CS 461 ARTIFICIAL INTELLIGENCE

Lecture # 06
March 25, 2021
SPRING 2021
FAST – NUCES, CFD Campus

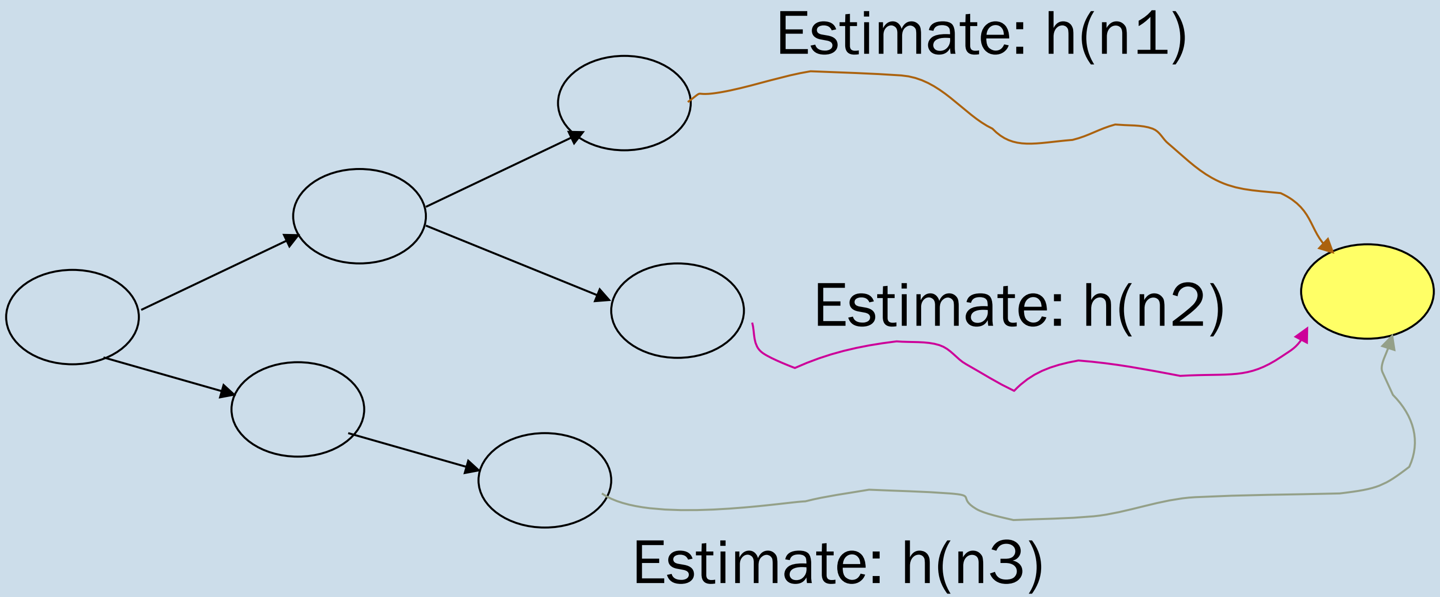
Dr. Rabia Maqsood
rabia.maqsood@nu.edu.pk

Today's Topics

- Search strategies
  - *Informed search algorithms*
    - Quick recap: A* algorithm
    - Heuristics
      - *Admissibility*
      - *Consistency*
    - IDA*
    - Recursive Best-First Search

Search Heuristic

A search heuristic *h(n)* is an estimate of the cost of the optimal (cheapest) path from node *n* to a goal node.



A* Search

- Avoid expanding paths that are already expensive
- Evaluation function:
  - *$f(n) = g(n) + h(n)$*
    - $g(n)$ = exact cost so far to reach n
    - $h(n)$ = estimated cost to goal from n
    - $f(n)$ = estimated total cost of cheapest path from start to goal through n
  - *Also, $h(n) \geq 0$ and $h(G)=0$ for any goal G*

Optimality of A*

- A* is complete (finds a solution, if one exists)
- And is optimal (finds the optimal path to a goal) if:
  - *the branching factor is finite*
  - *arc costs are > 0*
  - *$h(n)$ is admissible*

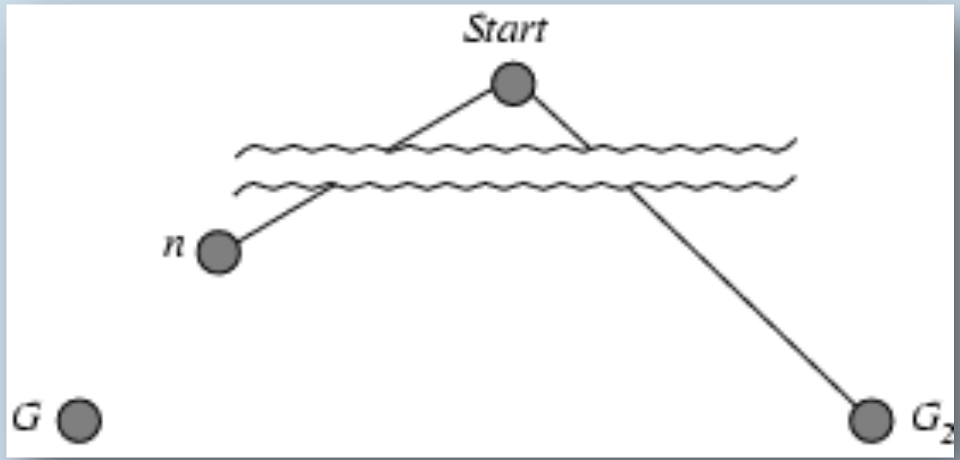
Admissibility of a heuristic

- A heuristic is admissible if it *never overestimates* the cost to reach the goal

- Let $c(n)$ denotes the optimal path from node n to any goal node. A search heuristic $h(n)$ is called admissible if $h(n) \leq c(n)$ for all nodes n, i.e., if for all nodes it is an underestimate of the cost to any goal.

Optimality of A* (tree-search proof)

- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let *n* be an unexpanded node in the fringe such that *n* is on a shortest path to an optimal goal *G*.



$f(G_2) = g(G_2)$	since $h(G_2) = 0$
$g(G_2) > g(G)$	since G_2 is suboptimal
$f(G) = g(G)$	since $h(G) = 0$
$f(G_2) > f(G)$	from above

Hence $f(G_2) > f(n)$, and A* will never select G_2 for expansion and thus A* is optimal

Optimality of A*

- A heuristic being **admissible** is not enough for graph-search problem
  - *Tree-search version of A* is optimal if h(n) is admissible*
- A* can return sub-optimal solutions, if we do not apply the uniform-cost approach (i.e., keep track of all generated paths, pick the one with least cost)
- However, this is really messy and expensive
- A much better solution is to ensure that the heuristic that you have selected is **consistent**

Consistent heuristic (monotonic)

- A heuristic *h(n)* is consistent if, for every node *n* and every successor *n'* of *n* generated by any action *a*, the estimated cost of reaching the goal from *n* is no greater than the step cost of getting to *n'* plus the estimated cost of reaching the goal from *n'*:

$$h(n) \le c(n,a,n') + h(n')$$

- If *n'* is a successor of n, then:

  $g(n') = g(n) + c(n,a,n')$

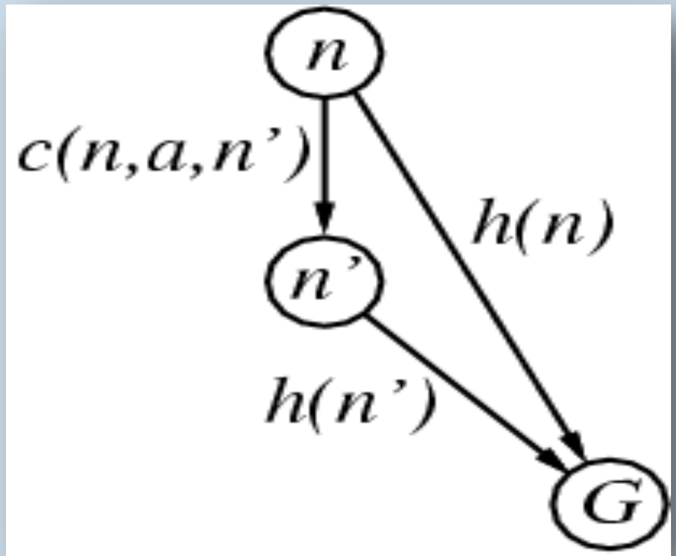
  And,

  $f(n') = g(n') + h(n')$

  $= g(n) + c(n,a,n') + h(n')$

  $\geq g(n) + h(n) = f(n)$

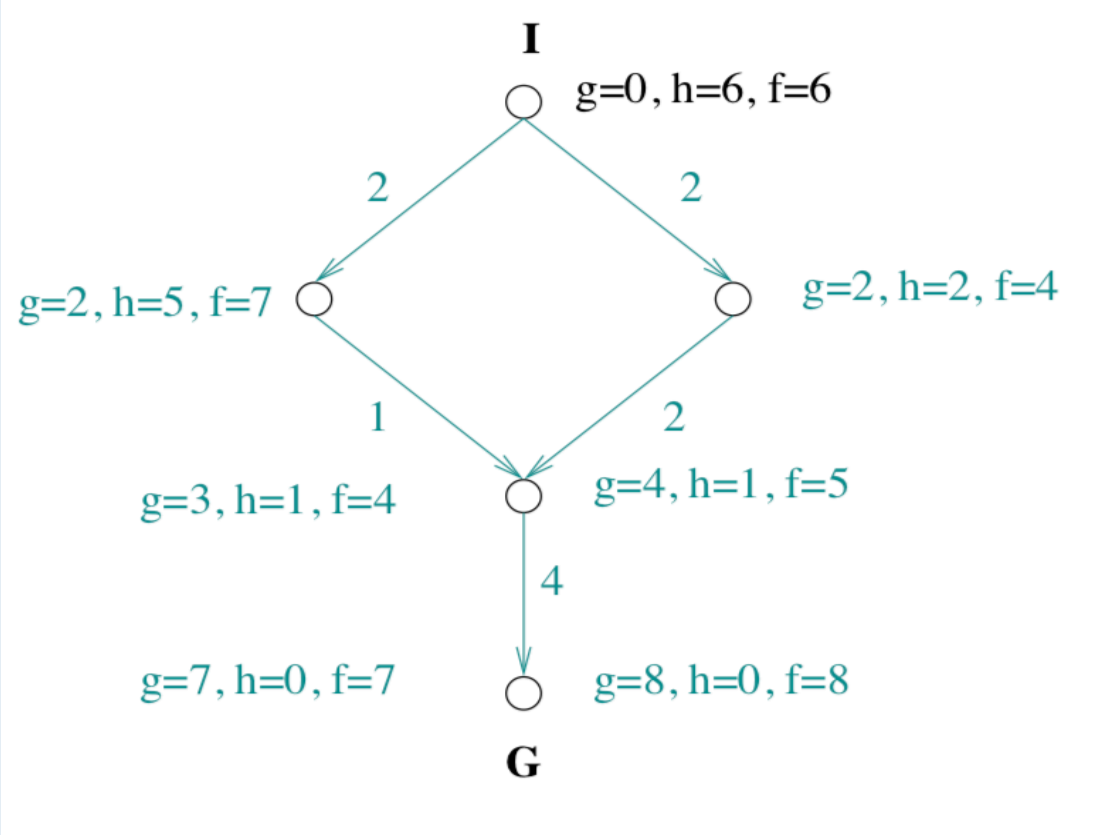
  i.e., *f(n)* is non-decreasing along any path



A consistent heuristic is admissible but not necessarily vice versa

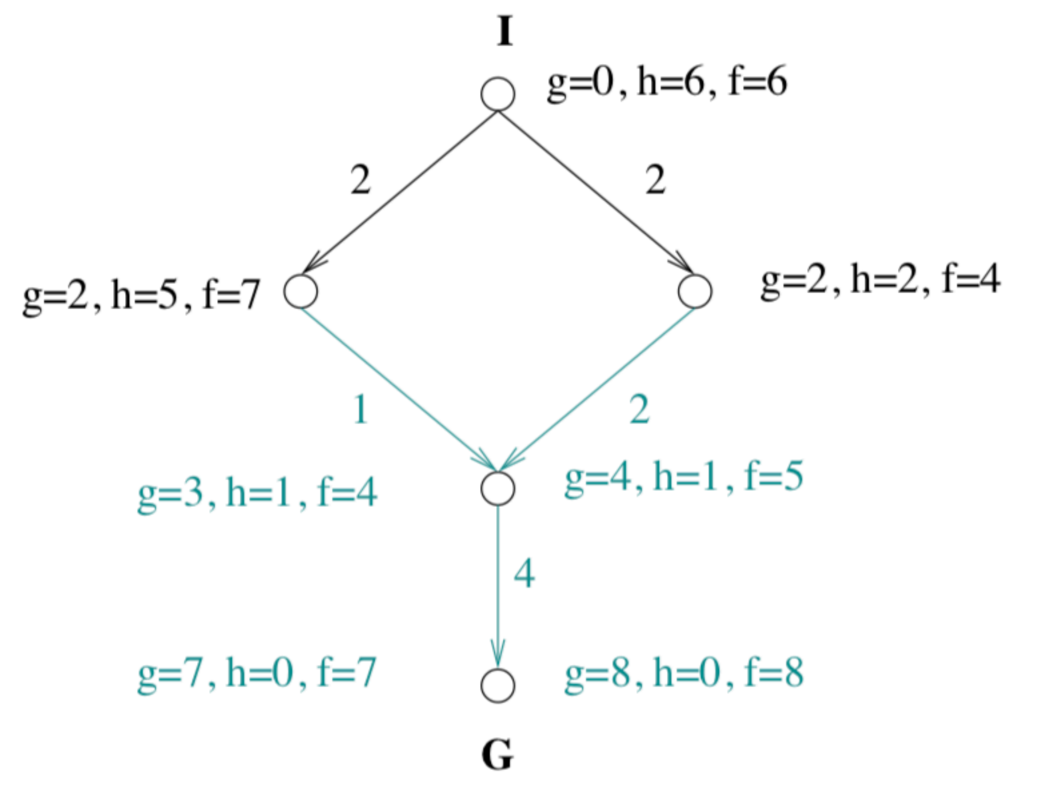
A* with an inconsistent heuristic

- Note that *h* is admissible, it never overestimates



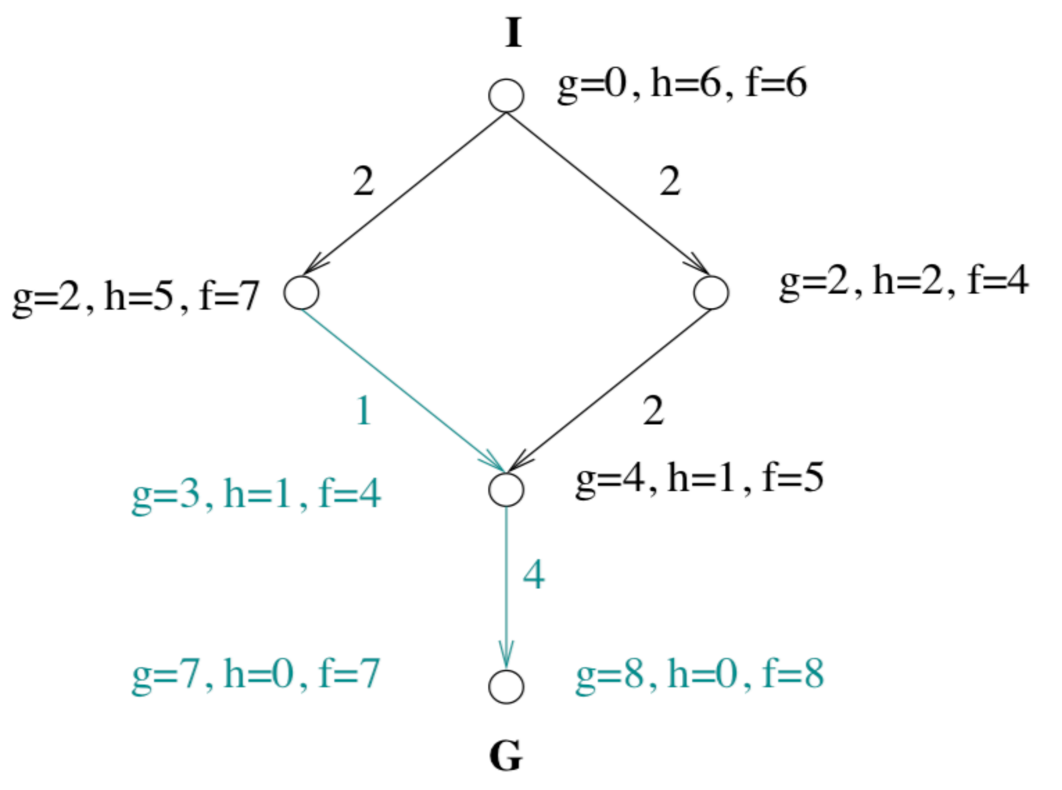
A* with an inconsistent heuristic

- The root node was expanded
- Note that *f* decreased from 6 to 4



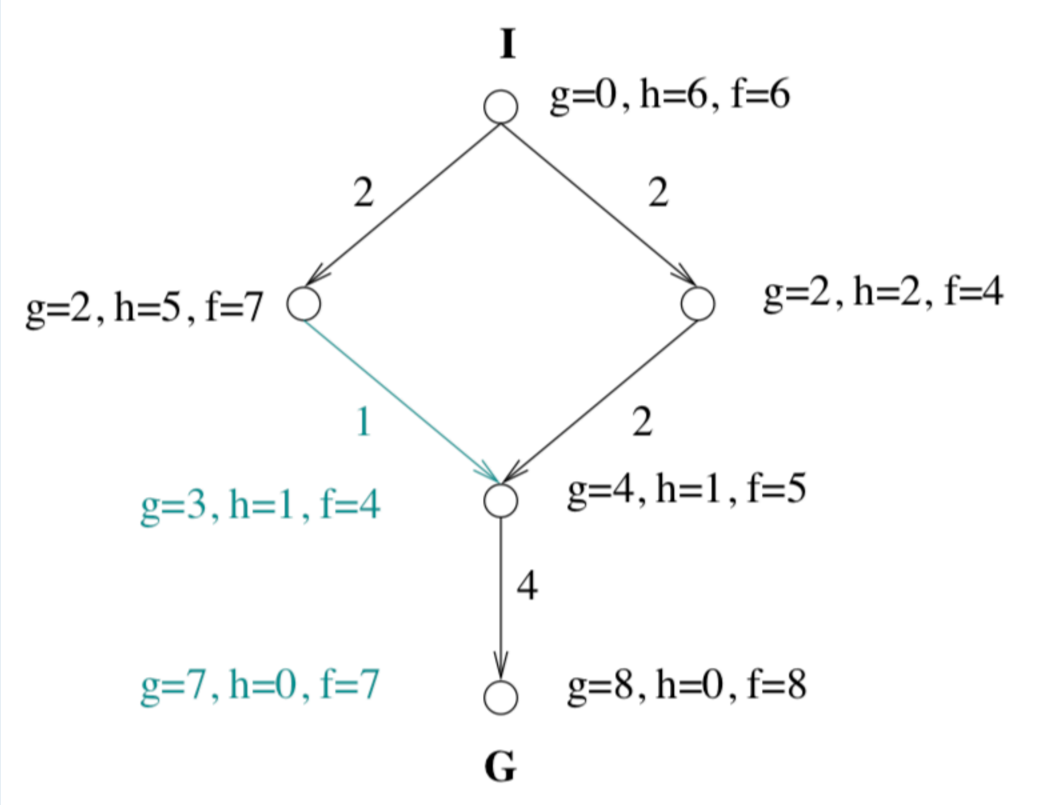
A* with an inconsistent heuristic

- The suboptimal path is being pursued.



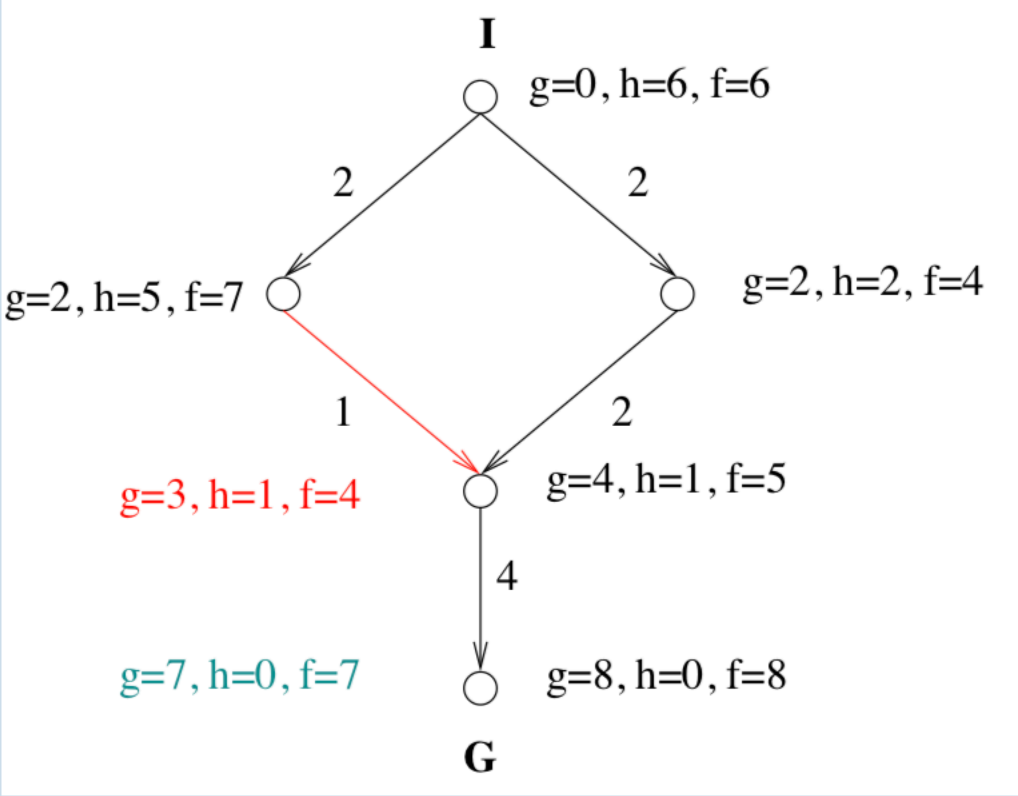
A* with an inconsistent heuristic

- Goal found, but we cannot stop until it is selected for expansion.



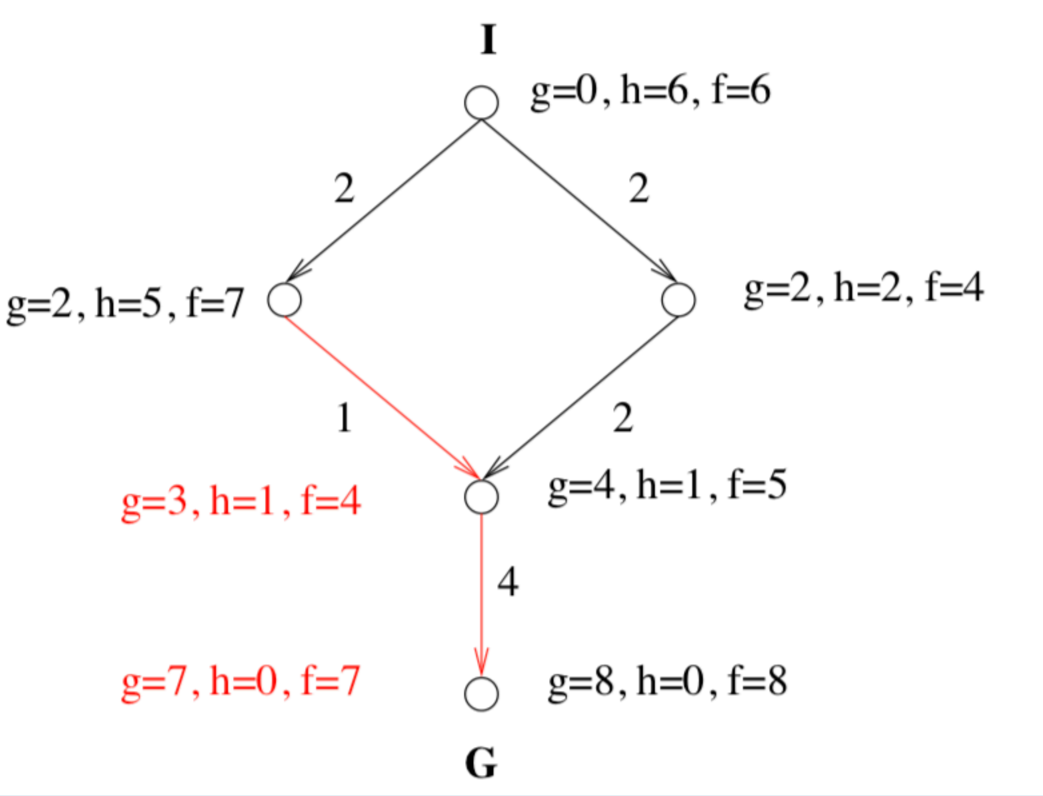
A* with an inconsistent heuristic

- The node with f = 7 is selected for expansion.



A* with an inconsistent heuristic

- The optimal path to the goal is found.



Consistent heuristic (monotonic)

- A heuristic *h(n)* is consistent if, for every node *n* and every successor *n'* of *n* generated by any action *a*, the estimated cost of reaching the goal from *n* is no greater than the step cost of getting to *n'* plus the estimated cost of reaching the goal from *n'*:

$$h(n) \le c(n,a,n') + h(n')$$

- If *n'* is a successor of n, then:

  $g(n') = g(n) + c(n,a,n')$

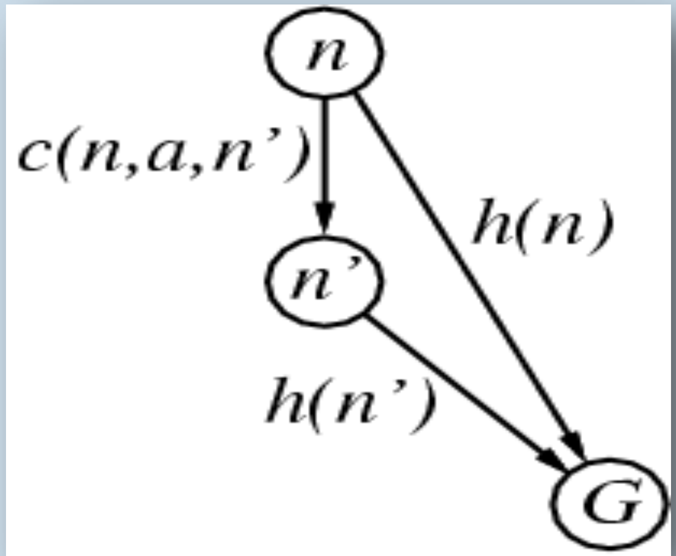
  And,

  $f(n') = g(n') + h(n')$

  $= g(n) + c(n,a,n') + h(n')$

  $\geq g(n) + h(n) = f(n)$

  i.e., *f(n)* is non-decreasing along any path

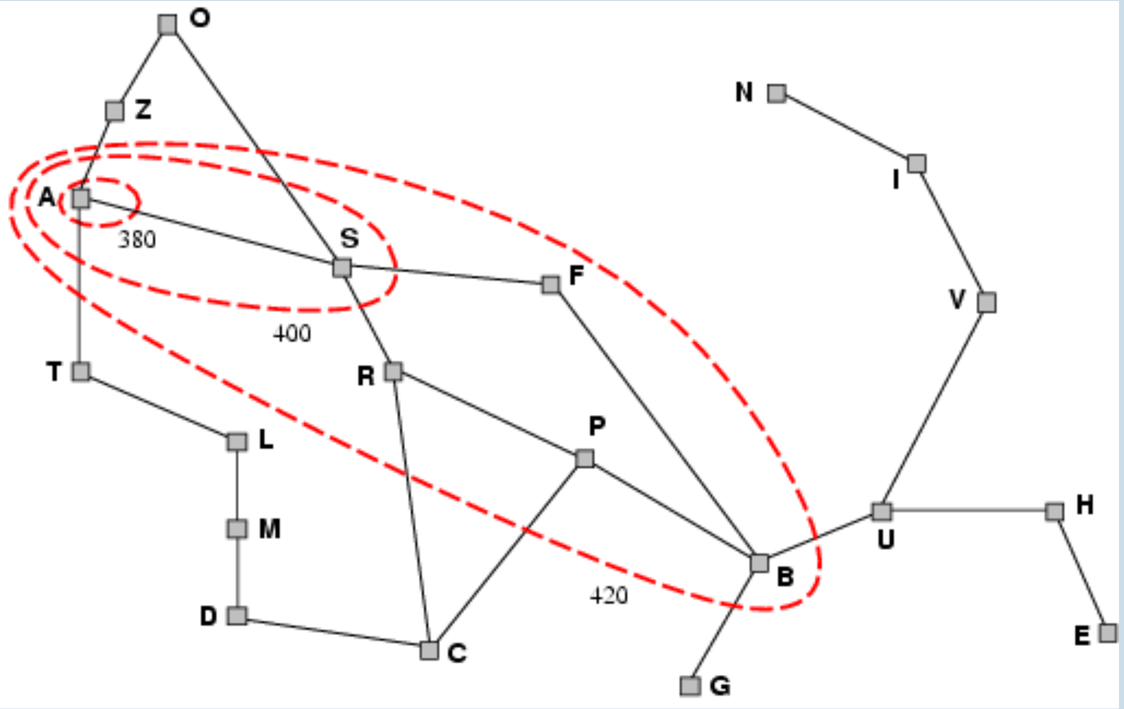


A consistent heuristic is admissible but not necessarily vice versa

Optimality of A* (graph-search)

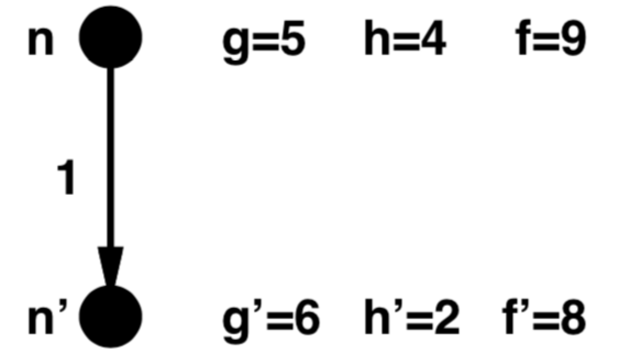
- Lemma: A* expands nodes in order of increasing *f* value
- Gradually adds "*f*-contours" of nodes
- Contour *i* has all nodes with $f=f_i$, where $f_i < f_{i+1}$

- With uniform-cost search (A* search with h(n)=0) the bands are “circular”.
With a more accurate heuristic, the bands will stretch toward the goal and become more narrowly focused around the optimal path.



Proof of Lemma: Pathmax

- For some admissible heuristic, *f may decrease along a path*
- For example, let's suppose *n'* is a successor of *n*



- But this throws away information!
- *f(n)* = 9 → true cost of a path through *n* is >= 9
- Hence, true cost of a path through *n'* is also >= 9

- Pathmax modification to A*:
  - Instead of using $f(n') = g(n') + h(n')$, use $f(n') = max(g(n') + h(n'), f(n))$
  - with pathmax, *f* is always nondecreasing along any path

Properties of A*

- Complete? Yes (unless there are infinitely many nodes with f ≤ $f(G)$)
- Time? Exponential
- Space? Keeps all nodes in memory
- Optimal? Yes – cannot expand f_{i+1} until f_i is finished
  - *A* expands all nodes with $f(n) < C^*$*
    A expands some nodes with $f(n) = C^*$*
  - *A* expands no nodes with $f(n) > C^*$*

Analysis of A*

- In fact, we can say something even stronger about A* (when it is admissible)

A* is optimally efficient among the algorithms that extend the search path from the initial state

It finds the goal with the minimum no. of expansions

Why A* is Optimally Efficient?

- No other optimal algorithm is guaranteed to expand fewer nodes than A* (given the same heuristic function)

- This is because any algorithm that **does not** expand every node with $f(n) < f(G)$ *(optimal goal)* risks missing the optimal solution

Effect of Search Heuristic

- A search heuristic that is a **better approximation** on the actual cost reduces the number of nodes expanded by A*

A* using h_2 will never expand more nodes than A* using h_1 (except possible for some nodes with f(n) = C*)

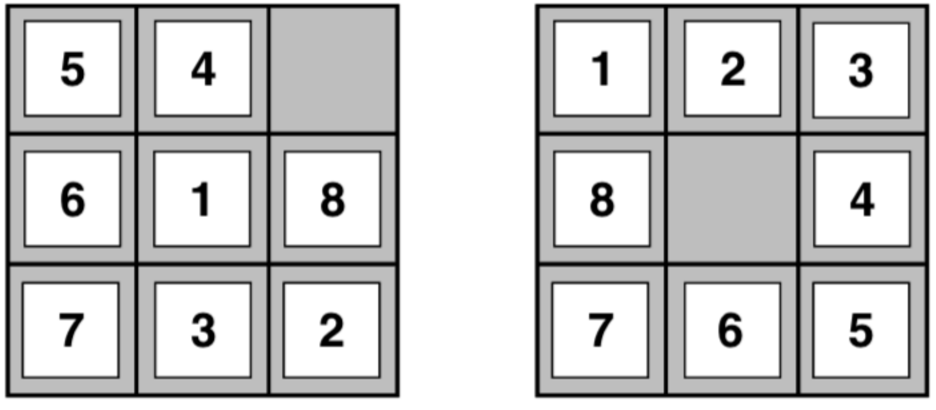
Example: 8puzzle:

(1) tiles can move anywhere

(h_1 : number of tiles that are out of place/misplaced)

(2) tiles can move to any adjacent square

(h_2 : sum of number of squares that separate each tile from its correct position, i.e. Manhattan distance)



Start State Goal State

$h_1(S) = 7$

$h_2(S) = 2 + 3 + 3 + 2 + 4 + 2 + 0 + 2 = 18$

If $h_2(n) >= h_1(n)$ for all *n* (both admissible) then h_2 <u>dominates</u> h_1 and is better for search

Effect of Search Heuristic

- A search heuristic that is a **better approximation** on the actual cost reduces the number of nodes expanded by A*

Example: 8puzzle:

(1) tiles can move anywhere

(**h_1 : number of tiles that are out of place)**

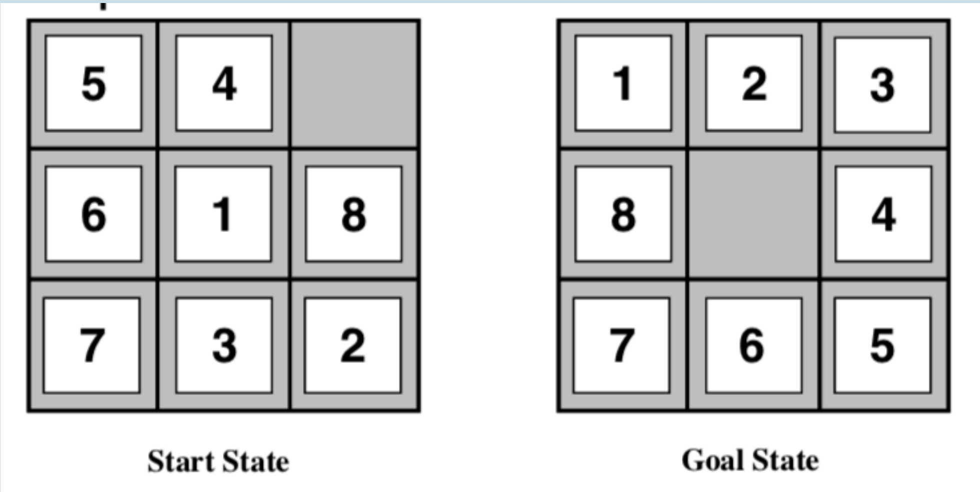
(2) tiles can move to any adjacent square

(**h_2 : sum of number of squares that separate each tile from its correct position)**

average number of paths expanded: (d = depth of the solution)

d=12 IDS = 3,644,035 paths
$A^*(h_1)$ = 227 paths
$A^*(h_2)$ = 73 paths

d=24 IDS = too many paths
$A^*(h_1)$ = 39,135 paths
$A^*(h_2)$ = 1,641 paths



	Complete	Optimal	Time	Space
DFS	N (Y if no cycles)	N	$O(b^m)$	$O(bm)$
BFS	Y	Y	$O(b^m)$	$O(b^m)$
IDS	Y	Y	$O(b^m)$	$O(bm)$
UCS (when arc costs available)	Y Costs > 0	Y Costs >=0	$O(b^m)$	$O(b^m)$
Best First (when *h* available)	N	N	$O(b^m)$	$O(b^m)$
A* (when arc costs > 0 and *h* admissible)	Y	Y	$O(b^m)$	$O(b^m)$

Source: UC Berkeley CS 188 Lecture Slides

Search algorithms often used in practice

- IDS (iterative deepening search)
- A*: many times, with variations
  - *IDA* (iterative deepening A*)*
    - Idea: perform iterations of DFS. The cutoff is defined based on the f-cost rather than the depth of a node.

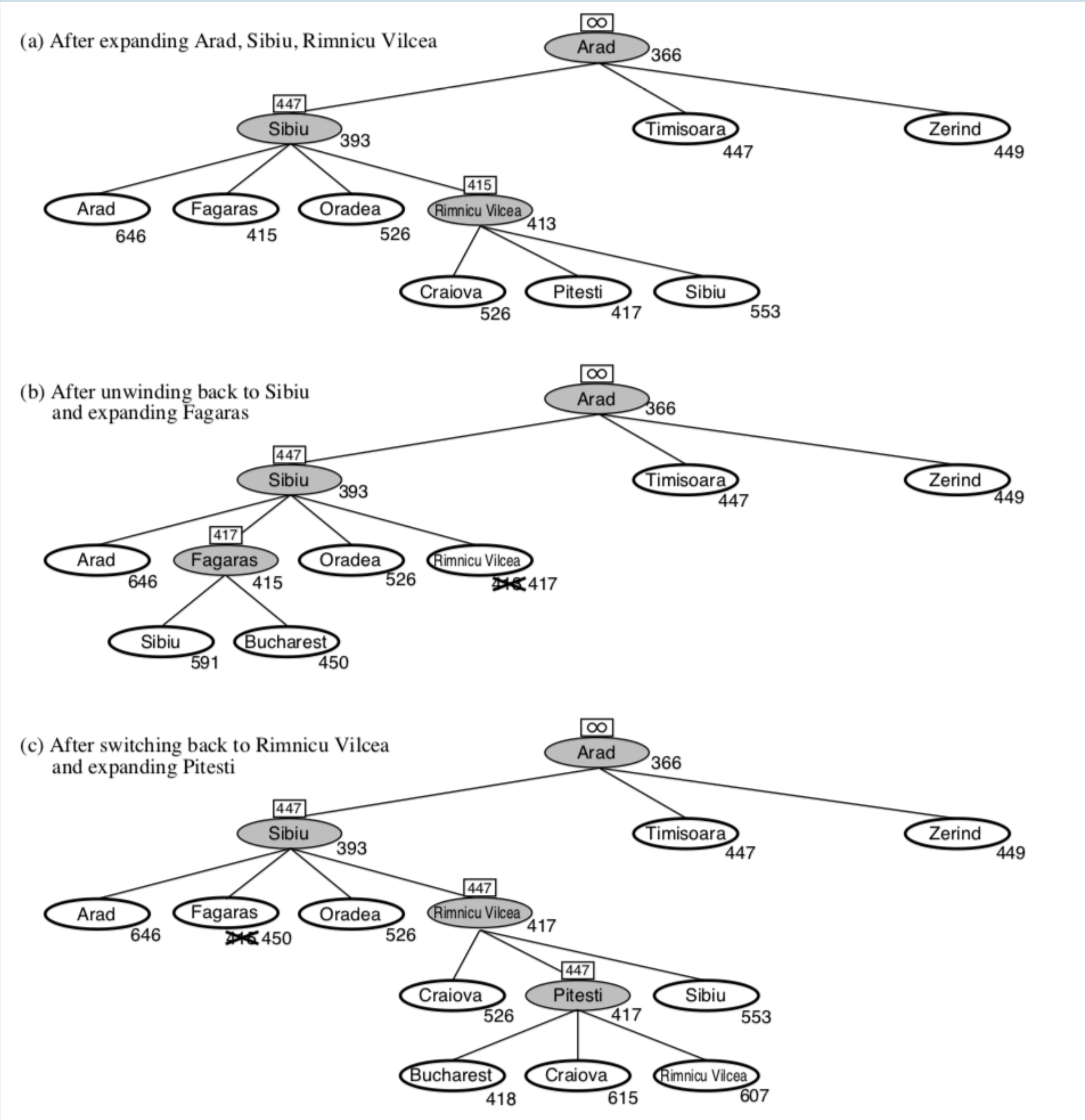
Recursive best-first search (RBFS)

- Idea: mimic the operation of standard best-first search, but use only linear space

- Runs similar to recursive depth-first search, but rather than continuing indefinitely down the current path, it uses the *f-limit* variable to keep track of the best alternative path available from any ancestor of the current node.

- If the current node exceeds this limit, the recursion unwinds back to the alternative path. As the recursion unwinds, RBFS replaces the *f-value* of each node along the path with the best *f-value* of its children. In this way, it can decide whether it's worth re-expanding a forgotten subtree.

RBFS



(a) After expanding Arad, Sibiu, Rimnicu Vilcea
∞
Arad
366
447
Sibiu
393
Timisoara
447
Zerind
449
Arad
646
Fagaras
415
Oradea
526
415
Rimnicu Vilcea
413
Craiova
526
Pitesti
417
Sibiu
553
(b) After unwinding back to Sibiu
and expanding Fagaras
∞
Arad
366
447
Sibiu
393
Timisoara
447
Zerind
449
Arad
646
417
Fagaras
415
Oradea
526
Rimnicu Vilcea
417
Sibiu
591
Bucharest
450
(c) After switching back to Rimnicu Vilcea
and expanding Pitesti
∞
Arad
366
447
Sibiu
393
Timisoara
447
Zerind
449
Arad
646
Fagaras
450
Oradea
526
447
Rimnicu Vilcea
417
Craiova
526
447
Pitesti
417
Sibiu
553
Bucharest
418
Craiova
615
Rimnicu Vilcea
607

Properties of RBFS

- Complete? Yes – similar to A*
- Optimal? Yes – similar to A*
- Time? difficult to characterize: it depends both on the accuracy of the heuristic function and on how often the best path changes as nodes are expanded. Each mind change corresponds to an iteration of IDA∗, and could require many reexpansions of forgotten nodes to recreate the best path and extend it one more node. RBFS is somewhat more efficient than IDA∗, but still suffers from excessive node regeneration.
- Space? IDA∗ and RBFS suffer from using too little memory. Between iterations, IDA∗ retains only a single number: the current *f-cost* limit. RBFS retains more information in memory, but only uses O(bd) memory. Even if more memory is available, RBFS has no way to make use of it.

Remember Deep Blue?

- Deep Blue's Results in the second tournament:
  - *second tournament: won 3 games, lost 2, tied 1*

- 30 CPUs + 480 chess processors
- Searched 126.000.000 nodes per sec
- Generated 30 billion positions per move reaching depth 14 routinely

- Iterative Deepening with evaluation function (similar to a heuristic) based on 8000 features (e.g., sum of worth of pieces: pawn 1, rook 5, queen 10)

Reading Material

- Russell & Norvig: Chapter # 3
- David Poole: Chapter # 3
- Reading material on “Search algorithms” uploaded on the Google Classroom
- An article: *A*’s use of Heuristic*

http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html