International Journal of Advanced Science and Technology
Volume 9, August, 2009

# Simulation of Path Finding Algorithm – a Bird's Eye Perspective

R.Anbuselvi[1] and R.S.Bhuvaneswaran[2]

*[1]Lecturer in Computer Science,*
*Bishop Heber College, Trichy-17.*
*[2]Assist.Professor, Annauniversity, Chennai.*
*[1]r.anbuselvi10@yahoo.co.in, [2]bhuvan@annauniv.edu*

***Abstract***

*Path finding is one of those required elements of the computer network. Any path finding algorithm will work as long as there are no obstacles or distractions along the way. If there is an obstacle, then the character needs to figure out a way to move around and still reach the goal.*

*In this paper, we have simulated the existing path-finding algorithms and state that A* is the most popular choice for path-finding and it is very flexible and can be used in a wide range of contexts.*

***Keywords:*** *Simulation, Path finding, Bird's eye.*

## 1. Introduction

Some path-finding algorithms are implemented with a generic A* search. The search algorithm allows finding the best path no matter where we are.

Also, a good path-finding algorithm is essential to artificial intelligence, so we will use what we create in this chapter Path-Finding Basics Path finding can be reduced to answering the question, "How do I get from point A to point B?" Generally, the path from an object (or, in this case, a bot) to another location could potentially have several different solutions, but ideally, we want a solution that solves these goals:

1. How to get from A to B
2. How to get around obstacles in the way
3. How to find the shortest possible path
4. How to find the path quickly

Some path-finding algorithms solve none of these problems, and some solve all of them. And in some cases, no algorithm could solve any of them—for example, point A could be on an island completely isolated from point B, so no solution would exist.[1], [3], [5]

## 2. Simulation of path finding algorithm

If the environment is flat with no obstacles, path finding is no problem: The bot can just move in a straight line to its destination. The bot is the white circle. The player is the black circle. But when obstacles occur between point A and point B, things get a little hairy. For example, in figure 1 if the bot just moves straight toward the player, it will end up just hitting a wall and getting stuck ther

Not the best solution: The bot just runs straight toward the player, even if there are obstacles in the way. This behavior might be fine for some games because it does not really matter what the bots do; it matters only what they appear to do. In a 3D world from the

player's perspective, it might look like the bot is just waiting behind the wall for the player instead of being stuck.

A bird's-eye perspective is a different story. Path finding is virtually required for games with bird's-eye perspectives, such as real-time strategy games or for games like The Sims. We would not want to tell a team of fighters to attack something but have the team not be able to do it because the destination was on the other side of the wall [2], [4]

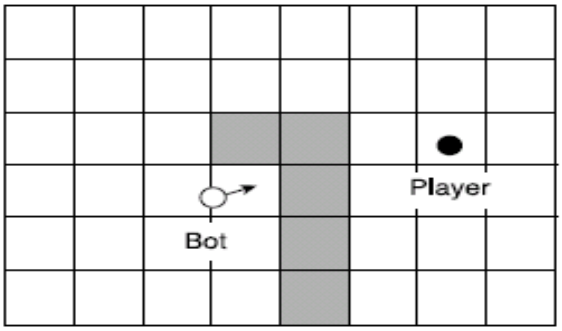


Figure 1. Not the best solution: The bot (white circle) just runs straight toward the player (black circle), even if there are obstacles in the way

Come on now, use the door.

Also, making a bot smart enough to find a path around an obstacle makes it more interesting for the player and harder to "trick" the bot into doing something or moving a certain way. Several techniques can be used to get around obstacles. A randomized algorithm finds a solution eventually, if there is one. This technique makes the bot walk around randomly, like in figure 2 until it finds what it is looking for. The bot moves around randomly until the player is found.

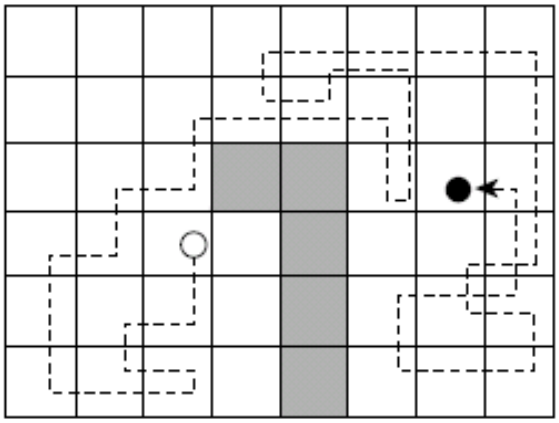

Figure 2. The bot (white circle) moves around randomly until the player (Black circle) is found.

Of course, the randomized algorithm doesn't give the best-looking results or the shortest path. Who wants to watch a bot move around randomly like it's missing a circuit or two?

This random algorithm could be modified to move randomly only occasionally and at other times move directly toward the player, which would make the bot's movement appear slightly more intelligent.

Another solution can be found when the environment is a simple maze with no loops, like in Figure 3 Here; the path-finding algorithm is the "right hand on the wall" algorithm.
Just keep our right hand on the wall as we move through the maze, and eventually we will find the destination.

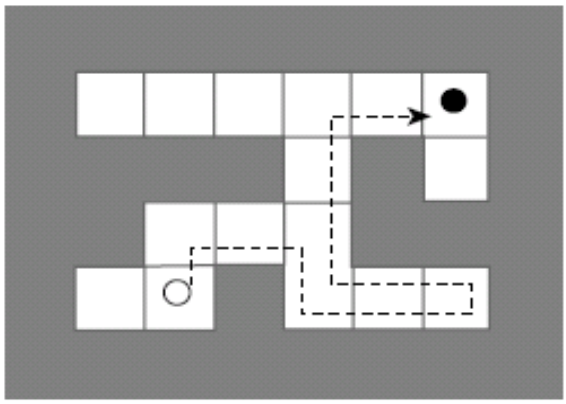

Figure 3. The bot moves in a maze, keeping its right hand on the wall until it finds the player

The bot moves in a maze, keeping its right hand on the wall until it finds the player If a bot follows the "right hand" algorithm literally, it will rarely find the shortest path from A to B. Alternatively, the path could be calculated ahead of time, before the bot makes any moves. Then any backtracking could be removed from the path, leaving a shortest-path solution.

However, this algorithm works only for the simple case of a maze with no loops. If a loop was in the path, the bot would veer to the right forever. Also, if rooms had wide spaces, the bot might never find its destination in the middle of a room because the bot is always next to a wall. Ideally, we will probably want to create more intricate environments than this.
Instead of looking at the environment as a grid, we will look at it as a graph, like in Figure 4 A graph is simply a bunch of nodes grouped by various edges.

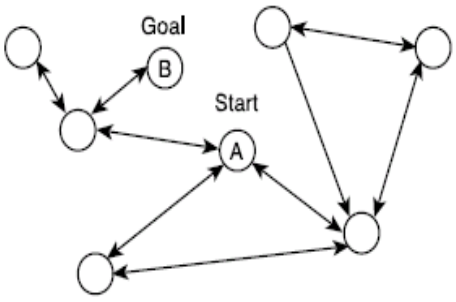


Figure 4. A simple graph

Each node of the graph could be anything. For example, a node could be a cell in a 2D grid. Or, a node could be a "city," with the edges of the graph representing highways. And remember, when finding the path from A to B, any node can be the start or goal node.
A graph is similar to a tree, "3D Scene Management Using BSP Trees," except that, instead of each node having up to two children, each node can have an indefinite number of children, or neighbors. Some graphs are directed, meaning that an edge between two nodes can be traversed in only one direction. Looking at the example in Figure 4 all the edges are bidirectional except for one. A unidirectional edge could be useful in such situations as when traveling involves jumping down a cliff that is too high to jump back up.

In this example, we want to find the shortest path from node A to node B, or the fewest number of traversed edges to get to the goal. Looking at the figure, the solution is easy to determine, but how would we find the solution in a computer program? An easy solution is to use a breadth-first search [1], [3], [5]

Movement for a single object seems easy. Pathfinding is complex. Why bother with pathfinding? Consider the following situation:

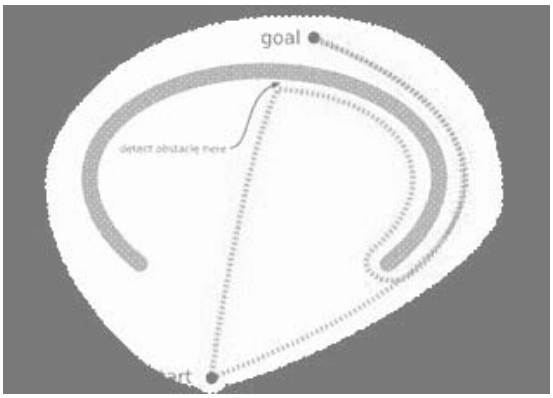


Figure 5. Unit at the bottom of the map

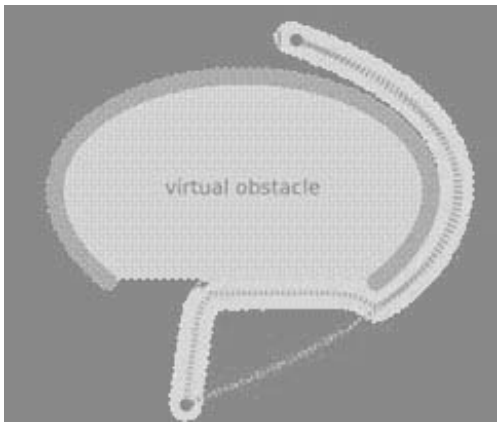


Figure 6. Unit at the top of the map

The unit is initially at the bottom of the map and wants to get to the top. There is nothing in the area it scans (shown in pink) to indicate that the unit should not move up, so it

continues on its way. Near the top, it detects an obstacle and changes direction. It then finds its way around the "U"-shaped obstacle, following the red path. In contrast, a pathfinder would have scanned a larger area (shown in light blue), but found a shorter path (blue), never sending the unit into the concave shaped obstacle.

We can however extend a movement algorithm to work around traps like the one shown above. Either avoid creating concave obstacles, or mark their convex hulls as dangerous (to be entered only if the goal is inside):

Pathfinders let us look ahead and make plans rather than waiting until the last moment to discover there's a problem. Movement without path finding works in many situations, and can be extended to work in more situations, but path finding is a more general tool that can be used to solve a wider variety of problems.

## 3. Experiment and Analysis

Like traversing BSP trees, a breadth-first search involves visiting nodes one at a time. A breadth-first search visits nodes in the order of their distance from the start node, where distance is measured as the number of traversed edges.

So, with a breadth-first search, first all nodes one edge away from the goal are visited, then those two edges away are visited, and so on until all nodes are visited. This way, we find the path from the start to the goal with the minimum number of traversed edges.

Another way to word it is like this: Visit the neighbor nodes, then the neighbor's neighbor nodes, and so on until the goal node is found. An example of a breadth-first search is in Figure 7 in which the nodes are numbered in the order they are visited.

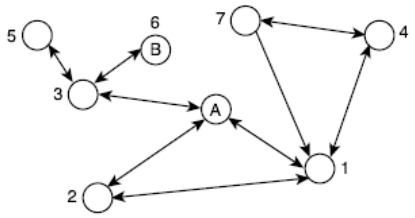


Figure 7 An example breadth-first search. The nodes are numbered in the order of the search.

The nodes are numbered in the order of the search. Later in this chapter, we use the A* search algorithm, which has several similarities to a breadth-first search. But first, to better understand A*, we'll implement the easy to- understand breadth-first search algorithm.

The neighbors list is simply a list of all the nodes' neighbors. The path Parent node is used for searching only. Think of the path from the start node to the goal node as a tree, with each node having only one child. The path Parent node is the node's parent in the path tree. When the goal is found, the path can be found by traversing up the path tree from the goal node to the start node.

For example, if A has neighbors B and C, and B has neighbor A, we do not want to visit A again or we will end up in an infinite loop. So, we will keep track of all nodes that have been visited by putting them in a "closed" list. If a node shows up in the search that's already in the closed list, we will ignore it.

Likewise, we will keep track of all the nodes we want to visit in an "open" list. The open list is a first in, first out list, effectively sorting the list from smallest number of edges from the start goal to the largest.

This function returns a list of nodes that represent the path, not including the start node. If a path cannot be found, it returns null. That's all there is for breadth-first search. However, taking a step back, it is easy to notice one problem with this search: we found the path with the least number of edges, but edges could have different "costs" associated with them.

For example, the cost of one edge could be 10km, while the cost of another could be 100km. obviously, traversing two 10km edges would be faster than traversing a single 100km edge. Breadth-first search assumes all edges have the same cost, which isn't good enough. This is where the A* algorithm comes in.

Basics of the A* Algorithm An A*—pronounced "A-star"—search works like a breadth-first search, except with two extra factors:
The edges have different "costs," which is how much it costs to travel from one node to another.

The cost from any node to the goal node can be estimated. This helps refine the search so that we're less likely to go off searching in the wrong direction. The cost between nodes does not have to be distance. The cost could be time, if we wanted to find the path that takes the shortest amount of time to traverse. [2], [4]

For example, when we are driving, taking the back roads might be a shorter distance, but taking the freeway usually takes less time. Another example is terrain: Traveling through overgrown forests could take longer than traveling through a grassy area.
Or, we could get more creative with the cost. For instance, if we want a bot to sneak up on the player, having the bot appear in front of the player could have a high cost, but appearing from behind could have little or no cost. We could take this idea further and assign a special tactical advantage to certain nodes—such as getting behind a crate—which would have a smaller cost than just appearing in front of the player.

Some of the following materials reference the program I wrote in C++ and Blitz Basic but the points are equally valid in other languages.

## 4. Related works

There are lots of other path finding algorithms, but those other methods are not A*, which is generally considered to be the best of the lot. Eric Marchesin,[5] discusses many of them in the article referenced at the end of this article, including some of their pros and cons. Sometimes alternatives are better under certain circumstances, but we should understand what we are getting into.

## 5. Conclusion

For convenience and performance, we started with some basic path-finding routines on a tile-based map, such as a randomized search and the popular "right hand on the wall" trick. Then we went to more advanced environments based on graphs and implemented a generic A* algorithm that could be used for graph – or tile-based environments.
Finally, we merged A* searching with the BSP tree we've been working with for the past couple of chapters, and we created a generic path that follows any type of path, whether it was created by an A* search or not.

## References

[1] Craig Reynold's, (1999) "Sterring Behavior for Autonomous Characters:" 1st Feb 2000 [Reyn87] Gamasutra, cited by 68 – Related articles – all 12 versions.

[2] Dave Pottinger, "Coordinated Unit Movement:" 22nd Jan 1999. Gamasutra Vol.3: Issue 3. First in a two-part series of articles on formation and group-based movement by Age of Empires designer.

[3] Dave Pottinger's "Implementing Coordinated Movement:" 29th Jan 1999. Game Developer PP 48-58. Second in two-part series.
[4] Marco Pinter, "More Realistic Part Finding Article:" 14th Mar 2001. Gamasutra.
[5] Eric Marchesin, "A simple c# Genetic alogorithm Article:" 22nd June 2003, .NET 1.0, 4.72
[6] http://www.gamasutra.com

## Authors

R.Anbuselvi received M.Sc in computer science from Bharathidasan University, India in 1992. She received M.Phil from Mother Teresa women's University kodaikanal in 2001 .Her research include Artificial Intelligence. She is working as a lecturer in Bishop heber 's college, Trichy.

R.S. Bhuvaneswaran received Bachelor of Science in Mathematics with Gold Medal in 1987 from Madras University, Master of Technology in Computer Science and Engineering from Pondicherry University, in 1996 and Ph.D in Computer Science and Engineering from Anna University in2003. He is a Post Doctoral Fellow of JSPS, Japan (2004-2006).Presently, he is with Anna University as Assistant Professor. His research interests include design of algorithms, distributed systems, wireless networks and fault tolerant systems.