Any-Angle Path Planning in Real-Time Computing Environments

Chase Johnson

May 2019

Abstract

With an ever growing demand for online goods and shipments, automation of warehouses is becoming a more realistic and rewarding investment for companies. Many different algorithms have to go into these new employees, though. In this paper I explore 3 different traditional path-finding algorithms: Dijkstra's, Uniform Cost Search, and A* — and I'll explore 3 different any-angle searching algorithms: Basic-Theta*, Lazy-Theta*, and Basic Link*. I test these algorithms in different sparse, dense, random, and organized environments to see how they handle the change. I compare them to see which gives the best balance between shortest paths and shortest wait time. I discovered the best algorithm from the ones tested to balance time spent calculating and time saved by moving is the any-angle search Lazy-Theta*.

1 Introduction

As online-based retailers such as Amazon, eBay, and Overstock grow larger, there has been an ever-growing push to create and maintain fully automated warehouse systems. Warehouses today are high demand environments in a race for productivity.

They offer numerous benefits to the company who owns it: 24/7 service, reduced wages, increased speed, and guaranteed correctness. The ultimate show of automation is reaching the illustrious, "lights out" point — where the lights inside are shut off to save electricity while the workers continue fulfilling orders.

Kroger and Ocado, two grocery chain giants, signed a deal in 2018 in which Ocado would provide Kroger the technology to create 3 to 20 automated warehouses in North America [6]. This deal remains extremely important because Ocado's share prices soared 44% that day. The corporate demand is higher than ever.

A fully automated warehouse's workers must know how to carry different shapes and sizes of boxes, the locations of every other robot worker, the warehouse's layout, any irregularities on the floor, and most simply, how they should

move from point A to B most efficiently. This paper seeks to answer which algorithm would best suit a system of automated warehouse workers travelling from point A to point B.

Creating a pathway from point A to B must be performed in a real-time computing environment. It doesn't matter if the robot saved itself 5 feet of moving if calculating that path took 5 minutes. Any loss in computing time must be made up for in savings of travel time.

This paper focuses itself on analyzing the different approaches to A* for intelligent agents within a warehouse environment. Most notably the combination of Theta* in [4] with the additional weight placed on turning in [9]. Normally, A* is restricted to travel in as many heading as you explicitly set up. In the graph used here, A* picks the best 45 ° angle. The difference can be seen in Figure 1.

(a) Traditional shortest path

(b) Any-angle shortest path

Figure 1: Any-angle shortest path v. traditional shortest path

It will also look for advantages in utilizing modifications of them both. They will be compared to unintelligent agents' performance to discover which algorithm would yield both a solution fast enough and short enough.

The warehouse will be represented as a 2-dimensional array called a **Visibility Graph**. In this visibility graph, each space is of uniform distance. The corners of blocked objects are marked, and the corners of a block will be both the start and ending position for our robot. Figure 1 highlights blocked spaces by filling them in black. Blocked vertices are given a black dot on them. The start vertex is blue, the goal vertex is green, and a passable vertex is unmarked.

2 Related Work

We are interested in the relationship between computational time spent and travel time saved. To compare the artificial intelligence algorithms correctly, I've used two control algorithms.

2.1 Control Algorithms

Dijkstra's Algorithm was implemented as a pure control. It utilizes a min-priority queue to reduce the asymptotic runtime from $O(|E| + |V^2|) = O(|V^2|)$ to $\Theta((|E| + |V|) \log |V|)$. Dijkstra's was implemented modified from [3].

1. Create min-priority queue for all vertices in the graph. All vertices $f(s) = \infty$
2. Set $f(start) = 0$
3. Make the current node s the min-element from the priority-queue.
4. Get all of the s's neighbors s'. If $f(s) + Euclid_Dist(s, s') < f(s')$, then set $f(s')$ to the new distance and set $parent(s') = s$.
5. If s is *goal* return the path. Otherwise, return to 3.
6. If the queue is empty, there is no path.

Uniform Cost Search seemed an easy choice for its simplicity. It is essentially a modified version of Dijkstra's Algorithm, but since we are working with an undirected, uniform cost graph, UCS is a natural fit.

2.2 Testing Algorithms

Nilsson, Hart, and Raphael presented $\mathbf{A}^*$ in [7] as an intelligent search algorithm that uses a heuristic to guide its search towards the goal. [7] suggested sorting your search based on a function $f(n)$, such that $f(n) = g(n) + h(n)$ where $f(n)$ is how we weigh searching vertex n, $g(n)$ is the actual cost to travel to vertex n, and $h(n)$ is an estimate of the cost from node n to the goal. They proved that if $h(n)$ is an underestimate that A* would find the optimal path. A*'s adjusted pseudo-code is available in Algorithm 1 from [4].

Alex Nash is involved in most of the prevalent work in any-angle path planning. Nash gives a detailed description of implementation of a general any-angle search algorithm in [4] called **Theta***. Theta* works by checking for *lineofsight*. Line of sight can be determined with Bresenham's line drawing algorithm, presented in 1965 [1]. Nash prefers this line drawing algorithm because it avoids any float arithmetic.[1]

Theta* is also different than the standard A* because it allows any vertex to be the parent of any other vertex, not only successors.[2] This combined with a *lineofsight* check allows Theta* to essentially search all possible angles. In addition, we do not search *succ(s)*, but gather the neighbors of s by returning all adjacent cells that have *lineofsight* with s.

In Algorithm 2, the decision between Path 1 on Line 8 and Path 2 on Line 3 is essentially Theta*. $g(parent(s))$ is the path from the start to the $parent(s)$. If the cost from $parent(s)$ to c is less than $g(c)$, it will be replaced by that path. We can do this check every time as opposed to normal A* because if there is line of sight, at worst, it will be the same as $g(s) + c(s, c)$. Any other time, it is saving space by the triangle inequality by traveling the Euclidean distance as opposed to Manhattan.

[1] The full *lineofsight* code can be found in the appendix of this paper in Algorithm 7.

```
1  Function A-Star(G)
       Data: Visibility graph G
       Result: G with updated parents such that distance(G_start, G_goal) is minimized
2      open ← ∅
3      closed ← ∅
4      g(s_start) ← 0
5      parent(s_start) ← s_start
6      open ← open ∪ {s_start}
7      while open ≠ ∅ do
8          s ← open.Pop()
9          if s = s_goal then
10             return "path found"
11         end if
12         closed ← closed ∪ {s}
13         foreach s' ∈ succ(s) do
14             if s' ∉ closed then
15                 if s' ∉ open then
16                     g(s') ← ∞
17                     parent(s') ← NULL
18                 end if
19                 Update-Vertex(s, s')
20             end if
21         end foreach
22     end while
23     return "no path found"
24 end
25 Function Update-Vertex(s, s')
26     g_old ← g(s')
27     Compute-Cost(s, s')
28     if g(s') < g_old then
29         if s' ∈ open then
30             open ← open \ {s'}
31         end if
32         open ← open ∪ {s'}
33     end if
34 end
```

Algorithm 1: A* Pseudo Code

Nash has improved upon Theta* with **Lazy-Theta***. Lazy-Theta* minimizes the expensive *lineofsight* call by assuming that the *lineofsight* call on Line 2 of Algorithm 2 is true — only verifying so when that vertex is being expanded.

Xu, Shu, and Huang in [10] give a new heuristic that avoids turning to Theta*, making it one of the most practical of all algorithms. [10] calls their algorithm **Basic Link***. Basic Link* works off of a function $\Theta(A, B, C)$, which returns the angle change between 3 points in the graph after changing them into vectors. This is is explicitly shown in Algorithm 1.

$$\Theta(B, A, C) = \begin{cases} 0, & \text{if A=B or A=C (null vector)} \\ arccos\left(\dfrac{\overrightarrow{AB} \cdot \overrightarrow{AC}}{\left\|\overrightarrow{AB}\right\| \times \left\|\overrightarrow{AC}\right\|}\right), & \text{otherwise} \end{cases} \tag{1}$$

```
1  Function Compute-Cost(s, s')
   |  Data: Vertices s, s' ∈ G.
   |  Result: Minimize g(s') and update parent(s') to reflect that.
2  |  if Line-Of-Sight(parent(s), s') then
3  |  |  if g(parent(s)) + c(parent(s), s') < g(s') then          /* Path 2 */
4  |  |  |  parent(s') ← parent(s)
5  |  |  |  g(s') ← g(parent(s)) + c(parent(s), s')
6  |  |  end if
7  |  else
8  |  |  if g(s) + c(s, s') < g(s') then                          /* Path 1 */
9  |  |  |  parent(s') ← s
10 |  |  |  g(s') ← g(s) + c(s, s')
11 |  |  end if
12 |  end if
13 end
```

Algorithm 2: Theta* Pseudo Code

```
1  Function Compute-Cost(s, s')
2  |  if g(parent(s)) + c(parent(s), s') < g(s') then             /* Path 2 */
3  |  |  parent(s') ← parent(s)
4  |  |  g(s') ← g(parent(s)) + c(parent(s), s')
5  |  end if
6  end
7  Function Set-Vertex(s)
8  |  if NOT Line-Of-Sight(parent(s), s')) then                   /* Path 1 */
9  |  |  parent(s) ← argmin_{s'∈nghr_vis(s)∩closed}
10 |  |  g(s) ← min_{s'∈nghbr_vis(s)∩closed}
11 |  end if
12 end
```

Algorithm 3: Lazy-Theta* Pseudo Code

Within Basic Link*, the cost of a node now accounts for angle changes. This is a realistic feature as we want the robot workers to be turning as little as possible. In [10], Xu, Shu, and Huang show that while Basic Theta* does reduce the amount of turns considerably, it also considerably increases the running time and the distance of the final path.

3 Approach

The algorithm takes 2D grid of size $(m \times n)$:

- 0 for unblocked corner
- 1 for blocked corner
- S for the start node
- G for the goal node

Afterwards, this grid is exchanged into an 8-neighbor graph G of size $(m \times n)$. A new graph O of size $(m-1 \times n-1)$ graph is then created. O holds if the center spaces are blocked to signify line of sight. These data structures consist of vertices V in a 2D matrix.

All algorithms return a path $P =< G_{start}, \ldots, G_{goal} >$ such that equation 2 is minimized.

$$P(x)_d = \sum_{j=0}^{n-1} \sqrt{((v_j)_x - (v_{j+1})_x)^2 + ((v_j)_y - ((v_{j+1}))_y)^2} \tag{2}$$

To create the grids, I gathered warehouse designs from a publicly available resource [8] and randomly generated both sparse and dense grids for the algorithms to traverse.[2] The warehouse designs have the robot workers picking up an item in a drop off point or depot and bringing it to storage or onto a shelf. The randomly generated grids were created by pseudo-random algorithms. The sparse grids had a 10% likelihood of a space being blocked; whereas the dense grids had a 20% chance of being blocked. The start and goal positions were also pseudo-randomly decided. The grids were verified to have a valid path from start to goal.

I will test each graph G with the algorithms:

- Dijkstra's (Algorithm 6)
- Uniform Cost Search (Algorithm 5)
- A* (Algorithm 1)
- Lazy-Theta* (Algorithm 3)
- Basic Link* (Algorithm 4)

My approach to find the shortest travel time path was to minimize the travel distance as much as possible. Since this is computed in real time on the warehouse floor, G cannot be completely searched in time, once it is sufficiently large, but the computation time pales in comparison to the travel time. The normal search space on a visibility graph limited to 45 degree increments, or an eight-neighbor system. This is simply not enough to produce truly optimal movement. Not only does it travel further than its all-angle counterpart, but it turns more as well, so a grid-following algorithm has two major setbacks from the beginning. In addition, a warehouse's grid does not have to be extremely accurate. If there is a corner node for every foot, there would more than enough accuracy to traverse quickly and accurately before switching into a different mode in order to pick up the package or place it down.

[2] All grids are displayed in the Appendix in Figures 3, 4, and 5

4 Results

4.1 Length Efficiency

Distance	Dijkstra's	UCS	A*	Lazy-Theta*	Basic Link*
Warehouse 1	167.28	167.28	167.28	162.14	162.14
Warehouse 2	367.28	367.28	367.28	357.10	662.24
Warehouse 3	809.41	809.41	809.41	760.55	923.42
Warehouse 4	893.14	893.14	893.14	865.91	865.91

Table 1: Path lengths for each warehouse grid

Unsurprisingly, Table 1 shows Dijkstra's, UCS, and A* all yielded the same path sized for all tests. Given they are grid-based search algorithms, this is to be expected. Lazy-Theta* always matched or improved upon the grid-based search. However, Basic Link* was inconsistent. In Table 1, Basic Link*'s shortest path was an average of 94.15 units longer of the grid optimal. Lazy-Theta* had its largest average improvement on warehouse paths with an average of 22.85 units shorter. The warehouse had an average path length of 559.28 units, the largest of all experiments. Surprisingly, Basic Link* performed worse than the average by 94 units for this test group.

Distance	Dijkstra's	UCS	A*	Lazy-Theta*	Basic Link*
Sparse 1	94.85	94.85	94.85	93.07	93.07
Sparse 2	224.85	224.85	224.85	213.21	215.73
Sparse 3	732.25	732.25	732.25	705.39	707.45
Sparse 4	304.56	304.56	304.56	292.75	292.75

Table 2: Path lengths for each random sparse grid

Again, Table 2 shows grid-based search algorithms' consistency. However, any-angle algorithms created shorter paths for each sparse trial. Lazy Theta* saw an average of 13.02 units saved while the average grid path was 339.13. Basic Link* saw 11.88 units of saving. It would make sense that Basic Link* is too averse to turning. In a sparse graph it saves time but within the warehouse, where sharp corners are the best option, it failed.

Distance	Dijkstra's	UCS	A*	Lazy-Theta*	Basic Link*
Dense 1	124.14	124.14	124.14	120.83	120.83
Dense 2	296.57	296.57	296.57	286.51	286.51
Dense 3	601.42	601.42	601.42	582.16	602.59
Dense 4	731.13	731.13	731.13	703.94	739.88

Table 3: Path lengths for each random dense grid

In Table 3, Basic Link* takes longer paths than the grid optimal on dense grid

3 and 4. Lazy-Theta* performed better than the grid optimal every trial. Figure 2 shows the path lengths difference from grid-restricted optimal. In warehouse and dense trials, Basic-Link∗ performs poorly compared to Lazy-Theta*.

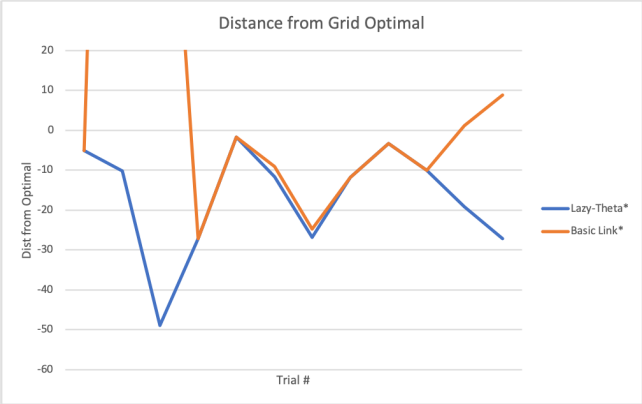


Figure 2: Lazy-Theta* and Basic Link* difference from grid optimal

5 Time Efficiency

Time (ms)	Dijkstra's	UCS	A*	Lazy-Theta*	Basic Link*
Warehouse 1	8.77	8.10	6.66	8.42	5.02
Warehouse 2	157.60	72.46	55.71	56.67	23.37
Warehouse 3	1319.50	957.81	668.75	664.07	152.65
Warehouse 4	3009.71	1370.49	954.62	968.05	140.78

Table 4: Time taken to compute optimal path for warehouse grids

Table 4 shows us the speed of Basic Link* is far ahead of the others. Lazy-Theta* has no comparable difference from A* at this point. Dijkstra's is too slow to reasonably use, and UCS is not as fast as A* — so it offers no real benefit at this time.

Time (ms)	Dijkstra's	UCS	A*	Lazy-Theta*	Basic Link*
Sparse 1	7.10	3.62	3.10	4.07	1.83
Sparse 2	243.13	28.78	21.91	26.07	3.48
Sparse 3	1377.69	1809.19	1216.11	1557.63	96.56
Sparse 4	1826.88	510.31	375.06	439.14	65.54

Table 5: Time taken to compute optimal path for sparse random grids

In Table 5 we observe Lazy-Theta* performed much slower than A*. In the warehouse tests, Lazy-Theta* was quite comparable. The difference from

A* to Lazy-Theta* in warehouse runs was an average of 2.87ms. Whereas in sparse graphs there is an average difference of 102.69ms. Perhaps Lazy-Theta*'s *lineofsight* calls are too expensive to keep up in large graphs.

Time (ms)	Dijkstra's	UCS	A*	Lazy-Theta*	Basic Link*
Dense 1	7.28	1.98	1.66	2.19	0.53
Dense 2	222.20	71.88	52.69	62.24	5.35
Dense 3	1219.11	540.14	373.20	478.67	43.33
Dense 4	2476.76	2366.88	1610.76	1964.00	2170.89

Table 6: Time taken to compute optimal path for dense random grids

In Table 6 we observe Basic Link* is not the fastest for the first time in Dense 4. It seems that Basic Link* is much faster than the rest of the algorithms analyzed in this paper, but the accuracy will falter at unknown times. On warehouse graphs especially, Basic Link* failed to match the grid-restricted optimum.

6 Analysis

From the scope of a warehouse worker, Basic Link* would result in worse productivity compared to its successful peers: hazards from errant path-finding, unreliable path-length, and unreliable computation time make it a poor fit. On average, Basic Link* goes 27 more units than a grid restricted search. Even though only 4 were worse than the grid optimum — they were so far off the optimum it is a net-loss.

A* is ideal if the warehouse cannot feasibly be a fully automated system or the grid must be maintained. Perhaps there needs to be paths delegated for automated use and others where humans can walk. A* saves on average 200.12 ms compared to UCS, and saves 544.63ms compared to Dijkstra's while still finding the grid-restricted optimum.

However fast A* is, Lazy-Theta* always improves on the grid-restricted optimum and finished within a tenth of a second of A* on average. Opening up the robot to the entire floor is a change that will improve the larger the warehouse is. For this kind of automation to work well, there needs to be predictability over pure speed.

7 Conclusion

In conclusion, Lazy-Theta* should be tested in an actual Real Time Computing environment. It requires many more calls to *lineofsight* than is ideal. Perhaps storing common locations' neighbors in memory would cut down on the time taken to compute them. A* is the most useful if sticking to the grid is desired. Even though it will increase the distance, it is reliable and by far the fastest of the grid-restricted algorithms.

For future work, Xu, Shu, and Huang [10] provide more optimized forms of the algorithm they call Link*. Further testing should be performed on these versions to discover if they yield a more reliable path-finding algorithm.

References

[1] Jack E Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems journal*, 4(1):25–30, 1965.

[2] František Duchoň, Andrej Babinec, Martin Kajan, Peter Beňo, Martin Florek, Tomáš Fico, and Ladislav Jurišica. Path planning with modified a star algorithm for a mobile robot. *Procedia Engineering*, 96:59–69, 2014.

[3] Nadira Jasika, Naida Alispahic, Arslanagic Elma, Kurtovic Ilvana, Lagumdzija Elma, and Novica Nosovic. Dijkstra's shortest path algorithm serial and parallel execution performance analysis. In *2012 proceedings of the 35th international convention MIPRO*, pages 1811–1815. IEEE, 2012.

[4] Alex Nash, Kenny Daniel, Sven Koenig, and Ariel Felner. Theta^*: Any-angle path planning on grids. In *AAAI*, volume 7, pages 1177–1183, 2007.

[5] Alex Nash, Sven Koenig, and Craig Tovey. Lazy theta*: Any-angle path planning and path length analysis in 3d. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

[6] BBC News. Ocado shares rise 44% on news of kroger tech deal. *BBC News*, may 2018.

[7] Nils Nilsson, Peter Hart, and Raphael Bertram. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions of Systems Science and Cybernetics*, 4(2):100–107, jul 1968.

[8] Varsha Saha. 10 great warehouse organization charts. *Camcode*, jan 2019.

[9] Chunbao Wang, Lin Wang, Jian Qin, Zhengzhi Wu, Lihong Duan, Zhongqiu Li, Mequn Cao, Xicui Ou, Xia Su, Weiguang Li, et al. Path planning of automated guided vehicles based on improved a-star algorithm. In *2015 IEEE International Conference on Information and Automation*, pages 2071–2076. IEEE, 2015.

[10] Hu Xu, Lei Shu, and May Huang. Planning paths with fewer turns on grid maps. In *Sixth Annual Symposium on Combinatorial Search*, 2013.

Appendices

```
1  Function INITIALIZE-VERTEX(s)
2      if s = start then
3          parent(s) ← s
4          s_f ← 0
5      else
6          parent(s) ← NULL
7          s_f ← ∞
8      end if
9  end
10 Function CHOOSE-PATH1(s, s')
11     f_temp ← parent(s)_f + Θ(G_goal, parent(s), s')
12     if f_temp < s'_f then
13         parent(s') ← parent(s)
14         s'_f ← f_temp
15         if s' ∉ open then
16             open ← open ∪ {s'}
17         end if
18     end if
19 end
20 Function CHOOSE-PATH2(s, s')
21     f_temp ← s_f + Θ(G_goal, s, s')
22     if f_temp < s'_f then
23         parent(s') ← s
24         s'_f ← f_temp
25         if s' ∉ open then
26             open ← open ∪ {s'}
27         end if
28     end if
29 end
```

Algorithm 4: Basic Link* Pseudo Code

```
Function UNIFORM-COST-SEARCH(G)
    Data: G such that G is a visibility graph.
    Result: G with updated parents such that cost(G_start, G_goal) is minimized
    node ← G_start
    cost ← 0
    frontier ← ∅
    explored ← ∅
    frontier ← frontier ∪ {0, node, [node]}
    while frontier ≠ ∅ do
        cost, node, path ← frontier.Pop()
        if node ∉ explored then
            explored ← explored ∪ {node}
            if node = G_goal then
                return "path found"
            end if
            foreach neighbor v of node do
                if v ∉ explored then
                    cost_new ← cost + distance(node, v)
                    frontier ← frontier ∪ {cost_new, v, path ∪ {v}}
                end if
            end foreach
        end if
    end while
    return "no path found"
end
```

Algorithm 5: Uniform Cost Search Pseudo Code

```
Function DIJKSTRA(G)
    Data: G such that G is a visibility graph.
    Result: G with updated parents such that cost(G_start, G_goal) is minimized
    foreach vertex v ∈ G do
        v_distance ← ∞
        parent(v) ← NULL
    end foreach
    (G_start)_distance ← 0
    Q ← the set of all vertices in G
    while Q ≠ ∅ do
        u ← Q.Pop()
        if u = G_goal then
            return "path found"
        end if
        foreach neighbor v of u do
            alt ← u_distance + distance(u, v)
            if alt < v_distance then
                v_distance ← alt
                parent(v) ← u
            end if
        end foreach
    end while
    return "no path found"
end
```

Algorithm 6: Dijkstra's Algorithm

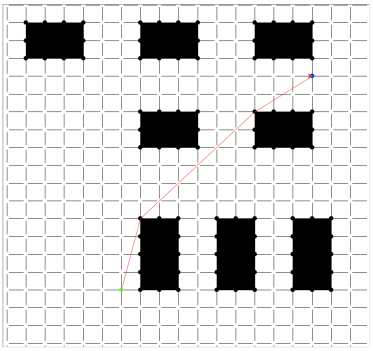
```
1  Function Line-Of-Sight(s, s')
       Data: Vertices s, s' ∈ G
       Result: Boolean if s can see s'
2      x_0 ← s.x
3      y_0 ← s.y
4      x_1 ← s'.x
5      y_1 ← s'.y
6      d_y ← y_1 − y_0
7      d_x ← x_1 − x_1
8      f ← 0
9      if d_y < 0 then
10         d_y ← −d_y
11         s_y ← −1
12     else
13         s_y ← 1
14     if d_x < 0 then
15         d_x ← −d_x
16         s_x ← −1
17     else
18         s_x ← 1
19     if d_x ≥ d_y then
20         while x_0 ≠ x_1 do
21             f ← f + d_y
22             if f ≥ d_x then
23                 if grid[x_0 + ((s_x − 1)/2), y_o + ((s_y − 1)/2)] then
24                     return false
25                 y_0 ← y_0 + s_y
26                 f ← f − d_x
27             if f ≠ 0 AND grid[x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)] then
28                 return false
29             if d_y = 0 AND grid[x_0 + ((x_x − 1)/2), y_0]  AND
                grid[x_0 + ((s_x − 1)/2), y_0 − 1] then
30                 return false
31             x_0 ← x_0 + s_x
32     else
33         while y_0 ≠ y_1 do
34             f ← f + d_x
35             if f ≥ d_y then
36                 if grid[x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)] then
37                     return false
38                 x_0 ← x_0 + s_y
39                 f ← f − d_y
40             if f ≠ 0 AND grid[x_0 + ((s_x − 1)/2), y_0 + ((s_y − 1)/2)] then
41                 return false
42             if d_x = 0 AND grid[x_0, y_0 + ((s_y − 1)/2)] AND
                grid[x_0 − 1, y_0 + ((s_y − 1)/2)] then
43                 return false
44             y_0 ← y_0 + s_y
45     return true
```

Algorithm 7: Line of sight Pseudo Code

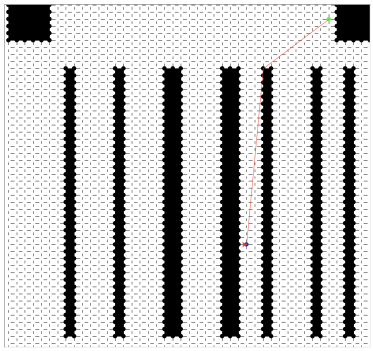
Data: Vertices $B, A, C \in G$
Result: Angle formed by vertices $\overrightarrow{AB}$ and $\overrightarrow{BC}$

Function $\Theta(B, A, C)$ `/* Null vector case */`
  if $A = B$ *or* $B = C$ **then**
    return *0*
  else
    return $arccos\left(\frac{\overrightarrow{AB} \cdot \overrightarrow{AC}}{\left\|\overrightarrow{AB}\right\| \times \left\|\overrightarrow{AC}\right\|}\right)$
  end if
end

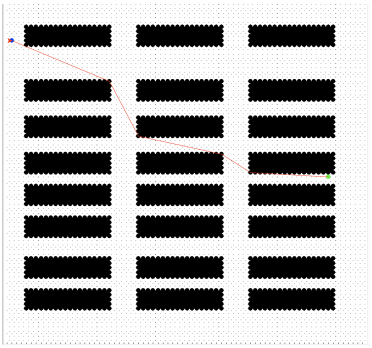
Algorithm 8: $\Theta(B, A, C)$ from Basic Link*



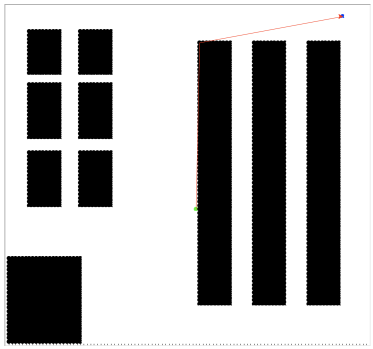
(a) (20x20) Warehouse 1



(b) (50x50) Warehouse 2

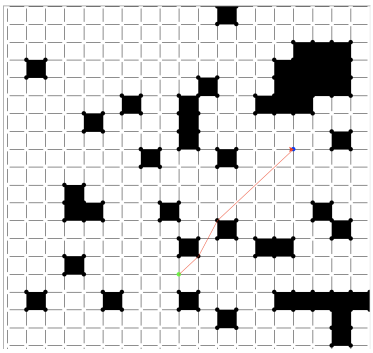


(c) (75x75) Warehouse 3

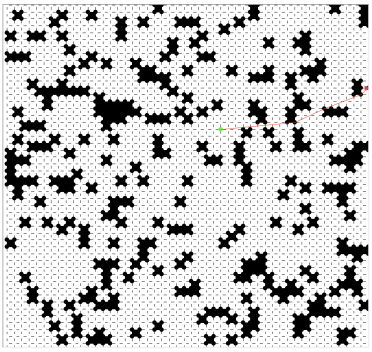


(d) (100x100) Warehouse 4

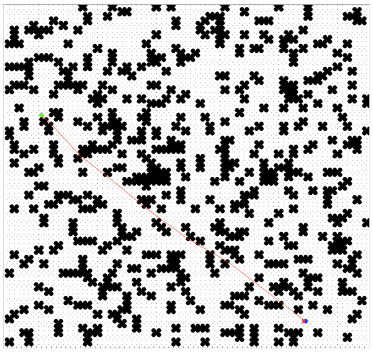
Figure 3: Theta* paths on warehouse grids



(a) (20x20) Sparse 1



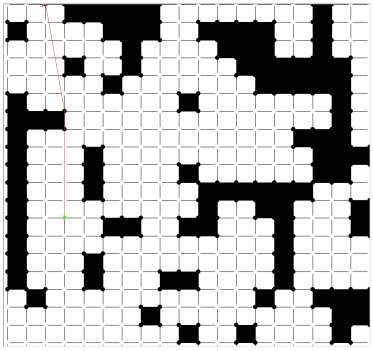
(b) (50x50) Sparse 2



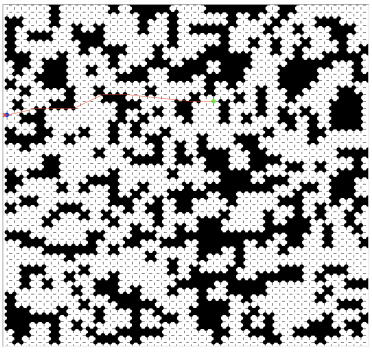
(c) (75x75) Sparse 3

(d) (100x100) Sparse 4

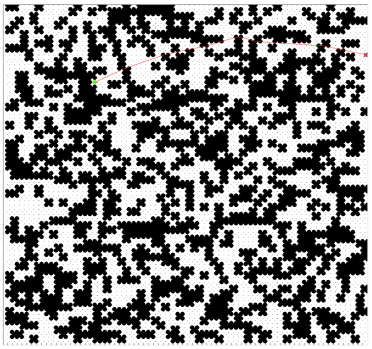
Figure 4: Theta* paths on sparse grids



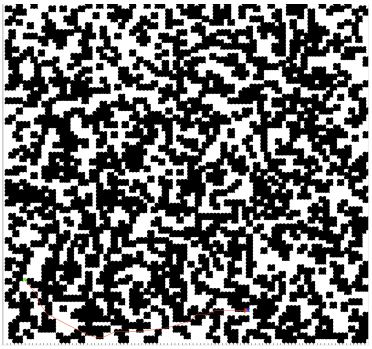
(a) (20x20) Dense 1



(b) (50x50) Dense 2



(c) (75x75) Dense 3



(d) (100x100) Dense 4

Figure 5: Theta* paths on dense grids