

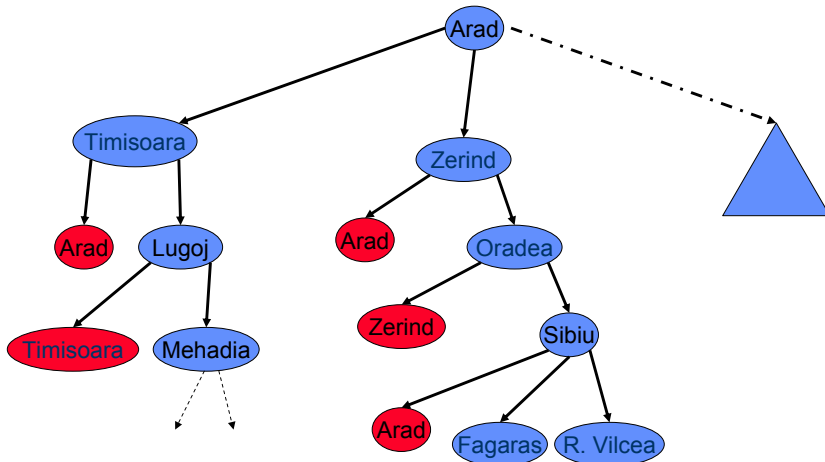
# CSC384: Intro to Artificial Intelligence Search III

- 4 Prolog Tutorials: T1, T2, T3, and T4
  - T1: Fri Sep 22 10–11am SS2127
  - T2: Fri Sep 22 4–5pm BA3012
  - T3: Fri Sep 29 10–11am SS2127
  - T4: Fri Sep 29 4–5pm BA3012
- Path Checking and Cycle Checking
- Heuristic Search
  - Required readings Chapter 4.1, 4.2, some of 4.3. Omit 4.4, 4.5

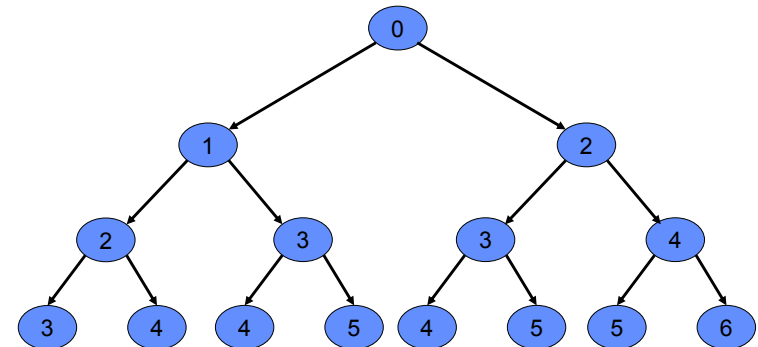
# Cycle Checking

- Path checking
  - Recall paths are stored on the frontier (this allows us to output the solution path).
    - If  $\langle S, n_1, \dots, n_k \rangle$  is a path to node  $n_k$ , and we expand  $n_k$  to obtain child  $c$ , we have
      - $\langle S, n_1, \dots, n_k, c \rangle$
      - As the path to “c”.
  - Path checking:
    - Ensure that the state  $c$  is not equal to the state reached by any ancestor of  $c$  along this path.

# Path Checking Example



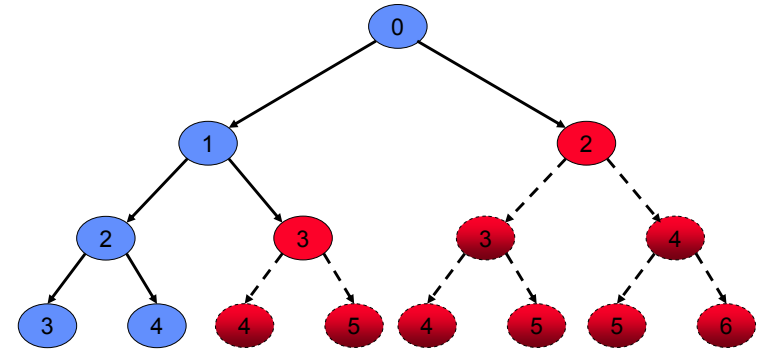
# Path Checking Example



## Cycle Checking

- Cycle Checking.
  - Keep track of **all states** previously expanded during the search.
  - When we expand  $n_k$  to obtain child  $c$ 
    - ensure that  $c$  is not equal to any previously expanded state.
  - This is called **cycle checking**, or **multiple path checking**.
  - Why can't we utilize this technique with depth-first search?
    - If we modify depth-first search to do cycle checking what happens to space complexity?

## Cycle Checking Example



## Cycle Checking

- High space complexity, only useful with breadth first search.
- There is an additional issue when we are looking for an optimal solution, **but**:
  - With uniform-cost search, we still find an optimal solution
    - **The first time uniform-cost expands a state it has found the minimal cost path to it.**
  - This means that the nodes rejected by cycle checking can't have better paths.
  - We will see later that we don't always have this property when we do heuristic search.

## 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**.

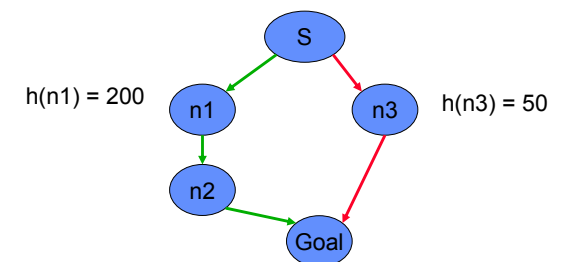
## Heuristic Search.

- 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:

→ cost = 10  
→ cost = 100



## 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

- $c(n_1 \rightarrow n_2) \geq \epsilon > 0$ . 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 ( $\infty$  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  $n_1, n_2$ ):
 
$$h(n_1) \leq c(n_1 \rightarrow n_2) + h(n_2)$$
- Note that there might be more than one transition (action) between  $n_1$  and  $n_2$ , the inequality must hold for all of them.
- Note that monotonicity implies admissibility. Why?

## Intuition behind admissibility

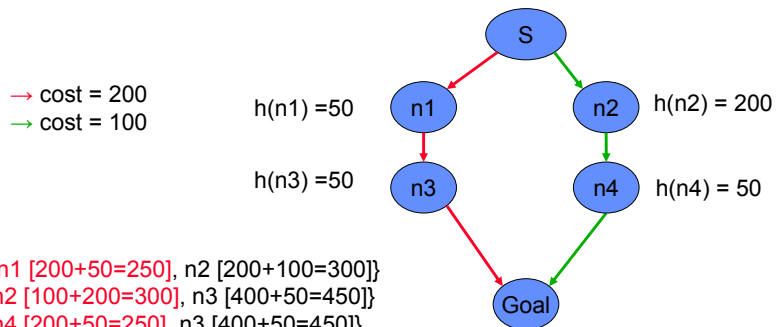
- $h(n) \leq h^*(n)$  means that the search won't miss any promising paths.
  - If it really is cheap to get to a goal via  $n$  (i.e., both  $g(n)$  and  $h^*(n)$  are low), then  $f(n) = g(n) + h(n)$  will also be low, and the search won't ignore  $n$  in favor of more expensive options.
  - This can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

## Example: admissible but nonmonotonic

- The following  $h$  is not consistent since  $h(n2) > c(n2 \rightarrow n4) + h(n4)$ . But it is admissible.



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

We **do find** the optimal path as the heuristic is still admissible. **But** we are misled 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  $\langle \text{Start} \rightarrow n1 \rightarrow n2 \dots \rightarrow nk \rangle$  be a path. We claim that

$$f(ni) \leq f(ni+1)$$

- Proof:

$$\begin{aligned}
 f(ni) &= c(\text{Start} \rightarrow \dots \rightarrow ni) + h(ni) \\
 &\leq c(\text{Start} \rightarrow \dots \rightarrow ni) + c(ni \rightarrow ni+1) + h(ni+1) \\
 &= c(\text{Start} \rightarrow \dots \rightarrow ni \rightarrow ni+1) + h(ni+1) \\
 &= g(ni+1) + h(ni+1) \\
 &= f(ni+1).
 \end{aligned}$$

## Consequences of monotonicity

### 2. If $n_2$ is expanded after $n_1$ , then $f(n_1) \leq f(n_2)$

#### Proof:

- If  $n_2$  was on the frontier when  $n_1$  was expanded,
  - $f(n_1) \leq f(n_2)$   
otherwise we would have expanded  $n_2$ .
- If  $n_2$  was added to the frontier after  $n_1$ 's expansion, then let  $n$  be an ancestor of  $n_2$  that was present when  $n_1$  was being expanded (this could be  $n_1$  itself). We have  $f(n_1) \leq f(n)$  since  $A^*$  chose  $n_1$  while  $n$  was present in the frontier. Also, since  $n$  is along the path to  $n_2$ , by property (1) we have  $f(n) \leq f(n_2)$ . So, we have
  - $f(n_1) \leq f(n_2)$ .

## 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  $\langle \text{Start}, n_0, n_1, n_{i-1}, n_i, n_{i+1}, \dots, n_k \rangle$  with  $f(n_k) < f(n)$  and  $n_i$  is its **last expanded** node.
- Then  $n_{i+1}$  must be on the frontier while  $n$  is expanded:
  - a) by (1)  $f(n_{i+1}) \leq f(n_k)$  since they lie along the same path.
  - b) since  $f(n_k) < f(n)$  so we have  $f(n_{i+1}) < f(n)$
  - c) by (2)  $f(n_{i+1}) \leq f(n)$  since  $n$  is expanded before  $n_{i+1}$ .
 \* 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  $\text{PATH1} = \langle \text{Start}, n_0, n_1, \dots, n_k, n \rangle$  be **the first** path to  $n$  found. We have  $f(\text{path1}) = c(\text{PATH1}) + h(n)$ .
- \* Let  $\text{PATH2} = \langle \text{Start}, m_0, m_1, \dots, m_j, n \rangle$  be another path to  $n$  found later. we have  $f(\text{path2}) = c(\text{PATH2}) + h(n)$ .
- \* By property (3),  $f(\text{path1}) \leq f(\text{path2})$
- \* hence:  $c(\text{PATH1}) \leq c(\text{PATH2})$

## Consequences of monotonicity

#### Complete.

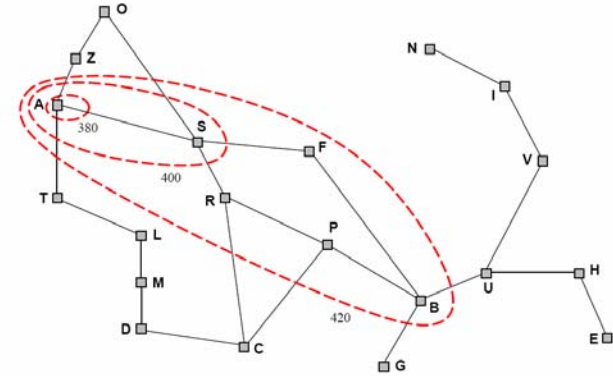
- Yes, consider a least cost path to a goal node
  - $\text{SolutionPath} = \langle \text{Start} \rightarrow n_1 \rightarrow \dots \rightarrow G \rangle$  with cost  $c(\text{SolutionPath})$
  - Since each action has a cost  $\geq \epsilon > 0$ , there are only a finite number of nodes (paths) that have cost  $\leq c(\text{SolutionPath})$ .
  - All of these paths must be explored before any path of cost  $> c(\text{SolutionPath})$ .
  - So eventually  $\text{SolutionPath}$ , 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  $\langle S, \dots, G \rangle$  found by  $A^*$  has cost bigger than the optimal cost: i.e.  $C^* < f(G)$ .
  - There must exist a node  $n$  in the optimal path that is still in the frontier.
  - We have:  $f(n) = g(n) + h(n) \leq g(n) + h^*(n) = C^* < f(G)$
  - 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.

## 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  $h(n) = \text{number of misplaced tiles} \leq h^*(n)$  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  $h(n) = \text{sum of the manhattan distances} \leq h^*(n)$ 
    - in a real solution we need to move each tile at least that far and we can only move one tile at a time.

## Building Heuristics: Relaxed Problem

Depth	IDS	A*(Misplaced)	A*(Manhattan)
10	47,127	93	39
14	3,473,94	539	113
24	1	39,135	1,641

## Building Heuristics: Pattern databases.

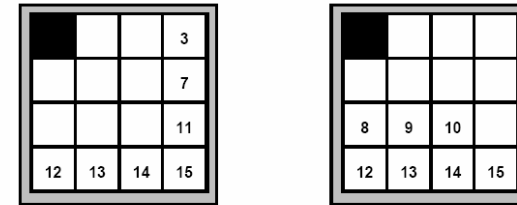


Fig. 2. The Fringe and Corner Target Patterns.

These are subpatterns of the goal state. By searching backwards from these 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 is added to "shake" out of local maxima.
- **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  $< 1$ .
- [read these two algorithms from the Book].