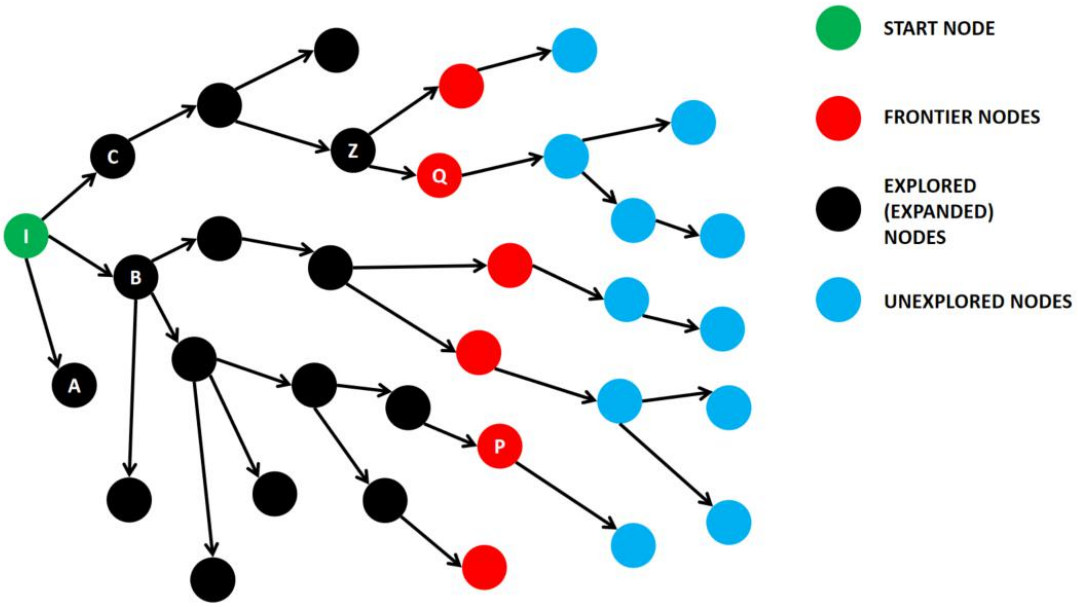
Implementing Greedy Best-First-Search / A* algorithms: Keeping track of nodes

First, let's distinguish two things:

- your input problem: state space graph,
- search tree for your problem.

It is better keep the state space graph as is at all times. Search tree CAN be (but does not have to be: you can create entire tree first and then explore it) dynamically constructed. You'd still need to keep at least some of it "alive".

To illustrate what I mean, let's start with this diagram:



Both search algorithms explore the state space in the way (but use different evaluation function to choose next move). They start with the start / initial node I and expand it to see available options (and to evaluate their "potential).

All three options (A, B, C) for the start node I will constitute the INITIAL set of frontier nodes (a set of nodes that are "promising": there's a chance that I will get to my goal state (not shown) through either A, B, and C). One of those A, B, C nodes will be the "most promising" (highest or lowest value of the evaluation function; lowest for our shortest path assignment). Let's make it A. However, B and C should be kept as an option, in case the path to the goal state through A will not pan out (either a dead end or it does not look like it is going to be the shortest path). In this case A would be a dead end and we'd need to choose from B and C for further exploration (again based on the evaluation function value). Say we picked B. We'd still need to keep C option open just in case. Moving from I to A makes A an explored ("**reached**"

using the textbook pseudocode naming convention) node. Moving from I to B makes B an explored ("**reached**" using the textbook pseudocode naming convention) node.

Now, this approach progresses throughout the state space (here left to right). At all times we keep a set of **frontier** nodes to keep our options open if we need to switch paths (BTW: this not backtracking in its fullest sense. More like jumping from one option to the other). All the **frontier** nodes are kept in a priority queue, so our "next best" option is always ready at the front of the queue.

Now, one thing that you need to remember, is that our solution is the ENTIRE path from start node to goal node (a sequence of nodes that the algorithm picks to get from one to the other). In the end, those are the nodes you need to "keep" (keeping means either keeping node objects "alive" or having some sort of a list that is constantly updated; a lot depends on your implementation).

For example, if my path to the goal state through node P does not pan out, my next best option could be getting to the goal state via node Q. However, if the path through Q becomes THE path, I need to know how did get to Q in the first place (through Z, ..., C, all the way to I). In other words, I should be able to get that information (either I keep track of it or I can extract it whenever I need. The latter is likely to be harder).

The way the **pseudocode in your textbook** envisions the node object is with the PARENT link. That parent link, **if done right**, automatically "preserves" the path and keeps all nodes of interest "alive". Think: all **frontier** nodes are kept "alive" and all their PARENT nodes are kept alive through their PARENT link (a reference keeps an object "alive"). PARENT's PARENT node object is also kept "alive" that way, all the way to the start node.

According to the textbook pseudocode, node A would be kept "alive" as part of the *reached* table. Even though it is a dead end and we coul discard it. And here we reach an implementation matters:

- you could be keeping a list of node LABELS only as *reached* instead of node objects,
- you could be re-creating nodes as needed / as you go.

In other words, a lot depends on your implementation (you could be discarding and re-creating nodes).

If you stick fairly closely to the pseudocode in the textbook, Python will take care of keeping necessary parts "alive" itself and you will not have to do any clean-ups. On the same token, I'd recommend keeping it simple and not discarding any nodes.