9/22/11

MASSACHUSETTS INSTITUTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.034 Artificial Intelligence
Fall, 2011
Recitation 2, Thursday September 22

Search Me! Prof. Bob Berwick

0. Notation, algorithm, terminology

graph is a data structure containing node-to-node connectivity information
start is the starting node
goal is the target node

Here is a simple generic search algorithm:

```
function search(graph, start, goal)
0. Initialize
   agenda = [ [start] ]           # a list of paths
   extended_list  = [  ]          # a list of nodes
   while agenda is not empty:
         1. path = agenda.pop(0)  # remove first element from agenda
         2. if path is-a-path-to-goal then return path  # we've reached the goal
         3. otherwise extend the current path if last node in this path
            is not in the extended_list:
            3a. add the last node of the current path to the extended_list
            3b. for each node connected to this last node  # look-up using graph
            3c.     make a new path                   # don't add paths with loops!
         4. add new paths from step 3c to agenda and reorganize agenda
   end while
return nil path # failure
```

Notes

• Search returns either a successful path from start to goal or a nil path indicating a failure to find such a path.
• **Agenda** keeps track of all the paths under consideration, and the way it is maintained is *the key* to the difference between most of the search algorithms.
• **Loops in paths:** *Thou shall not create or consider paths with cycles in step 3.*
• **Exiting the search:** Non-optimal searches may actually exit when they find or add a path with a goal node to the agenda (at step 3). But optimal searches *must only* exit when the path is the first removed from the agenda (steps 1, 2).
• **Backtracking**: When we talk about depth-first search (DFS) or DFS variants (like Hill Climbing) we talk about with or without "backtracking". You can think of backtracking in terms of the agenda. If we make our agenda size 1, then this is equivalent to having no backtracking. Having agenda size > 1 means we have some partial path to go back on, and hence we can backtrack.
• **Extended list (or set)**: the list of nodes that have undergone "extension" (step 3). Using an extended list/set is an *optional* optimization that could be applied to all algorithms (some with implications, see A* search). In the literature the extended list is also referred to as the "closed" or "visited" list, and the agenda the "open" list.
• If we **do not** use an extended list, then the underlined parts above are **not** used.

Terminology

Informed vs. uniformed search: A search algorithm is informed if is some evaluation function ***f*(*x*)** that helps guide the search. Except for breadth-first search (BFS), DFS, and the British Museum algorithm, all the other searches we studied in this class are informed in some way.
Complete vs. incomplete: A search algorithm is complete if, whenever there exists a solution (path from start to goal), then the algorithm will find it.
Optimal vs. Non-optimal: A search algorithm is optimal if the solution found is also the best one (as determined by the path cost)

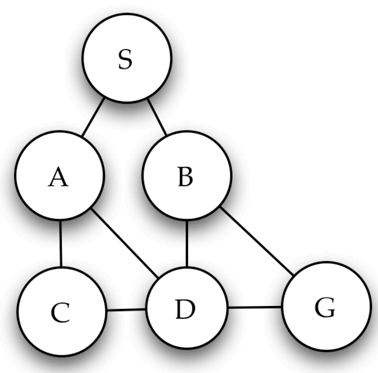
1. Now let's see how this works with the uninformed searches….

Search Algorithm	**Properties**	**Required Parameters**	**How the agenda is managed in step 4.**
Breadth-First Search (BFS)	Uninformed, Non-optimal (Exception: Optimal only if you are counting total path length), Complete		Add all new path extensions to the BACK of the agenda, like a **queue** (FIFO)
Depth-First Search (DFS)	Uninformed, Non-optimal, Incomplete if no backup; complete if backup		Add all new path extensions to the FRONT of the agenda, like a **stack** (FILO)
British Museum	Brutally exhaustive, Uninformed, Complete		Most likely implemented using a breadth-first enumeration of all paths

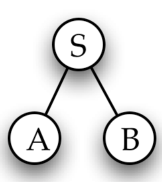
Let's try this out with the graph on the right, S= Start node; G= Goal node, for both BFS and DFS….

BFS: add path extensions to *back* of agenda. Node that is extended is in **bold.**

Step	agenda
1	[(*__S__*)]
2	[(*S __A__*), (*S B*)]
3	[(*S __B__*), (*S A C*), (*S A D*)]
4	[(*S A __C__*), (*S A D*), (*S B D*), (*S B G*)]
5	[(*S A __D__*), (*S B D*), (*S B G*), (*S A C D*)]
6	[(*S B __D__*), (*S B G*), (*S A C D*), (*S A D B*), (*S A D C*), (*S A D G*)]
7	[(*S B __G__*), (*S A C D*), (*S A D B*), (*S A D C*), (*S A D G*),(*S B D A*), (*S B D C*), (*S B D G*)]
8	Success - agenda.pop(0) has goal in path, (*S B G*)



(Note: we could have exited at Step 5 here, for non-optimal case, but for uniformity, exit as per code.)
Does adding an extended_list change anything in this example? (We will try it.)
Your turn – **DFS:** add extensions to *front* of agenda. You should start by expanding the graph as a tree…



Step	agenda
1	[(*__S__*)]
2	[(*S __A__*), (*S B*)]
3	[(*S A __C__*), (*S A D*), (*S B*)]
4	[(*S A C __D__*), (*S A D*), (*S B*)]
5	[(*S A C D __B__*), (*S A C D G*), (*S A D*), (*S B*)]
6	[(*S A C D B __G__*), (*S A C D G*), (*S A D*), (*S B*)]
7	Success - agenda.pop(0) has goal in path, (*S A C D B G*)

9/22/11

2. Informed search definitions – moving towards optimal search

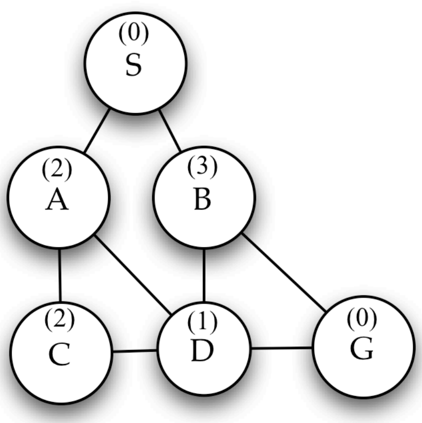
$f(x)$ = the total cost of the path that your algorithm uses to rank paths.
$g(x)$ = the cost of the path so far.
$h(x)$ = the (under)*estimate* of the remaining cost to the goal *g* node (Use *h* for 'heuristic')
$f(x) = g(x) + h(x)$
$c(x, y)$ is the *actua*l cost to go from node *x* to node *y*.

"Heuristics, Patient rules of thumb,
So often scorned: Sloppy, Dumb!
Yet, Slowly, common sense come" – Ode to AI

Search Algorithm	**Properties**	**Required Parameters**	**How the agenda is managed in step 4.**
Hill Climbing	Non-optimal, Incomplete unless backtracks Like DFS with a heuristic	$f(x)$ to sort newly added paths (usually, this is just h)	1. Keep only <u>newly-added</u> path extensions sorted by $f(x)$ 2. Add sorted <u>new</u> paths to the FRONT of agenda
Best-First Search	Depends on definition of $f(x)$ If $f(x) = h(x)$ (estimated distance to goal) then likely not optimal, and potentially incomplete. However, A* is a type of best-first search that is complete and optimal because of its choice of $f(x)$ which combines $g(x)$ and $h(x)$ (see below)	$f(x)$ to sort the entire agenda by.	Keep <u>entire</u> agenda sorted by $f(x)$

2.1 Hill-climbing (with backup)

Add path extensions (*sorted by heuristic value*) to the *front* of agenda.
Heuristic value is a measure of the 'goodness' of the path, e.g., an *estimate* of how far remaining to go, as the crow flies; or in some other terms if not a map. (We will see this a bit later how to work this into optimal search.)
Note that hill-climbing only looks at the next *locally* best step (*not* over *all* paths!).
(Below we tack the heuristic value to the front of the list, to keep track; note sorting.)
Our graph now has heuristic values that label each node, in parentheses inside the node:



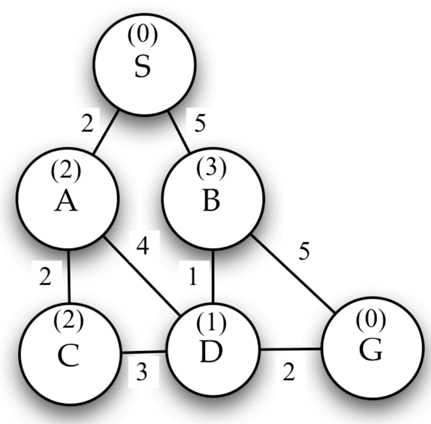
Step	agenda
1	[(0 ***S***)]
2	[(2 *S* ***A***), (3 *S B*)]
3	[(1 *S A* ***D***), (2 *S A C*), (3 *S B*)]
4	[(0 *S A D* ***G***), (2 *S A D C*), (3 *S A D B*), (2 *S A C*), (3 *S B*)]
5	Success - agenda.pop(0) has goal in path, (*S A D G*)

2.2 Optimal search methods

Search Algorithm	**Properties**	**Required Parameters**	**How the agenda is managed in step 4.**
Branch & Bound (B&B)	Optimal Like best-first except uses actual path costs	$g(x) = c(s, x)$ = the cost of path from s to node x. $f(x) = g(x) + 0$	Sort paths by $f(x)$ = $g(x)$=total path cost so far)
A* w/o extended list (or B&B w/o extended list + admissible heuristic)	Optimal if h is **admissible**	$f(x) = g(x) + h(x,g)$ $h(x,g)$ is the estimate of the cost from x to g. $h(x)$ must be an **admissible** heuristic	Sort paths by f(x)
A* with extended list	Optimal if h is **consistent**	$f(x) = g(x) + h(x)$ $h(x)$ must be a **consistent** heuristic	Sort paths by f(x)

2.3 Now let's try Branch & Bound, using $f(x) = g(x) + 0 = g(x)$ = cost of path so far to sort the agenda. We just pay attention to the numbers on the path links, *not* the 'heurisic' numbers in parentheses at each node. Let us also use an extended list.

Step	agenda	Extended
1	[(0 ***S***)]	{ }
2	[(2 *S* ***A***), (5 *S B*)]	{*S*}
3	[(4 *S A* ***C***), (5 *S B*), (6 *S A D*)]	{*S, A*}
4	[(5 *S* ***B***), (6 *S A D*), (7 *S A C D*)]	{*S, A, C*}
5	[(6 *S A* ***D***), (6 *S B D*), (7 *S A C D*), (10 *S B G*)]	{*S, A, B, C*}
6	[(6 *S B* ***D***), (7 *S A C D*), (8 *S A D G*), (10 *S B G*)]	{*S, A, B, C, D*}
7	[(8 *S A D* ***G***), (10 *S B G*)]	{*S, A, B, C, D, G*}
8	Success - agenda.pop(0) has goal in path, (8 *S A D G*), optimal	{*S, A, B, C, D, G*}



You can see here how B&B characteristically explores paths in order of *monotonically increasing* path length so far. Note that B&B is really finding the optimal path to <u>each</u> node in the graph. It is not 'biased' in the direction of the goal node. We will need to add in a heuristic function h to do that.

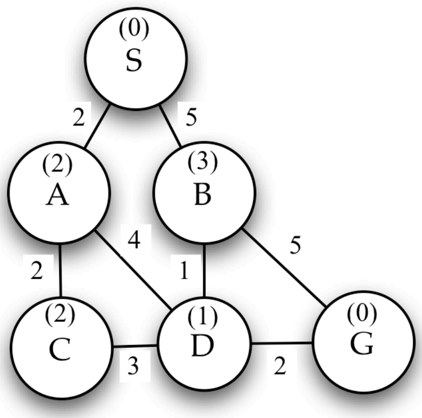
Note also how the extended list comes into play at Step 6 (how?).
(Note also the 'tie' in step 5…what about that? In quizzes you will always be instructed about how to break such ties.)

2.4 A* search = B&B + admissible heuristic

The main idea of A* is to avoid expanding paths that are already expensive. We use the evaluation function $f(n)=g(n)+h(n)$. We sort the entire agenda by this value, and pick the best path to work on next.

.

OK, let's try this. Now for f at a node we compute the **sum** of the path-length-so-far **plus** the h value at that node, the value in parentheses. For instance, the f value at node A, given that we start from S, is 2+2 = 4; for node B it is 5+3=8. We also use an extended list.

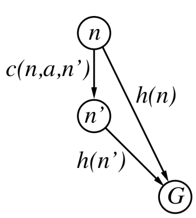


Step	Agenda	Extended
1	[(0 ***S***)]	{ }
2	[(4 *S* ***A***), (8 *S B*)]	{*S*}
3	[(6 *S A* ***C***), (7 *S A D*), (8 *S B*)	{*S, A*}
4	[(7 *S A* ***D***), (8 *S B*), (8 *S A C D*)]	{*S, A, C*}
5	[(8 *S* ***B***), (8 *S A C D*), (8 *S A D G*), (10 *S A D B*)]	{*S, A, C, D*}
6	[(7 *S B* ***D***), (8 *S A C D*), (8 *S A D G*), (10 *S A D B*), (10 *S B G*)]	{*S, A, B, C, D*}
7	[(8 *S A D* ***G***), (10 *S A D B*), (10 *S B G*)] Success! (8 *S A D G*)	{*S, A, B, C, D, G*}

Note that if the heuristic values at *S* and *D* were *S*=10 and *D* = 4, these would be <u>inadmissible</u> because, e.g., 4 is an <u>overestimate</u> of the remaining distance to the goal, 2. So the *h* value at *D* must be ≤ 2, and similarly the *h* value at *S* must be ≤ 8 to be admissible (in fact, for all node values, *h*≤ 8). Why is this important? Suppose an *h* value is inadmissible, say, 10^6 at some node. Then A* could fail: a path through that node will never get worked on, even though the *actual* path length through that node might be the optimal one.
Admissibility is a constraint that must hold between *every* node and the goal node. There is another, stronger constraint that is sometimes easier to check, that implies admissibility, namely, *consistency*, which amounts to the triangle inequality. This ensures that $f(n)$ is non-decreasing along *any* path, and it *must* hold if we are using A* with an extended list, as we will see below. (However, admissibility does **not** imply consistency, so this is not a bi-conditional.)

Definition: A heuristic is *consistent* if, for every node n, every successor node n' of n satisfies the following condition:

$$h(n) \le c(n,a,n') + h(n')$$



So if h is *consistent*, we have:

$$\begin{aligned} f(n') &= g(n') + h(n') && \text{[by dfn of } f\text{]} \\ &= g(n) + c(n,a,n') + h(n') \\ &\ge g(n) + h(n) = f(n) && \text{[substituting for dfn of consistent } h \text{ \& dfn of } f\text{]} \end{aligned}$$

So $f(n)$ is non-decreasing along any path. This is the same condition that B&B obeyed (since it uses actual costs or path values it holds for B&B).

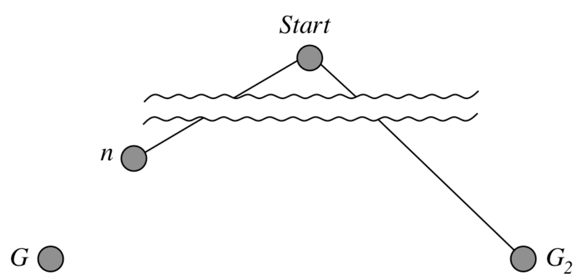
Question: is the search graph above *consistent*? (Hint: Look at paths from S to B to node D and calculate the f values, to see if they are non-decreasing, or look at what happened between Steps 5 and 6 above.) If a graph is *inconsistent* and we are using A* with an extended list, then A* might fail: consider what would happen above if the *S*-*B* link were of length 4 instead of 5.

Properties of A*

1. A* is <u>complete</u> unless there are infinitely many nodes s.t. $f \leq f(G)$
2. Time: Exponential in [relative error in h x length of solution path]
3. Space: Keeps all nodes in memory (the dark side of A*, usually runs out of memory)
4. Optimal: Yes, cannot expand f_{i+1} until f_i is finished

A* expands <u>all</u> nodes with $f(n) < C^*$, where C* is the optimum cost/distance
A* expands <u>some</u> nodes with $f(n) = C^*$
A* expands <u>no</u> nodes with $f(n) > C^*$

3.1 Enrichment portion 1: Optimality of A*

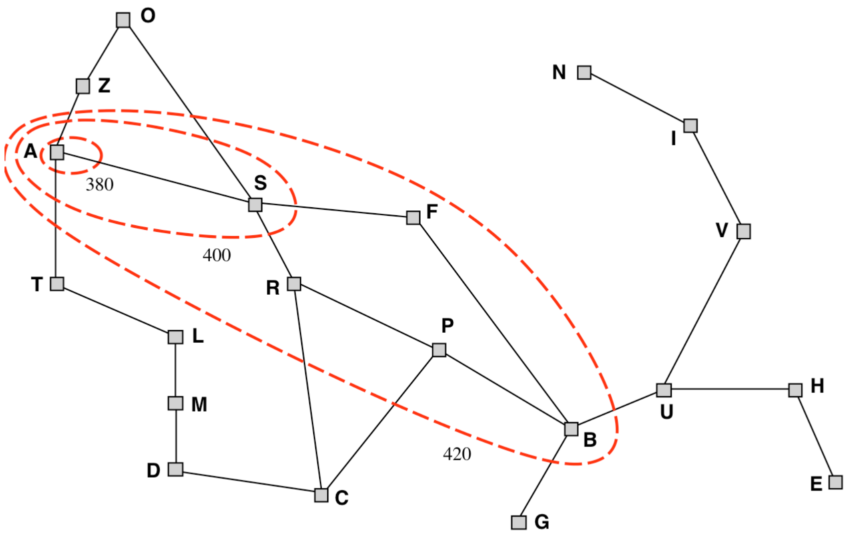


Suppose the algorithm generates some suboptimal goal G_2 and is in the fringe, as in the picture. Let n be an unexpanded node in the fringe such that n is on a shortest path to the optimal goal G. Then:

(1) $f(G_2) = g(G_2)$ since $h(G_2)=0$
(2) $g(G_2) > g(G)$ since G_2 is suboptimal
(3) $f(G) = g(G)$ since $h(G) = 0$
(4) $f(G_2) > f(G)$ from (1), (2), (3)
(5) $h(n) \leq h^*(n)$ since h is admissible
(6) $g(n) + h(n) \leq g(n) + h^*(n)$
(7) $f(n) \leq f(G)$ by dfn of $f(G)$ as $g(n) + h^*(n)$
But then:
(8) $f(G_2) > f(n)$ by (4) and (7), so A* will <u>never</u> select G_2 for expansion. QED.

Another picture, possibly more helpful (see properties of A*). A* expands in terms of increasing f values (like B&B), directed along contours 'pointing' towards the goal.

9/22/11



3.2 Enrichment 2: Cost and Performance of Various Search Strategies; Iterative Deepening Search

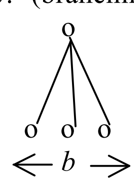
(branching factor = b, depth = d)
Worst case time = proportional to # nodes visited
Worst case space= proportional to maximum length of Q

Search Strategy	Worst Time	Worst Space	Fewest Nodes?	Guaranteed to find path?
Depth-first (with backup)	b^{d+1}	bd	No	Yes
Breadth-first	b^{d+1}	b^d	Yes	Yes
Hill-Climbing (no backup)	d	b	No	No
Hill-Climbing (with backup)	b^{d+1}	bd	No	Yes
Best-first	b^{d+1}	b^d	No	Yes
Beam (beam width k, no backup)	kd	kb	No	No

How could we combine the space efficiency of DFS with BFS? (BFS guaranteed to find path to goal with minimum number of nodes.) Answer: Iterative Deepening Search (IDS) – search DFS, level by level, until we run out of time. Let's see.

Counting Nodes in a Tree

Why is $(b^{d+1} – 1)/(b-1)$ the number of nodes in a tree? (branching factor = b, depth = d)
If each node has b immediate descendents:



Then Level 0 (the root) 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:

$$N = 1 + b + b^2 + b^3 + b^4 + \ldots + b^d$$
$$bN = b + b^2 + b^3 + b^4 + \ldots + b^d + b^{d+1}$$

Subtracting:

9/22/11

$(b - 1)N = b^{d+1} - 1$

$$N = \frac{b^{d+1} - 1}{b - 1}$$

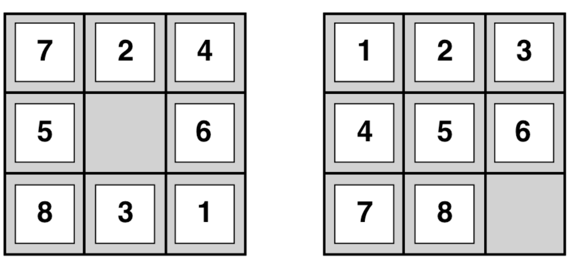
So we could do this to implement Iterative Deepening Search (IDS):

1: Initialize D_{max}=1. (The goal node is of unknown depth d)
2: **Do**
3:    DFS from S for fixed depth D_{max}
4:    **If** found a goal node, depth $d \leq D_{max}$ **then exit**
5:    $D_{max} = D_{max} + 1$

Cost is: $O(b^1+b^2+\ldots+b^L)=O(b^L)$ where L= length to goal. But isn't IDS wasteful because we repeat searches on the different iterations? No. For example, suppose b=10 and d=5. Then the total # number of nodes N we look at for in each case is:
N(IDS) ≈ $db+ (d–1)b^2+ \ldots + b^5$ = 123,450, while for BFS the # of nodes is approximately,
N(BFS) ≈ $b + b^2 + \ldots + b^5$ = 111,110, or only about 10% less. Most of the time is spent at depth d. So, IDS is asymptotically optimal; because 'most' of the time is spent in the fringe of the search tree. It is the preferred method over BFS, DFS when the goal depth is unknown.

Similarly, for A* search, in order to avoid HUGE memory costs, one will often use IDA*, i.e., iterative deepedning A*.

4. The value of good heuristics: the 8 puzzle



Start State **Goal State**

What is a 'legal move;?
What would be a good heuristic h for this puzzle? Note that even iterative deepening search is costly; if # tiles is 14, then IDS typically searches 3,473,941 nodes. If # tiles is 24, then this is about 54,000,000,000 nodes.

Two suggested heuristics, h_1=7; h_2= distance in terms of vertical + horizontal squares out of place from goal
The first is called: # tiles out of place.

The second is called: Manhattan distance

Question: can you guess what happens to the efficiency of search if it's always the case that $h_2(n) \geq h_1(n)$ for all n? (Both heuristics admissible). Why do you think this? What does this say about *how* to choose a heuristic?