Heuristically Informed Search Methods for Solving Path-finding Problem in Grid Square Environment and Its Optimization

Puja Pramudya

Program Studi Informatika
Sekolah Elektro dan Informatika
Institut Teknologi Bandung
Jl.Ganesha 10, Bandung
e-mail: if16058@students.if.itb.ac.id

ABSTRACT

Path-finding problem is a common problem we find daily. There are some algorithms to solve such problem. One of these is Heuristically Informed Methods, which is a group of algorithm that using an information to help the search. These methods consist of algorithm that having different approach to get the information about the search problem. They are Hill Climbing, Beam Search, Best First Search, and Branch and Bound. Some heuristic will explore different path and determine the result and computation. This paper shows these methods to solve path-finding problem in grid environment, common heuristic in grid environment and a creative way to optimize the search. Experiment show that the optimized algorithm perform better than standard algorithm.

Keywords : path finding, grid, search methods, heuristic.

1. PREFACE

Search methods aren't the perfect solution for every problem, but with creative applications it can solve many. If search is an appropriate solution, then choose the one which is guaranteed to find the solution. Furthermore, pick the most efficient one.

There are various search methods we can use to solve path-finding problem. Path-finding problem need us to search a path or a way from the start point to the goal point through some constraint may exist. This problem has a widen application in the real life, such as routing, shortest path, game and artificial intelligence and so on.

Each method has a different approach to solve the problem. Then, it let us to divide the methods into two kinds below:

1. Blind Search (Basic Search)
2. Heuristically Informed Methods

In this paper, square grid environment is choosen to narrow the scope of exploration to solve path-finding problem. Grids are built from a repetition of simple shapes, take as square.Grids are commonly used in games for representing playing areas such as maps (in games like Civilization and Warcraft), playing surfaces (in games like pool, table tennis, and poker), playing fields (in games like baseball and football), boards (in games like Chess, Monopoly, and Connect Four), and abstract spaces (in games like Tetris).

Square grid is easy to implement and cover most the pathfinding problem in a map. Each node in search method is represent the coordinate of each square altogether with the cost in the heuristically method.

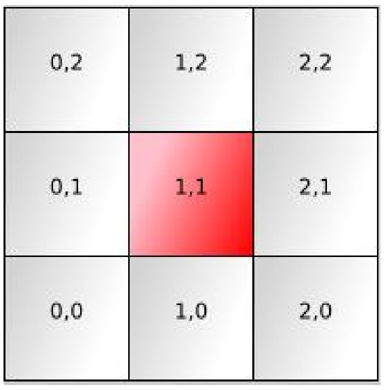


Figure 1 Squre grid representation

This paper will show how each method can help to find the path if one may exist and the different between them. To simplify the representation, just organize the possible solution into tree or graph structure. Searching starts with visiting each node in a tree or graph until all nodes has been looked or the solution is found. Then state S as the start point and G is the goal.

2. BLIND SEARCH

Assume there is no information about the graph / tree / network / road / or something alike being searched. It can't predict how many neighbors each node has until the goal is reached. From starting node, S, can be searched what neighbors it has (say A and B) but still lack information about the number of neighbors A and B have until they are reached.

Even not knowing things like that, DFS (Deep First Search) and BFS (Breadth First Search) are guaranteed to find a path if one exists.

BFS uses a data structure called a *queue*. Add the newly formed paths to the back of the list. When removing them to expand them, remove them from the front. This is the brief algorithm for BFS to solve path-finding problem:

```
Create a queue P
Add the start node S, to P giving it one element
Until first path of P ends with G, or P is empty
     Extract the first path from P
     Extend the path one step to all neighbors creating X new paths
     Reject all paths with blocks
     Add each remaining new path to the BACK of P
If G found -> success. Else -> failure.
```

BFS explores the tree uniformly checks all paths one step away from the start, then two steps, then three, and so on until the goal found or it comes failure.

DFS differs from BFS only in how the new paths are added to the list. In this case, it uses a *stack* rather than a queue. In a stack, new elements are added to the front of the list rather than the back, but when the remove the paths to expand them, still remove them from the front. The result of this is that DFS explores one path, ignoring alternatives, until it either finds the goal or it can't go anywhere else. This is the algorithm for DFS:

```
Create a stack P
Add the start node S, to P giving it one element
Until first path of P ends with G, or P is empty
     Extract the first path from P
     Extend the path one step to all neighbors creating X new paths
     Reject all paths with block
     Push each remaining new path to P
 If G found -> success. Else -> failure.
```

BFS and DFS are guaranteed to find a path to the goal (if one exists) but not necessarily the most efficient one. In this case, DFS gave the path, but one that had an unnecessary detour through A and B.

So if both are guaranteed to reach the destination, pick one that suit the problem. BFS is bad for those trees that have a high branching factor, that mean that each node has a lot of neighbors: use DFS for this. DFS is bad for those trees that have a lot of very long paths: use BFS for this.

3. HEURISTICALLY INFORMED METHODS

DFS and BFS searches are all fine and good if there isn't anything even a little about the tree searching. If it didn't, knowing even a little bit, though, that knowledge can help immensely. For one thing, if there is some clue about branching factor and average distance of the paths, it could be decided whether use BFS or DFS.

If it was provided more than that, for example, distance to goal, it can be used that to greatly improve the efficiency of search method and then it called the heuristically informed method.

3.1 HEURISTICALLY BASIC SEARCH

Take an example is gird below:

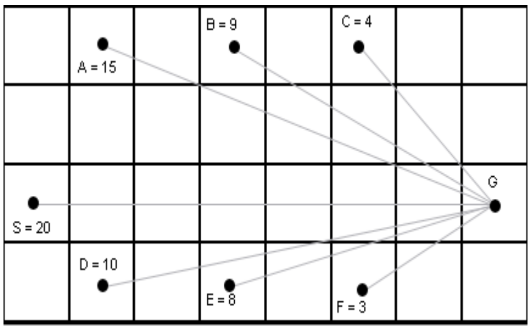


Figure 2 Grid with distance to goal

The gray lines represent distance, but not actual paths. Using these distance measurements and DFS, producing a method called ***Hill Climbing***. The algorithm for Hill Climbing follows:

```
Create a stack P
Add the start node S, to P giving it one element
Until first path of P ends with G, or P is empty
     Extract the first path from P
     Extend the path one step to all neighbors creating X new paths
     If any new paths exist
        Sort them by their distances from the last node to the goal
     Reject all paths with blocks
     Push each remaining new path to the FRONT of P
If G found -> success. Else -> failure.
```

Look the italicized addition. This determines the order nodes are added to the stack as in DFS. This method will simulate the search such below:

Add S (distance of 20) to the stack and enter the main loop. Remove S (20), expand it to S→A (15) and S→D (10). Sort these so the shortest remaining path goes first and add them to the stack. So, stack is now S→D (10) and S→A (15).

Remove the first one, S→D (10), and expand it to S→D→A (15), S→D→B (9) and S→D→E (8) and add them (sorted) to the stack which now has the following: S→D→E (8), S→D→B (9), S→D→A (15) and S→A (15).

Expand S→D→E (8) to S→D→E→F (3) which is still the shortest path so it then gets expanded to

S→D→E→F →G and reached the goal. Note that this takes care of the problem brought up in the DFS discussion. This does not mean that Hill Climbing solves all the problems of DFS, it just happened to find a more efficient path in this one example. In its worst-case scenario, Hill Climbing behaves as DFS.

While the Hill Climbing Method improves the efficiency of DFS, BFS has a potential improvement as well, called ***Beam Search***. Beam Search artificially limits the branching factor of the tree to some arbitrary value (for example, 2). This value is denoted W for width of the beam. The algorithm for the problem follows:

```
Create a queue P
Add the start node S, to P giving it one element
Until first path of P ends with G, or P is empty
      Extract the first path from P
      Extend ALL PATHS one step to all neighbors
creating X new paths
      Reject all paths with blocks
      Sort all paths by estimated distance to
goal
      Discard all but closest W paths
      Push each remaining new path to the BACK of
P
If G found -> success. Else -> failure.
```

The effect of this, it limits the number of neighbors that must be explored to only those that are closest to the goal. The estimated remaining distance is, in most cases the straight-line distance. However, because the beam search discards potential paths which it never examines again, it may be possible to discard paths which prove more efficient later on or, in some worse cases, discard the only paths to the goal

Hill Climbing and Beam search both have inherent problems and unless special care is taken (and sometimes its not practice to monitor the search and make sure its working correctly) they may not find a path, even if one exists. So, there must be a way to use heuristic knowledge in some way to guarantee a path will be found. The solution to this is ***Best First Search***. This is the algorithm for the current problem:

```
Create a list P
Add the start node S, to P giving it one element
Until first path of P ends with G, or P is empty
     Extract the first path from P
     Extend ALL PATHS one step to all neighbors
creating X new paths
     Reject all paths with blocks
     Add each remaining new path to P
     Sort entire list P by estimated distance to
goal
If G found -> success. Else -> failure.
```

This is guaranteed to find the path to the goal if any path exists and is likely (though not guaranteed) to do so efficiently. It may follow some unnecessary twists and turns but is still more efficient than BFS or DFS in most cases. In its worst-case scenario, however, it behaves just like BFS.

3.2 OPTIMAL SEARCH

Blind searches will find ANY path. Heuristic basic searches will usually find ANY path, but will do so faster usually than blind search. Sometimes it's fine to find just ANY path to the goal as long as reached there. But sometimes the problem is to find the BEST path to the goal. The fastest, cheapest, or easiest route to take is often times more important than just finding SOME path. That’s where optimal search comes in. The methods that follow are intended to find the optimal path, and path-finding problem now became the shortest path problem.

The first method is an exhaustive search. This method is guaranteed to find the best path, but is often quite inefficient. The method is simple and, at first glance, logical: explore every possible path and return the shortest one. One way to do this is to do BFS or DFS, but don’t stop when the goal is reached. Continue until EVERY node has been visited. During this, though, keep track of the distances traveled on each path and return the shortest one. This is practical for only small problems as this can get quite computationally expensive very fast,and it isn’t suit our scope because it doesn’t use any heuristic to help find the goal.

However, this is not much different than blind searches, so add a bit of heuristic tuning to improve efficiency as been done before. By using it, always expand shortest paths first (as in Best-First Search) and stop exploring certain paths if it hasn’t reached the goal yet but is still longer than an existing complete path. The end result of this is called Branch and Bound search.

3.2.1 BRANCH AND BOUND SEARCH

Branch and Bound (B&B) is implemented with BFS scheme. To speed up the search, every node is given a cost. Expanding process isn’t based on the sequenced expand but the node which has the lowest cost among the live node. The cost for node *i* give the estimated path cost from node *i* to goal node.

Otherwise, this function is the lower bound for the cost search for path-finding problem. This function is used to limit the expanding node which is not led to the goal node. For actual implementation, determine the bound function is hard and difficult to use exactly. Then, in practice we use an estimated, often called heuristic function. The function is shown below :

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

which are :
$f(x)$: total cost for node x
$g(x)$: cost to reach node x from start

h(x) : cost to reach goal node from node x

The heuristic function will influence B&B in running-time, to choose the next expanding nodes. Each function will act differently. Branch and Bound algorithm for path-finding problem can be shown below:

```
Create a list P
Add the start node S, to P giving it one element
Until first path of P ends with G, or P is empty
     Extract the first path from P
     Extend first path one step to all neighbors
creating X new paths
     Reject all paths with blocks
     Add each remaining new path to of P
     Sort all paths by total distance travelled,
shortest first.
If G found -> success. Else -> failure.
```

Therefore, the search can be stop expanding paths if the path's total underestimate is of greater distance than that of a complete path already found.

3.2.2 HEURISTIC FOR GRID MAPS

On a grid map, there are well-known heuristic functions to use. Some are explained below :

Manhattan Distance

The standard heuristic is the Manhattan distance. Compute total number of squares moved horizontally and vertically to reach the target square from the current square, ignoring diagonal movement, and ignoring any obstacles that may be in the way :

h(n) = (abs(n.x-goal.x) + abs(n.y-goal.y))

Diagonal Distance

If the map allow diagonal movement then the algorihtm need a different heuristic. Here is the function :

h_diagonal(n) = min(abs(n.x-goal.x), abs(n.y-goal.y))
h_straight(n) = (abs(n.x-goal.x) + abs(n.y-goal.y))

h(n) = h_diagonal(n) + (h_straight(n) - 2*h_diagonal(n))

Here compute h_diagonal(n) = the number of steps can take along a diagonal, h_straight(n) = the Manhattan distance, and then combine the two by considering all diagonal steps to cost D2, and then all remaining straight steps (note that this is the number of straight steps in the Manhattan distance, minus two straight steps for each diagonal step we took instead.

Euclidan Distance

This is equal to straight–line distance between two point. Take coordinate of each square and compute them just as a point. However, if this is the case, it may have trouble with using B&B directly because the cost function `g` will not match the heuristic function `h`. Since Euclidean distance is shorter than Manhattan or diagonal distance, it will still give shortest paths, but B&B will take longer to run

h(n) = sqrt((n.x-goal.x)^2 + (n.y-goal.y)^2)

Euclidan Distance-Squared

Some web pages recommend that avoid the expensive square root in the Euclidean distance by just using distance-squared:

h(n) = ((n.x-goal.x)^2 + (n.y-goal.y)^2)

This definitely runs into the scale problem. When B&B computes f(n) = g(n) + h(n), the square of distance will be much higher than the cost g and it will end up with an overestimating heuristic. For longer distances, this will approach the extreme of g(n) not really counting anymore, and B&B will degrade into BFS.

4. OPTIMIZING FOR HEURISTICALLY INFORMED SEARCH

4.1 DYNAMIC PROGRAMMING

There are many ways to improve efficiency, however, and that is to avoid doing the same work twice. Do this using Dynamic Programming. Dynamic Programming is a method to solve a problem with divide it to set of steps and stage in order to get solution from a sequence of binding decision. It has properties such as :

- Some possible solution
- Each solution is made from the latest stage solution
- Choose a function to bound the choice of solution

See graph below :

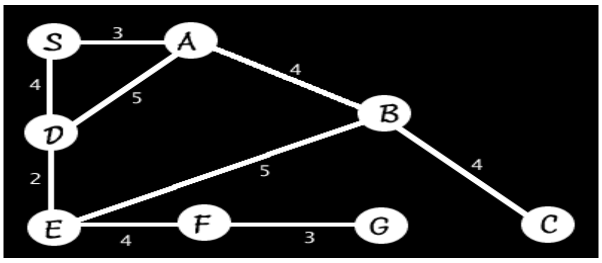


Figure 3 Graph simulation using DP

If it is implemented a Branch and Bound with Dynamic Programming, it save some steps.

Here is the show. As usual, expand S to S→A and S→D. These have partial paths of 3 and 4 respectively.

S→A (3) expands to S→A→B (7) and S→A→D (8) It is already had a path go to D, though with the path S→D

(4). Since it has a shorter path to D, ignore the longer path and discard S→A→D (8).

S-→D (4) expands to S→D→A (9) and S→D-→E (6) Again, S→D→A (9) is a longer path to A than simply S→A (3) so discard it.

S→D→E (6) expands to S→D→E→B (11) and S→D→E→F (10). Discard S→D→E→B (11) because already have a shorter path to B and continue.

S→A→B (7) expands to S→A→B→C (11) and S→A→B→E (12).Since E is reached with half that cost, discard this longer path.

S→D→E→F (10) expands to S→D→E→F→G (13) and it is reached the goal, again with the shortest path.

Here is the implementation of the combination:

```
Create a list P
Add the start node S, to P giving it one element
Until first path of P ends with G, or P is empty
    Extract the first path from P
    Extend first path one step to all neighbors
creating X new paths
    Reject all paths with blocks for all paths
that end at the same node, keep only the
shortest one.
    Add each remaining new path to of P
    Sort all paths by total distance travelled,
shortest first.
If G found -> success. Else -> failure.
```

Now, the dynamic programming saved some steps from the first examples. Combining Branch and Bound with dynamic programming and underestimates yields the favorite A* path-finding algorithm.

4.2 BREAKING TIES FUNCTION

In some maps there are many paths with the same length. For example, in flat areas without variation in terrain, using a grid will lead to many equal-length paths. B&B might explore all the paths with the same f value, instead of just one.

To solve this problem, either adjust the g or h values; it is usually easier to adjust h. The tie breaker needs to be deterministic with respect to the vertex (*i.e.,* it shouldn't just be a random number), and it needs to make the f values differ. Since B&B sorts by f value, making them different means only one of the "equivalent" f values will be explored.

One way to break ties is to nudge the scale of h slightly. If scale it downwards, then f will increase as it moves towards the goal. Unfortunately, this means that B&B will prefer to expand vertices close to the starting point instead of vertices close to the goal. We can instead scale h upwards slightly (even by 0.1%). B&B will prefer to expand vertices close to the goal.

$$heuristic = (1.0 + p)$$

The factor p should be chosen so that $p <$ *(minimum cost of taking one step) / (expected maximum path length)*. Assuming that program doesn't expect the paths to be more than 1000 steps long, choose p = 1/1000.

A different way to break ties is to prefer paths that are along the straight line from the starting point to the goal:

```
dx1 = current.x - goal.x
dy1 = current.y - goal.y
dx2 = start.x - goal.x
dy2 = start.y - goal.y
cross = abs(dx1*dy2 - dx2*dy1)
heuristic += cross*0.001
```

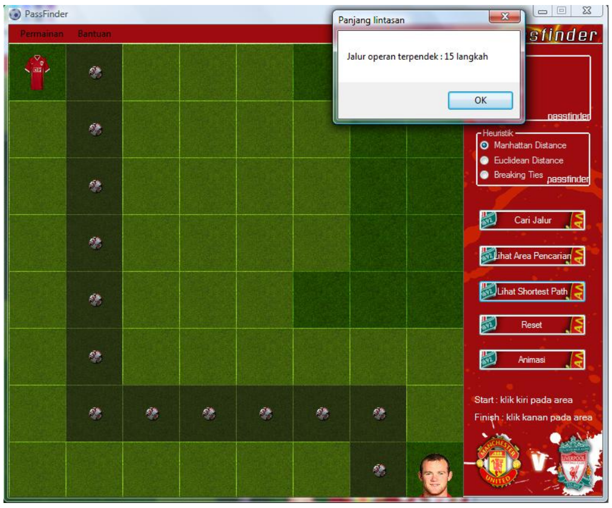
This code computes the vector cross-product between the start to goal vector and the current point to goal vector. When these vectors don't line up, the cross product will be larger. The result is that this code will give some slight preference to a path that lies along the straight line path from the start to the goal.

Yet another way to break ties is to carefully construct B&B priority queue so that *new* insertions with a specific `f` value are always ranked better (lower) than *old* insertions with the same `f` value.

5. COMPARISON SOME HEURISTIC

To compare how heuristic influence B&B program, i've add some improvement to my latest task about shortest path, which i added a breaking ties function, beside Manhattan Distance and Euclidean Distance in 8x8 grid map.

Here is the illustration of using Manhattan Distance.



Jalur operan terpendek : 15 langkah

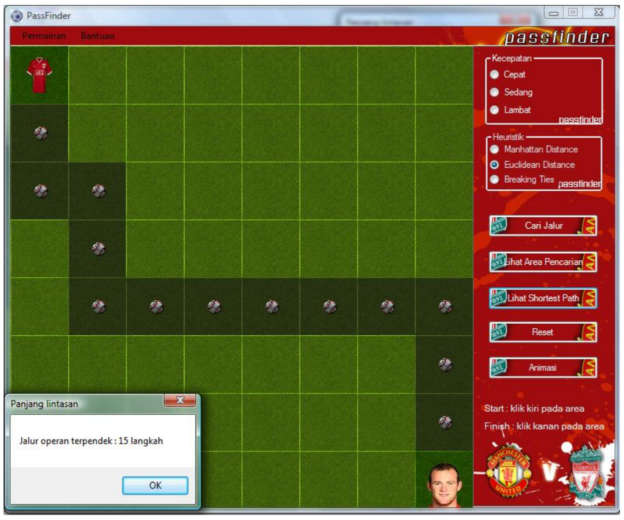
Heuristik
[✓ Manhattan Distance]
Euclidean Distance
Breaking Ties

Start : klik kiri pada area
Finish : klik kanan pada area

Figure 4 Manhattan distance heuristic

B&B explored almost three quarters of the grid area,and come with the best path : 15 grid during 65 milliseconds.

Otherwise, with Euclidean Distance, B&B is still find the best path : 15 grid, yet it also explores all grids in the map during 2 seconds and 67 milliseconds. It proved this heuristic will run much worse than Manhattan Distance.



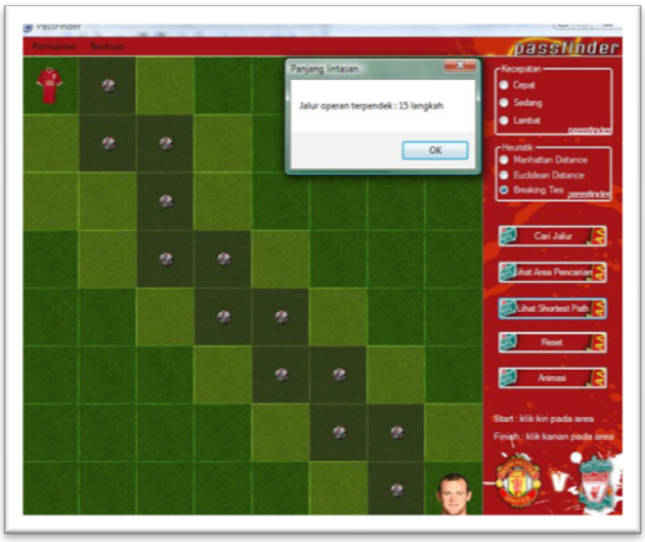
Panjang lintasan

Jalur operan terpendek : 15 langkah

Heuristik: Euclidean Distance

Figure 5 Euclidean distance heuristic

Then, when using Breaking Ties function in the heuristic, B&B is run quicker than Manhattan and Euclidean Distance in 39 milliseconds.



Panjang lintasan

Jalur operan terpendek : 15 langkah

Heuristik: Breaking Ties

Figure 6 Breaking ties optimization

It is not only explored the less grids ,the path look nice as very well.

6. CONCLUSION

After all explanation above, there are some point to notes, either a conclusion :

1. There are many search methods to solve path-finding problem in grid environment, one of them is Heuristically Informed Methods.
2. Heuristically Informed Methods need piece of information about the area being searched.
3. Optimal Search, which is the subset of Heuristic Methods, use information about the area, and take them as a heuristic for helping find the path.
4. Branch and Bound is an example of Optimal Search methods using a heuristic.
5. There are many heuristics for the grid-based area environtment and each heuristic influence how algorithm works in finding the path.
6. Optimizing the search can be done with using various ways, such as :
   1. Using dynamic programming
   2. Add breaking ties function in the heuristic
   3. Construct priority queue carefully

REFERENCES

[1] Russell, S. J., Norvig, P, Artificial Intelligence: A Modern Approach, 2003.

[2] Munir, Rinaldi, Diktat Kuliah IF2251 Strategi Algoritmik, Penerbit ITB, 2007.

[3] Museum search, http://bradley.bradley.edu/~chris/searches. html access date May, 18 2008 at 7.20 am.

[4] Beam Search, http://en.wikipedia.org/wiki/Beam_search. html access date May, 17 2008 at 3.20 am.

[5] Hill Climbing Search, http://en.wikipedia.org/wiki/ Hill_climbing. html access date May, 17 2008 at 3.05 am.

[6] Best First Search, http://en.wikipedia.org/wiki/Best-first _search. html access date May, 17 2008 at 1.56 am.

[7] Breadth First Search, http://en.wikipedia.org/wiki/Breadth-first _search. html access date May, 17 2008 at 1.35 am.

[8] Depth First Search, http://en.wikipedia.org/wiki/Depth-first _search. html access date May, 17 2008 at 1.56 am.