Fast A*: An A* Path Finding algorithm based on FastMap

Introduction

In this project our goal is to use fastmap in a* path finding algorithm instead of using Euclidian Distance as heuristic function. We use fastmap as an offline preprocessing step and map all of the free points in an k-dimensional space. After preprocessing, for finding the path between two points, we use the Euclidian distance between coordinates of points in k-dimensional space. We show that these results are more accurate and faster than A* in mazes with complicated paths. In the following sections, we introduce the original A* and fastmap algorithm and finally, explain our project as a combination of these two algorithms.

A* Algorithm:

A* is one of the most popular methods for finding the shortest path between two locations in a mapped area. A* combines heuristic approaches like Best-First-Search (BFS) and formal approaches like Dijsktra's algorithm.

The defining characteristics of the A* algorithm are the building of a "closed list" to record areas already evaluated, a "open list" to record areas adjacent to those already evaluated, and the calculation of distances traveled from the "start point" with estimated distances to the "goal point".

The heuristic used to evaluate distances in A* is:

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

Where g(n) represents the cost (distance) of the path from the starting point to any vertex n, and h(n) represents the estimated cost from vertex n to the goal.

Euclidean distance (straight line distance) is a common method to used for h(n).

A* Algorithm steps

There are two sets, open and closed. The open set contains those nodes that are candidates for examining. Initially, the closed set contains just one element: the starting position. The closed set contains those nodes that have already been examined. Initially, the closed set is empty. Graphically, the open set is the "frontier" and the CLOSED set is the "interior" of the visited areas. Each node also keeps a pointer to its parent node so that we can determine how it was found.

There is a main loop that repeatedly pulls out the best node n in open (the node with the lowest f value) and examines it. If n is the goal, then we're done. Otherwise, node n is removed from open and added to closed set. Then, its neighbors n' are examined. A neighbor that is in closed set has already been seen, so we don't need to look at it. A neighbor that is in open set will be examined if its f value becomes the lowest in open set. Otherwise, we add it to open, with its parent set to n. The path cost to n', g(n'), will be set to g(n) + movement-cost(n, n').

FastMap Algorithm

FastMap is a fast algorithm developed to map objects into points in some k-dimensional space (k is user-defined), such that the dis-similarities are preserved.It takes O(Nk) while N is the number of objects and k is the desired dimensionality.

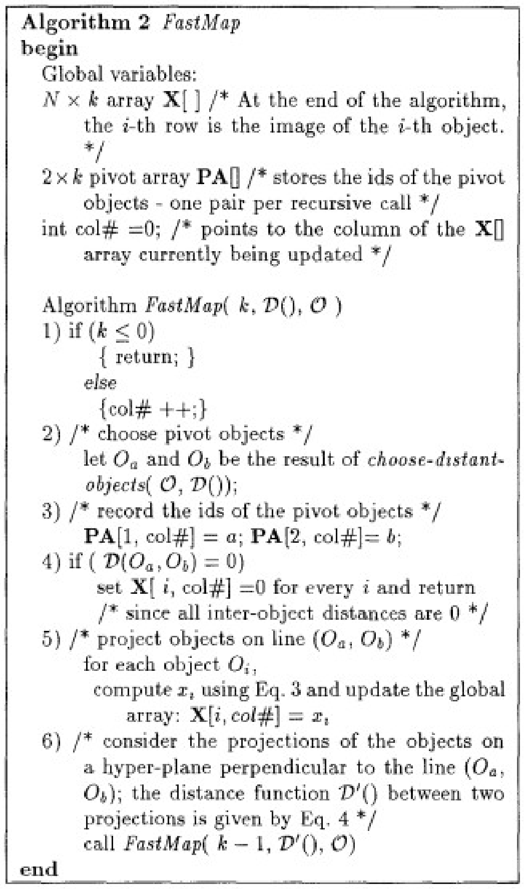
FastMap Algorithm Specifications

FastMap takes N objects and their matrix distances as its input. For example: Euclidian distance between two feature vectors as the distance function between the corresponding objects.
As the goal, FastMap tries to project these N points on K mutually orthogonal directions only by using distances among them. This algorithm assumes that distance is metric which means it is reflexive, symmetric, and obeys the triangular inequality.

FastMap Algorithm Steps

FastMap follows a few straightforward steps:

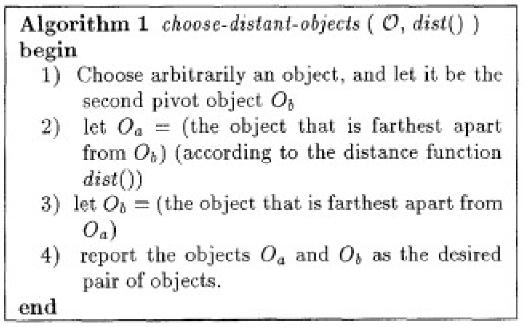
1) It finds two far away objects and makes a pivot using them
2) By projecting all of the points on that pivot, it reaches one of the coordinates of each point
3) It projects all of the objects on a hyperplane perpendicular to the defined pivot
4) Repeat for k-1 times



```
Algorithm 2 FastMap
begin
  Global variables:
  $N \times k$ array $\mathbf{X}[\,]$ /* At the end of the algorithm,
        the $i$-th row is the image of the $i$-th object.
        */
  $2 \times k$ pivot array $\mathbf{PA}[]$ /* stores the ids of the pivot
        objects - one pair per recursive call */
  int col# =0; /* points to the column of the $\mathbf{X}[]$
        array currently being updated */

  Algorithm FastMap( $k$, $\mathcal{D}()$, $\mathcal{O}$ )
  1) if ($k \leq 0$)
          { return; }
        else
          {col# ++;}
  2) /* choose pivot objects */
        let $O_a$ and $O_b$ be the result of choose-distant-
        objects( $\mathcal{O}$, $\mathcal{D}()$);
  3) /* record the ids of the pivot objects */
        $\mathbf{PA}[1, col\#] = a$; $\mathbf{PA}[2, col\#] = b$;
  4) if ( $\mathcal{D}(O_a, O_b) = 0$)
          set $\mathbf{X}[\, i, col\#] = 0$ for every $i$ and return
          /* since all inter-object distances are 0 */
  5) /* project objects on line ($O_a$, $O_b$) */
        for each object $O_i$,
         compute $x_i$ using Eq. 3 and update the global
              array: $\mathbf{X}[i, col\#] = x_i$
  6) /* consider the projections of the objects on
        a hyper-plane perpendicular to the line ($O_a$,
        $O_b$); the distance function $\mathcal{D}'()$ between two
        projections is given by Eq. 4 */
        call FastMap( $k - 1$, $\mathcal{D}'()$, $\mathcal{O}$)
end
```

Step1: Find two far away objects as pivot points in linear time
This algorithm takes O(N) in total.



```
Algorithm 1 choose-distant-objects ( $\mathcal{O}$, dist() )
begin
  1) Choose arbitrarily an object, and let it be the
     second pivot object $O_b$
  2) let $O_a$ = (the object that is farthest apart
     from $O_b$) (according to the distance function
     dist())
  3) let $O_b$ = (the object that is farthest apart from
     $O_a$)
  4) report the objects $O_a$ and $O_b$ as the desired
     pair of objects.
end
```

Step 2: Project points on the pivot
After making the pivot using O_a and O_b, two farthest points, we can reach the coordinate of O_i in the O_aO_b pivot using Cosine Law. Following formula is derived from this law with some simple math manipulation:

$$x_i = \frac{d_{a,i}^2 + d_{a,b}^2 - d_{b,i}^2}{2d_{a,b}}$$

Step3: Project points on a hyperplane orthogonal to the pivot
Suppose that points are in n-dimensional space. Now consider an n-1 dimensional hyperplane, H, that is perpendicular to the line (O_a ,O_b).By projecting points on this hyperplane, we have all of the points on a n-1 dimensional hyperplane, the problem is the same as the original problem with n and k decreased by one.

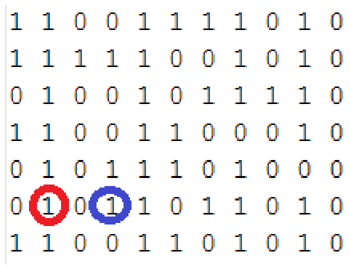
Calculate D'() using D():
We can calculate the new distances among points,D'(), by using original distances, D() using following formula derived from Pythagorean theorem:

$$d'(y',z') = \sqrt{d^2(y,z) - (c_z - c_y)^2}$$

Step 4: recurse until k features are chosen
In each iteration, by having the points mapped on a n-1 dimensional hyperplane, and D'() as distance function, we continue the recursively until k features are chosen. The final iteration gives us coordinates of objects in a k-dimensional space.

Fast A*

In this section we mainly explain how our algorithm is derived from A* search and FastMap algorithm, improving the speed and accuracy of A* in finding path between two points. We believe that heuristic of a* (Euclidian distance) is not accurate enough in complicates maze structures. As an example, look at the following maze considering 1 is free cell and 0 is blocked cell.



```
1 1 0 0 1 1 1 1 0 1 0
1 1 1 1 1 0 0 1 0 1 0
0 1 0 0 1 0 1 1 1 1 0
1 1 0 0 1 1 0 0 0 1 0
0 1 0 1 1 1 0 1 0 0 0
0 1 0 1 1 0 1 1 0 1 0
1 1 0 0 1 1 0 1 0 1 0
```

If we want to go from red cell to blue cell, Euclidian distance tells us that they are only two cells far away. However, we can see that having the blocks in the path, makes the actual distance to be 12. We know that if we could include the actual distance between the start and destination point, our A* would be more informed and so faster.

Now, FastMap as a fast algorithm in mapping points to k-dimensional space and by preserving the distances between the points comes to the picture.
We preprocess the map in an offline fashion by mapping the free cells to a k-dimensional space from 2D space. Passing these k-dimensional points to a* algorithm, the heuristic function would work more accurately as it is using the actual distance (considering block cells in the path) between points.

In this algorithm, the cost of passing from free cells is 1 but passing from blocked cells is expensive, 1 000 000.

Implementation

```
Fast A*():

Global variables:
 N x k array X[ ] // At the end of the algorithm, the i-th row is the image of the i-th object.
2 x k pivot array PA[] // stores the ids of the pivot objects - one pair per recursive call
 int col# =0; // points to the column of the Xo array currently being updated

FastA*(k,free_cells):

   if (k<=0):
     { return}
   else:
     { col++}

//Finds the two far points using Dijkstra algorithm and returns the distance tree in the form of map of {node:distance_to_node } based on
each pivot cell(root of tree)

   Oa,Ob,distOa,distOb=choose_distant_objects(FreeCells,dijkstra)
// record the ids of the pivot objects
   PA[0][col]= FreeCells.index(Oa)
   PA[1][col]= FreeCells.index(Ob)
   if (distOa(Ob)==0):
      set X[ i, col#] =0 for every i and return # since all inter-object distances are 0
// project objects on line (Oa, Ob). Use the distances that we got from running Dijkstra on pivot points.
   def x2(i,a,b):
      x=(pow(distOa[str(i)],2)+pow(distOa[str(b)],2)-pow(distOb[str(i)],2))*1.0/(2*distOa[str(b)]*1.0)
      return x


   for o in FreeCells:
      x=0;
      x=x2(o,Oa,Ob)
      X[FreeCells.index(o)][col]=x
   FastA*(k-1, FreeCells)

choose_distance_objects(FreeCells,Dijkstra):

  ob=random free cell
   oa=None

   distOb=Dijkstra(FreeCells,ob)//Finds shortest path from ob to all other freecells and returns the tree.
   oa,post_distOb=postprocess(ob,distOb)// calculates the distance based on current iteration, and
selects furthest point from ob,say oa
   distOa= Dijkstra(FreeCells,ob)//Find shortest path tree rooted at oa
   ob,post_distOa=postprocess(oa,distOa)// new distance and furthest point from oa
   distOb= Dijkstra(FreeCells,ob)//shortest path tree rooted at ob
   oc,post_distOb=postprocess(ob,distOb)// new distance and furthest point from ob

   return oa,ob,distOa,distOb

postprocess(root,tree):

furthest_point=None
max_distance=0
 for all of the nodes in free cells:
//find the difference between actual tree based distance and generated coordinate distances
            distance=tree_distance[node,root]^2- coordinate_distance[node,root]^2
            if distance>=0:
                        distance[node]=sqrt(distance)+tree_distance
//find the furthest point from the root
                        if (distance[node]>max_distance):
                                    furthest_point=node
                                    max_distance= distance[node]
            else:
                        distance[node]=0+tree_distance
                        furthest_point=node
                        max_distance= distance[node]
//reutn the furthest node from root and distance map
return furthest_point,distance;
```

We also provide an implementation of the Original FastMap algorithm(based on paper[1]) that take the original distance function as L1 distance from all walkable cells. To find L1 distance, we perform Breadth First Search starting from both pivot cells in each dimension and cache the distance result from each cell to 2 pivot cells. The benefit of this implementation is it can generate coordinates in higher dimensions faster and allow us to run it on mazes with a larger size.

We will refer to FastMap implementation for the upper path as **Modified FastMap A*** and the latter path as **Original FastMap A***.

Testing

TestSuite module provided the following function:

```
project.TestSuite.runTestSuite(maze_width_heights,k_tests,initial_maze_id = 0,maze_complexity = 0.9,maze_density =
0.9,is_use_modified_fastmap = True):
```

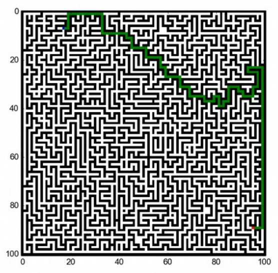
Researchers can specify
(1) maze_width_heights - list of (maze_width,maze_height)
(2) k_tests - a list of transformed dimensions for FastMap
(3) maze_id - initial maze ID to be report
(4) maze_complexity - how long a wall will be
(5) maze_density - how many walls will be present
(6) is_use_modified_fastmap – specify whether to use the Original FastMap algorithm(based on paper[1]) or the Modified FastMap algorithm

TestSuite will generate a randomized maze and run a test based on provided parameters. Running result is recorded in file result_data.csv. For one maze with a shape (maze_width,maze_height), TestSuite run A* based on Euclidean Distance on that maze and transform maze's cell coordinates into a list of dimension specified by k_tests and run FastMap A* on those transformed coordinates.

The following data is collected in the file result_data.csv:

Column	Description
maze_id	Id of the maze that this pathfinding algorithm is run on
maze_width	Width of the maze (even width will be transformed to odd width)
maze_height	Height of the maze (even height will be transformed to odd height)
alg_type	Type of path finding algorithm ("normal" - A* based on Euclidean Distance "fastmap" - A* based on FastMap)
k	Dimensionth that A* with FastMap run on (this value will be −1 for A* based on Euclidean Distance)
gen_coordinate_time	Time (in mills-seconds) taken to generate cell's coordinate in kth dimension (this value will be 0 for A* based on Euclidean Distance)
alg_time	Time (in mills-seconds) taken to find a path from the start cell to the end cell
alg_num_steps	Number of steps the algorithm take to walk from the start cell to the end cell
alg_path	Sequence of steps the algorithm take to walk from the start cell to the end cell
alg_image	Image depicts the path algorithm take to walk from the start cell to the end cell - For A* with Euclidean distance, the image name is {maze_id}_normal_{maze_width}_{maze_height}.png - For FastMap A*, the image name is {maze_id}_fastmap_{maze_width}_{maze_height}_{k_dimension}.png

The image depicting the path that each algorithm take is looks like as follows:



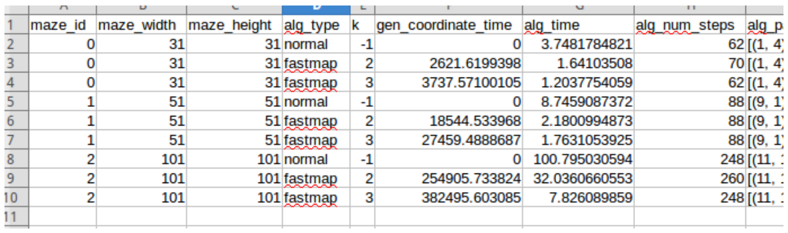
Black, white, green, blue, and red colors represent a wall, walkable cell, path taken, start cell, and end cell, respectively.

Experiment Result

Our testing machine

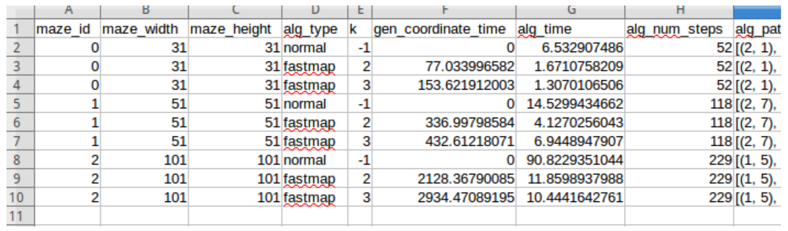
Dell Inspiron 15 5558 Laptop
Cpu: Intel Core i7-5500U
Ram: 8GB DDR3
OS: Elementary OS "Freya"(based on Ubuntu 14.04 LTS)

Due to time and computing resource constraints, we run Modified FastMap A* on just a number of small mazes and transformed dimensions. We also run Original FastMap A* on the maze with the same size. Here is a sample result:



maze_id	maze_width	maze_height	alg_type	k	gen_coordinate_time	alg_time	alg_num_steps	alg_p
0	31	31	normal	-1	0	3.7481784821	62	[(1, 4
0	31	31	fastmap	2	2621.6199398	1.64103508	70	[(1, 4
0	31	31	fastmap	3	3737.57100105	1.2037754059	62	[(1, 4
1	51	51	normal	-1	0	8.7459087372	88	[(9, 1
1	51	51	fastmap	2	18544.533968	2.1800994873	88	[(9, 1
1	51	51	fastmap	3	27459.4888687	1.7631053925	88	[(9, 1
2	101	101	normal	-1	0	100.795030594	248	[(11,
2	101	101	fastmap	2	254905.733824	32.0360660553	260	[(11,
2	101	101	fastmap	3	382495.603085	7.826089859	248	[(11,

(Modified FastMap A*)

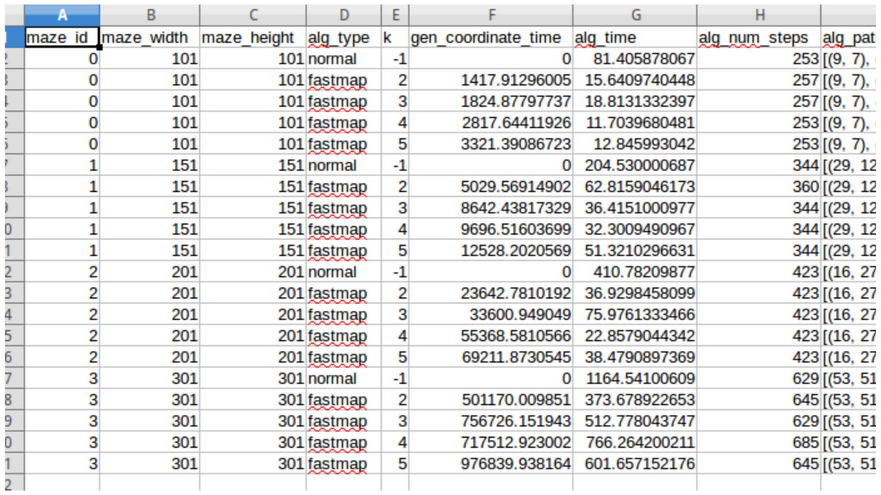


maze_id	maze_width	maze_height	alg_type	k	gen_coordinate_time	alg_time	alg_num_steps	alg_pat
0	31	31	normal	-1	0	6.532907486	52	[(2, 1),
0	31	31	fastmap	2	77.033996582	1.6710758209	52	[(2, 1),
0	31	31	fastmap	3	153.621912003	1.3070106506	52	[(2, 1),
1	51	51	normal	-1	0	14.5299434662	118	[(2, 7),
1	51	51	fastmap	2	336.99798584	4.1270256043	118	[(2, 7),
1	51	51	fastmap	3	432.61218071	6.9448947907	118	[(2, 7),
2	101	101	normal	-1	0	90.8229351044	229	[(1, 5),
2	101	101	fastmap	2	2128.36790085	11.8598937988	229	[(1, 5),
2	101	101	fastmap	3	2934.47089195	10.4441642761	229	[(1, 5),

(Original FastMap A*)

The full result and image depicting their path taking can be found in the folder "/2fastmap_compare".

We also run Original FastMap A* on mazes with a larger size, and transform coordinates into a higher dimension since their coordinates can be computed faster.

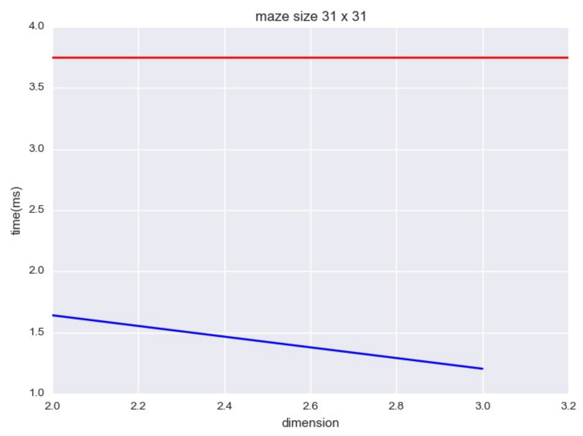


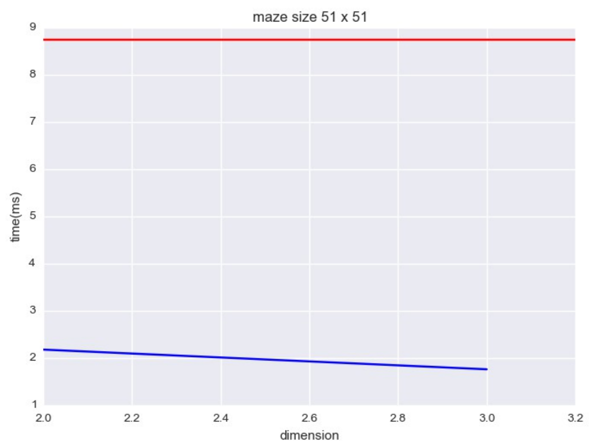
maze_id	maze_width	maze_height	alg_type	k	gen_coordinate_time	alg_time	alg_num_steps	alg_pat
0	101	101	normal	-1	0	81.405878067	253	[(9, 7),
0	101	101	fastmap	2	1417.91296005	15.6409740448	257	[(9, 7),
0	101	101	fastmap	3	1824.87797737	18.8131332397	257	[(9, 7),
0	101	101	fastmap	4	2817.64411926	11.7039680481	253	[(9, 7),
0	101	101	fastmap	5	3321.39086723	12.845993042	253	[(9, 7),
1	151	151	normal	-1	0	204.530000687	344	[(29, 12
1	151	151	fastmap	2	5029.56914902	62.8159046173	360	[(29, 12
1	151	151	fastmap	3	8642.43817329	36.4151000977	344	[(29, 12
1	151	151	fastmap	4	9696.51603699	32.3009490967	344	[(29, 12
1	151	151	fastmap	5	12528.2020569	51.3210296631	344	[(29, 12
2	201	201	normal	-1	0	410.78209877	423	[(16, 27
2	201	201	fastmap	2	23642.7810192	36.9298458099	423	[(16, 27
2	201	201	fastmap	3	33600.949049	75.9761333466	423	[(16, 27
2	201	201	fastmap	4	55368.5810566	22.8579044342	423	[(16, 27
2	201	201	fastmap	5	69211.8730545	38.4790897369	423	[(16, 27
3	301	301	normal	-1	0	1164.54100609	629	[(53, 51
3	301	301	fastmap	2	501170.009851	373.678922653	645	[(53, 51
3	301	301	fastmap	3	756726.151943	512.778043747	629	[(53, 51
3	301	301	fastmap	4	717512.923002	766.264200211	685	[(53, 51
3	301	301	fastmap	5	976839.938164	601.657152176	645	[(53, 51

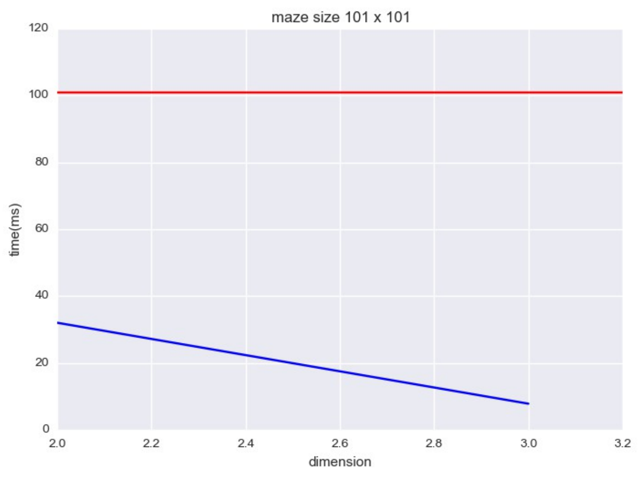
(Full result can be found in a file result_data.vanilla_fastmap.csv and folder img_vanilla_fastmap)

The time it took for each algorithm to execute A* search can be depicted as follows(the red line is the based line A* based on Euclidean distance)

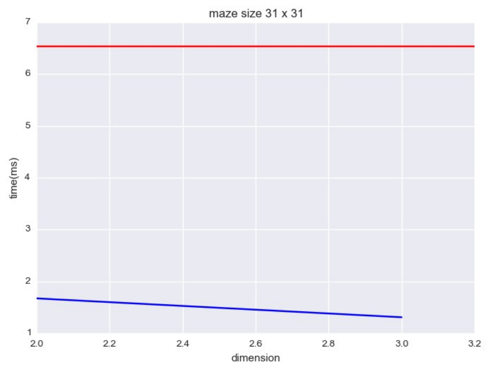
(1) Modified FastMap A* search

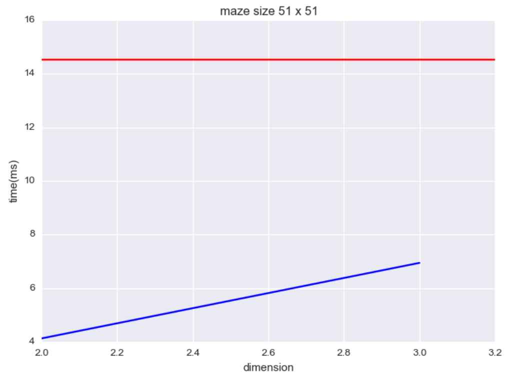


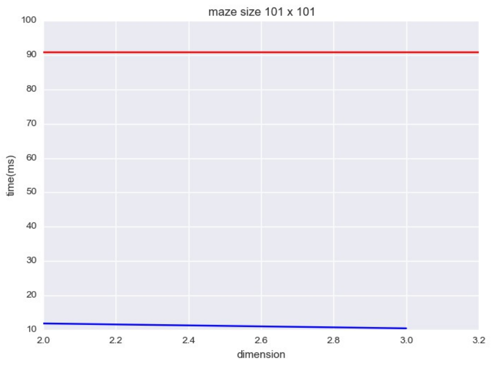




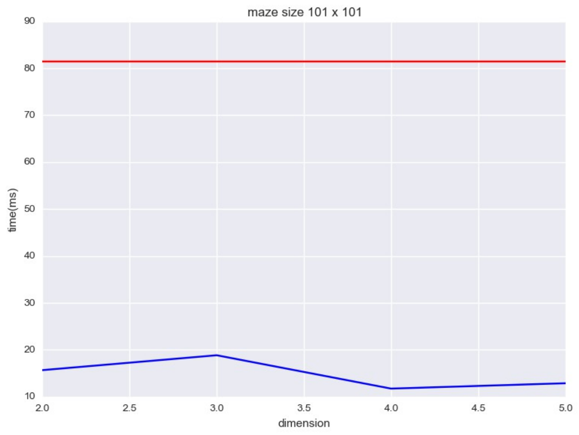
(2) Original FastMap A* search

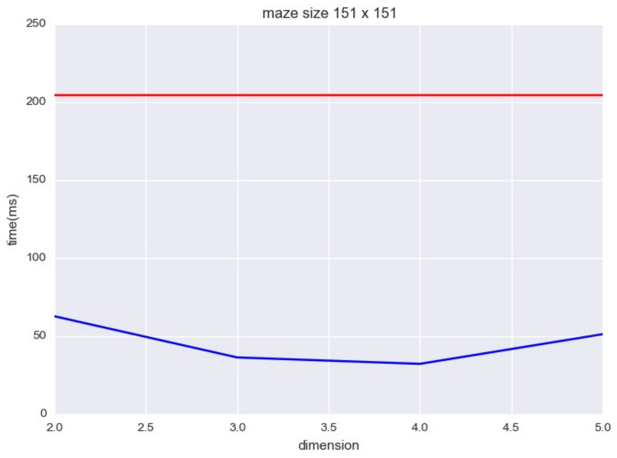


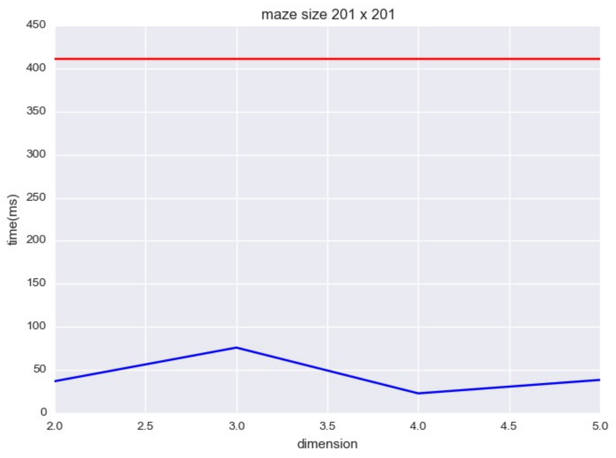


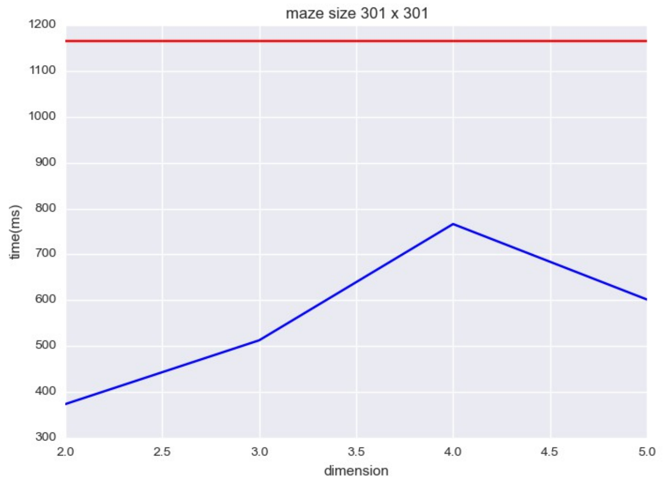


(3) Extend result for Original FastMap A* search









Based on the test result, we found that for a small maze, both Modified FastMap A* and Original FastMap A*perform better than A* with Euclidean distance. Increasing the number of transform dimension from k = 2 to k = 3 can sometimes lower the time it takes for A*, but the result is not conclusive, though. The number of steps taken is also not affected much. The benefit gain between Modified FastMap A* and Original FastMap A* is not conclusive as we need to test them on larger mazes.

For a result of running Original FastMap A* on a larger maze, FastMap A* also perform better than A* with Euclidean distance. Increasing the number of transformed dimension doesn't affect the time it takes and a number of steps in any significant ways.

It would be ideal to run our algorithm on a larger maze that put a stress on A* with Euclidean distance but even Original FastMap A* consume too much time to generate coordinates for a maze with a size larger that 300 x 300.

Further Suggestions

1. In order to generate the coordinate of a cell in each dimension for Modified FastMap A*, we run Dijkstra Algorithm to find the shortest path between that cell and 2 pivot cells. This method is not feasible and becomes a bottle neck as maze's size and the number of dimensions increased(we ran our algorithm on a maze with size 200x200, transformed the coordinates into 3 dimensions and decided to terminate the process after running for an hour). Although Original FastMap A* take advantage of cached BFS result to lower the time, it still required too much time to generate coordinates for a maze with a size higher than in the report. Obviously, it is desirable to devise an algorithm that can efficiently compute the coordinates.

2. Ideally, it would be better to run the algorithm on a bigger maze, transform the maze into various dimension settings and compare against benchmark A* algorithm. Researchers may vary maze's complexity and density to see how the algorithm performs. The same algorithm setting should be run on the same maze multiple times for consistent result.

3. The transformed coordinate for each cell can be computed independently, so we encourage researchers to take advantage of distributed computing, such as Apache Hadoop or Spark, to accelerate the running process.

4. Our maze generation algorithm only generate a maze with an odd size. It may be interesting to generate a maze with arbitrary size and arbitrary shape(not just rectangular) as well.

Conclusions

We found that, for a small maze, by transforming L1 distance in a maze based on FastMap algorithm and used it as a heuristic function, we can lower the time it takes for A* path finding algorithm. Increasing the number of dimensions doesn't provide obvious benefit and required further research. Although our finding is based on small mazes that can be easily solved by normal A*, it does provide a promising venue that researcher can continue to investigate on.

References

[1] Christos Faloutsos and King-Ip (David) Lin.
FastMap: A Fast Algorithm for Indexing, Data-Mining and Visualization of Traditional and Multimedia Datasets.
http://www.cs.bu.edu/faculty/gkollios/ada02/LectNotes/P163.PDF

[2] Laurent Luce
Based source code for A* algorithm:
https://github.com/laurentluce/python-algorithms/blob/master/algorithms/a_star_path_finding.py

[3] Wikipedia
Randomized Maze Generation algorithm:
https://en.wikipedia.org/wiki/Maze_generation_algorithm

[4] http://aidblab.cse.iitm.ac.in/cs625/6.FastMap.pdf

[5] https://en.wikipedia.org/wiki/Multidimensional_scaling

[6] http://www.cs.bu.edu/faculty/gkollios/ada02/LectNotes/P163.PDF