MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.034 Artificial Intelligence, Fall 2003
Notes on the Complexity of Search September 23

Introduction

One of the ways we evaluate search methods is as to their worst-case time or space complexity. This brief handout is meant to explain to you how we can derive the time and space complexity for various types of search, as outlined in the table of search methods below. "Proofs" will be intuitive and informal. After this, we will discuss two additional points: first, a summary of the pluses and minuses of each search method (rules of thumb for when you might use that method – but *remember* there is no 'fixed' answer here); and then, second, some subtleties about the complexity of A* search that you are **not** required to know, but are there for your learning enrichment (some folks like this sort of thing). We also omit a few other search methods, like iterative-deepening, that I'll cover in a later note.

Before getting started, two general points about time and space complexity. Say we have an algorithm that uses space $O(n)$ – what we call *linear space.* (If you can't remember what the order notation means, then, roughly and intuitively, it means, 'within a constant factor' or 'proportional to'.) It should be clear that such an algorithm must also use *at least* time $O(n)$ – that is, time used is at least, but often more, than the space used by an algorithm. That's because in order to 'examine' *anything* of size (space) n will take at least n time steps, however we measure time.

Second, these complexity results are *worst case* – that is, assuming some perverse adversary is out to make things turn out as horrible as possible. More often, we would be interested in *average case* results, but to prove such things requires assumptions about the distribution of problem spaces, problem inputs, etc. (but see the results for A* which do give a hint about this). **Note: I've truncated this material a bit so as to get the complexity results posted ASAP, so the material that is forward referenced, like this, will be posted later. In fact, it might be a good idea to real this later material in time for recitation.**

Assumptions

1. For the search method analysis, we shall assume that the search tree has a depth ***d*** and an *average* branching factor ***b***. The depth is just the number of levels in the tree. (If that's not clear to you, please ask me.) The average branching factor is a bit more complex, but not that challenging (i.e., if the tree had just 3 levels, and one level branched 3 and the next, 5, and the third 1, the average, and so b, would be 3).
(Note that we assume that the branching factor is never infinite – an assumption that might not have been made clear in lecture or in the tutor – is this a safe assumption? You might ponder that.)

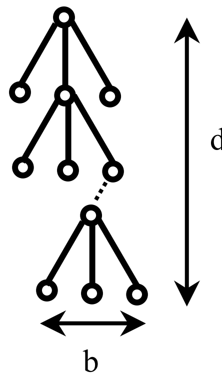
2. In general we assumed that our search methods announce that they have found a Goal when they **pop** a path that includes a Goal from the queue – **not** when that path is put on the queue. The effect of this is to make the algorithms go one more step beyond the node at which the Goal is found. Thus, if the Goal was found at depth d, then our algorithms will expand one more time, in effect making the depth $d+1$.
To see the same thing another way, if we don't 'wait' and do this then our algorithms will be in 'too much of a hurry' to find a Goal state. In this rush, they might miss an optimal goal. For suppose our algorithms returned 'success' as soon as an extended mother node puts a daughter path with a Goal on the path queue. Then if such mother node extended to another, better path, we would never find it. There is no guarantee that we would return the best successor. (Note this doesn't apply to the situations where we don't care about finding an optimal path.)

3. We have been deliberately vague about what counts as a primitive time step, and various other details as to the overhead of pushing & popping queues, sorting queues, etc. In general, all this bookkeeping won't make the time or space complexity any worse than it already is (but see the optional section on 'subtleties' at the end of this handout).

OK, that said, let's look at the search method table and time/space complexity results we shall cover.

Search Method	**Worst Time**	**Worst Space**	**Fewest Nodes to G?**	**Guaranteed to find path?**
Depth-first (w/ backup)	$O(b^{d+1})$	$O(bd)$	No	Yes
Breadth-first	$O(b^{d+1})$	$O(b^d)$	Yes	Yes
Hill-climbing (no backup)	$O(d)$	$O(b)$	No	No
Hill-climbing (w/backup)	$O(b^{d+1})$	$O(bd)$	No	Yes
Best-first	$O(b^{d+1})$	$O(b^d)$	No	Yes
Beam-search, width k	$O(kd)$	$O(kb)$	No	No
A* search	$O(b^{d+1})$	$O(b^d)$	No	Yes
Alpha-beta pruning	$O(b^{d/2})$- $O(b^{3d/4})$	$O(b^d)$	No	Yes

All these results are based on figuring out the size of a search tree – the number of nodes in a tree, with (average) branching factor b and depth d. For simplicity, assume constant branching factor b. Then such a search tree looks like this:



If each node has b immediate descendants, then level 0 (the root node) has 1 node.
Level 1 has b nodes
Level 2 has $b*b = b^2$ nodes
Level 3 has $b^2*b = b^3$ nodes
…
Level d has $b^{d-1}*b= b^d$ nodes
So the total number of nodes is:

$N= 1 + b + b^2 + b^3+ \ldots + b^d$ so therefore
$bN= \quad b + b^2 + b^3+ \ldots + b^d+ b^{d+1}$ so subtracting the 2nd line from the first, we have:

$(1\text{-}b)N = b^{d+1} – 1$ or $N= (b^{d+1} – 1)/b\text{-}1$ which is $O(b^d)$

OK, now let's use this result.

Depth-first search: In the worst case, the Goal will be at the far, right corner leaf of the search tree, so the algorithm will have to search *all* the nodes in the tree, plus one extra level of b nodes to account for our stopping condition i.e., $b^d * b = O(b^{d+1})$. So this is the worst case *time* for depth-first search. Note that if we are lucky, the Goal might be at the very left-most edge of the search tree, in which case the time would be $O(d)$ – or of course the Goal might even be the root node itself. That would be very lucky.

As for the worst-case space, we must consider the maximum storage space the algorithm could use at any one step in our simulation tableaus. This space is dominated by the size of the queue (the list of partial paths). Intuitively, this represents just one straight line zigging down the search tree to the leaves (at worst). How big could this get? You should be able to convince yourself that, in the worst case, suppose the Goal lies at a leaf

node of the search tree (say, the left-most leaf – the argument works for any Goal on a leaf node of the tree). In this case, at each iteration of the search, DFS will add b partial paths to the queue. At most, it can do this d times, until it reaches the bottom (leaf-node) of the search tree. At this point, the queue is of size bd. If the Goal is not found at this point, DFS unwinds (the stack is popped), perhaps all the back to the first level. So bd is the maximum space used. Note that DFS isn't guaranteed to find the Goal unless it does backtracking, and it certainly might waste a lot of time finding the Goal (if it's on the right side, say).

Breadth-first search: In the worst case, the Goal will be in the same spot as in DFS, at the far right corner leaf of the search tree. So the worst case time is exactly as for DFS, or $O(b^{d+1})$. What about the worst space? BFS adds nodes to the *end* of the queue. So, by the time the algorithm has arrived at the right-most leaf node, it will have pushed *all* the nodes in the search tree on its queue. So the worst-case space is or $O(b^d)$ (the space doesn't get any worse when we pop the Goal path off the stack after we find it –it actually gets a bit better).
It also should be pretty clear that, like the Mounties, BFS will always get its Goal, and, since it works its way down the search tree level by level, it will get to the 'earliest' Goal node in terms of tree depth, and so fewest nodes.

Hill-climbing (no backup): Intuitively, hill-climbing without backup just takes *one* path through the search tree, according to the local heuristic h. So, it's actually working like DFS – but just one path, in the worst case, all the way to a leaf node in the tree. Thus, the worst-case time is just the depth of the tree, d, and the since there's no backup, the worst space is just a single level's breadth, or b. Without backup, hill-climbing cannot guarantee that it will find an answer, or that it will find an answer by examining the fewest nodes.

Hill-climbing (with backup): This is exactly the same as DFS – the only difference is the *order* that nodes are expanded in. That doesn't change the time or space complexity in the worst case (though in the average case, the whole idea of a heuristic is to ensure that we get to a Goal faster…so, if it's a good heuristic, the average time complexity ought to improve). Just like DFS then, it will always find an answer, though not necessarily the one earliest in the search tree.

Best-first: This is simply breadth-first search, but with the nodes re-ordered by their heuristic value (just like hill-climbing is DFS but with nodes re-ordered). So, in the worst case, the time and space complexity for best-first search is the same as with BFS: $O(b^{d+1})$ for time and $O(b^d)$ for space. Of course, we would hope that our heuristic would give far better average case results. (In fact, you can generalize this fact: since we can imagine a stupid heuristic that assigns a value "1" to each node, or "1" to each path length, in general the heuristic or even branch-and-bound or A* search behave just like BFS – though you might want to take a look at some subtleties in the optional section below for figuring out when A* does better in terms of time/space – of course it does as good as we can do in terms of finding the optimal cost path).

Beam search (width k, **no** backup): This is like BFS, but with a fixed number of queue elements k (the k best) considered at each iteration. Since in the worst case, the algorithm might plunge all the way to the leaves of the search tree, depth d, the worst-case time is $O(kd)$. Since the queue contains at most proportional to k elements at any step, the worst-case space is $O(k)$.

A* search: See under the discussion for best-first search. A* has the same worst-case complexity as BFS.

Alpha-beta: Winston demonstrates that if a decent heuristic for ordering moves can be found (like the example where the right line of play is always on the left), then half the nodes need not be evaluated. In this case, the time complexity is cut in half, that is, to $O(b^{d/2})$ The space complexity here in the worst case is the same as for other depth-first searches, $O(bd)$, and for the same reasons. Note that for any reasonable game like chess, this is still a *huge* number. (More about this later.)