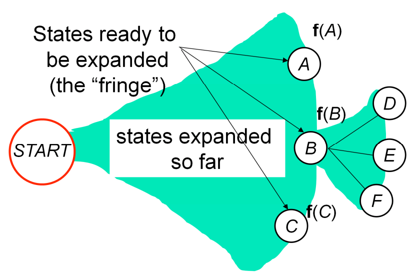
Lecture 3: Informed (Heuristic) Search

- Best-First (Greedy) Search
- Heuristic Search
- A^* search
- Proof of optimality of A^*
- Variations: iterative deepening, real-time search, macro-actions

Recall from last time: General search



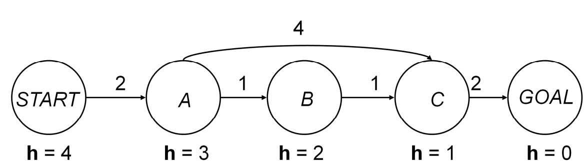
- Problems are described through states, operators and costs
- For each state and operator, there is a set of *successor states*
- In search, we maintain a set of *nodes*, each containing a state and other info (e.g. cost so far, pointer to parent etc)
- These nodes form a *search tree*
- The fringe of the tree contains *candidate nodes*, and is typically maintained using a *priority queue*
- Different search algorithms use different priority functions f

Uninformed vs. informed search

- Uninformed search methods expand nodes based on the distance from the start node. Obviously, we always know that!
- *Informed search methods* also use some estimate of the distance to the goal *h(n)*, called a *heuristic*.
- If we knew the distance to goal exactly, it would not even be "search" - we could just be greedy!
- But even if we do not know the exact distance, we often have some intuition about this distance:
  - The straight line between two points, in a navigation problem
  - The number of misplaced tiles in the 8-puzzle
- The heuristic is often the result of thinking about a *relaxed* version of the problem.

Best-First Search

- Algorithm: At any time, expand the most promising node according to the heuristic
- This is roughly the "opposite" of uniform-cost search
- Example:



At node A, we choose to go to node C, because it has a better heuristic value, instead of not B, which is really optimal

Properties of best-first search

- Time complexity: $O(b^d)$ (where b is the branching factor and d is the depth of the solution)
- If the heuristic is always 0, best-first search is the same as breadth-first search - so in the worst-case, it will have exponential space complexity
- However, depending on the heuristic, the expansion may look a lot like depth-first search - so space complexity may look like $O(bd)$.
- Like DFS, best-first search is *not complete in general*
  - Can go on forever in infinite state space
  - In finite state space, can get stuck in loops unless we use a closed list
- *Not optimal!* (as seen in the example)
- Best-first search is a *greedy method*.

  Greedy methods maximize short-term advantage without worrying about long-term consequences.

Fixing greedy search

- The problem with best-first search is that it is too greedy: it does not take into account the cost so far!
- Let g be the *cost of the path so far*
- Let h be a *heuristic* function (estimated cost to go)
- *Heuristic search* is a best-first search, greedy with respect to

$$f = g + h$$

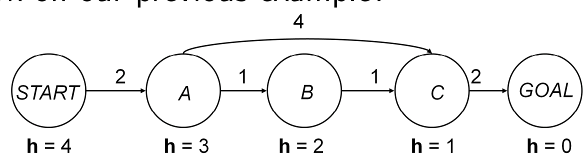
- Important insight: $f = g + h$ as an *estimate of the cost of the current path*

Heuristic Search Algorithm

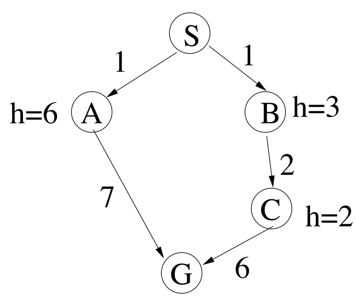
- At every step:
  1. Dequeue node n from the front of the queue
  2. Enqueue all its successors n' with priorities:

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= \text{cost of getting to } n' + \text{ estimated cost from } n' \text{ to goal} \end{aligned}$$

  3. Terminate when a goal state is popped from the queue.
- Does this work on our previous example?

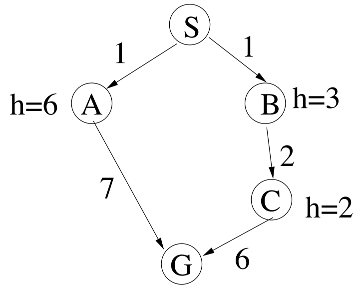


Example



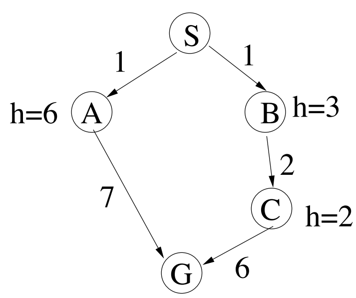
Priority queue: (S,h(S))

Example



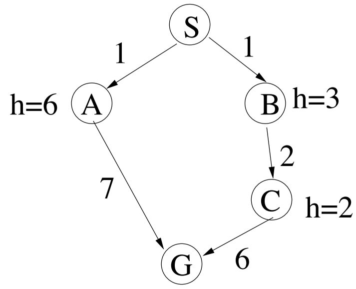
Priority queue: ((B,4),(A,7))

Example



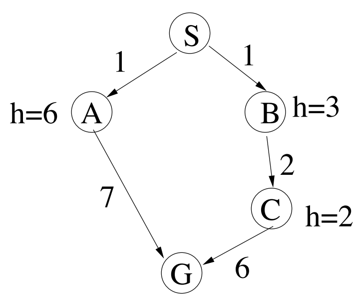
Priority queue: ((C,5),(A,7))

Example



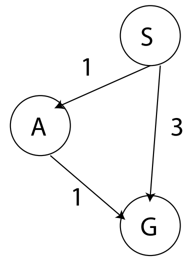
Priority queue: ((A,7), (G,9))

Example



Priority queue: ((G,8), (G,9)) $\Rightarrow$ the optimal path through A is found!

Does heuristic search always give the optimal solution?



- Whether the solution is optimal *depends on the heuristic*
- E.g., in the example above, any value of $h(A) \geq 3$ will lead to the discovery of a suboptimal path
- Can we put conditions on the choice of heuristic to guarantee optimality?

Admissible heuristics

- Let $h^*(n)$ be the shortest path from n to any goal state.
- Heuristic h is called *admissible* if $h(n) \leq h^*(n) \forall n$.
- Admissible heuristics are *optimistic*
- Note that if h is admissible, then $h(g) = 0, \forall g \in G$
- A trivial case of an admissible heuristic is $h(n) = 0, \forall n$.
  - In this case, heuristic search becomes uniform-cost search!

Examples of admissible heuristics

- Robot navigation: straight-line distance to goal
- 8-puzzle: number of misplaced tiles
- 8-puzzle: sum of Manhattan distances for each tile to its goal position (why?)
- In general, if we get a heuristic by solving a relaxed version of a problem, we will obtain an admissible heuristic (why?)

A^* search

- *Heuristic search with an admissible heuristic!*
- Let g be the cost of the path so far
- Let h be an admissible heuristic function

  I.e. h is optimistic, it never overestimates the actual cost to the goal
- Do a greedy search with respect to

$$f = g + h$$

A^* Pseudocode

1. Initialize the queue with $(S, f(S))$, where S is the start state
2. While queue is not empty:
   (a) Pop node n with lowest priority from the priority queue; let s be the associated state and $f(s)$ the associated priority value
   (b) If s is a goal state, return success (follow back pointers from n to extract best path)
   (c) Else, for all states $s' \in \text{Successor}(s)$
      i. Compute $f(s') = g(s') + h(s') = g(s) + cost(s, s') + h(s')$
      ii. If s' was previously expanded and the new $f(s')$ is smaller, or if s' has not been expanded, or if s' is already in the queue, then create node n' with priority $f(s')$ and insert it in the queue; else do nothing

Consistency

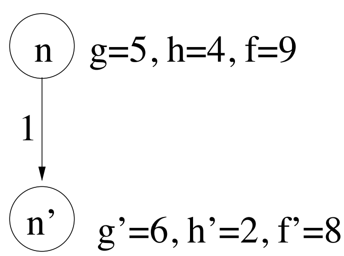
- An admissible heuristic h is called *consistent* if for every state s and for every successor s',
$$h(s) \leq c(s, s') + h(s')$$
- This is a version of triangle inequality, so heuristics that respect this inequality are metrics.
- If you think of h as estimating "distance to the goal", it is quite reasonable to assume this property
- Note that if h is monotone, and all costs are non-zero, then f cannot decrease along any path:

$$f(s) = g(s) + h(s) \leq g(s) + c(s, s') + h(s') = f(s')$$

Is A^* complete?

- Suppose that h is monotone $\Rightarrow$ f is non-decreasing
- Note that in this case, a node cannot be re-expanded
- If a solution exists, it must have bounded cost
- Hence A^* will have to find it! So it is complete

Dealing with inconsistent heuristics



- Make a small change to A*: instead of $f(s') = g(s') + h(s')$, use $f(s') = \max(g(s') + h(s'), f(s))$
- With this change, f is non-decreasing along any path, and the previous argument applies

Is A* search optimal?

- Suppose some suboptimal node containing a goal state has been generated and is in the queue (call this node G_2).
- Let n be an unexpanded node on a shortest optimal path, and call the end point of this path node G_1.
- We have:

$$\begin{aligned} f(G_2) &= g(G_2) \qquad \text{since } h(G_2) = 0 \\ &> g(G_1) \qquad \text{since } G_2 \text{ is suboptimal} \\ &\geq f(n) \qquad \text{since } h \text{ is admissible} \end{aligned}$$

- Since $f(G_2) > f(n)$, A* will select n for expansion before G_2
- Since n was chosen arbitrarily, *all* nodes on the optimal path will be chosen before G_2, so G_1 will be reached before G_2

Dominance

- If $h_2(n) \geq h_1(n)$ for all n (both admissible) then h_2 *dominates* h_1
- A^* using h_1 will expand all nodes expanded when using h_2, and more
- Eight-puzzle typical search example:

$d = 14$	IDS = 3,473,941 nodes
	A*(h_1) = 539 nodes
	A*(h_2) = 113 nodes
$d = 24$	IDS = too many nodes
	A*(h_1) = 39,135 nodes
	A*(h_2) = 1,641 nodes

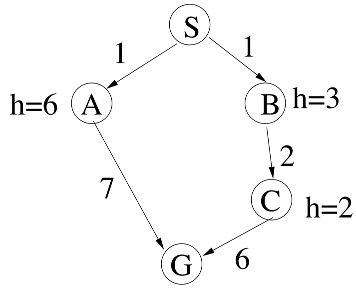
Properties of A^*

- *Complete!*
- *Optimal!*
- *Exponential worst-case time and space complexity* (why?)
  - But with a perfect heuristic, the complexity is $O(bd)$, because we would only expand the nodes along the optimal path
  - With a good heuristic complexity is often sub-exponential
- *Optimally efficient*: with a given h, no other search algorithm will be able to expand fewer nodes

Iterative Deepening A^* (IDA^*)

- Same trick as we used in last lecture to avoid memory problems
- The algorithm is basically depth-first search, but using the f-value to decide in which order to consider the descendents of a node
- There is an f-value limit, rather than a depth limit, and we expand all nodes up to $f_1, f_2, \ldots$
- Additionally, we keep track of the next limit to consider (so we will search at least one more node next time)
- IDA^* has the same properties as A^* but uses less memory
- In order to avoid expanding new nodes always, old ones can be remembered, if memory permits (a version know as SMA^*)

Iterative deepening example:



- Set $f_1 = 4 \longrightarrow$ only S, B are searched (no other nodes are put in the queue, because they exceed the cutoff threshold)
- Set $f_2 = 8 \longrightarrow$ now S, A, B, C, G are all searched

Real-Time Search

- *In dynamic environments, agents have to act before they finish thinking!*
- So instead of searching for a complete path to goal, we would like the agent to do a bit of search, then move in the direction of the currently "best" path
- Main issue: how do we avoid cycles, if we do not have enough memory to mark states?

Real-Time A^* (Korf, 1990s)

- When should the algorithm backtrack to a previously visited state s?
- Intuition: if the cost of backtracking to s and solving the problem from there is better than the cost of solving from the current state
- Korf's solution: do A^* but with the g function equal to the cost from the current state, rather than from the start.
  - This simulates physically going back to the previous state
- This is an *execution-time algorithm*!

How to decide the best direction?

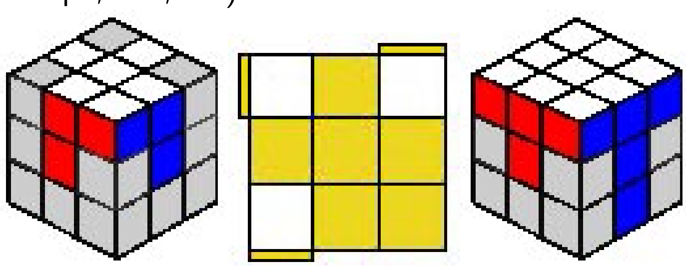
- Do we need to examine the whole frontier of a search tree to decide what node is best?
- *Not if we have a monotone f function!*
- First idea: bounding the search
  - Look at all the nodes on the frontier, but then *move one step* in the direction of the node with lowest f-value
- Second idea *pruning*
  - Maintain a variable α that has the lowest f-value of any node on the current search horizon
  - A node with cost higher than α will never get expanded
  - If a node with lower f-value is discovered, α is updated
- This is called *α-pruning*, and allows search to proceed deeper
- Same idea is used in adversarial search for game playing

Search improvements

- Consider Rubik's cube: 43,252,003,274,489,856,000 states!
- How do people solve this puzzle?

Changing the search problem

- *People do not think at the level of individual moves!*
- Instead, there are sequences of moves, designed to achieve a certain pattern (L-shape, fish, etc)



- Instead of choosing individual operators, you choose what *subgoal* you want to achieve next
- Then solve the subgoal, and choose the next subgoal
- Often the solution to a subgoal is quite standard (e.g a fish on any corner can be achieved in a similar way)

Abstraction and decomposition

- The key to solving complicated problems is to *decompose* them into smaller parts
- Each part may be easy to solve; then we put the solutions together
- *Abstraction* is a term used to refer to methods that choose to ignore information, in order to speed up computation

  E.g. in Rubik's cube, we focus only on a certain aspect of the state, like the fish, and ignore the rest of the tiles!
- Intuitively, abstraction means that we construct a smaller problem, in which *many states* of the original problem are *mapped to a single abstract state*
- A *macro-action* is a sequence of actions from the original problem (think large jump)
  - E.g. Swapping two tiles in the 8-puxxle
  - E.g. Making a T in Rubik's cube

Example: Landmark navigation

- Find a path from the current location to a well-known landmark (e.g. McGill metro)
- Find a path between landmarks (this can even be pre-computed!)
- Find a pat from last landmark to destination

Trade-offs

- By decomposing a problem and putting the solutions together, we may be *giving up optimality*
- But otherwise we may not be able to solve the problem at all!
- Solutions to subgoals are often cached in a database
- When we choose subgoals, we need to be careful that the overall problem still has a solution
  - Knoblock (1990s) showed conditions under which sub-solutions can be pieced together an completeness is preserved

Summary of informed search

- Insight: use knowledge about the problem, in the form of a heuristic.
  - The heuristic is a guess for the remaining cost to the goal.
  - A good heuristic can reduce the search time from exponential to almost linear.
- Best-first search is greedy with respect to the heuristic, not complete and not optimal
- Heuristic search is greedy with respect to $f = g + h$, where g is the cost so far and h is the estimated cost to go
- A^* is heuristic search where h is an admissible heuristic; it is complete and optimal
- *A^* is a key AI search algorithm*