CSE 3402: Intro to Artificial Intelligence Informed Search I

- Required Readings: Chapter 4. We won't cover the material in section 4.4 and 4.5, so reading them is optional.

Heuristic Search.

- In uninformed search, we don't try to evaluate which of the nodes on the frontier are most promising. We never "look-ahead" to the goal.
  - E.g., in uniform cost search we always expand the cheapest path. We don't consider the cost of getting to the goal.
- Often we have some other knowledge about the merit of nodes, e.g., going the wrong direction in Romania.

Heuristic Search.

- Merit of a frontier node: different notions of merit.
  - If we are concerned about the cost of the solution, we might want a notion of merit of how costly it is to get to the goal from that search node.
  - If we are concerned about minimizing computation in search we might want a notion of ease in finding the goal from that search node.
  - We will focus on the “cost of solution” notion of merit.

Heuristic Search.

- The idea is to develop a domain specific heuristic function h(n).
- h(n) guesses the cost of getting to the goal from node n.
- There are different ways of guessing this cost in different domains. I.e., heuristics are domain specific.

Heuristic Search.

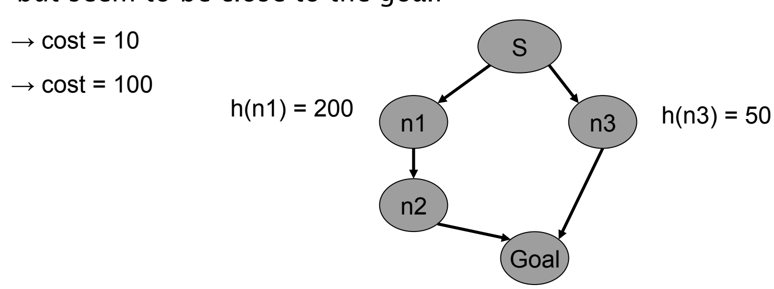
- Convention: If $h(n_1) < h(n_2)$ this means that we guess that it is cheaper to get to the goal from n_1 than from n_2.

- We require that
  - $h(n) = 0$ for every node n that satisfies the goal.
    - Zero cost of getting to a goal node from a goal node.

Using only h(n)
Greedy best-first search.

- We use h(n) to rank the nodes on open.
  - Always expand node with lowest h-value.
- We are greedily trying to achieve a low cost solution.

- However, this method ignores the cost of getting to n, so it can be lead astray exploring nodes that cost a lot to get to but seem to be close to the goal:



A* search

- Take into account the cost of getting to the node as well as our estimate of the cost of getting to the goal from n.
- Define
  - $f(n) = g(n) + h(n)$
    - $g(n)$ is the cost of the path to node n
    - $h(n)$ is the heuristic estimate of the cost of getting to a goal node from n.
- Now we always expand the node with lowest f-value on the frontier.
- The f-value is an estimate of the cost of getting to the goal via this node (path).

Conditions on h(n)

- We want to analyze the behavior of the resultant search.
- Completeness, time and space, optimality?
- To obtain such results we must put some further conditions on the heuristic function h(n) and the search space.

Conditions on h(n): Admissible

- $c(n1 \rightarrow n2) \geq \epsilon > 0$. The cost of any transition is greater than zero and can't be arbitrarily small.
- Let h*(n) be the cost of an optimal path from n to a goal node (∞ if there is no path). Then an admissible heuristic satisfies the condition
  - $h(n) \leq h^*(n)$
    - i.e. *h* always underestimates of the true cost.
- Hence
  - $h(g) = 0$
  - For any goal node "g"

Consistency/monotonicity.

- Is a stronger condition than $h(n) \leq h^*(n)$.
- A monotone/consistent heuristic satisfies the triangle inequality (for all nodes n1,n2):

  $$h(n1) \leq c(n1 \rightarrow n2) + h(n2)$$

- Note that there might be more than one transition (action) between n1 and n2, the inequality must hold for all of them.
- Note that monotonicity implies admissibility. Why?

Intuition behind admissibility

- $h(n) \leq h^*(n)$ means that the search won't miss any promising paths.
  - If it really is cheap to get to a goal via n (i.e., both $g(n)$ and $h^*(n)$ are low), then $f(n) = g(n) + h(n)$ will also be low, and the search won't ignore n in favor of more expensive options.
  - This can be formalized to show that admissibility implies optimality.

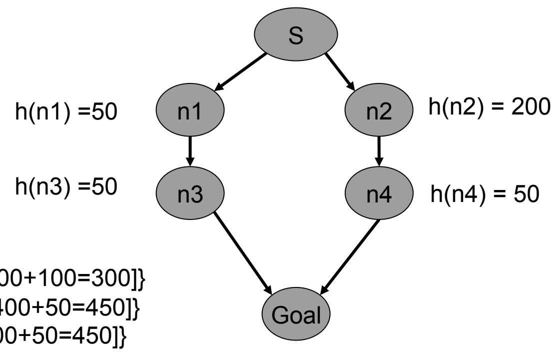
Intuition behind monotonicity

- $h(n1) \leq c(n1 \rightarrow n2) + h(n2)$
  - This says something similar, but in addition one won't be "locally" mislead. See next example.

Example: admissible but nonmonotonic

- The following *h* is not consistent since *h(n2)>c(n2→n4)+h(n4). But it is admissible.*

→ cost = 200
→ cost = 100



{S} → {n1 [200+50=250], n2 [200+100=300]}
→ {n2 [100+200=300], n3 [400+50=450]}
→ {n4 [200+50=250], n3 [400+50=450]}
→ {goal [300+0=300], n3 [400+50=450]}

We **do find** the optimal path as the heuristic is still admissible. **But** we are mislead into ignoring n2 until after we expand n1.

Consequences of monotonicity

1. The f-values of nodes along a path must be non-decreasing.

- Let <Start→ n1→ n2…→ nk> be a path. We claim that

  f(ni) ≤ f(ni+1)

- Proof:

  f(ni) = c(Start→ …→ ni) + h(ni)
  ≤ c(Start→ …→ ni) + c(ni→ ni+1) + h(ni+1)
  = c(Start→ …→ ni→ ni+1) + h(ni+1)
  = g(ni+1) + h(ni+1)
  = f(ni+1).

Consequences of monotonicity

2. If n2 is expanded after n1, then $f(n1) \leq f(n2)$

Proof:

- If n2 was on the frontier when n1 was expanded,
  - $f(n1) \leq f(n2)$

  otherwise we would have expanded n2.

- If n2 was added to the frontier after n1's expansion, then let n be an ancestor of n2 that was present when n1 was being expanded (this could be n1 itself). We have $f(n1) \leq f(n)$ since A* chose n1 while n was present in the frontier. Also, since n is along the path to n2, by property (1) we have $f(n) \leq f(n2)$. So, we have
  - $f(n1) \leq f(n2)$.

Consequences of monotonicity

3. When *n* is expanded every path with lower f-value has already been expanded.

- Assume by contradiction that there exists a path <Start, n0, n1, ni-1, ni, ni+1, …, nk> with $f(nk) < f(n)$ and ni is its **last expanded** node.

- Then ni+1 must be on the frontier while n is expanded:

  a) by (1) $f(ni+1) \leq f(nk)$ since they lie along the same path.

  b) since $f(nk) < f(n)$ so we have $f(ni+1) < f(n)$

  c) by (2) $f(n) \leq f(ni+1)$ since n is expanded before ni+1.

  * Contradiction from b&c!

Consequences of monotonicity

4. With a monotone heuristic, the first time A* expands a state, it has found the minimum cost path to that state.

- Proof:
  * Let PATH1 = <Start, n0, n1, …, nk, n> be **the first** path to n found. We have f(path1) = c(PATH1) + h(n).
  * Let PATH2 = <Start, m0,m1, …, mj, n> be another path to n found later. we have f(path2) = c(PATH2) + h(n).

  * By property (3), f(path1) ≤ f(path2)

  * hence: c(PATH1) ≤ c(PATH2)

Consequences of monotonicity

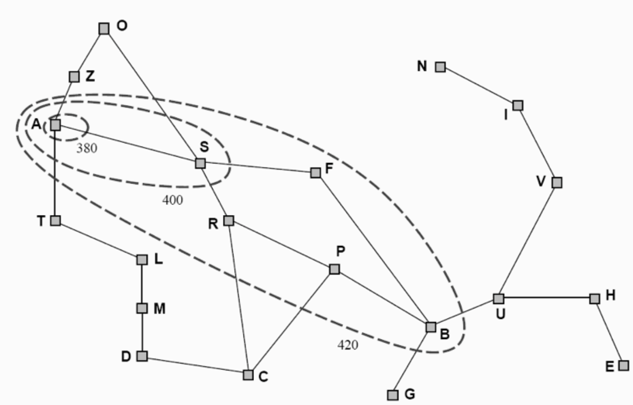
- Complete.
  - Yes, consider a least cost path to a goal node
    - SolutionPath = <Start→ n1→ …→ G> with cost
    - c(SolutionPath)
    - Since each action has a cost ≥ ϵ > 0, there are only a finite number of nodes (paths) that have cost ≤ c(SolutionPath).
    - All of these paths must be explored before any path of cost > c(SolutionPath).
    - So eventually SolutionPath, or some equal cost path to a goal must be expanded.
- Time and Space complexity.
  - When h(n) = 0, for all n
    - h is monotone.
  - A* becomes uniform-cost search!
  - It can be shown that when h(n) > 0 for some n, the number of nodes expanded can be no larger than uniform-cost.
  - Hence the same bounds as uniform-cost apply. (These are worst case bounds).

Consequences of monotonicity

- Optimality
  - Yes, by (4) the first path to a goal node must be optimal.
- Cycle Checking
  - If we do cycle checking (e.g. using GraphSearch instead of TreeSearch) it is still optimal. Because by property (4) we need keep only the first path to a node, rejecting all subsequent paths.

Search generated by monotonicity

Gradually adds "f-contours" of nodes (cf. breadth-first adds layers)
Contour i has all nodes with $f = f_i$, where $f_i < f_{i+1}$



Admissibility without monotonicity

- When "h" is admissible but not monotonic.
  - Time and Space complexity remain the same. Completeness holds.
  - Optimality still holds (without cycle checking), but need a different argument: don't know that paths are explored in order of cost.

- Proof of optimality (without cycle checking):
  - Assume the goal path <S,…,G> found by A* has cost bigger than the optimal cost: i.e. $C^* < f(G)$.
  - There must exists a node n in the optimal path that is still in the frontier.
  - We have: $f(n)=g(n)+h(n) \le g(n)+h^*(n)=C^* < f(G)$
  - Therefore, f(n) must have been selected before G by A*. contradiction!

Admissibility without monotonicity

- No longer guaranteed we have found an optimal path to a node ***the first time*** we visit it.
- So, cycle checking might not preserve optimality.
  - To fix this: for previously visited nodes, must remember cost of previous path. If new path is cheaper must explore again.
- contours of monotonic heuristics don't hold.

Space problem with A* (like breath-first search):

IDA* is similar to Iterative Lengthening Search: It puts the newly expanded nodes in the front of frontier! Two new parameters:

- curBound (any node with a bigger f value is discarded)
- smallestNotExplored (the smallest f value for discarded nodes in a round) when frontier becomes empty, the search starts a new round with this bound.

Building Heuristics: Relaxed Problem

- One useful technique is to consider an easier problem, and let h(n) be the cost of reaching the goal in the easier problem.
- 8-Puzzle moves.
  - Can move a tile from square A to B if
    - A is adjacent (left, right, above, below) to B
    - and B is blank
- Can relax some of these conditions
  1. can move from A to B if A is adjacent to B (ignore whether or not position is blank)
  2. can move from A to B if B is blank (ignore adjacency)
  3. can move from A to B (ignore both conditions).

Building Heuristics: Relaxed Problem

- #3 leads to the misplaced tiles heuristic.
  - To solve the puzzle, we need to move each tile into its final position.
  - Number of moves = number of misplaced tiles.
  - Clearly h(n) = number of misplaced tiles $\leq$ the h*(n) the cost of an optimal sequence of moves from n.
- #1 leads to the manhattan distance heuristic.
  - To solve the puzzle we need to slide each tile into its final position.
  - We can move vertically or horizontally.
  - Number of moves = sum over all of the tiles of the number of vertical and horizontal slides we need to move that tile into place.
  - Again h(n) = sum of the manhattan distances $\leq$ h*(n)
    - in a real solution we need to move each tile at least that that far and we can only move one tile at a time.

Building Heuristics: Relaxed Problem

- The optimal cost to nodes in the relaxed problem is an admissible heuristic for the original problem!

  Proof: the optimal solution in the original problem is a (*not necessarily optimal*) solution for relaxed problem, therefore it must be at least as expensive as the optimal solution in the relaxed problem.
- Comparison of IDS and A* (average total nodes expanded):

Depth	IDS	A*(Misplaced)	A*(Manhattan)
10	47,127	93	39
14	3,473,941	539	113
24	---	39,135	1,641

Let h1=Misplaced,   h2=Manhattan

- Does h2 **always** expand less nodes than h1?
  - Yes! Note that h2 dominates h1, i.e. for all n: $h1(n) \leq h2(n)$. From this you can prove h2 is faster than h1.
  - Therefore, among several admissible heuristic the one with highest value is the fastest.

Building Heuristics: Pattern databases.

- Admissible heuristics can also be derived from solution to **subproblems: Each state is mapped into a partial specification, e.g. in 15-puzzle only *position of specific tiles matters.***

- Here are goals for two sub-problems (called Corner and Fringe) of 15puzzle. If you want to know how they came up with these subproblems? Here is the paper.
- Note that the goal state here for15-puzzle is **different** than what we have defined in Assignment1).

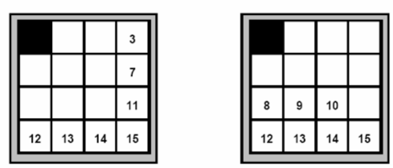


Fig. 2. The Fringe and Corner Target Patterns.

- By searching backwards from these goal states, we can compute the distance of any configuration of these tiles to their goal locations. We are ignoring the identity of the other tiles.
- For any state n, the number of moves required to get these tiles into place form a lower bound on the cost of getting to the goal from n.

Building Heuristics: Pattern databases.

- These configurations are stored in a database, along with the number of moves required to move the tiles into place.
- The maximum number of moves taken over all of the databases can be used as a heuristic.
- On the 15-puzzle
  - The fringe data base yields about a 345 fold decrease in the search tree size.
  - The corner data base yields about 437 fold decrease.
- Some times disjoint patterns can be found, then the number of moves can be added rather than taking the max.

Local Search

- So far, we keep the paths to the goal.
- For some problems (like 8-queens) we don't care about the path, we only care about the solution. Many real problem like Scheduling, IC design, and network optimizations are of this form.
- Local search algorithms operate using a single Current state and generally move to neighbors of that state.
- There is an objective function that tells the value of each state. The goal has the highest value (global maximum).
- Algorithms like Hill Climbing try to move to a neighbor with the highest value.
- Danger of being stuck in a local maximum. So some randomness can be added to "shake" out of local maxima.

Local Search

- Simulated Annealing: Instead of the best move, take a random move and if it improves the situation then always accept, otherwise accept with a probability <1. Progressively decrease the probability of accepting such moves.
- Local Beam Search is like a parallel version of Hill Climbing. Keeps K states and at each iteration chooses the K best neighbors (so information is shared between the parallel threads). Also stochastic version.
- Genetic Algorithms are similar to Stochastic Local Beam Search, but mainly use crossover operation to generate new nodes. This swaps feature values between 2 parent nodes to obtain children. This gives a hierarchical flavor to the search: chunks of solutions get combined. Choice ofstate representation becomes very important. Has had wide impact, but not clear if/ when better than other approaches.