Search

Informed Algorithms

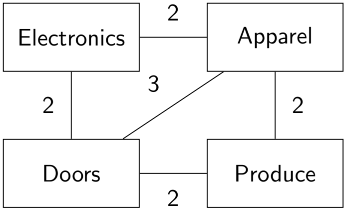
Introduction to Artificial Intelligence

Chandra Gummaluru
University of Toronto

Version W22.1

- The following is based on material developed by many individuals, including (but not limited to):
  - Sheila McIlraith
  - Bahar Aameri
  - Fahiem Bacchus
  - Sonya Allin

- An ideal search algorithm would explore "more promising" paths first.
- **Example**: Informed Search in a Superstore
  - You enter a super-store and want a T-shirt. Where would you look first?

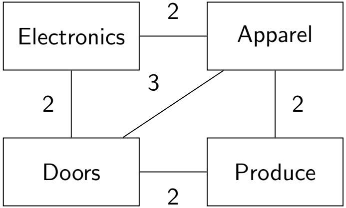


  - You would probably check the "Apparel" section even though it is farther away.

- We want to develop a metric to guide the order in which paths are explored.
- We will represent this metric as a function, f, of the path; paths with smaller f-values should be explored earlier.
- So far, we have been defining the f-value of a path in terms of its length/cost.
- However, we now also want to use info about the states traversed by the path. In particular, we want to measure how "good" the final state in the path is.
- Such algorithms are said to be **informed**.

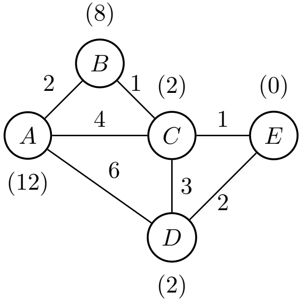
- We define a function, h, called a **heuristic**, so that $h(s)$ ideally estimates the minimum cost of getting from a state, s, to some goal state.
- We extend h to operate on paths by defining $h(p) = h(s_n)$ where $p = \langle s_0, \ldots, s_n \rangle$.
- In other words, the h-value of a path is the h-value of its terminal state.
- We can now use h as part of f. Two common choices include:
  1. $f(p) = h(p)$, which is called **greedy best-first search** (GBFS)
  2. $f(p) = c(p) + h(p)$, which is called **A-star** (A*)

- **Example**: Informed Search in a Superstore
  - You want to minimize the total estimated cost, i.e., $f(p) = c(p) + h(p)$.



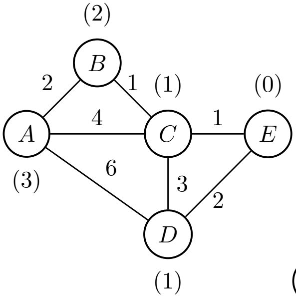
  - Although $c(\langle \text{Doors}, \text{Electronics}\rangle), c(\langle \text{Doors}, \text{Produce}\rangle) < c(\langle \text{Doors}, \text{Apparel}\rangle)$, we check "Apparel" first since we feel $h(\text{Apparel}) \ll h(\text{Electronics}), h(\text{Produce})$.

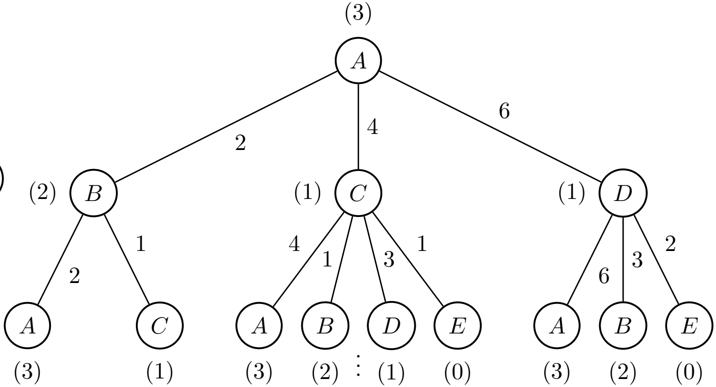
- To compare the effects of different f-functions, we will run the corresponding algorithms on the following search graph (starting at A and looking for E):



- We will use path-checking and break ties alphabetically.

- Recall that the generalized search algorithm explores a tree of possible paths.

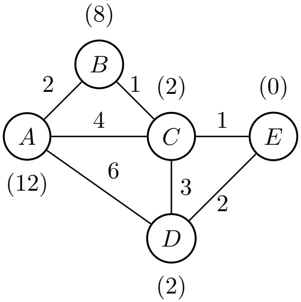




- We wish to determine the properties of the various search algorithm in terms of *b*, *d*, ε, *m* and c^*, where:
  - *b*, *m*, and ε, be the branching factor, depth, and minimum edge weight of the tree
  - *d* and c^* denote the length and cost of the optimal solution

- In greedy best-first search (GBFS), we use $f(p) = h(p)$. In other words, a partial path is more promising if we estimate that the remaining cost to the goal is lower.

Itr.	p	$\mathcal{O}$
1	–	$A\|12$
2	A	$AB\|8, AC\|2, AD\|2$
3	AC	$AD\|2, AB\|8, ACB\|8, ACD\|2, ACE\|0$
4	ACE	$AD\|2, AB\|8, ACB\|8, ACD\|2$



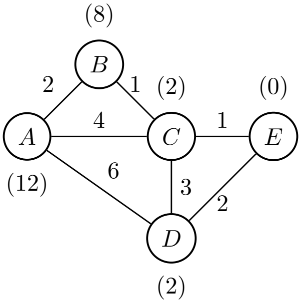
- Notice that this particular heuristic yielded a sub-optimal solution, but it did reduce the number iterations needed to find a solution (relative to UCS).

- **Complete** for $b, d < \infty$ if path-checking is used.
  - There are at most $1 + b + \ldots b^d$ paths in the tree.
  - The search will never explore any path more than once.
  - The search must explore all paths reachable from the root.
- **Optimal** never[1].
  - f does not incorporate c.
- **Time Complexity**: $O(b^m)$.
  - In the worst-case, we will need to explore every path.
- **Space Complexity**: $O(bm)$.
  - In the worst-case, we will need to explore every path.

[1] except by chance.

- In A-Star (A*), we use $f(p) = c(p) + h(p)$. In other words, a partial path is more promising if we estimate its total cost to the goal to be lower.

Itr.	p	$\mathcal{O}$
1	–	$A\|12$
2	A	$AB\|10, AC\|6, AD\|8$
3	AC	$AB\|10, AD\|8, ACB\|13, ACD\|9, ACE\|5$
4	ACE	$AD\|8, AB\|10, ACB\|13, ACD\|9$



- Notice that this particular heuristic yielded a sub-optimal solution, but it did reduce the number iterations needed to find a solution (relative to UCS).

- The optimal path from A to E is $\langle A, B, C, E \rangle$.
- However, both GBFS and A* found the same sub-optimal path, $\langle A, C, E \rangle$.
- This is expected in GBFS since it does not make use of the costs.
- Since A* does make use of the costs, ideally the path it finds would be optimal.
- The problem was that the heuristic we used over-estimated the cost to get from B to E, making the partial path $\langle A, B \rangle$ seem far less promising.

- If $h^*(s)$ is the true cost from s to the nearest goal, then ideally, our heuristic would be such that $h(s) \leq h^*(s)$ for all s, i.e., h never over-estimates the true cost.
- Such a heuristic is said to be **admissible**.
- If a heuristic does not over-estimate the cost of individual actions, i.e.,

$$h(s) - h(a(s)) \leq c(a), \forall s, a \in A(s),$$

we say that the heuristic is **consistent** or **monotone**.

- Any consistent heuristic, h, where $h(s) = 0$ for every $s \in G$, is also admissible.
- **Proof**:
  - Pick an arbitrary initial state, s_0.
  - If no goal-terminating path from s_0 exists, then $h^*(s_0)$ is infinite and the claim trivially holds.
  - Otherwise, let $\langle s_0, \ldots, s_n \rangle$ be resulting state sequence of the optimal goal-terminating path, $\langle a_{0,1}, \ldots, a_{n-1,n} \rangle$ from s_0.
  - Since $s_n \in \mathcal{G}$, it follows that $h(s_n) = 0 = h^*(s_n)$. Thus, $h(s_n) \leq h^*(s_n)$.
  - Assume $h(s_{i+1}) \leq h^*(s_{i+1})$ for some i. Then, $h(s_i) \leq h^*(s_i)$:

$$h(s_i) \leq c(a_{i,i+1}) + h(s_{i+1}) \leq c(a_{i,i+1}) + h^*(s_{i+1}) = h^*(s_i).$$

  - Since s_0 was arbitrary, it follows that $h(s) \leq h^*(s)$ for all s, i.e., h is admissible.

- It turns out that if h is consistent, then A-star satisfies the condition needed to use global path checking.

- We prove this in three parts, all of which rely on h being consistent:
  1. If $f \equiv c + h$, then f must be non-decreasing along any path.
  2. The f-values of paths explored in A* are non-decreasing.
  3. The first time A* finds a path to a state, it has found the optimal path.

1. If $f \equiv c + h$, then the f-value must be non-decreasing along any path.
   - Let $(s_0, \langle a_{0,1}, \ldots, a_{n-1,n} \rangle)$ be a path, and $\langle a_0, \ldots, s_n \rangle$.
   - By the definition of f in A*,

$$f(s_i) = c(a_{0,1}, \ldots, a_{i-1,i}) + h(s_i).$$

   - If h is consistent, then by definition, $h(s_i) \leq c(a_{i,i+1}) + h(s_{i+1})$.
   - It follows that $f(s_i) \leq c(a_{0,1}, \ldots, a_{i,i+1}) + h(s_{i+1}) = f(s_{i+1})$

2. The f-values of paths explored in A* are non-decreasing.
   - If p' is explored after p, then either:
     - p' was already on the open when p was explored
     - p' is an extension of p
     - p' is an extension of a path, p'' such that $f(p'') \geq f(p)$.
   - In any case $f(p') \geq f(p)$.

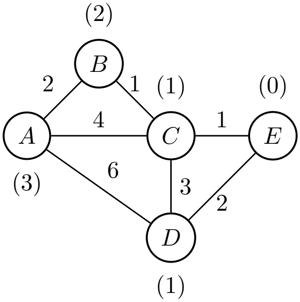
3. The first time A* finds a path to a state, it has found the optimal path.
   - Let $p = \langle s_0, s_1, \ldots, s_{n-1}, s_n \rangle$ and $p' = \langle s_0, s'_1, \ldots, s'_{n-1}, s_n \rangle$ be two state sequences to the goal state, s_n found by A*.
   - If p was found before p', then by (2), $f(p') \geq f(p)$.
   - Using the definition of f in A*, we have

$$c(p') + h(p') \geq c(p) + h(p).$$

   - Since h is consistent (so admissible), $h(p') = h(p) = 0$, and so $c(p') \geq c(p)$.

- Let us reconsider A* using a consistent heuristic.

Itr.	p	$\mathcal{O}$
1	–	$A\|3$
2	A	$AB\|4, AC\|5, AD\|7$
3	AB	$AC\|5, AD\|7, ABC\|4$
4	ABC	$AC\|5, AD\|7, ABCE\|5, ABCD\|7$
5	$ABCE$	$AC\|5, AD\|7, ABCD\|7$



- Notice that this particular heuristic yielded the optimal solution.

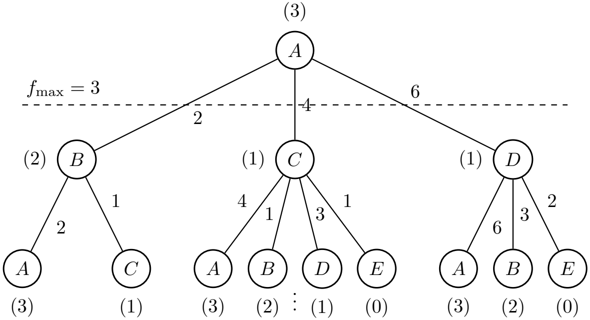
- **Complete** for $b, d < \infty$, $\varepsilon > 0$.
  - There are at most $1 + b + \ldots b^d$ paths in the tree.
  - The search will never explore any path more than once.
  - The search must explore all paths reachable from the root.
- **Optimal** if $h(\cdot)$ is admissible.
  - Let s_n be a goal state and $p = \langle s_0, \ldots, s_n \rangle$ be the path found by A*.
  - Recall that for A*, $f(\cdot) = c(\cdot) + h(\cdot)$.
  - If h is admissible, then $h(s_n) = h(s_n) = 0$, and so $f(p) = c(p)$.
  - If p is sub-optimal, i.e., $c(p) \geq h^*(s_0)$, then $f(p) \geq h^*(s_0)$.
  - Let $p^* = \langle s_0^*, \ldots s_m^* \rangle$ where $s_0^* = s_0$ and $s_m^* = s_n$ be the optimal path to s_n.
  - A subpath of p^*, say $p_{0:i}^*$ must still be on the frontier.
  - We compute $f(p_{0:i}^*) = c(p_{0:i}^*) + h(s_i^*) \leq c(p_{0:i}^*) + h^*(s_i^*) = h^*(s_0) \leq f(p)$.
  - However, this means that we should have explored $p_{0:i}^*$ before p.

- **Time Complexity**: $O(b^{c^*/\varepsilon})$.
    - In the worst-case, the time-complexity of A* is the same as UCS.
- **Space Complexity**: $O(b^{c^*/\varepsilon})$.
    - In the worst-case, the space-complexity of A* is the same as UCS.

- Like IDDFS, we can implement an iterative version of A*.
- We start with an f-limit of $f_{\max} = h(s_0)$, and repeatedly perform A*, increasing the f-limit each time, until a solution is found
- In each iteration of IDA*, we increase $f_{\max}$ to the minimum f-value of all pruned paths in the previous iteration.
- This is called **iterative-deepening A-star search** (IDA*).

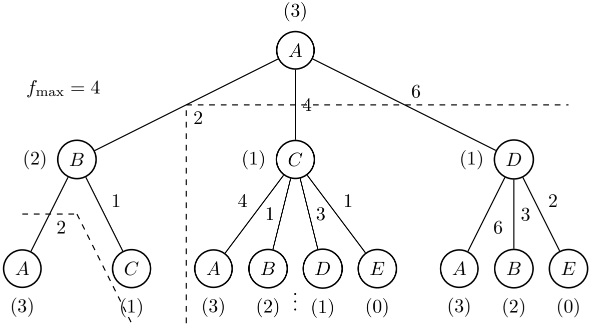
- IDA* is like performing A* multiple times on a sub-tree of possible paths.

Itr.	p	$\mathcal{O}$
1	–	A
2	A	



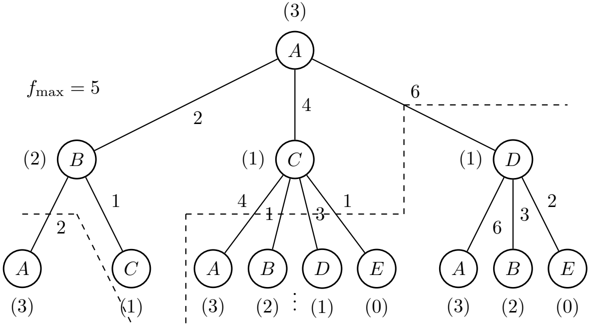
- IDA* is like performing A* multiple times on a sub-tree of possible paths.

Itr.	p	$\mathcal{O}$
1	–	A
2	A	AB
3	AB	ABC
4	ABC	



- IDA* is like performing A* multiple times on a sub-tree of possible paths.

Itr.	p	$\mathcal{O}$
1	–	A
2	A	AB, AC
3	AB	AC, ABC
4	ABC	AC
5	AC	



- Below, we summarize the properties of GBFS, A*, and IDA*:

Property	GBFS	A*
Complete	$b, d < \infty$, PC	$b < \infty, \varepsilon > 0$
Optimal	never[2]	h admissible
Time Complexity	$O(b^m)$	$O(b^{c^*/\varepsilon})$
Space Complexity	$O(bm)$	$O(b^{c^*/\varepsilon})$

[2] except by chance.

- Weighted A* is a generalization of the A* algorithm where we let

$$f(\cdot) = wc(\cdot) + (1 - w)h(\cdot),$$

  for some $0 \leq w \leq 1$.
- This way, we can interpret UCS as weighted A* with $w = 1$, GBFS as weighted A* with $w = 0$, and A* as weighted A* with $w = 0.5$.
- The weight, w, can be tuned to balance the trade-off between the time required to find a solution versus the quality of the solution found; increasing w results in a better solution, but smaller w results in a quicker solution.

- Designing heuristics is an art.
- There are many techniques used in practice. We shall consider two of them:

    1. using a relaxation of the problem
    2. using pattern databases

- One way to design a heuristic for a problem is to find the solution of a relaxed problem and use its' cost as a heuristic for the original problem.

Example: Slider Puzzle Relaxation (l_1 distances)

- In the slider puzzle, tiles could only be moved into an unoccupied, horizontally/vertically adjacent space.
- Suppose we can move tiles to any horizontally/vertically adjacent space (including occupied ones).
- In this case, the sum of the l_1 distances between each tile's correct position and its current position gives the number of moves required.
- For the example on the right, this value is $3 + 2 + 0 + 1 + 1 + 2 + 0 + 1 = 10$.

4	6	3
2		1
7	5	8

s_0

1	2	3
4	5	6
7	8	

$s \in \mathcal{G}$

Designing Heuristics via Problem Relaxation

Example: Slider Puzzle Relaxation (misplaced tiles)

- In the slider puzzle, tiles could only be moved into an unoccupied, horizontally/vertically adjacent space.
- Suppose we can move tiles to any adjacent space (including diagonally adjacent or occupied ones).
- In this case, the number of misplaced tiles gives the number of moves required.
- For the example on the right, this value is $1+1+0+1+1+1+0+1=6$.

4	6	3
2		1
7	5	8

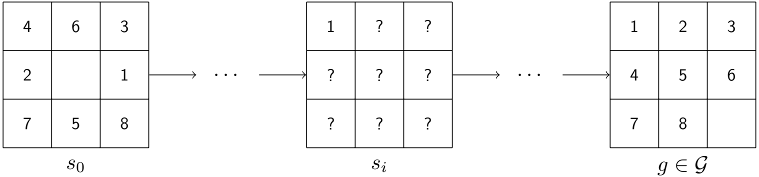
s_0

1	2	3
4	5	6
7	8	

$s \in \mathcal{G}$

- Another way to design a heuristic for a problem is to break it up into sub-problem such that solving the original problem is equivalent to solving each of the sub-problems.

- **Example**: Slider Puzzle Pattern Database
    - It turns out that the Slider puzzle can be solved by first moving the ‘1’ tile to the top-left position and then never moving it again.



For this particular example, it takes at least 11 moves to get from s_0 to s_i.

- Suppose the costs are reversible, i.e., the cost to get from s_i to s_j is the same as the cost to get from s_j to s_i.

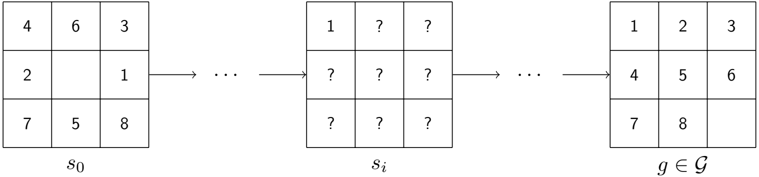
- For each s, we can search backwards from the solution(s) of the sub-problem until we find s. The cost of the resulting path is the cost of solving the sub-problem when the initial state is s.

- We store these values in a table. We can then compute a heuristic for the original problem using this table.

Designing Heuristics via Pattern Databases (continued)

Example: Slider Puzzle Pattern Database

- Suppose we had a function, $h_{0,i}(s)$, which gives the cost of getting from any state, s, to one of the form of s_i, and h_i is a heuristic for the sub-problem of finding g from s_i.



- We can then define $h(p) = h_{0:i}(p) + h_i(p)$ as a heuristic for the original problem.

- In general, we can create many admissible heuristics for a given problem.
- However, some heuristics are better than others, in the sense that they will allow us to explore less paths.
- Given two admissible heuristics, h_1, and h_2, if $h_1(s) \geq h_2(s)$ for all s, then we say that h_1 **weakly dominates** h_2.
- If $h_1(s) > h_2(s)$ for some s, then we say that h_1 **strongly dominates** h_2.
- A strongly dominant heuristic is guaranteed to explore fewer paths.
- Clearly, if h_1 and h_2 are admissible heuristics, then $h = \max\{h_1, h_2\}$ is also admissible and weakly dominates h_1 and h_2.