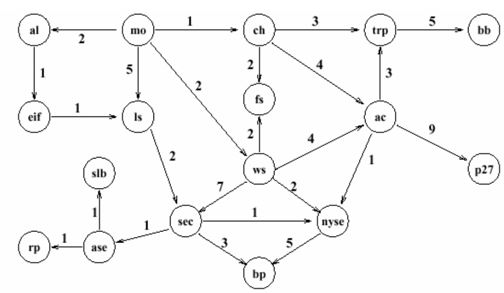
CSC384: Lecture 5

- Last time
  - search, DFS & BrFS; cycle checking & MPC
- Today
  - arc costs; heuristics; LCFS, BeFS, A*
  - misc: iterative deepening, etc.
- Readings:
  - Today: Ch.4.5, 4.6
  - Next Weds: class notes (no text reading)

Manhattan Bike Courier (Acyclic)



Arc Costs

- DFS/BrFS make sense when no arc costs
  - e.g., BrFS ensures shortest path (fewest arcs)
- If arc costs & **aim of finding least-cost path**, BFS in not suitable
  - e.g., goal=ls, start=mo: BrFS finds shortest path [ls,mo] with cost 5; but least-cost path is [ls,eif,al,mo] with cost 4 (even though it has more arcs)
- **Least-cost first search (LCFS)** : least cost path
  - works much like BrFS, except paths are ordered according to cost, rather than “length”

Least-cost First Search

- Implementing LCFS is straightforward
- Let cost of any path *p* to node *n* be denoted *g(n)*
  - note: this notation is misleading but conventional
- Organize frontier as a priority queue
  - with each path on frontier, attach cost *g(n)*
  - paths with lower cost are at the head of the frontier
  - new paths (nbrs) are inserted in order of cost
  - so *add_to_f* is just priority queue insertion
- Selecting a path from the head of the frontier
  - thus, you always get least cost path from the frontier

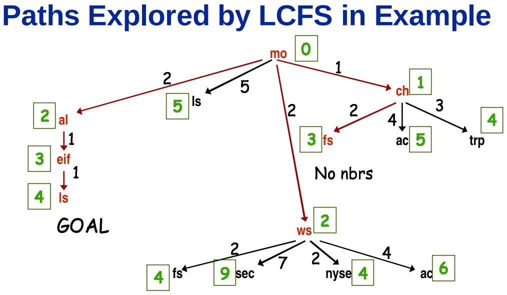
Trace of LCFS (with paths: *mo* to *ls*)

Frontier evolution:

1. [mo]:0
2. [ch,mo]:1 [al,mo]:2 [ws,mo]:2 [ls,mo]:5
3. [al,mo]:2 [ws,mo]:2 [fs,ch,mo]:3 [trp,ch,mo]:4 [ac,ch,mo]:5 [ls,mo]:5
4. [ws,mo]:2 [eif,al,mo]:3 [fs,ch,mo]:3 [trp,ch,mo]:4 [ac,ch,mo]:5 [ls,mo]:5
5. [eif,al,mo]:3 [fs,ch,mo]:3 [fs,ws,mo]:4 [myse,ws,mo]:4 [trp,ch,mo]:4 [ac,ch,mo]:5 [ls,mo]:5 [ac,ws,mo]:6 [sec,ws,mo]:9
6. [fs,ch,mo]:3 [ls,eif,al,mo]:4 [fs,ws,mo]:4 [myse,ws,mo]:4 [trp,ch,mo]:4 [ac,ch,mo]:5 [ls,mo]:5 [ac,ws,mo]:6 [sec,ws,mo]:9
7. **[ls,eif,al,mo]:4** [fs,ws,mo]:4 [myse,ws,mo]:4 [trp,ch,mo]:4 [ac,ch,mo]:5 [ls,mo]:5 [ac,ws,mo]:6 [sec,ws,mo]:9

Goal found after 7 node expansions; least-cost path to ls

Paths Explored by LCFS in Example



Red paths: expanded
Black paths: added to frontier, but not expanded

Properties of LCFS

- Guaranteed to find least-cost path under certain circumstances
- If all arc costs are greater than 0 (assume a solution exists)
  - exercise: prove it will find least-cost path
  - what can happen if we have negative arc costs?
- Space and time complexity similar to BrFS
  - note: BrFS is a special case of LCFS when all arc costs are “uniform” (e.g., all arc costs are 1)

Uninformed Search Strategies

- For any search strategy so far (DFS, BFS, LCFS) suppose I give you goal *g1* and ask you to trace the paths explored. Then I change the goal to *g2* and ask you to repeat the process.
- Both traces will look the same (up to the point that the goal is found)
- These search strategies are *blind* or *uninformed*
  - search process in uninfluenced by the goal
  - e.g., in LCFS (goal=ls), first step is toward ch
  - e.g., Craig often turns right at red lights no matter what direction he’s heading

CSC 384 Lecture Slides (c) 2002, C. Boutilier

Heuristics

- **Heuristics** generally refer to any rules of thumb that provide some help when solving a problem
  - e.g., an estimate/guess as to best way to proceed
  - generally guidance is not perfect
- In graph search, a *heuristic function* h(n) is an estimate of cost to goal g from node n
  - Why an estimate? What if h(n) were perfect?
  - Exercise: prove that if h(n) is true cost to goal for each n, you can find best path without backtracking
  - Note: h(n) will vary with goal g; so we sometimes write $h(n,g_1)$, $h(n,g_2)$, etc. for emphasis

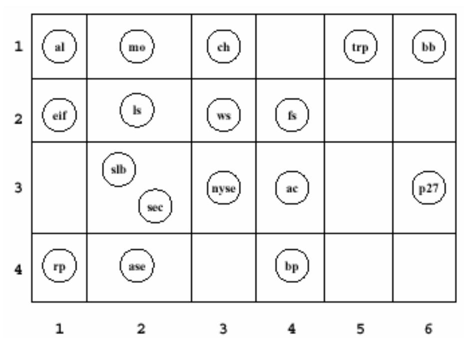
CSC 384 Lecture Slides (c) 2002, C. Boutilier

Good Heuristics

- Where do heuristics come from?
  - depends on the problem we’re trying to solve
  - planning? we’ll look at some
  - chess? rules of thumb about board position (vulnerability, number of pieces, etc.)
  - Manhattan bike courier? see handout of “grid”
- Features of a good heuristic function
  - should be somewhat accurate
  - should be easy to compute (e.g., if it requires lots of search, that defeats the purpose!)
  - should underestimate true cost (for reasons we’ll see)

CSC 384 Lecture Slides (c) 2002, C. Boutilier

Heuristic for MBC



CSC 384 Lecture Slides (c) 2002, C. Boutilier

Heuristic for MBC (see handout)

For instance, if our Goal location was `slb`, we could represent our heuristic function directly as follows:

```
h(mo, 2).  h(slb, 0).  h(trp, 5).  h(sec, 0).  h(fs, 3).  h(ch, 3).
h(bb, 6).  h(ws, 2).   h(eif, 2).  h(nyse, 1). h(ac, 2).  h(rp, 2).
h(al, 3).  h(p27, 4).  h(ase, 1).  h(ls, 1).   h(bp, 3).
```

A generic heuristic for arbitrary goals h(n,g):

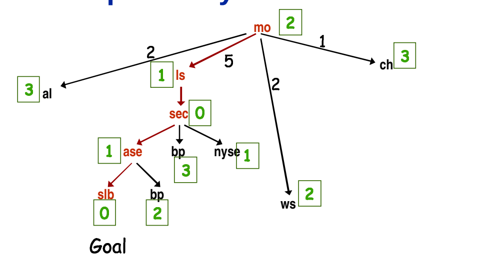
```
md(Loc,G,D) :- coord(G,X1,Y1), coord(Loc,X2,Y2), dist(X1,Y1,X2,Y2,D).

dist(X1,Y1,X2,Y2,D) :- dist2(X1,X2,X), dist2(Y1,Y2,Y), D is X+Y.

dist2(X1,X2,Z) :- X1 >= X2, Z is X1-X2.
dist2(X1,X2,Z) :- X1 < X2, Z is X2-X1.

coord(al,1,1).  coord(mo,1,2).  coord(ch,1,3).  coord(trp,1,5). etc...
```

CSC 384 Lecture Slides (c) 2002, C. Boutilier

Best-first Search (BeFS)

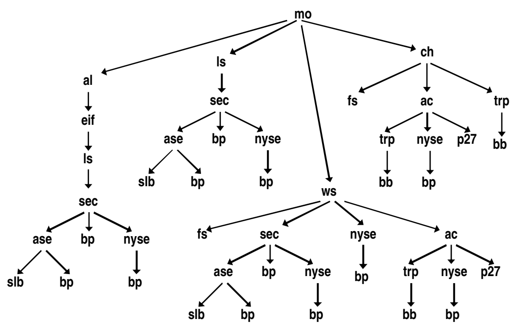
- We can use heuristics to guide search in heuristic DFS (see text), best-first search, A*
- **Best-first search** works just like LCFS except we attach h(n) to each path instead of g(n)
  - i.e., priority queue sorts paths based on h(n) value
  - we explore paths whose end points *appear to be closest to the goal* (according to h)

Paths Explored by BeFS: *mo* to *slb*



Red paths: expanded
Black paths: added to frontier, but not expanded

Search Tree: MBC Acyclic; Start *mo*



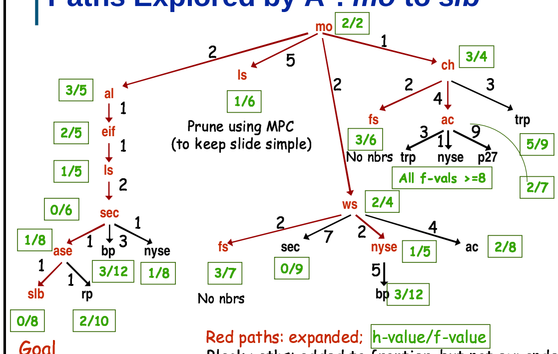
Problem with BeFS

- In previous example, BeFS guides us *very* directly to a path to slb (in fact, *no* backtracking)
- Unfortunately, not the least-cost path
- Indeed, BeFS ignores arc costs altogether!
  - chooses path to expand based only on estimated cost-to-go, h(n), and is uninfluenced by cost of path so far g(n)
  - makes sense if you've already "gone" to the node, but not if you're searching for the shortest path

A* Search

- A* search combines aspects of LCFS and BeFS
  - we use both h(n) and g(n) when choosing paths
- Quality of path on frontier is given by the **evaluation function**: f(n) = g(n) + h(n)
- Paths are ordered on the frontier according to f-value f(n)
  - if expanded path is not a soln, it is extended by its neighbors; which are inserted according to f-values
  - always select path from frontier with minimal f-value
  - Implementation: priority queue sorted on f-value

Paths Explored by A*: *mo* to *slb*

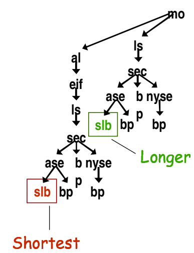


A* Analysis

- In this example, A* leads pretty directly to the goal *slb*
  - it expands six "false leads" and "prunes" one more
- A* also found the least-cost path to *slb*
- Seems to combine the best of LCFS (best path) and BeFS (goes fairly directly to the goal)
- Space and time complexity similar to BrFS
  - note: BrFS and LCFS are special cases of A* (under what conditions?)

Properties of A* (Informally)

- Will A* always find shortest path?
- Not necessarily:
  - suppose *h(al) = 17* in our example?
  - this very misleading (and pessimistic!) estimate of cost-to-go from *al* means it won't get expanded before *[ls, mo]*
  - will find longer path to *slb*

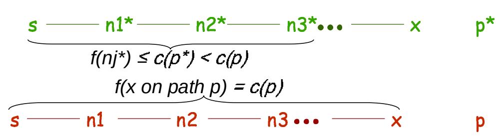


Admissible Heuristics

- Suppose h(n) never overestimates the true cost-to-goal from n?
  - A* will find least-cost path (assuming arcs costs > 0)
  - a heuristic s.t. $h(n) \leq mincost(n,g)$ is **admissible**
  - our example heuristic turns out to be admissible
- Special case: let *h(n) = 0* for all *n*
  - since *f(n) = h(n) + g(n) = g(n)*: reduces to LCFS
  - an admissible, but uninformative heuristic
- In general, the more "informative" *h(n)* is, the better A* will perform (more "direct" search)
  - Exercise: Prove that if *h(n) = mincost(n,g)* – that is, *h(n)* is perfect – A* will find optimal path directly (no backtracking)

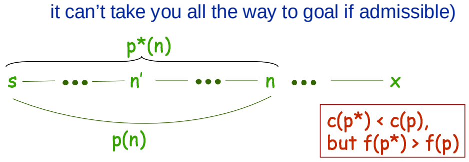
Optimality of A* (Intuitions)

- Assume admissible heuristic h
  - Let p be a nonoptimal path to goal x with cost c(p)
  - Let p* be optimal path to goal x with cost c(p*) < c(p)
  - Note: every subpath q of p* has f-value ≤ c(p*) < c(p) since h is admissible
  - So every such path—including p* -- will be expanded (removed from frontier) before p
  - Note: *some* subpaths of p can be expanded, but not p

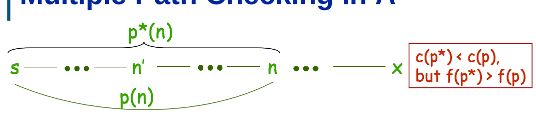


Multiple Path Checking in A*

- MPC: If you find a path to node n that you've already expanded, don't expand it again
  - was OK for BFS and LCFS, since first path expanded to any node n was assured to be shortest/cheapest
  - In A*, you can be misled by heuristic that takes you all the way to node *n* along an "expensive path" (though it can't take you all the way to goal if admissible)



Multiple Path Checking in A*



- In example, p expanded before p*, and MPC ignores shorter path p* to node n
  - MPC can destroy optimality of A*
- But this can only happen if:
  - some n' on p* is on frontier, with $f_{p*}(n') > f_p(n)$
- But $g_{p*}(n') + dist(n',n) < g_p(n)$
- So we must have h(n') > h(n) + dist(n,n')
  - thus h(n') makes n' look worse than n by more than the actual distance it takes to get from n' to n
  - this can happen even if h is admissible: basically it means heuristic is too optimistic about n relative to n'

The Monotone Restriction

- Can insist h satisfy the **monotone restriction:**

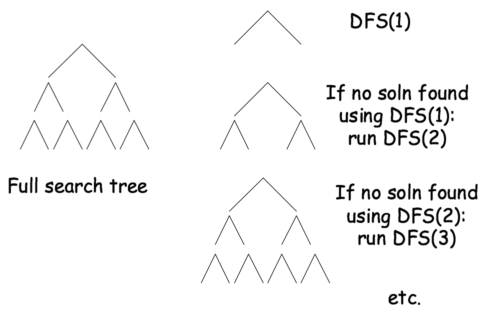
  $|h(n,) - h(n)| \le d(n',n)$ *for all nodes n, n'*

- This is enough to ensure that MPC can be performed safely with A* (i.e., MPC will preserve optimality)

Iterative Deepening (IDS)

- IDS is motivated by the following tension:
  - BFS guarantees optimal soln, requires expnt'l space
  - DFS requires linear space, can't guarantee optimality
  - How can we get best of both worlds?
- Trick: add a depth bound d to DFS
  - normal DFS, but never expand path with length > d
- How do I ensure I find solution if one exists?
  - if failure at depth bound d, increase bound and repeat
- How do I ensure shortest path is found first?
  - use the depth bounds: d=1, d=2, d=3, d=4, etc.

Iterative Deepening Graphically



Properties of IDS

- Guaranteed to find shortest solution
- Will only use linear space:
  - O(db) space with depth bound d, branching factor b
  - Important: do *not* "save" results from previous iteration
- How do we get this benefit?
  - we're repeating computation!
  - At depth bound d, we repeat all computation done at all earlier depth bounds. The only "new" steps are the expansion of leafs from previous iteration
- Why redo? Why not store previous tree?
  - requires exponential space

What Price do We Pay?

- IDS seems silly: a lot of wasted effort it seems!
  - but how bad is it compared to BFS?
  - Assume shortest soln has length d
- BFS generates:

  $b^d + b^{d-1} + b^{d-2} + \ldots + b^0 = O(b^d)$ nodes

- IDS generates:

  $b^d + 2\, b^{d-1} + 3\, b^{d-2} + \ldots + d\, b^0$ nodes

  which is roughly $b^d (1-1/b)^{-2} = O(b^d)$ nodes

Benefit of IDS

- We pay a **constant** time overhead (compared to BFS) for **exponential** space savings!
- Note: constant factor $(1-1/b)^{-2}$ is pretty small
  - if b = 2, overhead factor is 4 (4 times as long as BFS)
  - if b = 4, overhead factor is 1.8
  - overhead factor decreases with b!
- Iterative Deepening can be used with A*: **IDA***
  - basically, do DFS, but let "depth bound" be maximum f-value you consider, and increase f-value-bound gradually

Implicit Search Graphs

- Most search problems are not specified with explicit search graphs; nbr predicate "creates" neighboring states *on the fly*
  - chess, SLD-derivations, planning robot activity, etc.
- Example: 8-puzzle
  - Each board position a state
  - 9! = 362880 states
  - each state has 2, 3, or 4 nbrs
  - nbrs correspond to possible moves
  - nbr predicate: returns list of states reachable
- State Representation? Neighbor implementation? Possible Heuristics? see assignment 2!

1	2	3
8		4
7	6	5

Other Issues

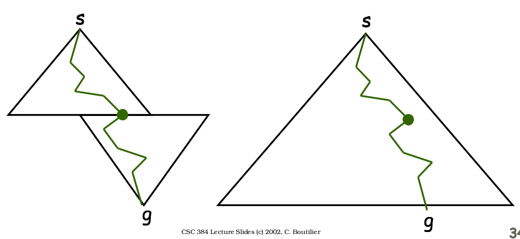
- Suppose list of neighbors is too large:
  - to add to frontier? to calculate all heuristic values?
  - What might one do? How could you use heuristic info to limit your attention?
  - One possibility: *generate* neighbors in heuristic order (only a subset of nbrs ever put on frontier)
  - can destroy optimality unless more nbrs added when backtracking
- Other things we can do to increase efficiency?
  - control the direction of search

Backward Search

- *Backward* branching factor is the (avg) set of moves that can be made *to* a specific node
  - if I have the inverse nbr relation available, I can search in the graph backwards from the goal to the start state
- Advantage: if backward BF *b-* less than forward BF *b+,* then search algth'm (any type) benefits
  - examples: planning (as we'll see later)
  - lower time and space complexity since optimal path length still the same
  - heuristic methods need a *backwards* heuristic, though

Bidirectional Search

- Search simultaneously in both directions
  - if two frontiers intersect, you can "join" forward and backward paths to node in intersection to get a sol'n
  - contrast # expansions for b-d BrFS vs. normal BrFS



Bidirectional Search

- Suppose we do BrFS
  - length of sol'n (shortest path) is k
  - branching factor (frwd/bkwd) is b
- Each component of the bidirectional search expands $O(b^{k/2})$ nodes
- Normal BrFS expands $O(b^k)$ nodes
- Bidirectional is exponential, but offers exponential savings
- Issues: need bkwd dynamics, need to test intersection, must choose search alg. carefully

Island Search

- Suppose you know that any (good) path to goal must pass through *island states* $i_1, i_2, \ldots i_k$
  - e.g., must pass through specific tunnels to deliver pkg
- Complexity can be cut significantly by searching for path from s to i_1, i_1 to i_2, …, i_{k-1} to i_k, i_k to g
  - what is potential savings (say) for BrFS using this strategy if avg subpath between islands has length m?