A* Implementation Details, Guidelines, and Tips

Here is some additional information that will help you to implement the A* algorithm. While the algorithm itself is usually easy to understand, implementing it can be tricky because it involves a lot of data structures for bookkeeping.

- *How do I implement a state?*

  The implementation of a state depends a lot on the problem being solved. Often it can be a simple data type such as an integer or a string, but sometimes it needs to be something more structured like a class or other aggregate data type. In general, if your conception of a "state" involves two or more pieces of information that are not the same data type themselves, you will need to create a new data type (e.g., a class).

  Also, remember that values of the f(n), g(n), and h(n) functions are not part of a state representation. The exception might be h(n), because usually h(n) depends only on the state itself, not on extra information in the node, but usually we store all three of these values in a node, rather than a state.

- *How do I implement an action?*

  The book makes many references to the concept of an "action," which is what moves the agent from one state to another state. Whether an action needs to be explicitly represented as a variable in your code depends upon the problem you are solving A* with. Because A* makes it easy to trace the sequence of states from the goal state back to the initial state once a goal state is found, if this sequence of states is all that is needed to present the solution to the user, then actions often do not need to be explicitly represented at all. In other words, if given a state *s1* and a successor state *s2*, the action that the agent took to move from *s1* to *s2* is not important or can be easily recovered by inspecting the two states, then explicitly representing the action as a variable is often not necessary.

  However, for some problems, representing the action explicitly is useful. Consider the 8-puzzle, where the possible actions are to slide a tile left, right, up, or down. While representing an action is not technically required here, since given two 8-puzzle boards that differ in only one tile position, it is *possible* to reconstruct the action that transformed the first board into the second, it is helpful to represent the action (e.g., as the string "left," "right," etc.) because recovering the action by computing the difference between two boards requires extra computational work.

- *How do I implement a node?*

  Unlike a state, the implementation of a node (as in the data structure itself) does **not** usually depend on the problem being solved. In other words, whether you are solving the navigation problem, the 8-puzzle, or the Roomba problem, the implementation of a node does not really change.

  I recommend following the book's suggestion for a node, with a few extra components. I suggest creating a data structure (e.g., a class) that stores the following pieces of information:

  - a state (the state to which the node corresponds)
  - a parent pointer or reference (the node in the search tree that generated this node)
  - an action, if needed (the action that was applied to the parent's state to generate this node)
  - the f(n), g(n), and h(n) values for this node (the book only makes reference to "path-cost," which is g(n), but I think it's nice to have all three values in one place)

- *How do I implement the functions in the book like ACTIONS(s), RESULT(s, a), IS-GOAL(s), and ACTION-COST(s, a, s')?*

  Whether you want to explicitly write these functions as functions in your code is up to you. For some problems, it may be overkill. For instance, if you are choosing not to store actions in your nodes, then implementing the ACTIONS function is probably not necessary. Because ACTIONS and RESULT are only used in the EXPAND function, often people will combine those functions into one function that generates all possible child states from a given state.

  ACTION-COST is also a function that often does not need to be explicitly implemented, often because you will have already written a function somewhere else in your code that does the same thing (or the cost for each action is constant, and writing a function would be overkill).

  Similarly, whether IS-GOAL needs its own function depends on the problem. For problems with a single goal state, writing a separate function might be overkill because it might just be a simple boolean test in your code. On the other hand, for problems with multiple goal states that are evaluating complex conditions about the state, writing this function might be prudent and will make your code clearer.

  In summary, you don't have to write any of these functions, but then you will have to adapt the pseudocode slightly to account for that. This is truly fine, but if you don't feel 100% secure about doing that, feel free to write all of these functions, even if they are super-short.

- *How do I implement the frontier?*

  The *frontier* is a priority queue that keeps track of nodes. It is always sorted by f(n), which for A* is equal to g(n) + h(n). (*Note that in other search algorithms, f(n) might be equal to g(n) alone or even h(n) by itself*.) You should use a priority queue data structure and have it sort nodes so lower values of f(n) come off the queue first.

- *How do I implement the reached map?*

  The *reached* map, in the pseudocode, is a map from states (not nodes!) to nodes. It is used in situations where a node is encountered in the search tree that contains a state which we have already seen in a different part of the search tree. In this situation, we must evaluate if the path to the newly-discovered node is faster than the path to the previously-discovered node. This is difficult because the frontier does not let us easily access the previously-discovered node to check what its value for f(n) is. The reached map is lets us solve this problem, because it lets us look up previously-found nodes, (and therefore their f(n) or g(n) values) based on a state. And because the states for the newly-discovered node and the previously-discovered node which we're trying to compare are *identical*, this solves the problem.

  To implement this map, you can follow the pseudocode exactly, and create a map from states to nodes. Or, because we actually only care about looking up the values of f(n) for a state once we find it (that is, we don't need the entire node), you can create a map from states to **numbers** (whatever numeric type you are using for f(n), like a double or int).

Clearer Pseudocode for A*

This pseudocode is adapted from the best-first search algorithm presented on page 73 of the 4th edition of AIMA. The pseudocode assumes that a node has an explicit entry for storing its value of *f(n)* [called F_COST]. If this is not stored, it must be recalculated each time by calculating g(n) and h(n) and summing them.

```
A-STAR-SEARCH(initial_state)  // returns a solution node or null, indicating failure
    node ← a new node corresponding to initial_state, with PARENT = null, ACTION = null,
                    G_COST = 0, H_COST = h(initial_state), F_COST = G_COST + H_COST
    frontier ← a priority queue of nodes ordered by f(n) [F_COST], initialized to contain only node
    reached ← a map from states to nodes with one entry mapping initial_state to node
    while not IS-EMPTY(frontier):
        node ← POP(frontier)  // remove lowest cost node from frontier [smallest value of f(n)]
        if IS-GOAL(node.STATE) then return node
        for each child_node in EXPAND(node):
            child_state ← child_node.STATE
            if child_state is not in reached or child_node.F_COST < reached[child_state].F_COST:
                reached[child_state] ← child_node
                ADD child_node to frontier
    return null

EXPAND(node)  // returns a list or set of child nodes for node
    child_node_collection ← an empty list or set to hold the child nodes
    state ← node.STATE
    for each action in ACTIONS(state):
        child_state ← RESULT(state, action)
        child_gcost ← node.G_COST + ACTION-COST(state, action, child_state)
        child_node ← a new node corresponding to child_state, with PARENT = node,
                    ACTION = action, G_COST = child_gcost, H_COST = h(child_state),
                    F_COST = G_COST + H_COST
        ADD child_node to child_node_collection
    return child_node_collection
```