World Applied Programming, Vol (2), No (4), April 2012. 251-256
ISSN: 2222-2510

Investigation of the * (Star) Search Algorithms: Characteristics, Methods and Approaches

Masoud Nosrati *
Dept of Computer Enginnering
Shaneh Branch,
Islamic Azad University,
Shaneh, Iran.
minibigs_m@yahoo.co.uk

Ronak Karimi
Dept of Computer Engineering
Shaneh Branch,
Islamic Azad University,
Shaneh, Iran.
rk_respina_67@yahoo.com

Hojat Allah Hasanvand
Department of Graphic
Shaneh Branch,
Islamic Azad University,
Shaneh, Iran.
hasanvand_6@yahoo.com

Abstract: In this study, a branch of search algorithms that are called * (star) algorithms is taken to look. Star algorithms have different types and derivatives. They are A*, B*, D* (including original D*, Focused D* and D* Lite), IDA* and SMA*. Features, basic concepts, algorithm and the approaches of each type is investigated separately in this paper.

Keywords: Star search algorithm, A*, B*, D*, original D*, Focused D*, D* Lite, IDA* and SMA*.

I. INTRODUCTION

A search algorithm is an algorithm for finding an item with specified properties among a collection of items. There is a type of search algorithms called * (star) search algorithms. This paper will briefly take a look to them. So, in the second section, the most popular star algorithms will be talked and their features and details will be investigated separately. Finally, conclusion is placed in the end of paper.

II. MOST POPULAR STAR ALGORITHMS

In this section we will cover the most important star algorithms. Due to it, we will study the A*, B*, D* (including original D*, Focused D* and D* Lite), IDA* and SMA*. The details for them will be talked separately.

A*

A* is a search algorithm that is widely used in path finding and graph traversal, the process of plotting an efficiently traversable path between points, called nodes. Noted for its performance and accuracy, it enjoys widespread use. Peter Hart, Nils Nilsson and Bertram Raphael first described the algorithm in 1968 [1]. It is an extension of Edsger Dijkstra's 1959 algorithm. A* achieves better performance (with respect to time) by using heuristics.
In 1964 Nils Nilsson invented a heuristic based approach to increase the speed of Dijkstra's algorithm. This algorithm was called A1. In 1967 Bertram Raphael made dramatic improvements upon this algorithm, but failed to show optimality. He called this algorithm A2. Then in 1968 Peter E. Hart introduced an argument that proved A2 was optimal when using a consistent heuristic with only minor changes. His proof of the algorithm also included a section that showed that the new A2 algorithm was the best algorithm possible given the conditions. He thus named the new algorithm in Kleene star syntax to be the algorithm that starts with A and includes all possible version numbers or A* [2].

As A* traverses the graph, it follows a path of the lowest known cost, keeping a sorted priority queue of alternate path segments along the way. If, at any point, a segment of the path being traversed has a higher cost than another encountered path segment, it abandons the higher-cost path segment and traverses the lower-cost path segment instead. This process continues until the goal is reached.
A* uses a best-first search and finds a least-cost path from a given initial node to one goal node (out of one or more possible goals).

It uses a distance-plus-cost heuristic function (usually denoted $f(x)$) to determine the order in which the search visits nodes in the tree. The distance-plus-cost heuristic is a sum of two functions:

- the path-cost function, which is the cost from the starting node to the current node (usually denoted $g(x)$)
- and an admissible "heuristic estimate" of the distance to the goal (usually denoted $h(x)$).

The *h(x)* part of the *f(x)* function must be an admissible heuristic; that is, it must not overestimate the distance to the goal. Thus, for an application like routing, *h(x)* might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any two points or nodes.
If the heuristic *h* satisfies the additional condition $h(x) <= d(x,y)+h(y)$ for every edge *x, y* of the graph (where *d* denotes the length of that edge), then *h* is called monotone, or consistent. In such a case, A* can be implemented more efficiently—roughly speaking, no node needs to be processed more than once - and A* is equivalent to running Dijkstra's algorithm with the reduced cost:

$d'(x,y):=d(x,y)-h(x)+h(y)$.

The time complexity of A* depends on the heuristic. In the worst case, the number of nodes expanded is exponential in the length of the solution (the shortest path), but it is polynomial when the search space is a tree, there is a single goal state, and the heuristic function *h* meets the following condition:

$$|h(x)-h^*(x)|=O(\log h^*(x))$$

where h^* is the optimal heuristic, the exact cost to get from *x* to the goal. In other words, the error of *h* will not grow faster than the logarithm of the "perfect heuristic" h^* that returns the true distance from *x* to the goal [3][4].

B*

B* is a best-first graph search algorithm that finds the least-cost path from a given initial node to any goal node (out of one or more possible goals). First published by Hans Berliner in 1979 [5]. The algorithm stores intervals for nodes of the tree as opposed to single point-valued estimates. Then, leaf nodes of the tree can be searched until one of the top level nodes has an interval which is clearly "best."
Leaf nodes of a B*-tree are given evaluations that are intervals rather than single numbers. The interval is supposed to contain the true value of that node. If all intervals attached to leaf nodes satisfy this property, then B* will identify an optimal path to the goal state.

To back up the intervals within the tree, a parent's upper bound is set to the maximum of the upper bounds of the children. A parent's lower bound is set to the maximum of the lower bound of the children. Note that different children might supply these bounds.

B* systematically expands nodes in order to create "separation," which occurs when the lower bound of a direct child of the root is at least as large as the upper bound of any other direct child of the root. A tree that creates separation at the root contains a proof that the best child is at least as good as any other child.
In practice, complex searches might not terminate within practical resource limits. So the algorithm is normally augmented with artificial termination criteria such as time or memory limits. When an artificial limit is hit, then you must make a heuristic judgment about which move to select. Normally, the tree would supply you with extensive evidence, like the intervals of root nodes.

B* is a best-first process, which means that the whole tree is kept in memory, and repeatedly descended to find a leaf to expand. This section describes how to choose the node to expand.

At the root of the tree, the algorithm applies one of two strategies, called prove-best and disprove-rest. In the prove-best strategy, the algorithm selects the node associated with the highest upper bound. The hope is that expanding that node will raise its lower bound higher than any other node's upper bound.

The disprove-rest strategy selects the child of the root that has the second-highest upper bound. The hope is that by expanding that node you might be able to reduce the upper bound to less than the lower bound of the best child.
Note that applying the disprove-rest strategy is pointless until the lower bound of the child node that has the highest upper bound is the highest among all lower bounds.

The original algorithm description did not give any further guidance on which strategy to select. There are several reasonable alternatives, such as expanding the choice that has the smaller tree.
Once a child of the root has been selected (using prove-best or disprove-rest) then the algorithm descends to a leaf node by repeatedly selecting the child that has the highest upper bound.

When a leaf node is reached, the algorithm generates all successor nodes and assigns intervals to them using the evaluation function. Then the intervals of all nodes have to be backed up using the backup operation.

When transpositions are possible, then the back-up operation might need to alter the values of nodes that did not lie on the selection path. In this case, the algorithm needs pointers from children to all parents so that changes can be propagated. Note that propagation can cease when a backup operation does not change the interval associated with a node.

If intervals are incorrect (in the sense that the game-theoretic value of the node is not contained within the interval), then B* might not be able to identify the correct path. However, the algorithm is fairly robust to errors in practice.
The Maven (Scrabble) program has an innovation that improves the robustness of B* when evaluation errors are possible. If a search terminates due to separation then Maven restarts the search after widening all of the evaluation intervals by a small amount. This policy progressively widens the tree, eventually erasing all errors.

The B* algorithm applies to two-player deterministic zero-sum games. In fact, the only change is to interpret "best" with respect to the side moving in that node. So you would take the maximum if your side is moving, and the minimum if the opponent is moving. Equivalently, you can represent all intervals from the perspective of the side to move, and then negate the values during the back-up operation.

Andrew Palay applied B* to chess. Endpoint evaluations were assigned by performing null-move searches. There is no report of how well this system performed compared to alpha-beta pruning search engines running on the same hardware.

The Maven (Scrabble) program applied B* search to endgames. Endpoint evaluations were assigned using a heuristic planning system. This program succeeded splendidly, establishing the gold standard for endgame analysis.
The B* search algorithm has been used to compute optimal strategy in a sum game of a set of combinatorial games [6].

D*
D* is any one of the following three related incremental search algorithms:

- The original D*[7], by Anthony Stentz, is an informed incremental search algorithm.
- Focused D*[8] is an informed incremental heuristic search algorithm by Anthony Stentz that combines ideas of A*[9] and the original D*. Focused D* resulted from a further development of the original D*.
- D* Lite[10] is an incremental heuristic search algorithm by Sven Koenig and Maxim Likhachev that builds on LPA*[11], an incremental heuristic search algorithm that combines ideas of A* and Dynamic SWSF-FP[12].

All three search algorithms solve the same assumption-based path planning problems, including planning with the freespace assumption[13], where a robot has to navigate to given goal coordinates in unknown terrain. It makes assumptions about the unknown part of the terrain (for example: that it contains no obstacles) and finds a shortest path from its current coordinates to the goal coordinates under these assumptions. The robot then follows the path. When it observes new map information (such as previously unknown obstacles), it adds the information to its map and, if necessary, replans a new shortest path from its current coordinates to the given goal coordinates. It repeats the process until it reaches the goal coordinates or determines that the goal coordinates cannot be reached. When traversing unknown terrain, new obstacles may be discovered frequently, so this replanning needs to be fast. Incremental (heuristic) search algorithms speed up searches for sequences of similar search problems by using experience with the previous problems to speed up the search for the current one. Assuming the goal coordinates do not change, all three search algorithms are more efficient than repeated A* searches.

*Original D**
The original D* was introduced by Anthony Stentz in 1994. The name D* comes from the term "Dynamic A*", because the algorithm behaves like A* except that the arc costs can change as the algorithm runs.

The basic operation of D* is outlined below:
Like Dijkstra's algorithm and A*, D* maintains a list of nodes to be evaluated, known as the "OPEN list". Nodes are marked as having one of several states:

- NEW, meaning it has never been placed on the OPEN list
- OPEN, meaning it is currently on the OPEN list
- CLOSED, meaning it is no longer on the OPEN list
- RAISE, indicating its cost is higher than the last time it was on the OPEN list
- LOWER, indicating its cost is lower than the last time it was on the OPEN list

The algorithm works by iteratively selecting a node from the OPEN list and evaluating it. It then propagates the node's changes to all of the neighboring nodes and places them on the OPEN list. This propagation process is termed "expansion". In contrast to A*, which follows the path from start to finish, D* begins by searching backwards from the goal node. Each expanded node has a backpointer which refers to the next node leading to the target, and each node knows the exact cost to the target. When the start node is the next node to be expanded, the algorithm is done, and the path to the goal can be found by simply following the backpointers.

When an obstruction is detected along the intended path, all the points that are affected are again placed on the OPEN list, this time marked RAISE. Before a RAISED node increases in cost, however, the algorithm checks its neighbors and examines whether it can reduce the node's cost. If not, the RAISE state is propagated to all of the nodes' descendants, that is, nodes which have backpointers to it. These nodes are then evaluated, and the RAISE state passed on, forming a wave. When a RAISED node can be reduced, its backpointer is updated, and passes the LOWER state to its neighbors. These waves of RAISE and LOWER states are the heart of D*.

By this point, a whole series of other points are prevented from being "touched" by the waves. The algorithm has therefore only worked on the points which are affected by change of cost.

This time, the deadlock cannot be bypassed so elegantly. None of the points can find a new route via a neighbor to the destination. Therefore, they continue to propagate their cost increase. Only outside of the channel can points be found, which can lead to destination via a viable route. This is how two Lower waves develop, which expand as unattainably marked points with new route information.

*Focused D**
As its name suggests, Focused D* is an extension of D* which uses a heuristic to focus the propagation of RAISE and LOWER toward the robot. In this way, only the states that matter are updated, in the same way that A* only computes costs for some of the nodes.

D Lite*
D* Lite is not based on the original D* or Focused D*, but implements the same behavior. It is simpler to understand and can be implemented in fewer lines of code, hence the name "D* Lite". Performance-wise, it is as good or better than Focused D*. D* Lite is based on Lifelong Planning A*, which was introduced by Amita few years earlier.
Minimum cost versus current cost

For D*, it is important to distinguish between current and minimum costs. The former is only important at the time of collection and the latter is critical because it sorts the OpenList. The function which returns the minimum cost is always the lowest cost to the current point since it is the first entry of the OpenList.

IDA*
IDA* [14] is a variant of the A* search algorithm which uses iterative deepening to keep the memory usage lower than in A*. It is an informed search based on the idea of the uninformed iterative deepening depth-first search.
The main difference to IDS is that it uses the f-costs (g + h) as the next limit and not just an iterated depth.
In the following pseudocode, *h* is the heuristic function that estimates the path from a node to the goal.

```
algorithm IDA*(start):
    cost_limit = h(start)

    loop:
        solution, cost_limit = depth_limited_search(0, [start], cost_limit)
        if solution != None:
            return solution
        if cost_limit == ∞:
            return None

# returns (solution-sequence or None, new cost limit)
algorithm depth_limited_search(start_cost, path_so_far, cost_limit):
    node = last_element(path)
    minimum_cost = start_cost + h(node)
    if minimum_cost > cost_limit:
        return None, minimum_cost
    if is_goal(node):
        return path_so_far, cost_limit

    next_cost_limit = ∞
    for s in successors(node):
```

```
        new_start_cost = start_cost + edge_cost(node, s)
        solution, new_cost_limit = depth_limited_search(new_start_cost, extend(path_so_far, s),
cost_limit)
        if solution != None:
            return solution, new_cost_limit
        next_cost_limit = min(next_cost_limit, new_cost_limit)

    return None, next_cost_limit
```

The difference to A* can be seen from this pseudo code: it doesn't remember the current shortest path and costs for all visited nodes like in A* (that is why space-complexity is linear in A* to all visited nodes until it finds the goal) but it only remembers one single path at a time [15].

SMA*

SMA* Search is one of the Memory bounded heuristic search that comes under Informed Search Strategies. The main advantage of this search is that it only makes use of available memory to carry out the search.
The Simplified Memory-Bounded Algorithm (SMA*) is a variant of A* search which is memory-bounded [16]

Pseudo code:

```
function SMA-star(problem): path
  queue: set of nodes, ordered by f-cost;
begin
  queue.insert(problem.root-node);

  while True do begin
    if queue.empty() then return failure;
    node := queue.begin(); // min-f-cost-node
    if problem.is-goal(node) then return success;

    s := next-successor(node)
    f(s) := max(f(node), g(s) + h(s))
    if no more successors then
      update node-s f-cost and those of its ancestors if needed

    if node.successors □ queue then queue.remove(node);
    if memory is full then begin
      badNode := queue.popEnd(); // removes node with highest f-cost out of queue
      for parent in badNode.parents do begin
        parent.successors.remove(badNode);
        if needed then queue.insert(parent);
      end;
    end;

    queue.insert(s);
  end;
end;
```

III. 3. CONCLUSION

In this paper, we had an investigation on the basic concepts of star search algorithms, and the features of them. Different types of star algorithms were specified and they were talked separately. They were: A*, B*, D* (including original D*, Focused D* and D* Lite), IDA* and SMA*.

REFERENCES

[1] 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.
[2] A* search algorithm article. Available at: http://en.wikipedia.org/wiki/A*_search_algorithm.
[3] Pearl, Judea (1984). Heuristics: Intelligent Search Strategies for Computer Problem Solving. Addison-Wesley. ISBN 0-201-05594-5.
[4] Russell, S. J.; Norvig, P. (2003). Artificial Intelligence: A Modern Approach. Upper Saddle River, N.J.: Prentice Hall. pp. 97–104. ISBN 0-13-790395-2.
[5] Berliner, Hans (1979). "The B* Tree Search Algorithm. A Best-First Proof Procedure.". Artificial Intelligence 12 (1): 23–40. doi:10.1016/0004-3702(79)90003-1.
[6] Sheppard, Brian (2002). "World-championship-caliber Scrabble.". Artificial Intelligence 134 (1-2): 241–275. doi:10.1016/S0004-3702(01)00166-7.

[7] Stentz, Anthony (1994), "Optimal and Efficient Path Planning for Partially-Known Environments", Proceedings of the International Conference on Robotics and Automation: 3310–3317
[8] Stentz, Anthony (1995), "The Focussed D* Algorithm for Real-Time Replanning", In Proceedings of the International Joint Conference on Artificial Intelligence: 1652–1659
[9] P. Hart, N. Nilsson and B. Raphael, A Formal Basis for the Heuristic Determination of Minimum Cost Paths, IEEE Trans. Syst. Science and Cybernetics, SSC-4(2), 100–107, 1968.
[10] S. Koenig and M. Likhachev. Fast Replanning for Navigation in Unknown Terrain. Transactions on Robotics, 21, (3), 354–363, 2005.
[11] S. Koenig, M. Likhachev and D. Furcy. Lifelong Planning A*. Artificial Intelligence Journal, 155, (1–2), 93–146, 2004.
[12] G. Ramalingam, T. Reps, An incremental algorithm for a generalization of the shortest-path problem, Journal of Algorithms 21 (1996) 267–305.
[13] S. Koenig, Y. Smirnov and C. Tovey. Performance Bounds for Planning in Unknown Terrain. Artificial Intelligence Journal, 147, (1–2), 253–279, 2003.
[14] Korf, Richard (1985). "Depth-first Iterative-Deepening: An Optimal Admissible Tree Search". Artificial Intelligence 27: 97–109.
[15] IDA* article. Available at: http://en.wikipedia.org/wiki/IDA*
[16] Russell, S. 1992. Efficient memory-bounded search methods. In Proceedings of the 10th European Conference on Artificial intelligence (Vienna, Austria). B. Neumann, Ed. John Wiley & Sons, New York, NY, 1-5.