International Journal of Engineering Research-Online
A Peer Reviewed International Journal
Email:editorijoer@gmail.com http://www.ijoer.in

Vol.4., Issue.2., 2016
(Mar-Apr)

RESEARCH ARTICLE

ISSN: 2321-7758

HARDWARE IMPLEMENTATION OF A* ALGORITHM SHORTEST PATH ALGORITHM

E. BHARAT BABU[1], V.MANIDEEP[2],T .LAKSHMI MANOGNA[2],V.GEETHASRI[2],HARI KRISHNA[2]

[1]Assistant professor [2]Undergraduate students
Department of ECE
Padmasri Dr B.V. Raju Institute of Technology, Medak (Dist), Telangana, India

ABSTRACT

This electronic document presents the implementation of A* Algorithm in a known environment. We will keep watch over the environment by regularly walking or travelling around it.A* Algorithm can be implemented only in known environment. A* algorithm is useful for an environment that can be represented as a graph with the presence of nodes that act as the decision making centers. It solves problems by searching among all possible paths to the solution (goal) for the one that incurs the smallest cost (least distance travelled, shortest time, etc.), and among these paths it first considers the ones that appear to lead most quickly to the solution. It is formulated in terms of weighted graphs: starting from a specific node of a graph, it constructs a tree of paths starting from that node, expanding paths one step at a time, until one of its paths ends at the predetermined goal node. We can implement A* Algorithm by using any programming language . But we consider the case of implementing A* algorithm by using FPGA technology in Verilog code. Here we use FPGA as it has unique feature like parallel processing and it is reprogrammable. In this regard, we are proposing VLSI efficient scheme for the A* algorithm with FPGA implementation. It is simulated using Xilinx ISE14.7 software and implemented on Spartan 6 FPGA board.

Keywords—A* Algorithm, Nodes, path planning

INTRODUCTION

Basically , A* search algorithm is an algorithm for finding an item with specified properties among a collection of items which are coded into a computer program, that look for clues to return what is wanted. The items may be stored individually as records in a database; or may be elements of a search space defined by a mathematical formula or procedure. A graph is connected if there is a path between every pair of vertices. A connected component of a graph G is a maximal connected sub graph of G.A tree is an undirected graph T such that T is connected. T has no cycles. This definition of tree is different from the one of a rooted tree. A forest is an undirected graph without cycles. The connected components of a forest are trees.

A* is a algorithm that is widely used in path finding and graph traversal, the process of plotting an efficiently traversable path between multiple points, called nodes. Noted for its performance and accuracy, it enjoys widespread use. However, in practical travel-routing systems, it is generally outperformed by algorithms which can pre-process the graph to attain better performance, although other work has found A* to be superior to other approaches.

Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute first described the algorithm in 1968. It is an extension of Edsger Dijkstra's 1959 algorithm. A* achieves better performance by using heuristics to guide its search. Hewas trying to improve the path planning done by Shakey the Robot, a prototype robot that could navigate through a room containing obstacles. This path-finding algorithm, that Nilsson called A1, was a faster version of the then best known method, Dijkstra's algorithm, for finding shortest paths in graphs. Bertram Raphael suggested some significant improvements upon this algorithm, calling the revised version A2. Then Peter E. Hart introduced an argument that established A2, with only minor changes, to be the best possible algorithm for finding shortest paths. Hart, Nilsson and Raphael then jointly developed a proof that the revised A 2 algorithm was optimal for finding shortest paths under certain well-defined conditions.

Main Idea

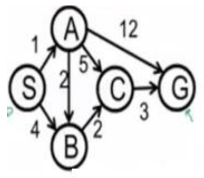
The main idea of this paper is to develop a FPGA based robot that moves along a path that has been decided. A* is an algorithm that is used to find a shortest path. A* is calculated using heuristic approach which includes cost function. The cost function is given by

F(n)=g(n)+h(n)

Where g(n) determines the distance between goal and the current position h(n) indicates the distance between starting point and current point. Typical implementations of A* use a priority queue to perform the repeated selection of minimum (estimated) cost nodes to expand. This priority queue is known as the open setor fringe. At each step of the algorithm, the node with the lowest f(x) value is removed from the queue, the f and g values of its neighbors are updated accordingly, and these neighbors are added to the queue. The algorithm continues until a goal node has a lower f value than any node in the queue (or until the queue is empty). The f value of the goal is then the length of the shortest path, since h at the goal is zero in an admissible heuristic.

The algorithm described so far gives us only the length of the shortest path. To find the actual sequence of steps, the algorithm can be easily revised so that each node on the path keeps track of its predecessor. After this algorithm is run, the ending node will point to its predecessor, and so on, until some node's predecessor is the start node.

Environment and algorithm



A-Star requires a distance function from start configuration to goal configuration. It does not need an exact distance, just an estimate of how long it would take to get from start configuration to the goal configuration, in the best case. The A-Star algorithm combines features of uniform-cost search and pure heuristic search to efficiently compute optimal solutions. A-star uses the distance between the current configuration and the target configuration and moves to the node that has the smallest distance.

Implementation

There are a number of simple optimizations or implementation details that can significantly affect the performance of an A* implementation. The first detail to note is that the way the priority queue handles ties can have a significant effect on performance in some situations. If ties are broken so the queue behaves in a LIFO manner, A* will behave like depth-first search among equal cost paths (avoiding exploring more than one equally optimal solution).

When a path is required at the end of the search, it is common to keep with each node a reference to that node's parent. At the end of the search these references can be used to recover the optimal path. If these references are being kept then it can be important that the same node doesn't appear in the priority queue more than once (each entry corresponding to a different path to the node, and each with a different cost). A standard approach here is to check if a node about to be added already appears in the priority queue. If it does, then the priority and parent pointers are changed to correspond to the lower cost path. A standard binary heap based priority queue does not directly support the operation of searching for one of its elements, but it can be augmented with a hash table that maps elements to their position in the heap, allowing this decrease-priority operation to be performed in logarithmic time. Alternatively, a Fibonacci heap can perform the same decrease-priority operations in constant amortized time.

For the first iteration, the current intersection will be the starting point, and the distance to it (the intersection's label) will be zero. For subsequent iterations (after the first), the current intersection will be the *closest* unvisited intersection to the starting point (this will be easy to find).

From the current intersection, update the distance to every unvisited intersection that is directly connected to it. This is done by determining the sum of the distance between an unvisited intersection and the value of the current intersection, and relabeling the unvisited intersection with this value (the sum), if it is less than its current value. In effect, the intersection is relabeled if the path to it through the current intersection is shorter than the previously known paths.

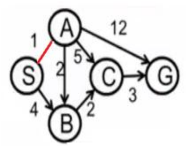
To facilitate shortest path identification, in pencil, mark the road with an arrow pointing to the relabeled intersection if you label/relabel it, and erase all others pointing to it. After you have updated the distances to each neighboring interaction, mark the current intersection as visited, and select the unvisited intersection with lowest distance (from the starting point) – or the lowest label—as the current intersection. Nodes marked as visited are labeled with the shortest path from the starting point to it and will not be revisited or returned to.

Algorithm

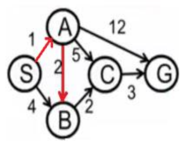
1) Put the start node on the list OPEN and calculate the cost function f (n). {h (n) = 0; g(n) = distance between the goal and the start position, f(n) = g(n).}
2) Remove from the List OPEN the node with the smallest cost function and put it on CLOSED. This is the node n. (Incase two or more nodes have the cost function, arbitrarily resolve ties. If one of the nodes is the goal node, then select the goal node
3) If n is the goal node then terminate the algorithm and use the pointers to obtain the solution path. Otherwise, continue
4) Determine all the successor nodes of n and compute the cost function for each successor not on list CLOSED.
5) Associate with each successor not on list OPEN or CLOSED the cost calculated and put these on the list OPEN, placing pointers to n (n is the parent node).
6) Associate with any successors already on OPEN the smaller of the cost values just calculated and the previous cost value. (min(new f(n), old f(n)))
7) Go to step 2.

Let us assume the h values of S=7,A=6,B=2,C=1,G=0. By taking into consideration these values we may find the path as shown in the following figures below

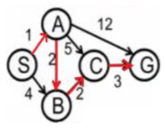
Initially the path from every node is taken as zero except the source and goal nodes .The cost from is determined as the distance between each path i.e, from source to destination. The cost from S TO Similarly the cost from AS =1,AB=2,SB=4,BC=2,AC=5,CG=3,AG=12.In this environment the source is taken as S and the destination position is G.



From the above diagram the three costs are compared i.e, SA, SB. By comparing these three costs the shortest path is taken into consideration.SA=1 is the shortest path. The shortest path of SA is taken into consideration.

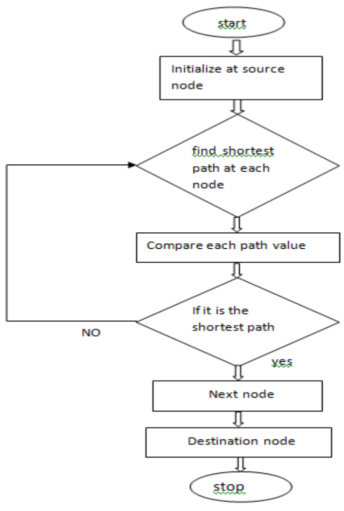


The goal is to find the shortest path from S to G. The ways to travel from S to G is through SABCG. This shortest path is calculated by comparing each and every node it visits to the destination node. The nodes which are calculated or visited are stored in a closed list and the nodes which are to be visited are stored in the open list. The values or distances between the nodes is refreshed every time and new cost value is entered into the list. The above graph indicates the shortest path is SABCG=8 .To find the shortest path from A to G ,the path needs to travel from SABCG path. Which gives the total distance as 8.



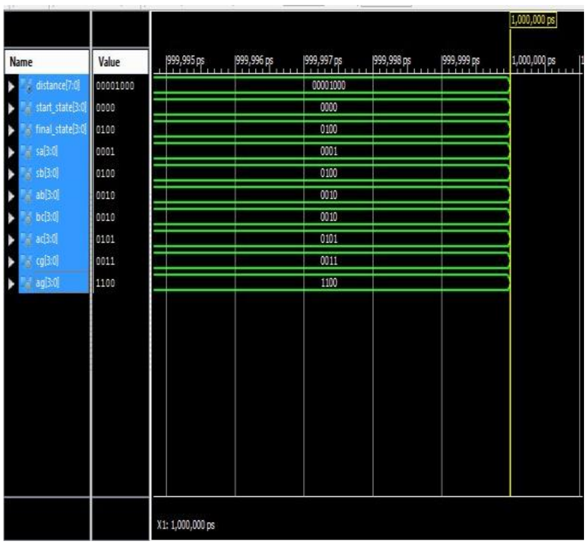
Flow Chart

The logical decision for the movement of the robot is shown from the flowchart shown below. Accordingly the robot moves in the shortest path possible by following the steps as shown in the figure below


start
Initialize at source node
find shortest path at each node
Compare each path value
If it is the shortest path
NO
yes
Next node
Destination node
stop

Simulation

The simulation result would be of the kind as shown in the following fiSystem-level testing may be performed with ISIM or the Model Sim logic simulator, and such test programs must also be written in HDL languages. Test bench programs may include simulated input signal waveforms, or monitors which observe and verify the outputs of the device under test.figure shown below



```
Name              Value
distance[7:0]     00001000
start_state[3:0]  0000
final_state[3:0]  0100
sa[3:0]           0001
sb[3:0]           0100
ab[3:0]           0010
bc[3:0]           0010
ac[3:0]           0101
cg[3:0]           0011
ag[3:0]           1100
```

The distance is calculated in terms of counts for the movement of the robot. The nodes are taken as the inputs for simulation purpose.

Conclusion

This paper deals with finding the shortest path from a given source to destination using a* algorithm. This a* algorithm is implemented using heuristic approach. This heuristic approach deals with the cost function where the distance between the nodes is taken as costs. This a* code is implemented on Xilinx ISE and result is simulated. This A* gives the minimum time to find the shortest path.The future work of this project intends to implement this Verilog code to implement on FPGA board. Straightforward extensions to our algorithm include improving the execution time of the space carving portion of the algorithm and demonstrating parallelization of the whole algorithm. In addition, more aggressive space carving may be possible by making inferences about sensor lines of sight that return no range data. In the future, we hope to apply our methods to other scanning technologies and to large scale objects such as terrain and architectural scenes

References

[1]. Delling, D.; Sanders, P.; Schultes, D.; Wagner, D. (2009). "Engineering route planning algorithms". Algorithmic of Large and Complex Networks: Design, Analysis, and Simulation. Springer. pp. 117–139. doi:10.1007/978-3-642-02094-0_7.

[2]. Zeng, W.; Church, R. L. (2009). "Finding shortest paths on real road networks: the case for A*". International Journal of Geographical Information Science 23 (4): 531–543.doi:10.1080/13658810801949850.

[3]. Hart, P. E.; Nilsson, N. J.; Raphael, B. (1968). "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". IEEE Transactions on Systems Science and Cybernetics SSC4 4 (2): 100–107. doi:10.1109/TSSC.1968.300136.

[4]. De Smith, Michael John; Goodchild, Michael F.; Longley, Paul (2007), Geospatial Analysis: A Comprehensive Guide to Principles, Techniques and Software Tools, Troubador Publishing Ltd, p. 344, ISBN 9781905886609.

[5]. Hetland, Magnus Lie (2010), Python Algorithms: Mastering Basic Algorithms in the Python Language, Apress, p. 214, ISBN 9781430232377.

[6]. Dechter, Rina; Judea Pearl (1985). "Generalized best-first search strategies and the optimality of A*". Journal of the ACM 32 (3): 505–536. doi:10.1145/3828.3830.

[7]. Koenig, Sven; Maxim Likhachev; Yaxin Liu; David Furcy (2004). "Incremental heuristic search in AI". AI Magazine 25 (2): 99–112.

[8]. Pohl, Ira (1970). "First results on the effect of error in heuristic search". Machine Intelligence 5: 219–236.

[9]. Pearl, Judea (1984). Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley. ISBN 0-201-05594-5.