MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.034 Artificial Intelligence, Fall 2003
Recitation 4: Optimal Search & Games, September 25/26th

Introduction: Branch & Bound; A* search

A* search is the most widely used approximation search algorithm on search spaces (graphs). This is because it is complete (it will always find an optimal solution if one exists) and it is optimal (it examines the fewest nodes possible in doing so). (***There are other search methods beyond A* that are used because, as we'll see, A* can use too much space, but these are no longer guaranteed to be complete.) So, let's see what A* is all about.

While uninformed (blind) searches are simple, they really are not of much use in actual problem spaces because they take far too much time and space (There are exceptions: if there are many, many solutions then DFS may work more quickly than informed search. And even with heuristics, even, as we'll see, with A*, there may be too many options to explore and this might take up too much memory, so something like a depth-limited or width-limited, i.e., beam search constraint, might prove useful. This latter case often occurs in speech recognition problems, for example.)

To see how bad things can get even with a simple problem, consider the 8-puzzle again. With 9 squares, one can show that there are 9! = 362,880 non-repeated states. That's already too many. Remarkably, A* can cut this down to 25 or 39 examined nodes (the first heuristic = the # of tiles out of place; the 2nd – the sum of the 'Manhattan' distances. See the table below, taken from Russell & Norvig).

Where does A* come from? So far, we have used only the following notion of a heuristic function *h* in order to 'prune' paths and tell the search algorithm which paths to pursue. Roughly, *h* estimates the (true) distance **remaining** to the Goal – the cost of **future** search. In what follows, we'll make this notion precise and see that in order to produce an optimal search we must also measure the distance ***g*** we have **already gone,** or **past search.** Intuitively, this is the accumulated path length traveled up to some node.

If we use *g* alone, and simply enumerate paths in order of their known lengths already traveled, along with the principle, 'avoid paths with *g* values already larger than what we've got' – this gives us **branch and bound** search.

However, it should be clear that B&B could be very inefficient in terms of # of nodes enqueued and extended – because we don't use our forward estimate of future distance remaining. If we combine *h* and *g*, both satisfying certain properties, along with best-first search, we get A* search, which, as we'll see, is guaranteed to find an optimal path by examining as few nodes as possible – it does no extra work. (We will do an example in a moment.) Ultimately then, A* uses the following function *f* as its evaluation metric, and it attempts to *minimize* this value of $f(n)$, consistent with *n* being a Goal node:

$$f(n) = g(n) + h(n)$$

Again intuitively, *h* represents a 'depth-first' factor in *f,* while *g* adds a 'breadth-first' factor to *f.* Note that $g(n)$ is the actual cost from S(tart) to *n* while $h(n)$ is the estimated cost from *n* to *G*(oal). A* extends nodes in a search tree according to *f** values: the node (or leaf) with the smallest *f*-value is the node to be extended next (other texts like the online tutor use the word 'expanded' instead of 'extended').

Definition of a *heuristic function:* A **heuristic** is a function *h* on search tree nodes that satisfies the following conditions: (i) $h(n) \geq 0$ for all nodes *n*; (ii) $h(n)=0$ implies that *n* is a Goal node; (iii) $h(n)= \infty$ implies that *n* is a dead end from which no goal can be reached. Informally then, this function *h* estimates the 'goodness' of a node *n* with respect to reaching a goal. (Remember, we want the *smallest cost* or *shortest* path.) Therefore, *h* is an estimate of 'remaining distance to go' or **the cost of future search.** $h(n)$ is an **estimate** of the ***true*** **value** of the remaining distance, *h*, based* on domain-specific information, that is **computable** from the current state description. Heuristics do not guarantee feasible solutions and are often without theoretical basis.

Examples of heuristics.

- Fox and geese: # of items on starting river bank
- 8-puzzle: # of tiles out of place
- Checkers: [# pieces taken]
- Tic-tac-toe [# winnable lines – corner is 3]
- Route finding: [straight line distance]
- VLSI Design layout: [amount of area used]

Note 0: The *h* value for a goal state (node) is always 0.
Note 1: A **good** heuristic will reduce the branching factor as close as possible to 1.
Note 2: The **null** heuristic is the function $h(n) = 0$. (It **does nothing**. In fact, it acts like 'uniform path cost' search – we haven't talked about that one, it just uses *g*).

Note 3: One heuristic h_2 is **better** than a heuristic h_1 iff $h^* \geq h_2 \geq h_1$ for all non-goal nodes. (Do you see why?) When h is 0, it does no pruning at all. As h gets larger and larger, it starts approaching the true cost remaining, and it will prune more and more nodes from consideration by the search algorithm, until it becomes exactly equal to the true cost value, h^*.)
Ans: If A_1^* uses h_1, and A_2^* uses h_2, then every node expanded by A_2^* is also expanded by A_1^*, that is, A_1^* expands at least as many nodes as A_2.
Note 4: A heuristic h is **perfect** iff $h=h^*$ In this case, *no* extra work is performed.

So what happens if a heuristic gets *larger* than the true value, h^*?? Oops! We might miss an optimal path…We call such errant heuristics *inadmissible,* while their counterparts, heuristics that always behave, and never get larger than the true cost remaining to the Goal, are called *admissible:*

Definition of an *admissible heuristic*: Let $h^*(n)$ be the actual or true cost of the optimal path from n to the Goal state. A heuristic $h(n)$ is **admissible** iff for all nodes n, $h(n) \leq h^*(n)$. In other words, an admissible heuristic is optimistic and *never* over-estimates the cost (distance) remaining to the Goal (that is, it estimates that the cost of solving the problem is less than it actually is). Suppose that h did over-estimate the distance remaining to the Goal. Then we can show a search algorithm using this h might miss the optimal path. (Can you see why this would happen? Assign some extreme values to a graph…)

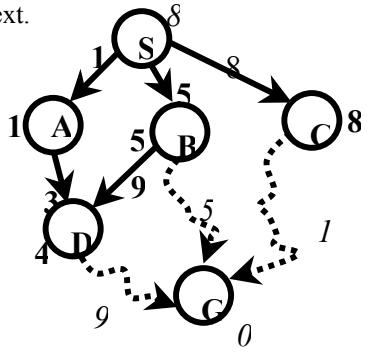
Examples of admissible heuristics.

- Routing: h=travel distance along a straight line. Must be shorter than actual travel path.
- 8-puzzle: h=tiles out of place. Each move can reorder at most one tile.
- 8-puzzle: h= 'Manhattan' distance sum (movement along right angles) from proper final position.

OK, so much for h. What about $g(n)$, the actual distance (or cost) accumulated so far to a particular node n? First of all, we always want $g \geq 0$ (no negative numbers), which makes sense; also, g cannot be infinitesimally small. Second, if we set g to 0, then $f(n) = h(n)$ and we are simply using our old-style heuristic measures (best-first, hill-climbing, whatever). If we set then $f(n) = g(n)$, then this is called 'uniform cost search', aka 'branch and bound': if we ever accumulate a cost higher than a known path cost we already have, there's no need to look at it. But such a search is also 'too greedy', and does useless work because a locally great path so far could have a high cost at the very end. That's why we want to paste g and h together – one to get rid of poor starts, and the other to get rid of poor finishes. In a nutshell, that is A*.

Summary: $f = g + h \rightarrow$ A* search $g=0 \rightarrow$ best-first search $g=1, h=0 \rightarrow$ breadth-first search g only $\rightarrow$ B&B

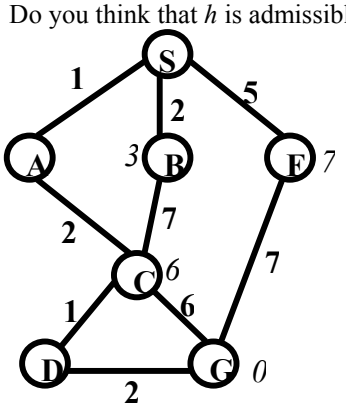
To check your understanding of this *f, g, h* business along with a single step of A*, consider the graph below, and suppose we have expanded S to nodes D, B, and C. Let's compute the *f* value at each, and note that A* will simply pick the node with the smallest *f* value to expand next.

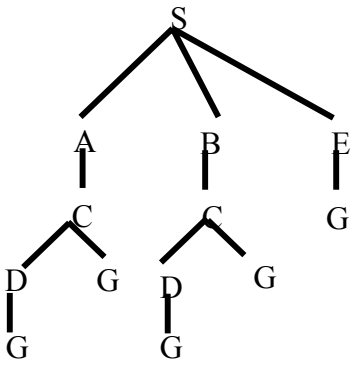


f(D)=
f(B)=
f(C)=

Which node does A* pick to expand?

Let's now try out all this stuff with an example search space, below. S is the Start node, G the Goal. Each directed arc between nodes has its cost in **bold,** also given as in the table below, along with estimates in *italics* of the distance from G from each node. Do you think that h is admissible?





Arc (link) lengths (costs):			
S-A	1	C-D	1
S-B	2	C-G	6
S-E	5	D-G	2
A-C	2		
B-C	7		
E-G	7		

Estimates of distance to G(oal) from:	
A	1
B	3
C	6
D	2
E	7
G	0
S	1

Problem 1: Let's first try branch-and-bound on this search tree – that is, use just *g* (cost so far), and write down partial paths in order of increasing accumulated length, starting from S and **not** writing down any path that is greater than one we already have. You just have to show the contents of the queue after each node extension step. To start, we expand S to get paths (A S), (B S), and (E S). The shortest is (A S), with length 1, so that's what goes on the queue first. [in the table below we don't show the other elements of the queue – you would typically have to keep these to decide which node to expand next. In our case, (B S) and (E S) have values 2 and 5 respectively. So we don't expand them yet, but (You do the rest – note how the lengths just increase as we go along.)

Step	**Partial Path**	**Length of path** [Queue]
1	**(S A)**	**1** **(1 S A) (2 S B) (5 S E)**
2	**(S ?)**	
3		
4		
5		
6		

Now let's do the same search using A*. This time, we will use $f=g+h$ as our 'goodness' metric, and place that value at the head of each partial path. A* then picks the lowest f value as the next node to extend. We keep track of nodes already examined by means of an 'extended list' giving the nodes in the path we're looking at (so that we don't duplicate work). Assume new nodes are added to the queue in left to right order, and that ties in the choice of which node to expand are broken by picking the node closer to the front of the queue. Do we get the same answer as with B&B? (Should we always?) Is *h* admissible? Why or why not?

STEP	**QUEUE**	**Extended list**	**Next node to extend**	Not extended
1	(1 S) [g=0, h=1]	S	(1 **S**)	
2	(?___ A)(?___ B)(?__E)			
3				
4				
5				
6				
7				

OK, now for some details as to why A* is so cool.
Here's a table taken from Russell & Norvig's AI book showing A* working on the 8-puzzle, for different 'board' sizes (2 x 2, 4 x 4, 8 x 8, 10 x 10, etc.) and two heuristics, the first the '# of tiles out of place,' the second the Manhattan distance. The 'effective branching factor' shows how close to 1, or 'perfect' these heuristics operate – 1.26 in the case of the second heuristic. Pretty darn close to perfect.

		Search Cost		Effective Branching		Factor
d	IDS	A*(h_1)	A*(h_2)	IDS	A*(h_1)	A*(h_2)
2	10	6	6	2.45	1.79	1.79
4	112	13	12	2.87	1.48	1.45
6	680	20	18	2.73	1.34	1.30
8	6384	39	25	2.80	1.33	1.24
10	47127	93	39	2.79	1.38	1.22
12	364404	227	73	2.78	1.42	1.24
14	3473941	539	113	2.83	1.44	1.23
16	–	1301	211	–	1.45	1.25
18	–	3056	363	–	1.46	1.26
20	–	7276	676	–	1.47	1.27
22	–	18094	1219	–	1.48	1.28
24	–	39135	1641	–	1.48	1.26

From Russell & Norvig, 1995; used with permission. And, to put that in perspective, if it takes a millisecond to process a node:

Depth	Nodes		Time	Memory	
0	1	1	millisecond	100	bytes
2	111	.1	seconds	11	kilobytes
4	11,111	11	seconds	1	megabyte
6	10^6	18	minutes	111	megabytes
8	10^8	31	hours	11	gigabytes
10	10^{10}	128	days	1	terabyte
12	10^{12}	35	years	111	terabytes
14	10^{14}	3500	years	11,111	terabytes

Definition of *complete* search algorithm: A search algorithm is **complete** if, given a search space that has a Goal state (node), that algorithm always finds a Goal state (node).

Definition of *optimal & optimally efficient:* A search algorithm is **optimal** if it always finds the optimal solution path (if one exists); it is **optimally efficient** if it expands the fewest nodes possible (does the least work) while finding an optimal solution (i.e., the first goal node or solution found is also the optimal one).

***Claim.** A* search using an admissible h is complete and optimal. (Assuming that the branching factor b is finite and that there is a fixed positive cost per node generator for next states.)

Proof.

Claim (i). A* is complete. Set $h(n) = 0$ and $g(n)=1$ for all n. Clearly h is admissible. Then A* is the same as BFS (why?), which is already known to be complete. Or, to see this another way: we can see this by observing that A* expands nodes in order of increasing f so it will reach the goal unless there are infinitely many nodes N with $f(N) < f^*$.

Claim (ii). A* is optimal. We can prove this by contradiction. Suppose we have 2 goal states G and G'. Suppose G is optimal (shortest, smallest) and G' is suboptimal, but the A* algorithm is about to select G'. Let the cost of the optimal path to G be denoted by f^*. Then the actual cost of the path to the suboptimal goal G' is $g(G') > f^*$. Consider an unexpanded node N on the optimal path to the goal G. Because h is admissible, it follows that $f^* \geq f(N)$. But since N is not chosen over G' it also follows that $f(N) \geq f(G')$. But then, by transitivity, $f^* \geq f(G') = h(G') + g(G') = 0 + g(G')$. But then $f^* \geq g(G')$, which contradicts our assumption that G' was suboptimal.

As for optimal efficiency, suppose C^* is the cost of the optimal solution path. A* expands **all** nodes with $f(n) < C^*$. A* might expand **some** of the nodes with $f(n) = C^*$ on the goal contour. A* will expand **no** nodes with $f(n) > C^*$ - these are pruned! No other optimal algorithm is guaranteed to expand fewer nodes than A*. An algorithm might miss the optimal solution if it does not expand all nodes with $f(n) < C^*$.

As to time and space complexity, A* must examine all the nodes on the goal contour, which might be exponential in number in the worse case. A* must keep **all** of the queue and the expanded nodes in memory – A* usually runs out of space long before it runs out of time.

But, even A* can be thwarted. In many cases, it takes too much memory. The space complexity can become exponential if the heuristic is not good enough.