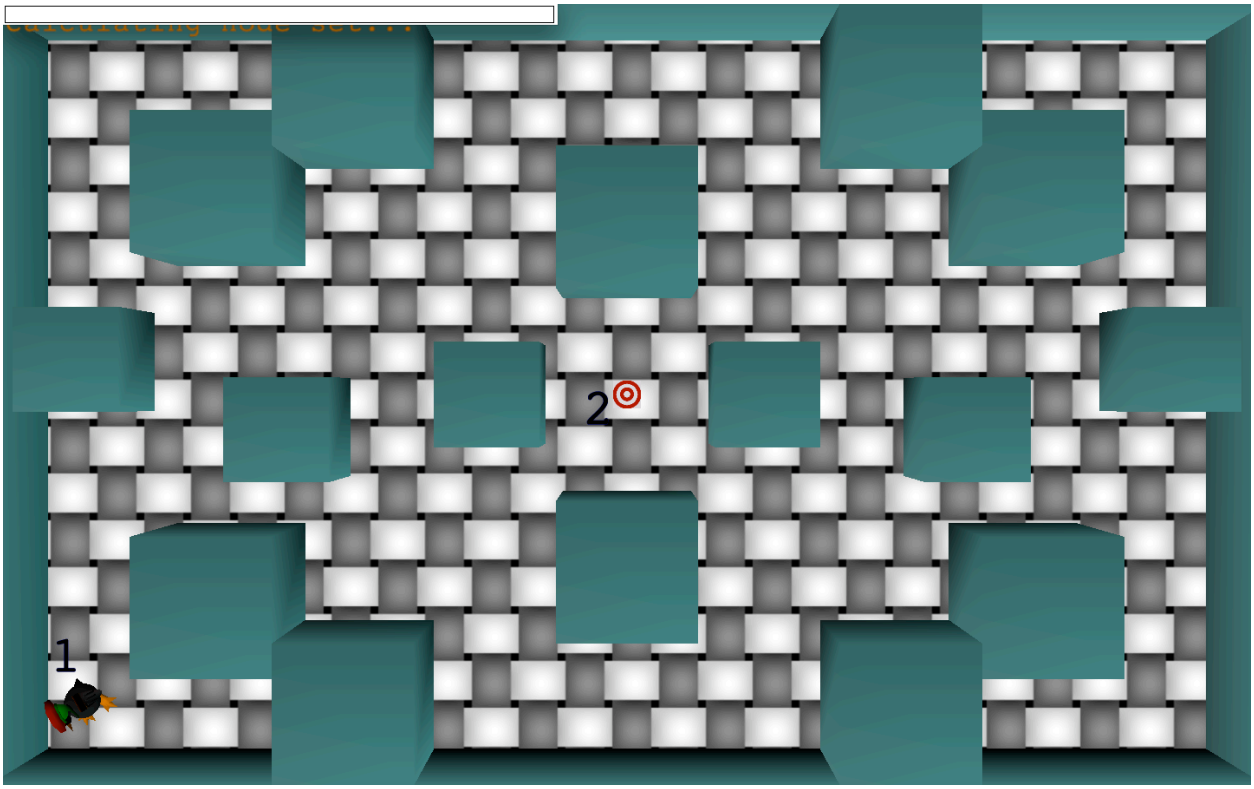


Figure 2. Environment 2: Obstacles arranged symmetrically, not in a grid

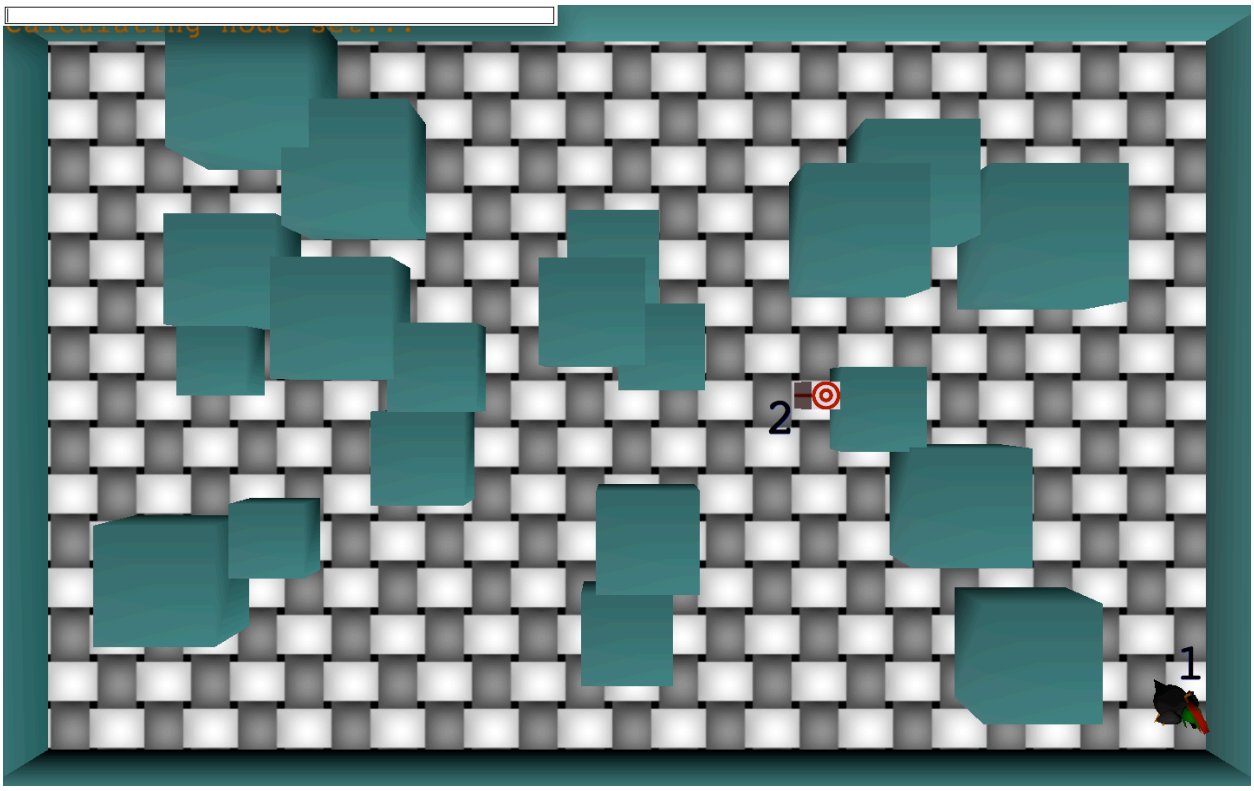


Figure 3. Environment 3: Obstacles placed randomly

The application then generates a series of valid path nodes within the environment. These tests use 32 nodes. Valid path nodes are randomly generated points on the map, that do not lie within a given radius of any of the obstacles. For these tests, this radius is two. Two units allows an object the size of the agent to maneuver along the paths with minimal obstacle avoidance. Once the list of nodes is complete, the application finds a list of valid links between nodes. A valid link is a straight line between two nodes that does not intersect any other objects. With the list of valid links complete, the simulation can now run.

Below is a sample graph tree complete with nodes and links, shown before the algorithm begins its search. The agent and target are indicated as referenced above. The small boxes (1) show the location of path nodes and the lines connecting them are valid links.

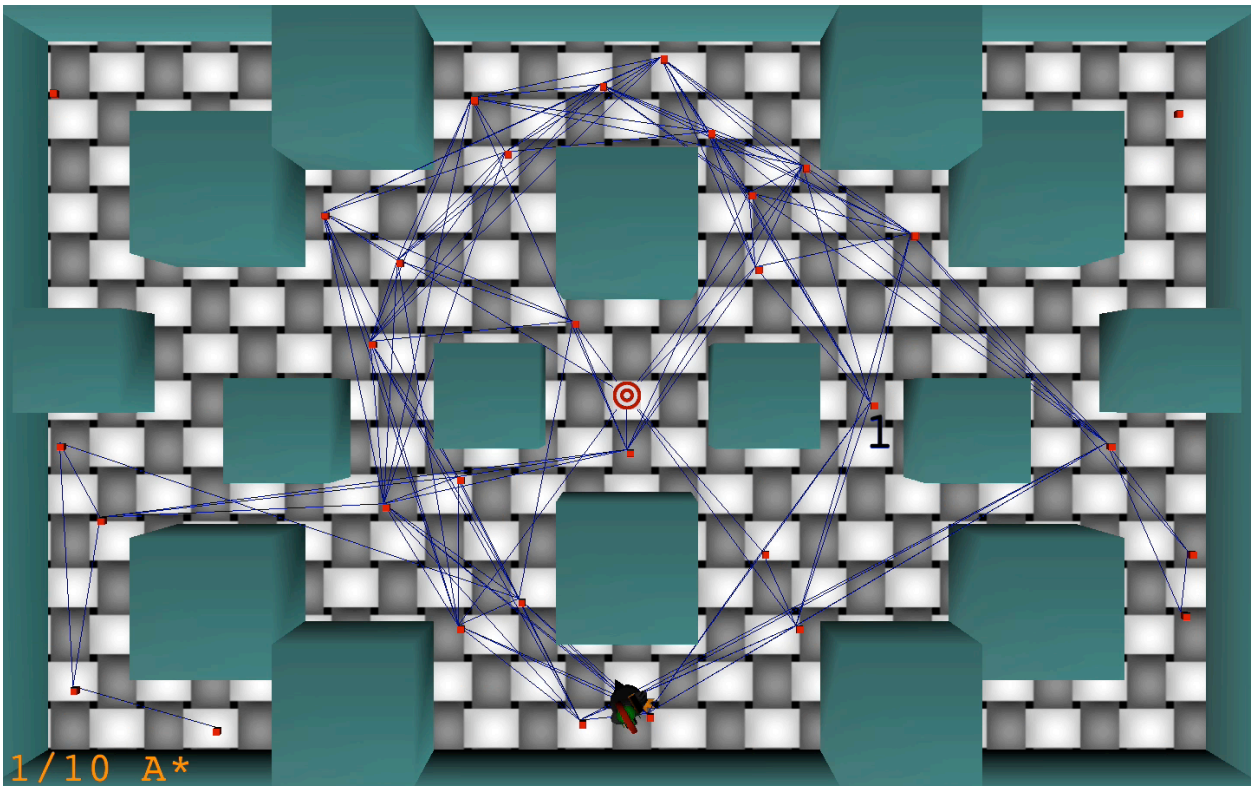


Figure 4. A sample graph tree

3. Overview of Algorithms

3.1. A*

The A* search algorithm is generally regarded as the *de facto* standard in game pathfinding. It was first described in 1968 by Peter Hart, Nils Nilsson, and Bertram Raphael.

For every node in the graph, A* maintains three values: *f(x)*, *g(x)*, and *h(x)*. *g(x)* is the distance, or cost, from the initial node to the node currently being examined. *h(x)* is an estimate or heuristic distance from the node being examined to the target.

There are a variety of methods available for calculating the distance, but for these tests, the method used is known as the Manhattan distance, and is simply the distance between the two points along the x-axis plus the distance between the two points along the z-axis. This is calculated by subtracting the x value of the second point from the x value of the first point, taking the absolute value of the result, then subtracting the z value of the second point from the z value of the first point, taking the absolute value of this result, and then adding the two result together. The reason for using this formula is based in the assumption that, from most nodes, the distance to the target likely passes through at least one more node, which makes the path deviate from a straight line, and therefore a horizontal displacement-plus-vertical displacement formula is likely to result in value closer to the actual distance that will be traveled than an estimate using plain Euclidean distance.

The value of *g(x)* is the distance from the initial node to the current node through all

previous nodes traversed to get to that point. Therefore if A* is examining node *C* as a possible next step in the path after traversing node *B*, then the *g(x)* value of node *C* is equal to the distance from the origin *A* to node *B*, plus the distance from node *B* to node *C*. This makes the *g(x)* value for a given node equal to the actual distance required to travel from the origin to that node, through all preceding nodes. *h(x)* is an estimate of the distance from the current node to the node located at the target. *f(x)* is the sum of *g(x)* and *h(x)*.

A* also maintains an "open list," which is a list of all unvisited nodes, and a "closed list," or list of visited nodes. At the beginning of the search, all nodes are on the open list, and the initial node is marked as current. The values of *g(x)*, *h(x)*, and *f(x)* are calculated for each of its neighbors. If the new *f(x)* value of a node being examined is less than the previous *f(x)* value for that node, the new *f* value replaces the old *f* value. The current node is moved from the open list to the closed list, the neighbor node with the lowest *f(x)* value is marked as the new current node and the process repeats until the target is added to the closed list, or there are no more nodes on the open list. [1]

In pseudocode:

```
procedure A*(graph, start, target)

        /* Path finding phase */
num_visits = 0
for each vertex v in graph
        g[v] = infinity
        h[v] = abs(target_x-v_x)+abs(target_z-v_z)
        f[v] = g[v]+h[v]
        previous[v] = undefined
end for
g[start] = 0
h[start] = abs(target_x-start_x)+abs(target_z-start_z)
f[start] = g[start]+h[start]
closedlist = empty set
openlist = the set of all nodes in graph

while openlist is not empty
        x = vertex in openlist with smallest f[]
        if f[x] == infinity
                break
        remove x from openlist
        add x to closedlist
        num_visits++
end while

if x is not target
        for each neighbor v of x
```

```
                    if v is not in closedlist
                        alt = dist_between(x,v)+g[v]+h[v]
                        if alt < f[v]
                            f[v] =  alt
                            previous[v] = x
                        end if
                    end if
                end for
            end if

            /* Path reconstruction phase */
            s = empty sequence
            x = target
            while previous[x] is defined
                insert x at the beginning of s
                x = previous[x]
            end while
```

3.2. Dijkstra's Algorithm

Named for its creator, Edsger Dijkstra, Dijkstra's algorithm was first proposed in 1959 and is the immediate precursor of A*.

The basic process is to assign each node a distance value, at first set to zero for the initial node and infinity for all other nodes. All nodes are marked as unvisited and the initial, node is marked as the current node.

All nodes that are neighbors to the current node are examined and their distance *D* from the initial node is calculated through the current node is calculated (*D* = distance from current node to neighbor node + distance from initial node to current node). This value is equivalent to A*'s *g(x)* value and is calculated the same way. If this new distance *D* is less than the previously recorded distance *D* for that node, the new distance value replaces the old distance value for that node.

When all neighbors of the current node are examined, the current node is marked as visited and will not be looked at again. The neighbor node with the lowest distance value is marked as the new current node and the process repeats until the target is marked as visited or all nodes are marked as visited without the target being found. [2]

Dijkstra's algorithm can also be considered to be a special case of the A* algorithm, where the *h(x)* value is always equal to zero. Dijkstra's algorithm, however, does not make use of the open and closed lists that A* does.

In pseudocode:

```
procedure Dijkstra(graph, start, target)

            /* Path finding phase */
```

```
num_visits = 0
for each vertex v in graph
        dist[v] = infinity
        previous[v] = undefined
        visited[v] = false
end for
dist[start] = 0
Q = the set of all nodes in graph

while Q is not empty
        x = vertex in Q with smallest dist[]
        if dist[x] == infinity
                break
        remove x from Q
        visited[x] = true
        num_visits++
end while

if x is not target
        for each neighbor v of x
                if visited[v] is not true
                        alt = dist_between(x,v)
                        if alt < dist[v]
                                dist[v] =  alt
                                previous[v] = u
                        end if
                end if
        end for
end if

        /* Path reconstruction phase */
s = empty sequence
x = target
while previous[x] is defined
        insert x at the beginning of s
        x = previous[x]
end while
```

3.3. D*

D* is a dynamic variant of the A* search algorithm optimized for partially obscured environments. It was first described by Anthony Stentz in 1994.

D* maintains the same three values as A*: the cost-to-current-node value, the heuristic-

to-end estimate, and the cost-plus-heuristic value (Stentz refers to these as *c*, *h*, and *k*, respectively). The value for *c* is calculated the same way as the value for *g(x)* in the A* algorithm, and *h* is calculated the same way as A*'s *h(x)* value. *k* is functionally equivalent to A*'s *f(x*). D* also tags nodes with a place on an open and closed list, like A*, but uses an additional tag of "new" for a node that has not yet been added to the open list. Unlike A* and Dijkstra's Algorithm, however, the algorithm starts at the node representing the goal, and works toward the node at which the agent starts. The goal node is first tagged as "open," and values for *c*, *h*, and *k* are calculated for each neighbor node, and the node with the lowest *k* value is chosen as the next node. The process is repeated until the starting node is tagged as "closed" or no next node can be found.

D* then uses the path found as a starting path and runs it until an error is found (i.e. an obstacle has moved into the agent's path or the next node turns out to be a dead end). In the event, the last node the agent traversed is set as a starting node and D* recalculated the *h*, *c*, and *k* values for all nodes examined between the new start and the goal. These values are compared to the previously computed values. If the *k* value for a node is lower than its old *k* value, then the node is placed in a "lower" state. If it is higher, the node is placed in a "raise" state. Nodes in the "lower" state are given higher priority in subsequent calculation loops. The entire process repeats until the starting node is tagged as "closed" or no next node can be found. [3]

In pseudocode:

```
procedure D*(graph, start, target)

        /* Path finding phase */
num_visits = 0
for each vertex v in graph
        c[v] = infinity
        h[v] = abs(start_x-v_x)+abs(start_z-v_z)
        k[v] = c[v]+h[v]
        tag[v] = open
        next[v] = undefined
end for
c[target] = 0
h[target] = abs(start_x- target_x)+abs(start_z- target_z)
k[target] = c[start]+h[start]
Q = the set of all nodes in graph

while Q is not empty
        x = vertex in Q with smallest k[]
        if k[x] == infinity
                break
        remove x from Q
        tag[x] = closed
end while
```

```
if x is not target
        for each neighbor v of x
                if tag[v] is not closed
                        alt = dist_between(x,v)+c[v]+h[v]
                        if alt < k[v]
                                k[v] =  alt
                                next[v] = u
                        end if
                end if
        end for
end if

        /* Path reconstruction phase */
s = empty sequence
x = start
while next[u] is defined
        insert x at the end of s
        x = next[x]
end while
```

4. Examples of simulations

In these stills from the simulator, small red boxes (1) signify nodes, and blue lines show the links between the nodes, signifying the entire graph tree. Yellow boxes (2), slightly larger, signify nodes that have been examined by the search algorithm, and green boxes (3), slightly larger than the yellow boxes, signify nodes that were determined to be part of the best path chosen by the algorithm. Black boxes (not shown) are nodes that were determined to be dead ends. The read out in the bottom left corner shows how many simulations (of all specified algorithms) out of the total have been run, followed by the name of the algorithm being used in the current test. The agent is the penguin model and the target is the bull's-eye texture. In each case, the path calculated by the algorithm has been shown using a thicker line.

4.1. Environment 1

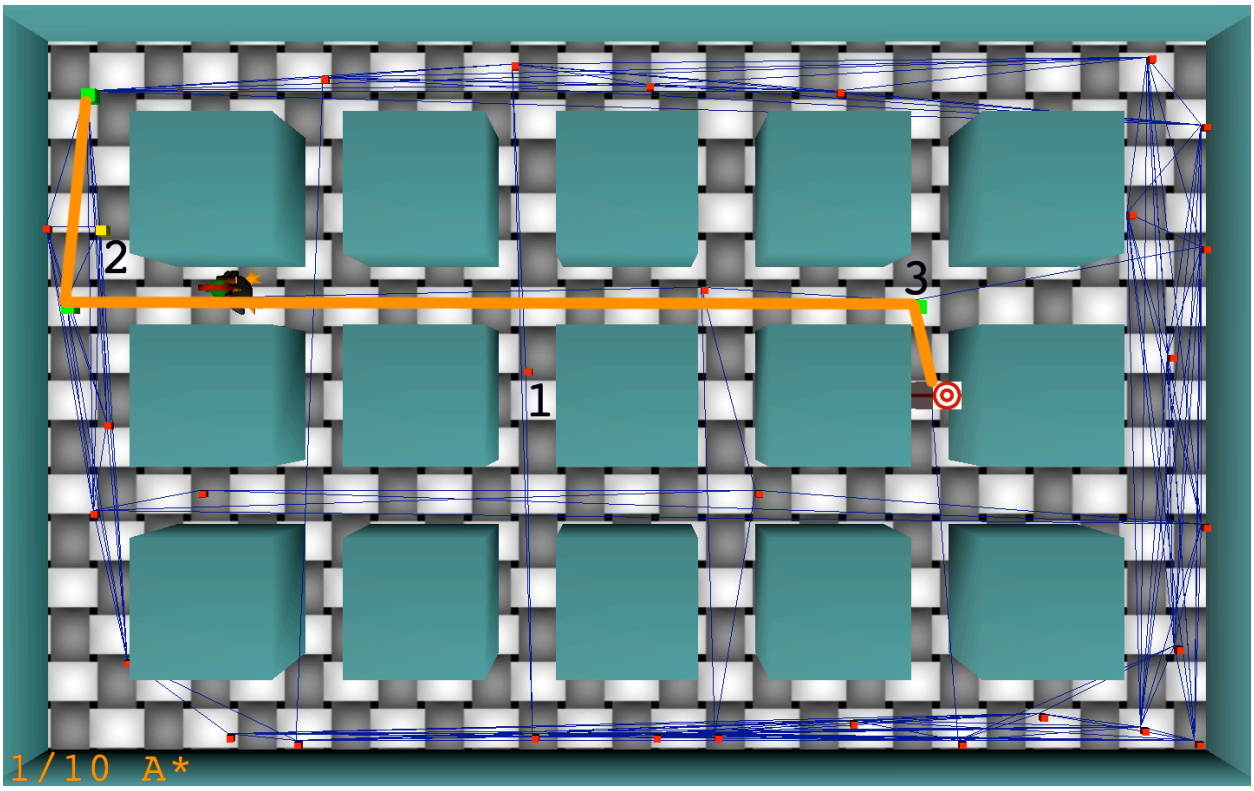


Figure 5-1. Simulation in Environment 1, A*

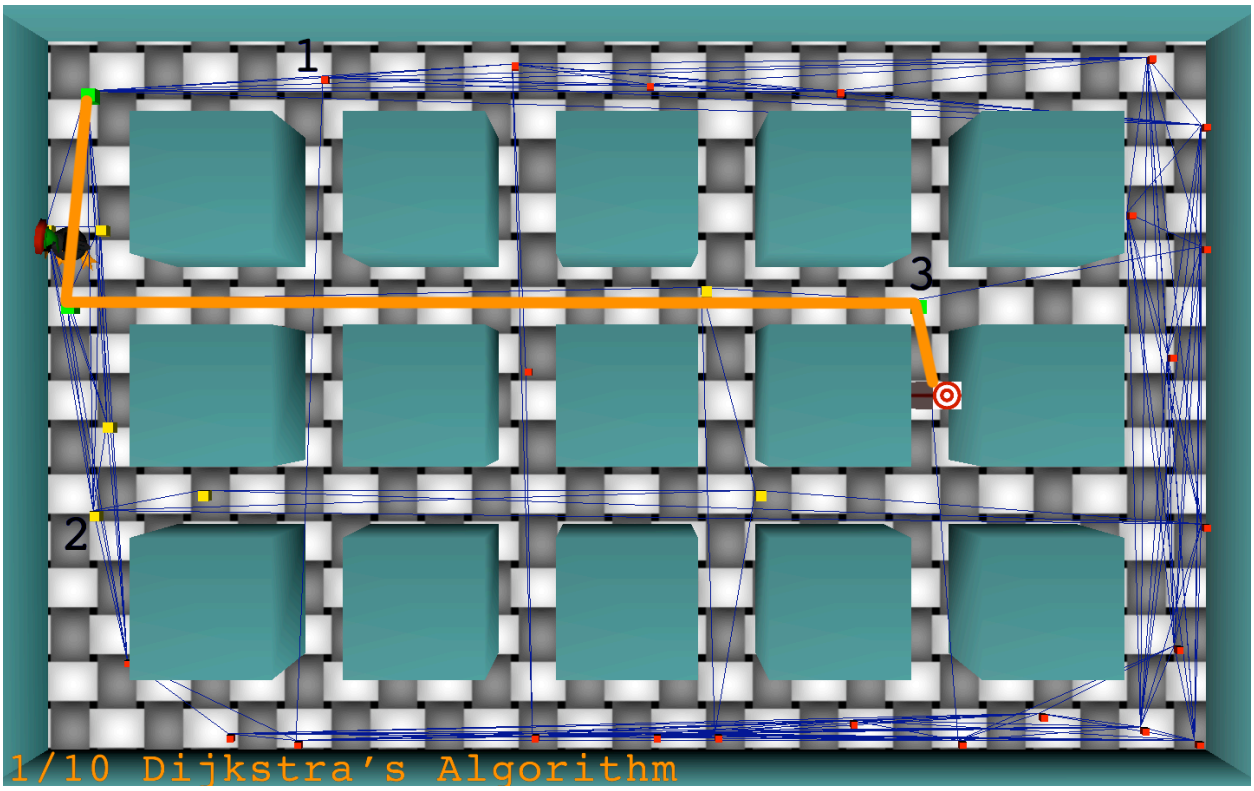


Figure 5-2. Simulation in Environment 1, Dijkstra's algorithm

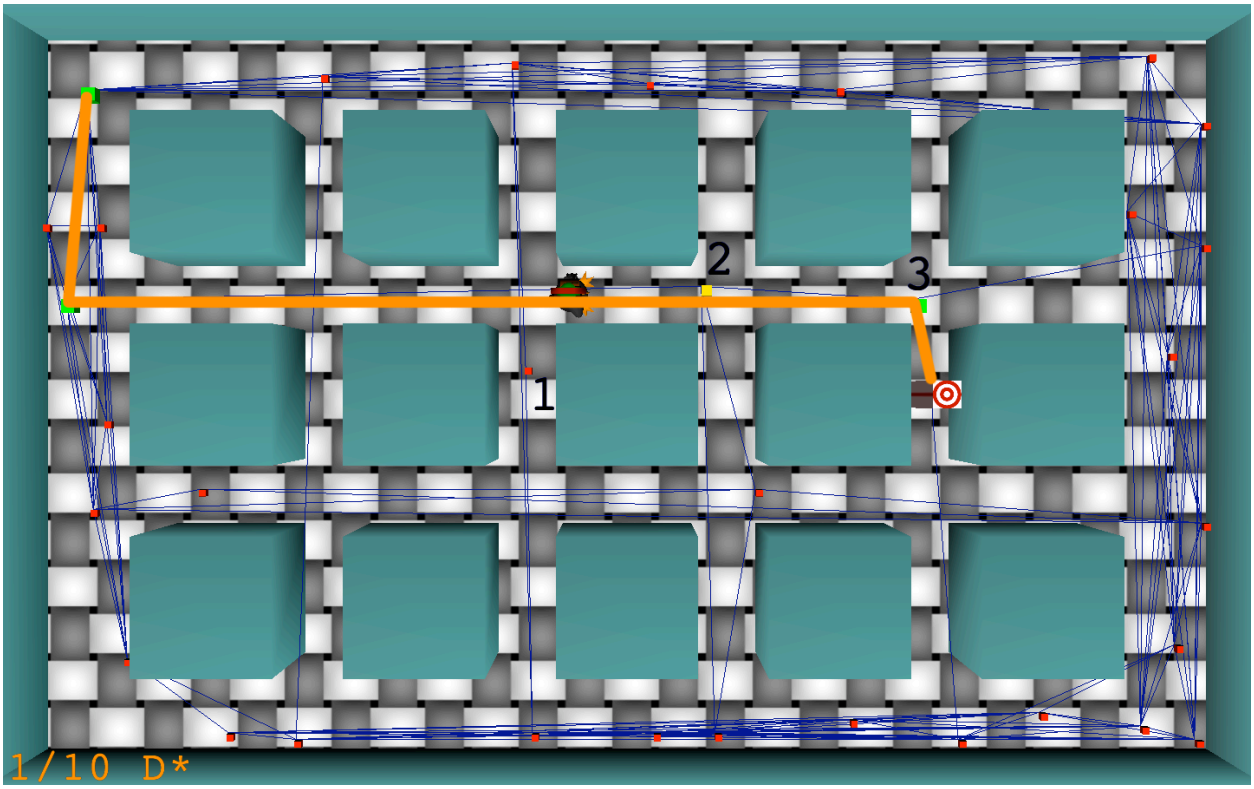


Figure 5-3. Simulation in Environment 1, D*

In these examples from Environment 1, all algorithms found the same path. However, in the screenshot from the run using Dijkstra's algorithm, 10 nodes are marked as examined, including the nodes that are part of the path, as opposed to 5 each in the runs using A* and D*. Dijkstra's algorithm examined more nodes than either of the other algorithms.

4.2. Environment 2

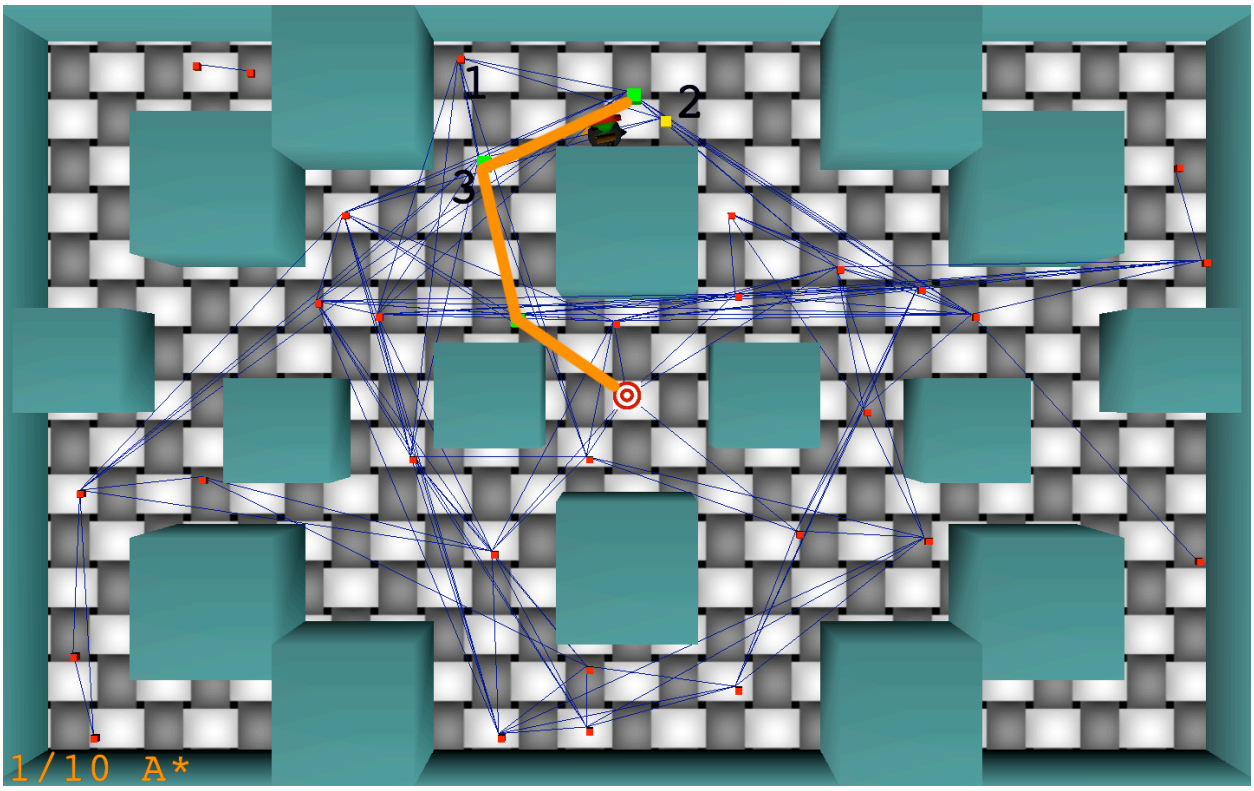


Figure 6-1. Simulation in Environment 2, A*

Figure 6-2. Simulation in Environment 2, Dijkstra's algorithm

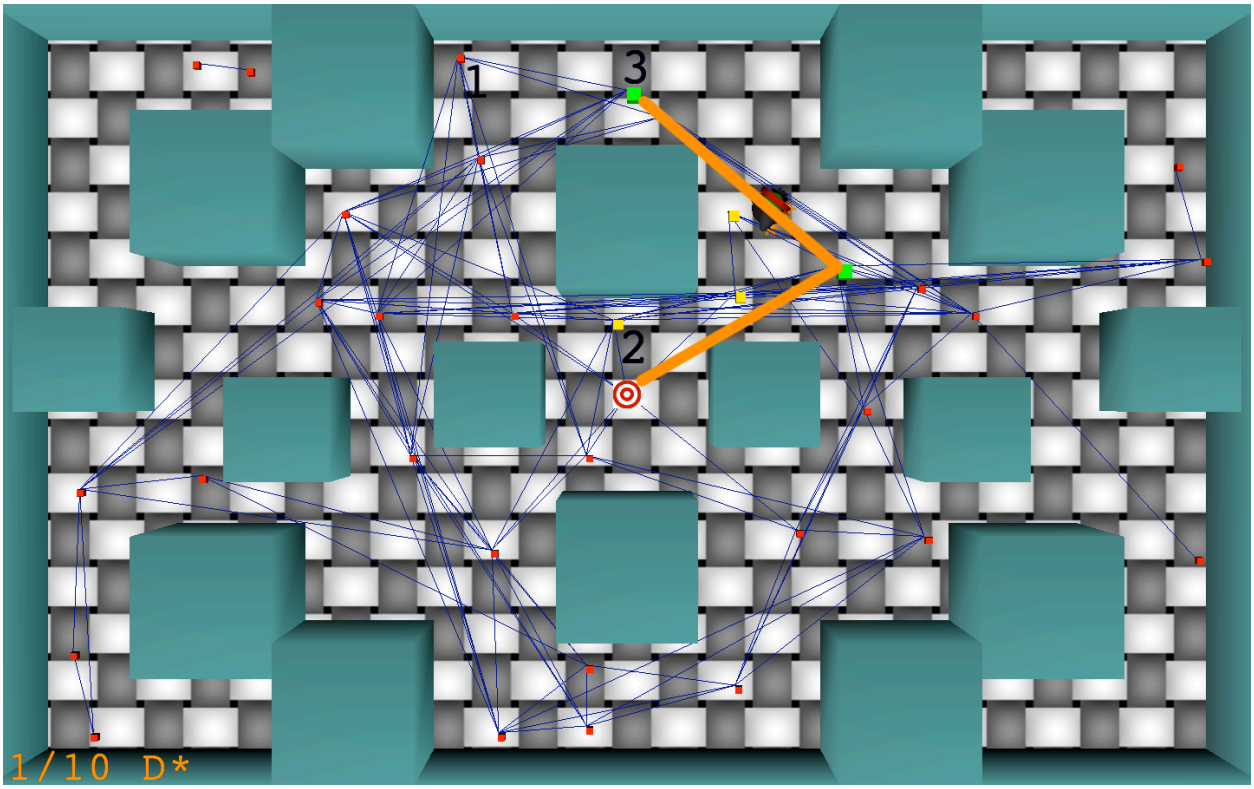
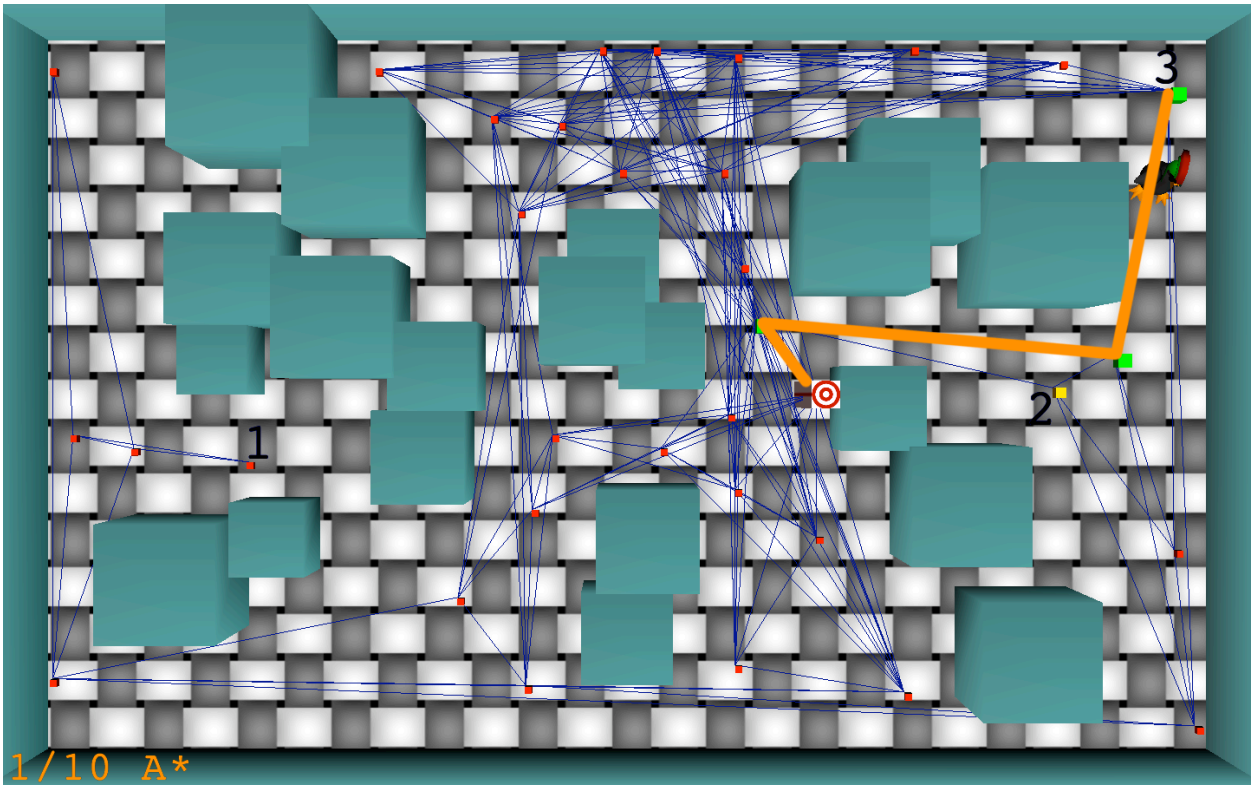


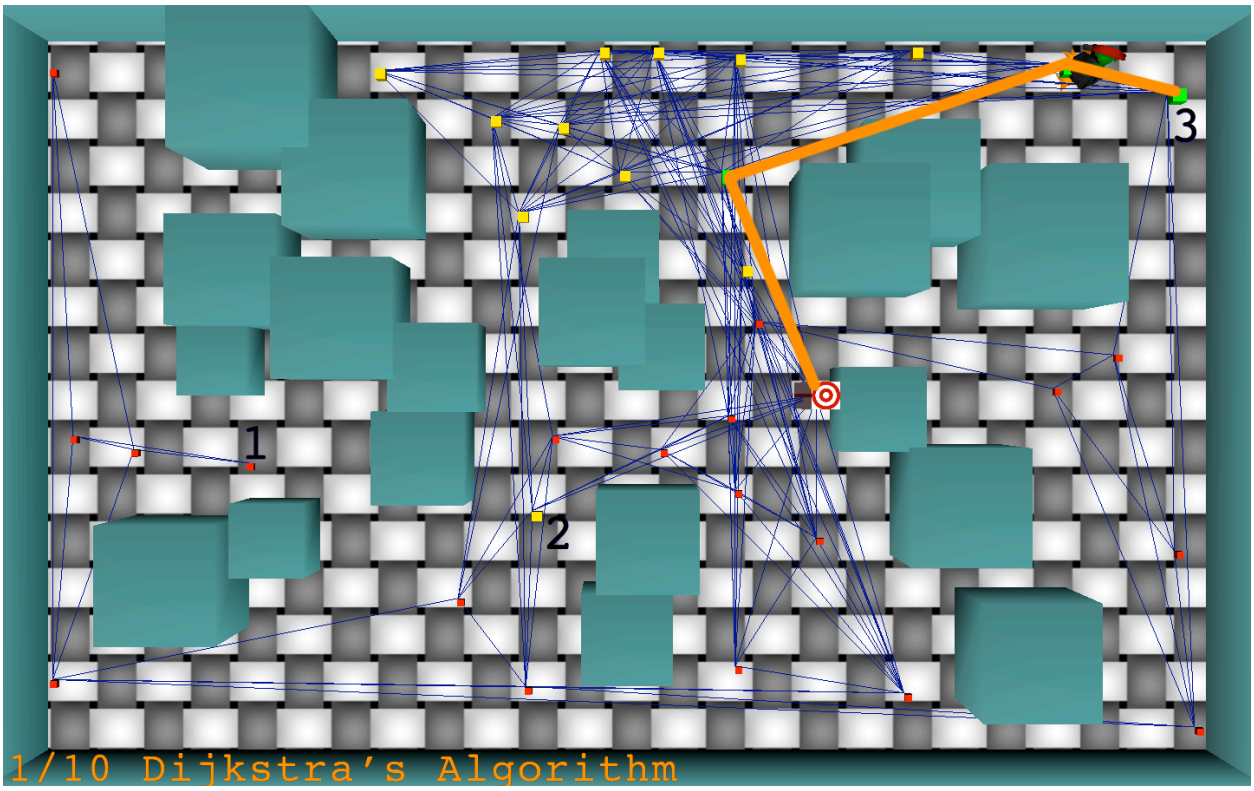
Figure 6-3. Simulation in Environment 2, D*

In this sample from Environment 2, A* and Dijkstra's algorithm found the same path, but Dijkstra's algorithm examined 16 nodes to find it, while A* examined 5. D* found a path that traversed fewer nodes than the other path. This path is 53.081 units long, versus the other path, which is 56.835 units long. D* calculated this path by working backwards from the target to the start, rather than the other way around.

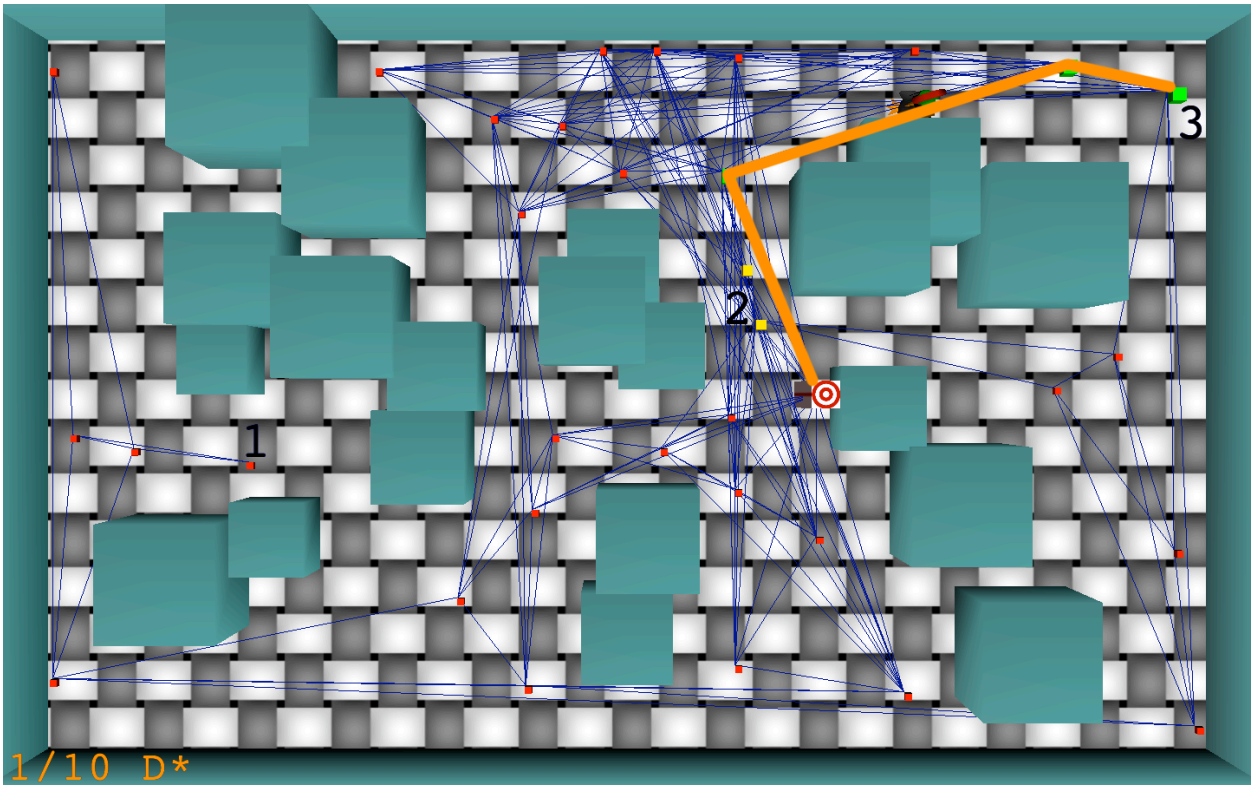
4.3. Environment 3



Figures 7-1. Simulation in Environment 3, A*



Figures 7-2. Simulation in Environment 3, Dijkstra's algorithm



Figures 7-3. Simulation in Environment 3, D*

In a sample test of Environment 3, A* found a path that was different from the path found by Dijkstra's algorithm and D*. Dijkstra's algorithm examined more nodes than D* when finding that common path – 15 as opposed to 6. A*, taking into account the heuristic estimate of distance to the target, chose a different path, proceeding downward from the starting node instead of leftward like the other path, and, in the process, examined only 5 nodes.

5. Data

The analysis uses data gathered from 50 tests run in each of the three environments using each of the three algorithms – a total of 450 tests. The data collected from these tests includes:

1. The number of times any node was visited and examined during the course of the path calculation. This includes repeat visits to nodes, as a single node may have been examined multiple times.
2. Whether a path was successfully found or not.
3. Path traversal time, in seconds.
4. Path traversal time, in loops.
5. The number of nodes traversed in the path, including the target node.
6. The length of the path, in game world units.

The tables below show selected averages for all the tests. The tables contain the following data:

1. The average of the total number of node visitations over all tests for the given algorithm in the given environment. This parameter is designated V.
2. The average number of nodes in a calculated path over all tests for the given algorithm and environment. This parameter is designated N.
3. The average length, in game world units, of a calculated path over all tests for the given algorithm and environment. This parameter is designated L.
4. The average time, in seconds, spent executing a path from start to finish over all tests for the given algorithm and environments. This parameter is designated E_s.
5. The average number of main application loop completed during path execution over all tests for the given algorithm and environment. This parameter is designated E_l.
6. The total number of times a path was successfully found by all three algorithms
7. The total number of times all algorithms found the same path over all tests in the given environment.

5.1. Data from Environment 1

	Avg. # node visitations (V)	Avg. # nodes in a path (N)	Avg. length of a path (units) (L)	Avg. path execution time (seconds) (E_s)	Avg. path execution time (app. loops) (E_l)
A* Search Algorithm	**23.915**	**2.900**	**137.650**	**3.564**	**108.970**
Dijkstra's Algorithm	**145.665**	**2.855**	**134.014**	**3.507**	**102.330**
D* Search Algorithm	**3.640**	**2.770**	**132.412**	**3.472**	**106.050**

Total number of successful path calculations: **48/50 (96.00%)**
Total number of times all algorithms found the same path: **39/48 (81.25%)**

The lengths of all paths, as well as the execution times, are similar. However, the average number of node visitation made by the D* algorithm is far less than the number of visitations made by either of the other algorithms, indicating the shorter search time for D* in this environment. In second place is A*, which made, on average, a higher number of node visitations than D*, but a lower number that Dijkstra's algorithm. All three algorithms calculated the same path over 81% of the time – the highest rate for any environment.

5.2. Data from Environment 2

	Avg. # node visitations (V)	Avg. # nodes in a path (N)	Avg. length of a path (units) (L)	Avg. path execution time (seconds) (E_s)	Avg. path execution time (app. loops) (E_l)
A* Search Algorithm	**5.062**	**3.353**	**108.004**	**3.046**	**94.682**
Dijkstra's Algorithm	**24.494**	**3.271**	**105.206**	**2.930**	**86.833**
D* Search Algorithm	**12.773**	**3.271**	**106.417**	**3.006**	**93.282**

Total number of successful path calculations: **44/50 (88.00%)**
Total number of times all algorithms found the same path: **28/44 (63.63%)**

In Environment 2, the average length of a calculated path, as well as the execution times, are again very close - within 3 units in length, 1 second, or 8 loops of each other. However, this time, A* makes the fewest node visitations, with D* in second place, making, on average, about half the number of node visitations as Dijkstra's algorithm. The algorithms had the lowest rate of successful path calculation in this environment. The algorithms also produced the most variations in paths in this environment. All three algorithms calculated the same path 63% of the time.

5.3. Data from Environment 3

	Avg. # node visitations (V)	Avg. # nodes in a path (N)	Avg. length of a path (units) (L)	Avg. path execution time (seconds) (E_s)	Avg. path execution time (app. loops) (E_l)
A* Search Algorithm	**4.960**	**2.380**	**113.254**	**2.997**	**91.040**
Dijkstra's Algorithm	**23.360**	**2.380**	**111.552**	**2.956**	**85.960**
D* Search Algorithm	**7.180**	**2.300**	**109.805**	**2.899**	**87.980**

Total number of successful path calculations: **50/50 (100.00%)**
Total number of times all algorithms found the same path: **37/50 (74.00%)**

Once again, the paths found by all algorithms were very close to each other in average length and execution time. In this environment, as in Environment 2, A* made the fewest node visitations, on average. D* is a close second, and its average number of node visitations are closer to the average number of node visitations made by A* than to the number made by Dijkstra's algorithm. This environment also had the highest raw success rate. All algorithms found a path in all tests. All algorithms found the same path 74% of the time, which is a higher rate of consistency than in Environment 2, but a lower rate than in Environment 1.

Refer to the attached Appendix for the full data tables.

6. Analysis

An efficient algorithm is one that calculates the shortest path with the fewest number of node visitations. A basic "efficiency score", *S*, can be calculated for each algorithm in each environment by multiplying the average number of node visitations by the average length of a path. Multiply the reciprocal of this number by a constant *k* to increase understandability (in this case, *k*=10000). Therefore, $\boldsymbol{S = k/(V*L)}$, where *S* is the efficiency score, *V* is the average number of node visitation, and *L* is the average length of a path in units. A larger value indicates a more efficient algorithm. These values are specific to the environment and can only be used to compare the efficiency of different algorithms in the same environment.

6.1. Efficiency scores

	Environment 1	Environment 2	Environment 3
A* Search Algorithm	**3.037**	**18.291**	**17.802**
Dijkstra's Algorithm	**0.512**	**3.881**	**3.838**
D* Search Algorithm	**20.748**	**7.357**	**12.684**

In all three environments, Dijkstra's algorithm was the least efficient. This is because Dijkstra's algorithm has no method of cutting down on the search space. As seen in the data tables above, the average length of a path found by all algorithms was similar (within about five units of each other), but in all environments, Dijkstra's algorithm made far more node visitations during the path calculation phase than either A* or D*. This behavior contributed to its lower efficiency scores.

In Environment 1, D* was by far the most efficient algorithm, with an average of fewer than 4 node visitations per path calculation. D* also had the lowest average number of nodes in a path, and shortest average path length. However, in the other two environments, A* proved to be more efficient than D*. Likely, this can be attributed to the restrictiveness of the respective environments. Environment 1 had the least unobstructed space, and therefore there only ended up being a few (usually one or two) links between the target node and any other node. Since D* begins calculating the path at the target node, the search space expanded less rapidly. This resulted in fewer node visitations. Additionally, since node visitations are counted even when no path could be found, D* could determine that no path existed more quickly than A*.

These results were reversed in the other environments, which were more open, and had greater potential for there to be a larger number of nodes connected directly to the target compared to those connected to the starting point. A* could either determine more quickly that no path existed. D* took more time and had to check a greater number of nodes, expanding the search space more quickly.

Dijkstra's algorithm produced the shortest average path length in Environment 2. However, it was the slowest.

7. Conclusions

Depending on the environment, the most efficient of these three algorithms is either A* or D*. The difference between the two appears to be most closely correlated to unobstructed space in the environment. In Environment 1, where the target was in a more restricted area than the starting node, D* usually had to visit fewer nodes, and often found a shorter path than A*. In the other two environments, the opposite was true. The differences between the two algorithms' results in these environments were not as extreme as in Environment 1.

8. Future work

Artificial intelligence is an evolving field. This study is limited to three algorithms. It does not test all pathfinding algorithms available or all situations that one could encounter. It also raises further questions, which could be answered using similar methods. These may include:

• The path nodes in these tests were placed randomly. In a game, it would be possible to have a level designer place path nodes manually and encode them in the level layout. How much of an effect would this have on the navigability of the environment? Might it be possible to write an algorithm that generates an optimal placement of path nodes, allowing maximum navigability?

• Points of intersection of two or more links between path nodes can be valid path nodes themselves. If these points were added to the list of path nodes, would this significantly decrease the time needed to find the path, or would the increased number of path nodes actually widen the search space and decrease the efficiency of any or all algorithms?

• A larger number of path nodes increases the probability of a path being available, but also widens the search space, and lengthens the time needed to examine nodes. Is there a number of nodes in the graph tree that is optimal between time of path calculation and time of path execution?

• Could a developer determine a method and write code to scan the environment and choose which pathfinding algorithm would be the most effective?

9. Acknowledgements

The author would like to thank Professors Joseph Linhoff and Noriko Tomuro of DePaul University's College of Computing and Digital Media for their advice, insight, and assistance on this project.

10. References

[1] P. E. Hart, N. J. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths", July 1968. [Online]. Available:

http://www.cs.auckland.ac.nz/compsci709s2c/resources/Mike.d/astarNilsson.pdf. [Accessed: January 22, 2009].

[2] E. W. Dijkstra, "A Note on Two Problems in Connexion with Graphs", June 11, 1959. [Online]. Available: http://www-m3.ma.tum.de/foswiki/pub/MN0506/WebHome/dijkstra.pdf. [Accessed: March 12, 2009].

[3] A. Stentz, "Optimal and Efficient Path Planning for Partially-Known Environments", May 13, 1994. [Online]. Available: http://www.frc.ri.cmu.edu/~axs/doc/icra94.pdf. [Accessed: April 22, 2009].

Appendix: Data Tables