CS 380: Artificial Intelligence

Lecture 4: Informed Search, A*, & Heuristics

Where we are…

- Problem Solving:
  - Problem Formulation
- Search — finding a solution:
  - Uninformed search
  - Informed search

- Let's finish up Uninformed Search…

Uniform-cost search

Expand least-cost unexpanded node

Implementation:

$fringe$ = queue ordered by path cost, lowest first

with path cost function *g*

Equivalent to breadth-first if step costs all equal

each action costs at least ε

Complete?? Yes, if step cost $\geq \epsilon$

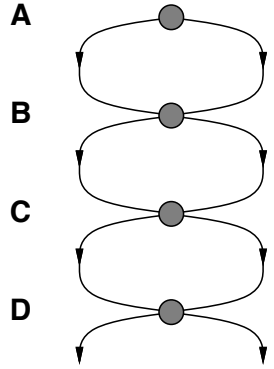
Time?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$
where C^* is the cost of the optimal solution

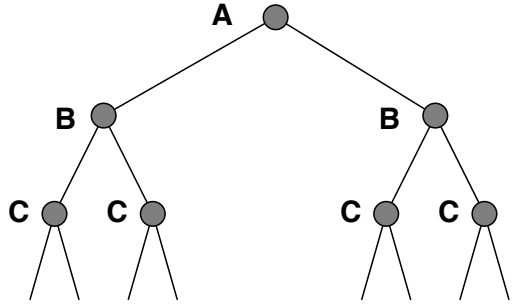
Space?? # of nodes with $g \leq$ cost of optimal solution, $O(b^{\lceil C^*/\epsilon \rceil})$

Optimal?? Yes—nodes expanded in increasing order of $g(n)$

Repeated states

Failure to detect repeated states can turn a linear problem into an exponential one!





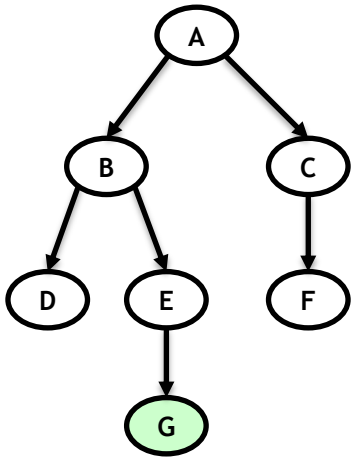
Graph search

```
function GRAPH-SEARCH( problem, fringe) returns a solution, or failure
  closed ← an empty set
  fringe ← INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)
  loop do
      if fringe is empty then return failure
      node ← REMOVE-FRONT(fringe)
      if GOAL-TEST(problem, STATE[node]) then return node
      if STATE[node] is not in closed then
          add STATE[node] to closed
          fringe ← INSERTALL(EXPAND(node, problem), fringe)
  end
```

Review

- Search methods:
  - They manage a list of **OPEN** nodes (or "fringe")
  - Different strategies (BFS, DFS, etc.) constructed by expanding nodes in that list in different orders.
  - For example, BFS:

Iteration 1: [A]
(expand A): Remove A, Insert B,C
Iteration 2: [B,C]
(expand B): Remove B, Insert D,E
Iteration 3: [C,D,E]
(expand C): Remove C, Insert F
Iteration 4: [D,E,F]
(expand D): Remove D
Iteration 5: [E,F]
(expand E): Remove E, Insert G
Iteration 6: [F,G]
(expand F): Remove F
Iteration 7: [G]



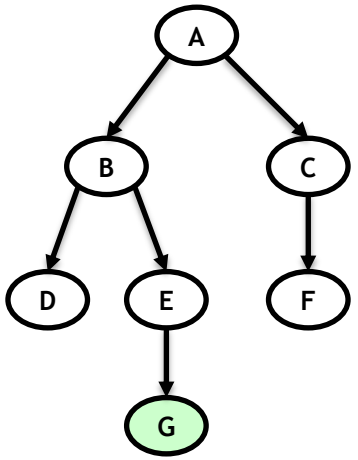
Review

- Search methods:
  - They manage a list of **OPEN** n
  - Different strategies (BFS, DFS, expanding nodes in that list ir
  - For example, BFS:

BFS: we remove the first element in the list, and add its children at the end.

DFS: we remove the first element in the list and add its children at the beginning.

Iteration 1: [A]
(expand A): Remove A, Insert B,C
Iteration 2: [B,C]
(expand B): Remove B, Insert D,E
Iteration 3: [C,D,E]
(expand C): Remove C, Insert F
Iteration 4: [D,E,F]
(expand D): Remove D
Iteration 5: [E,F]
(expand E): Remove E, Insert G
Iteration 6: [F,G]
(expand F): Remove F
Iteration 7: [G]



Review

- Search methods:
  - They manage a list of **OPEN** n
  - Different strategies (BFS, DFS, expanding nodes in that list ir
  - For example, BFS:

Iteration 1: [A]
(expand A): Remove A, In
Iteration 2: [B,C]
(expand B): Remove B, In
Iteration 3: [C,D,E]
(expand C): Remove C, In
Iteration 4: [D,E,F]
(expand D): Remove D
Iteration 5: [E,F]
(expand E): Remove E, Ins
Iteration 6: [F,G]
(expand F): Remove F
Iteration 7: [G]

BFS: we remove the first element in the list, and add its children at the end.

DFS: we remove the first element in the list and add its children at the beginning.

In other words:

BFS: we expand the node with the smallest depth

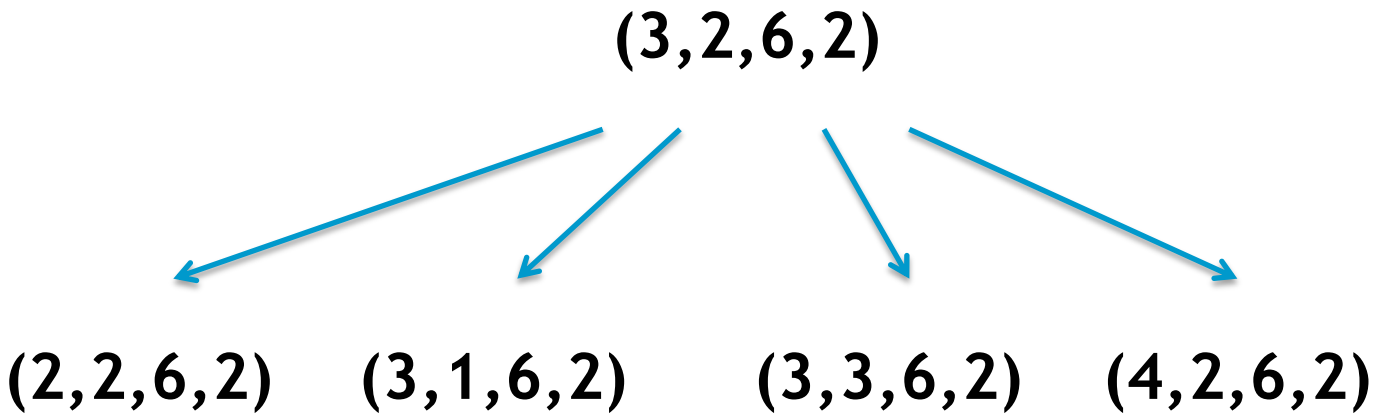
DFS: we expand the node with the largest depth.

Uninformed → Informed Search

- All the problem solving strategies we have seen so far use the following knowledge:
  - **Goal check**: knowing if the problem is solved
  - **States**
  - **Actions**: an agent knows which actions can be executed in the current state.

- Note that this is very little information

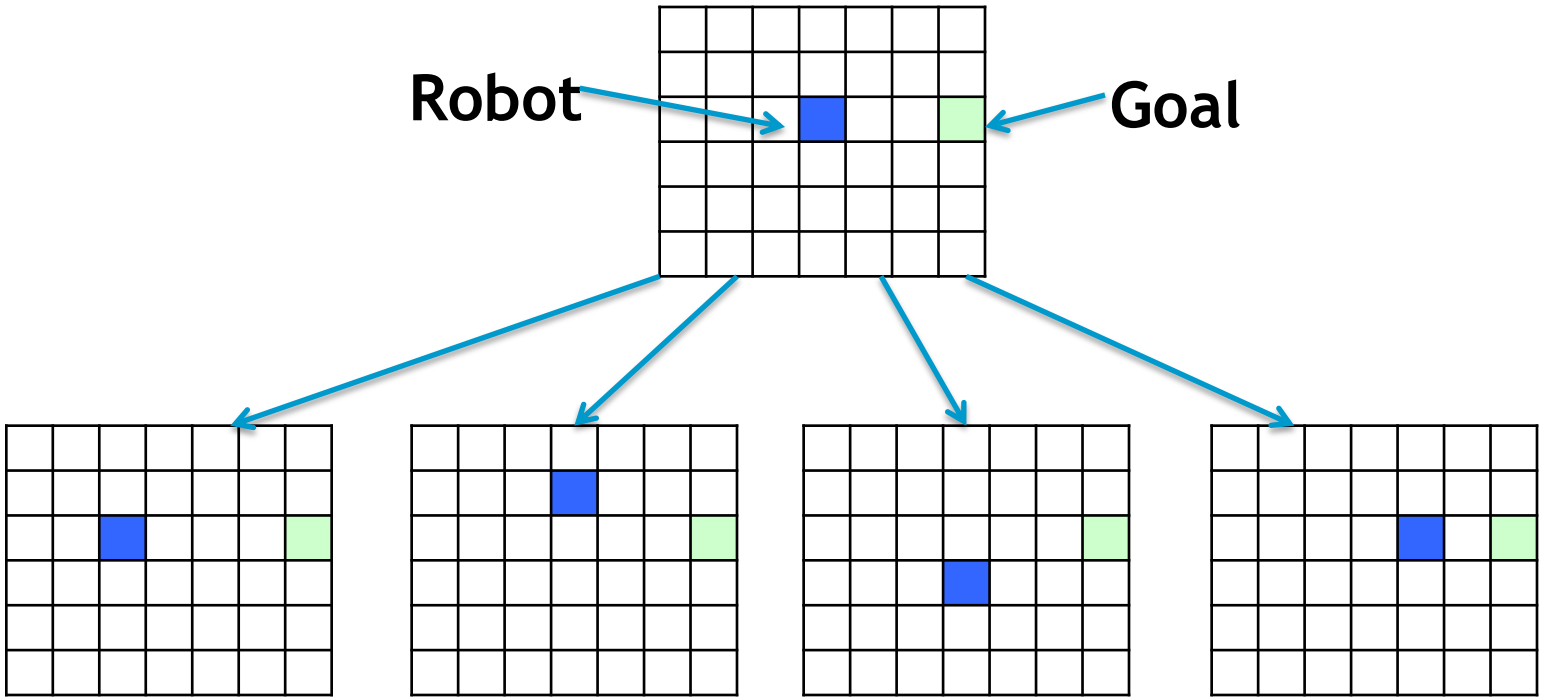
Example



There are 4 possible actions to execute. Now, if you were the robot, which one would you try first?

Example

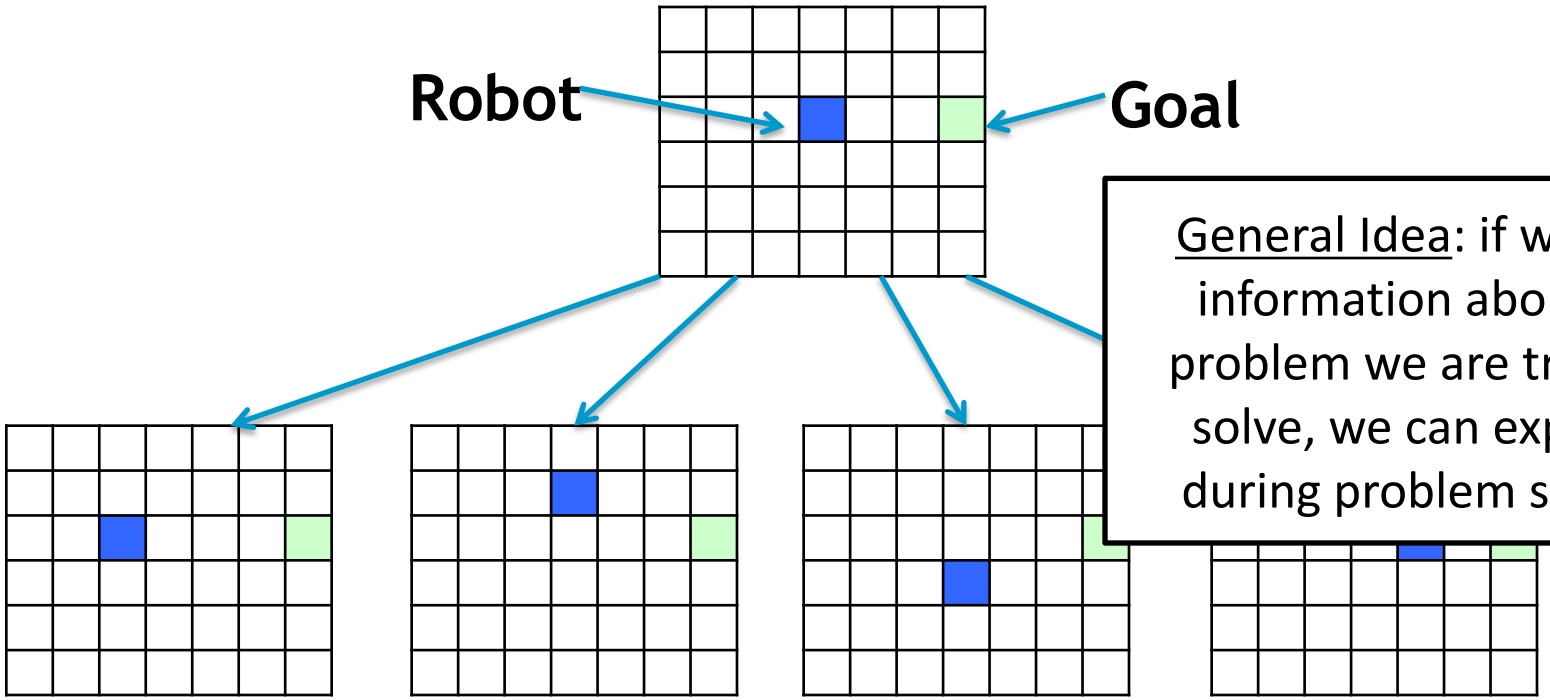
What if now you know what the coordinates mean?



There are 4 possible actions to execute. If you were the robot, which one would you try first?

Example

What if now you know what the coordinates mean?



General Idea: if we have information about the problem we are trying to solve, we can exploit it during problem solving.

There are 4 possible actions to execute. If you were the robot, which one would you try first?

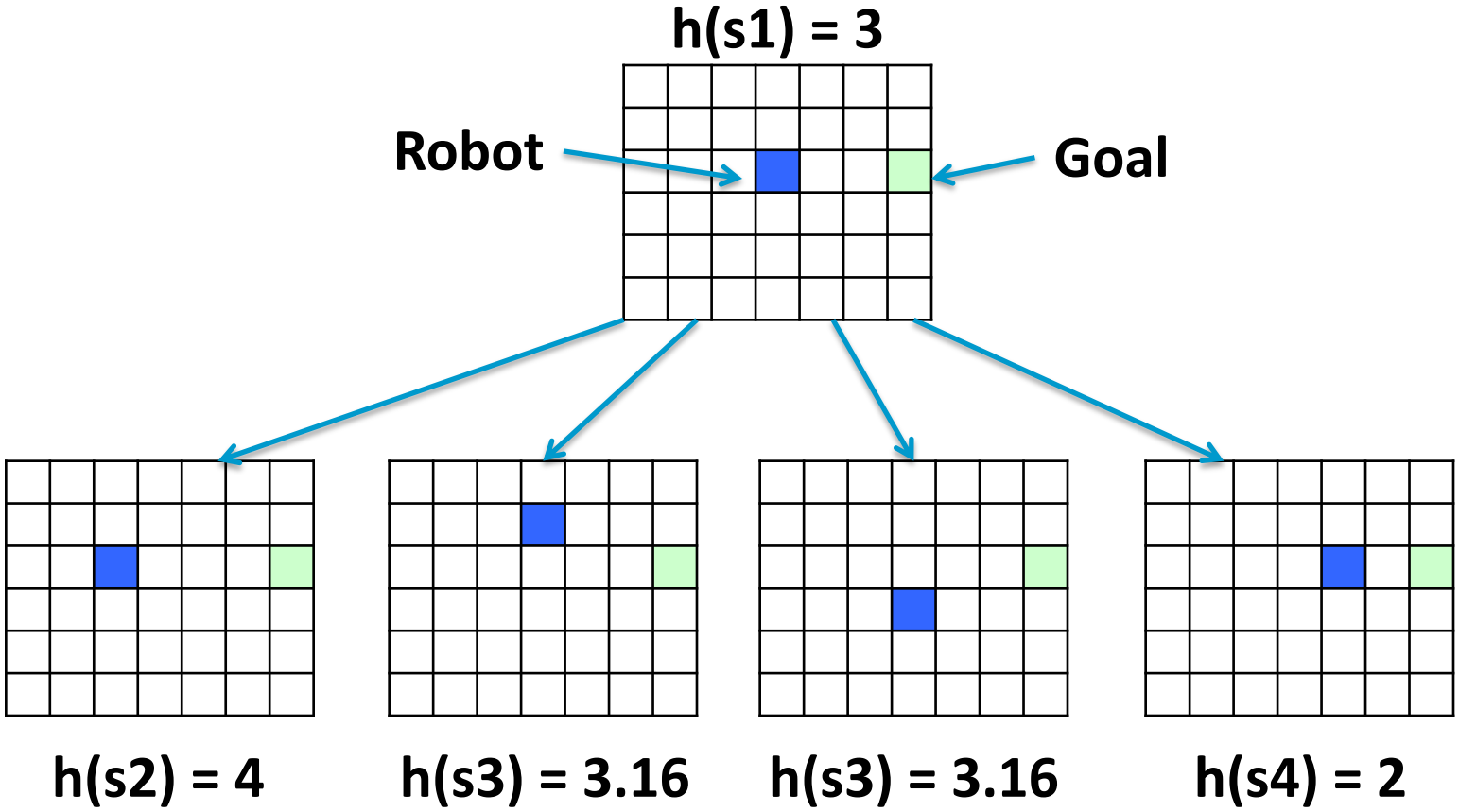
Evaluation Function

- Idea: represent the information we have about the domain as an "evaluation function" ***h***

- Evaluation function or "***heuristic***":
  - Given a state ***s*** …
  - ***h(s)*** estimates how close or how far it is from the goal

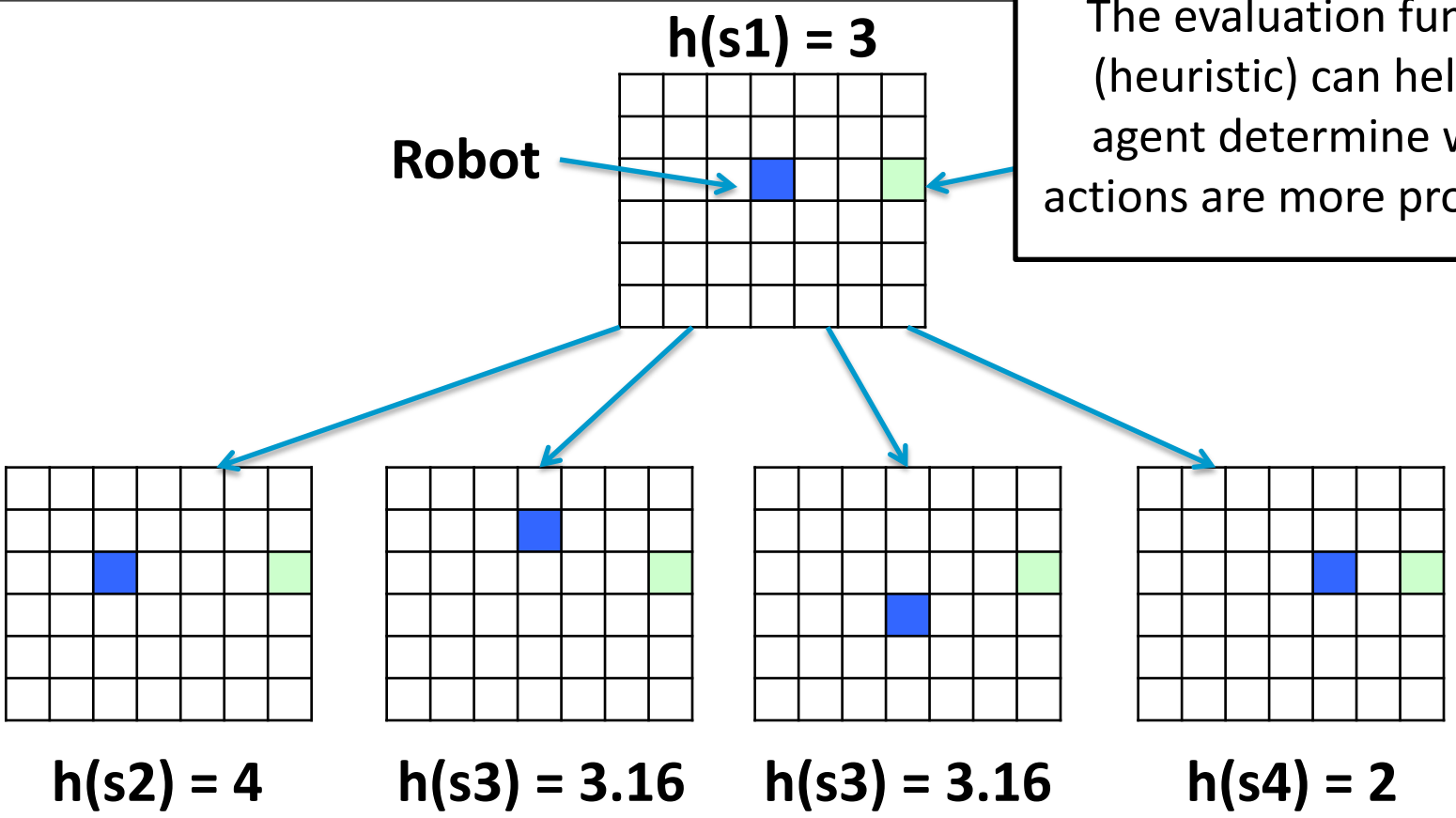
- For example, in a maze solving problem…
  - Euclidean distance to the goal?
  - Manhattan distance?
  - Etc.

Example



h: Euclidean distance to the goal

Example



h: Euclidean distance to the goal

Best-first search

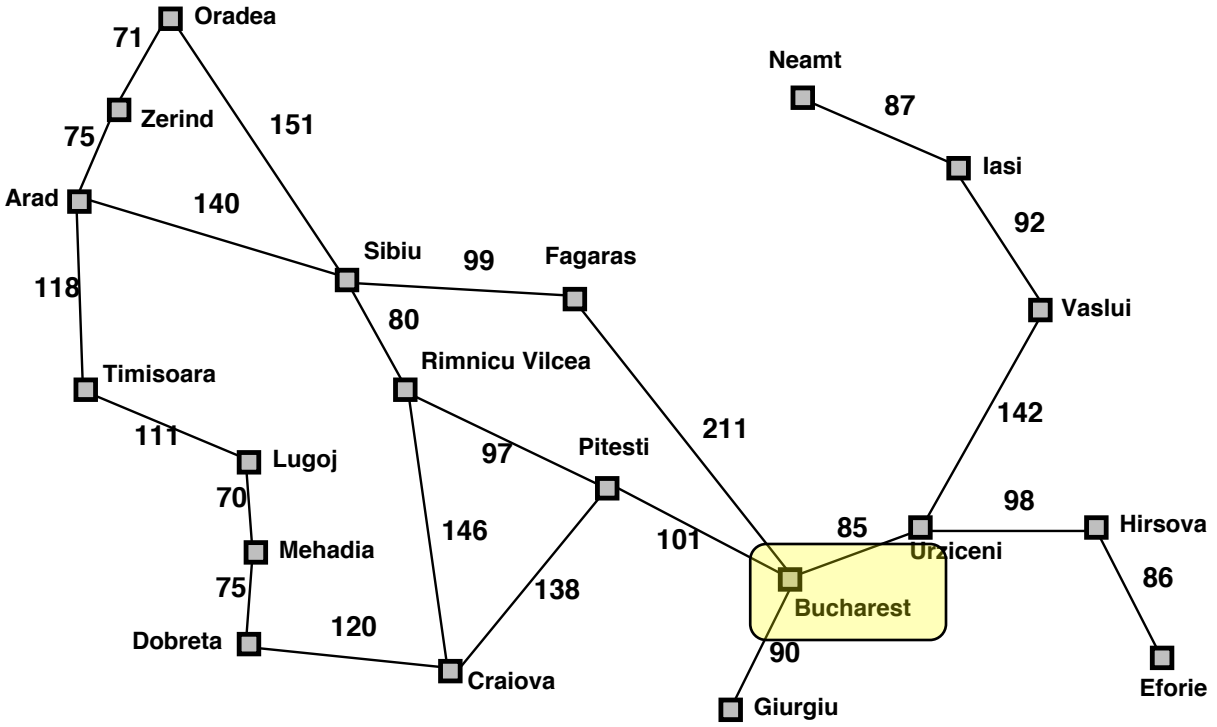
Idea: use an evaluation function for each node
– estimate of "desirability"

$\Rightarrow$ Expand most desirable unexpanded node

Implementation:
fringe is a queue sorted in decreasing order of desirability

Special cases:
greedy search
A* search

Romania with step costs in km



Straight–line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	178
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	98
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy search

Evaluation function $h(n)$ (**h**euristic)
$=$ estimate of cost from n to the closest goal

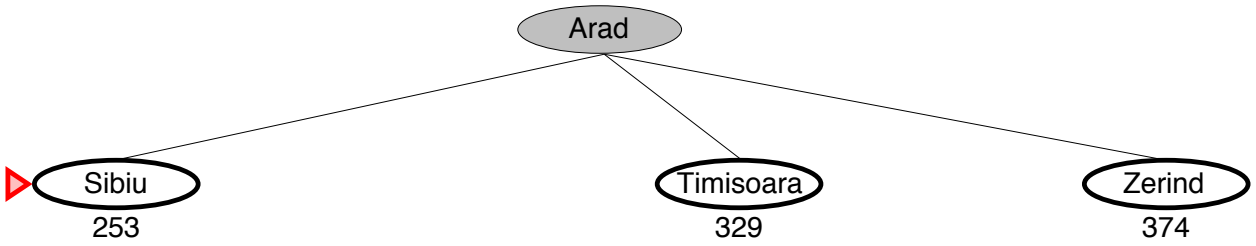
E.g., $h_{\mathrm{SLD}}(n) =$ straight-line distance from n to Bucharest

Greedy search expands the node that **appears** to be closest to goal

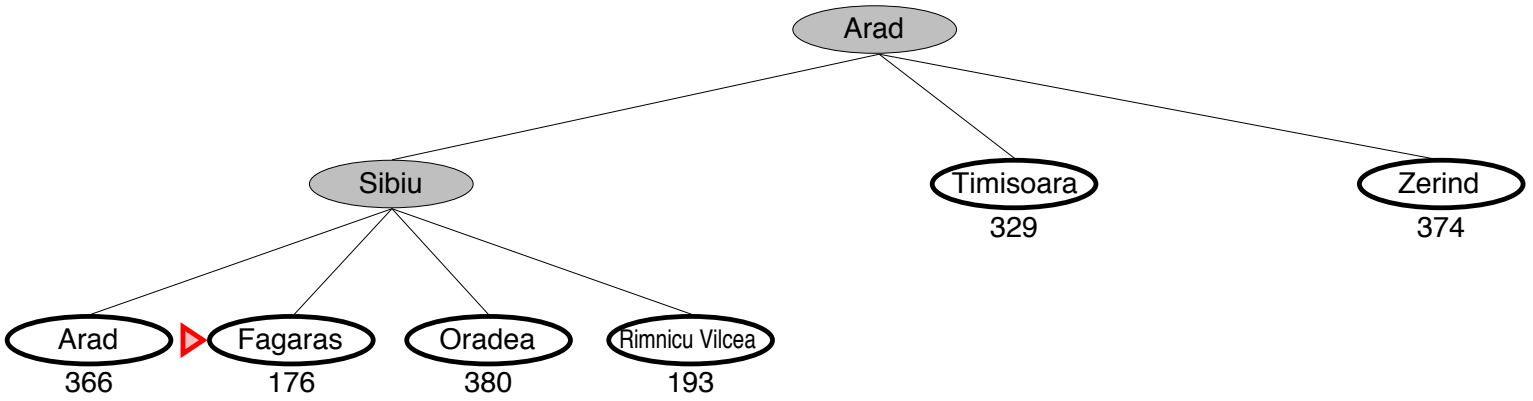
(What might go wrong with this heuristic?)

Greedy search example

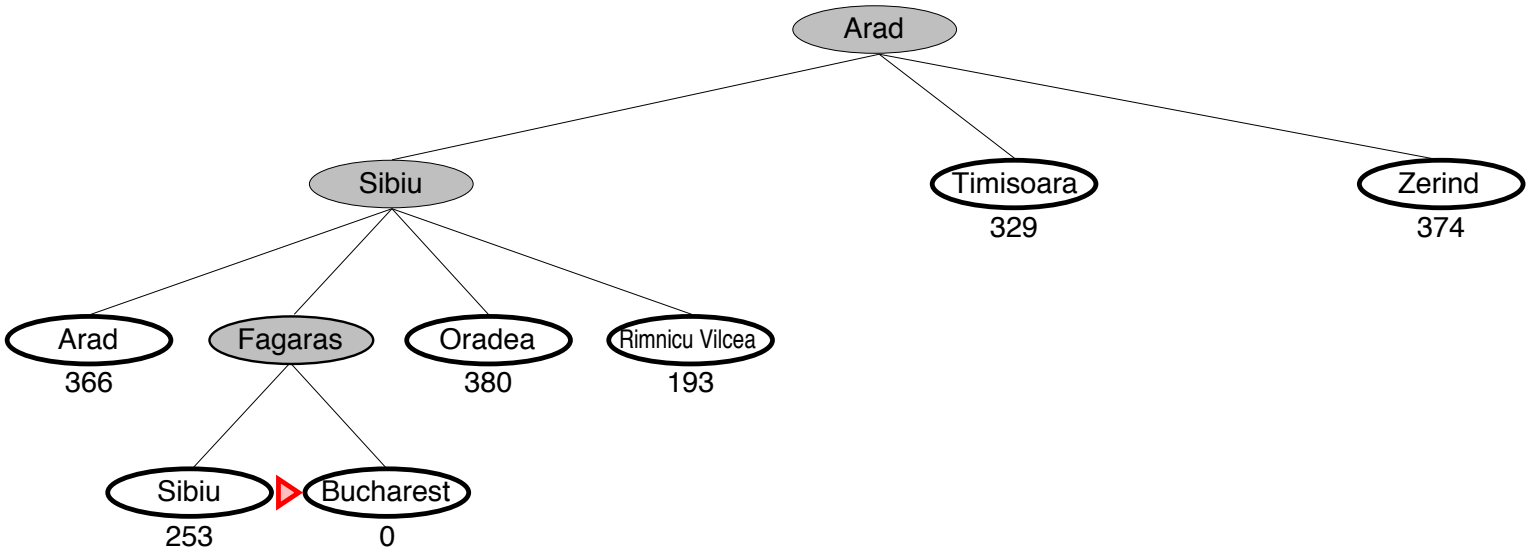
Greedy search example



Greedy search example



Greedy search example



Properties of greedy search

Complete??

Properties of greedy search

Complete?? No–can get stuck in loops, e.g., with Oradea as goal,

Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$

Complete in finite space with repeated-state checking

Time??

Properties of greedy search

Complete?? No–can get stuck in loops, e.g.,

Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$

Complete in finite space with repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

Space??

Properties of greedy search

Complete?? No–can get stuck in loops, e.g.,
Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$
Complete in finite space with repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

Space?? $O(b^m)$—keeps all nodes in memory

Optimal??

Properties of greedy search

Complete?? No–can get stuck in loops, e.g.,
Iasi $\rightarrow$ Neamt $\rightarrow$ Iasi $\rightarrow$ Neamt $\rightarrow$
Complete in finite space with repeated-state checking

Time?? $O(b^m)$, but a good heuristic can give dramatic improvement

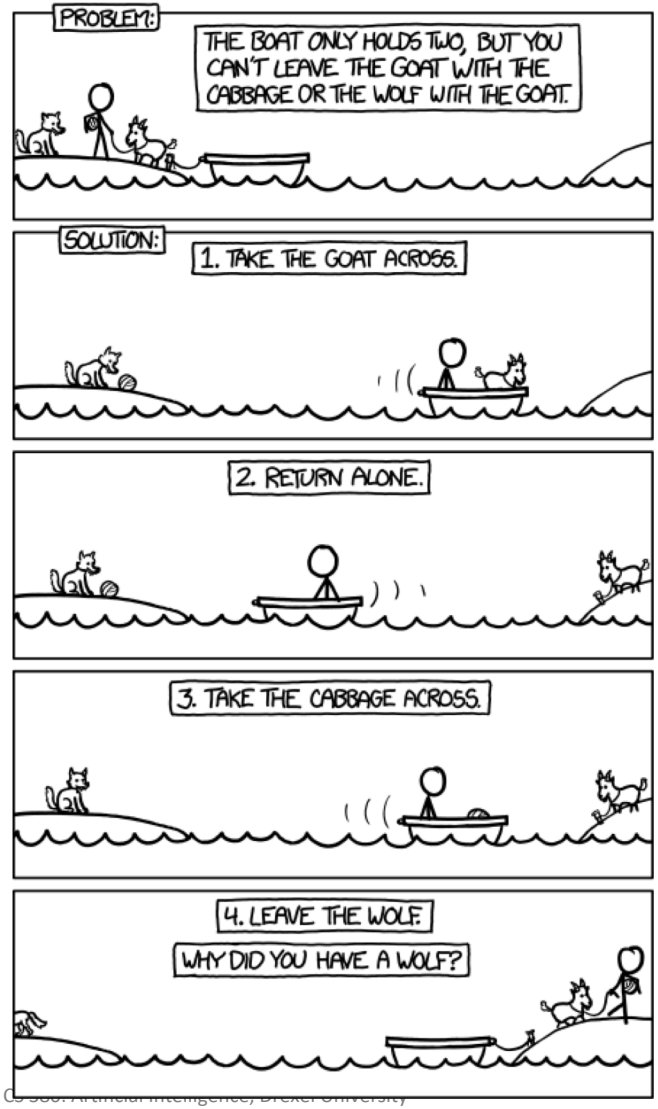
Space?? $O(b^m)$—keeps all nodes in memory

Optimal?? No

Heuristics

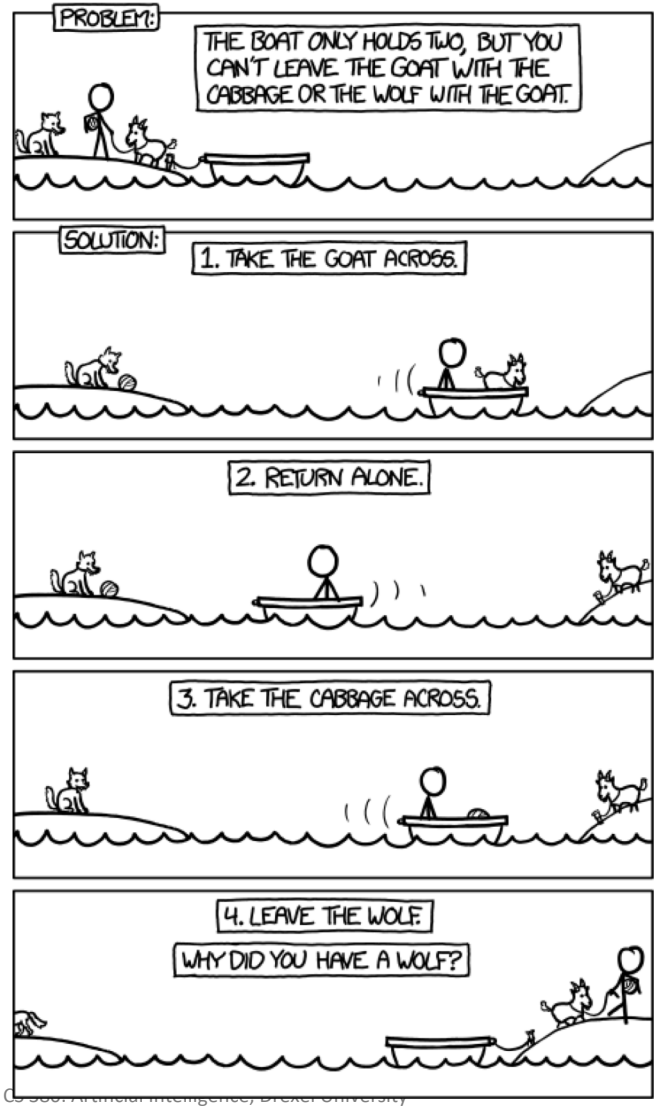
- Idea: estimate the distance to the solution.
- Must be computationally efficient (otherwise, one could just run the complete search algorithm to determine the actual cost!).
- Finding good heuristics makes a huge difference in computation time.

Heuristics: Fox-Goat-Cabbage



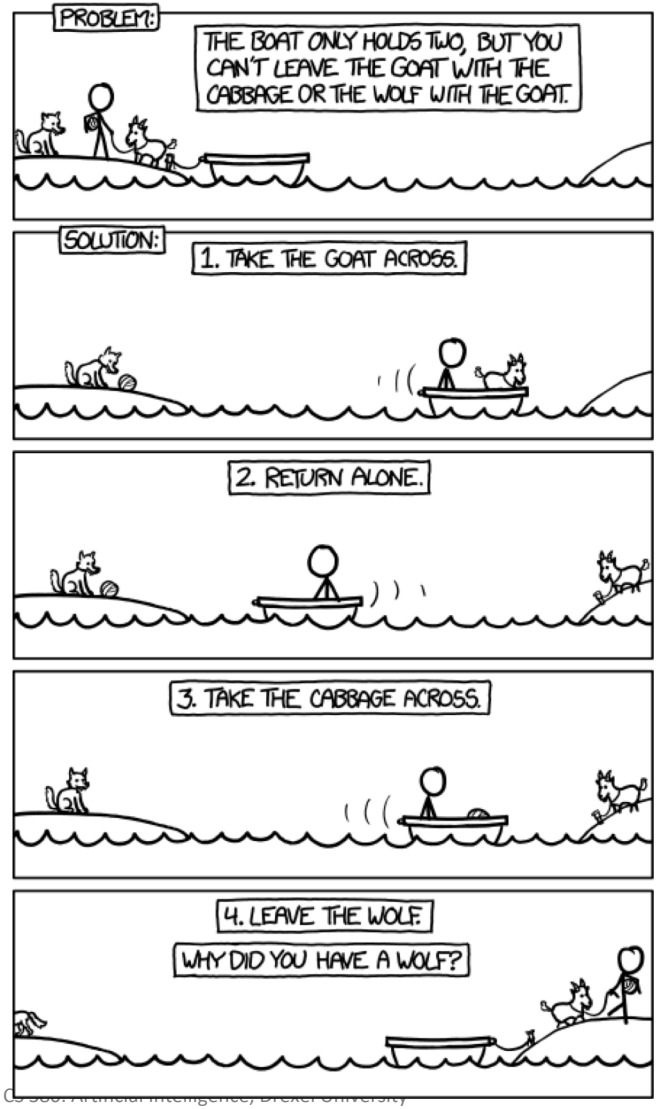
- Let's assume that "cost" is the number of boat trips.
- What would be a good heuristic?

Heuristics: Fox-Goat-Cabbage



- Let's assume that "cost" is the number of boat trips.
- What would be a good heuristic?
- Example:
  - Number of "items" on the left bay (since we know that at least we have to make that number of trips)

Heuristics: Fox-Goat-Cabbage



- Let's assume that "cost" is the number of boat trips.
- What would be a good heuristic?
- Example:
  - Number of "items" on the left bay (since we know that at least we have to make that number of trips)
  - Better:
    - ([*number of items in left bay*] – 1) * 2.
    - Add 1 if boat on right bay and [*number of items in left bay*] > 0

Heuristics: 8-Puzzle

8		1
3	2	6
7	4	5

- Cost: number of moves
- What would be a good heuristic?

Heuristics: 8-Puzzle

8		1
3	2	6
7	4	5

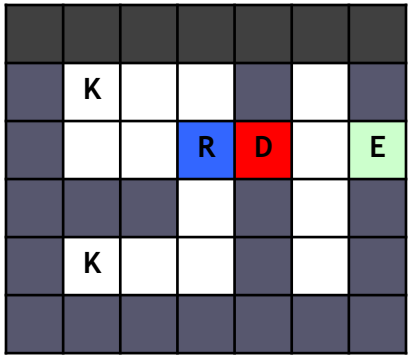
- Cost: number of moves
- What would be a good heuristic?
  - Heuristic 1:
    - Number of blocks out of place

Heuristics: 8-Puzzle

8		1
3	2	6
7	4	5

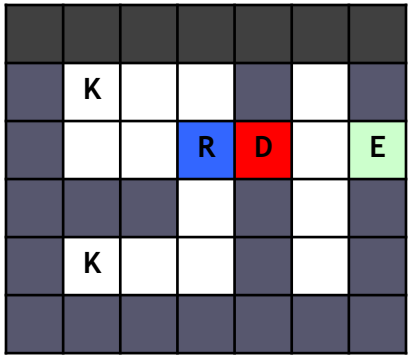
- Cost: number of moves
- What would be a good heuristic?
  - Heuristic 1:
    - Number of blocks out of place
  - Heuristic 2:
    - Distance of each block to its target position

Heuristic: Robot navigation with keys



- A robot (blue) needs to exit a maze (light green), and has to pick up keys to cross doors (red)

Heuristic: Robot navigation with keys



- A robot needs to exit a maze, and has to pick up keys to cross doors
- Heuristic 1:
  - Simply Euclidean distance to goal (ignoring keys and doors)
  - Can you improve it?

A Note on Graph Search

- Repeated-state checking:
  - When the search state is a graph, strategies like DFS can get stuck in loops.
  - Algorithms need to keep a list (CLOSED) of already-visited nodes.
- In DFS:
  - If we want to avoid repeating states completely, we need to keep ALL the visited states in memory (in the CLOSED list)
  - If we just want to avoid loops, we only need to remember the current branch (linear memory as a function of “m”)

A* Search

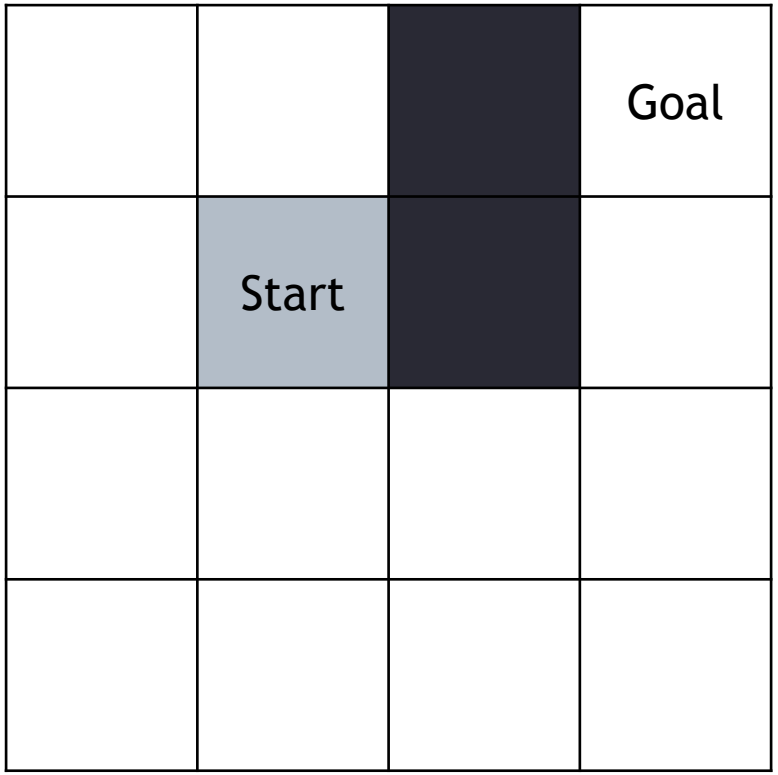
- Pronounced “A-Star”
- At each cycle, expand the node with the lowest *f(n)*

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

- A* implementations assume repeated-state checking (i.e., assume the search space is a graph)
- Thus, they use two lists simultaneously:
  - OPEN: list of nodes that need to be expanded
  - CLOSED: list of nodes that have already been expanded

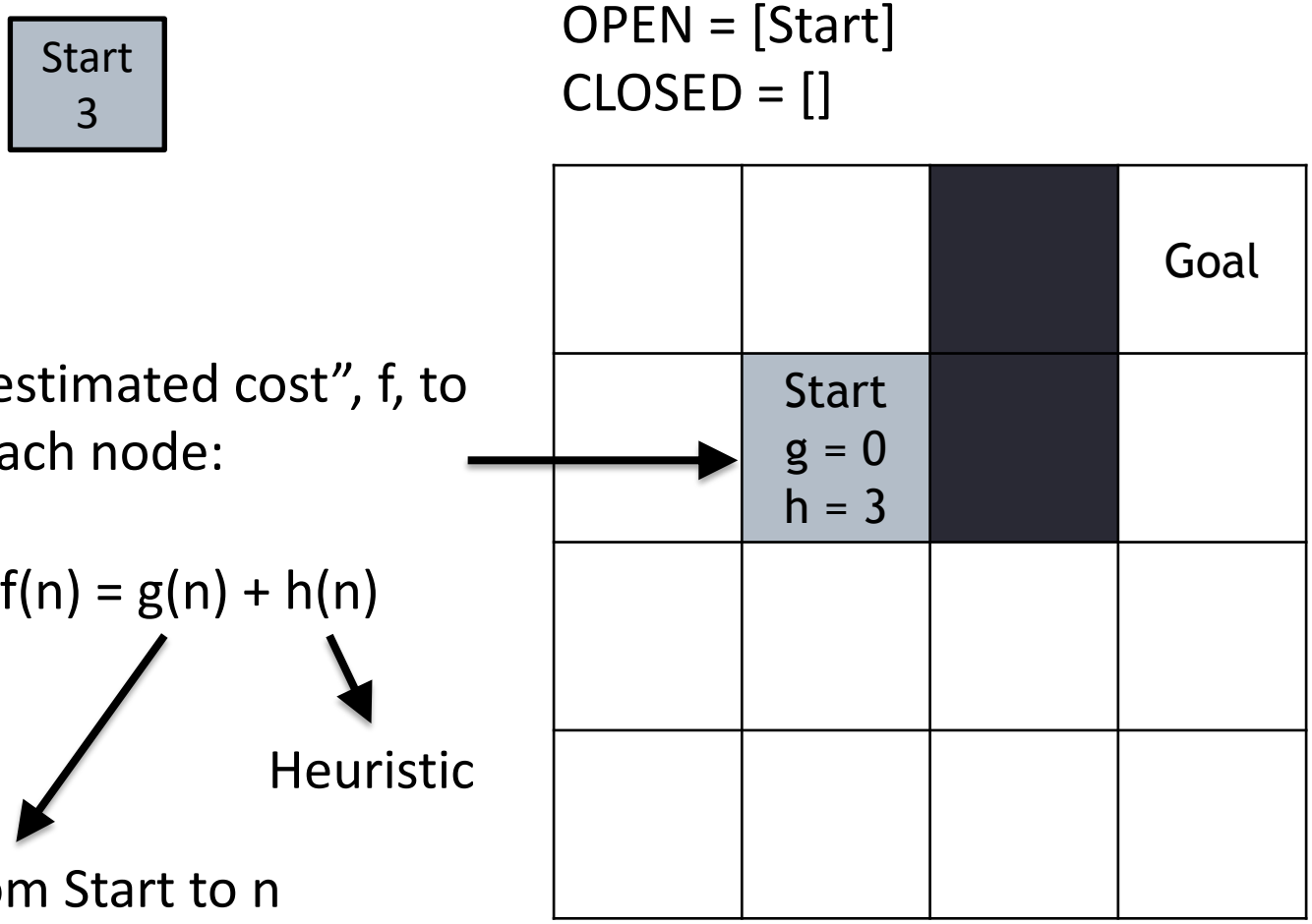
Example: A*

OPEN = [Start]
CLOSED = []



Heuristic used: **Manhattan Distance**

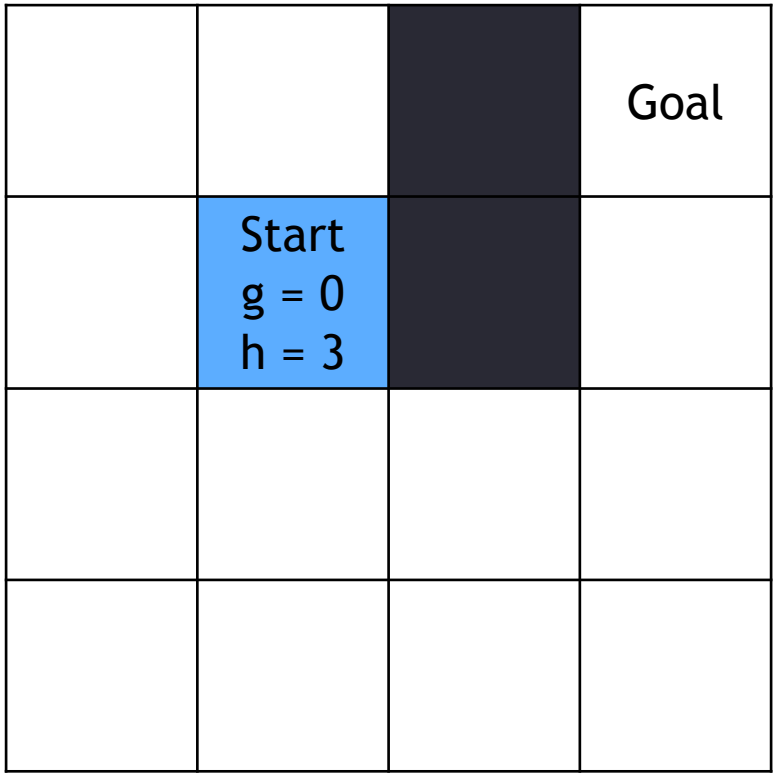
Example: A*



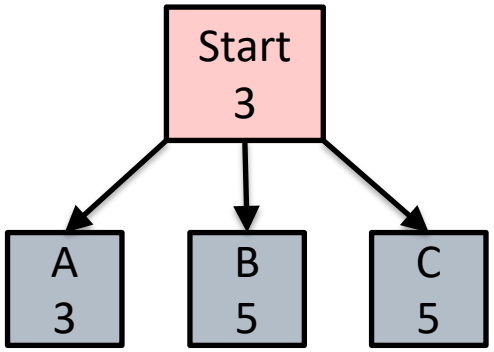
Heuristic used: **Manhattan Distance**

Example: A*

OPEN = [Start]
CLOSED = []



Example: A*

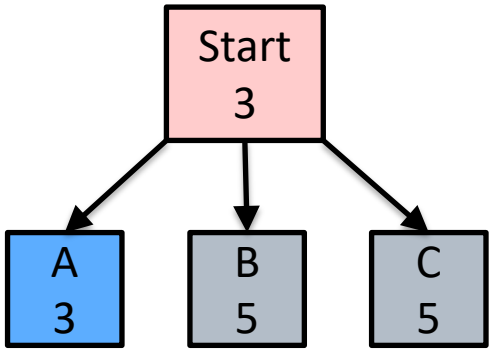


Expands the node with the lowest Estimated cost first

OPEN = [A,B,C]
CLOSED = [Start]

	A g = 1 h = 2		Goal
B g = 1 h = 4	Start g = 0 h = 3		
	C g = 1 h = 4		

Example: A*

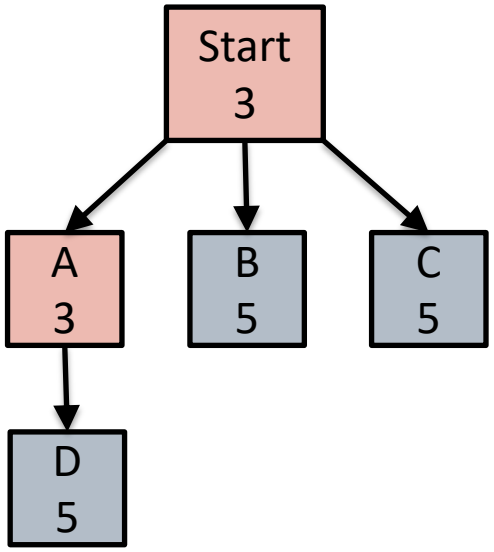


Expands the node with the lowest Estimated cost first

OPEN = [A,B,C]
CLOSED = [Start]

	A g = 1 h = 2		Goal
B g = 1 h = 4	Start g = 0 h = 3		
	C g = 1 h = 4		

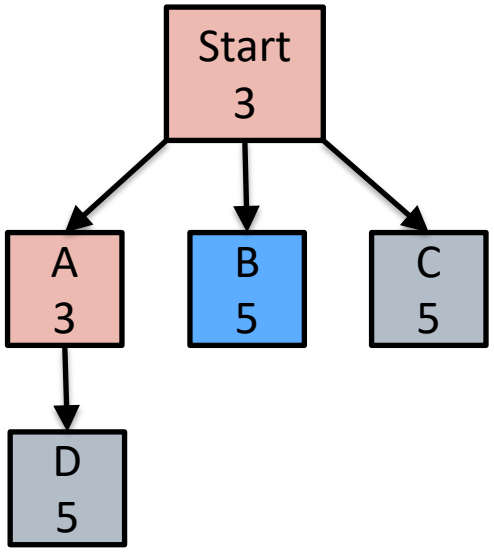
Example: A*



OPEN = [B, C, D]
CLOSED = [Start, A]

D g = 2 h = 3	A g = 1 h = 2		Goal
B g = 1 h = 4	Start g = 0 h = 3		
	C g = 1 h = 4		

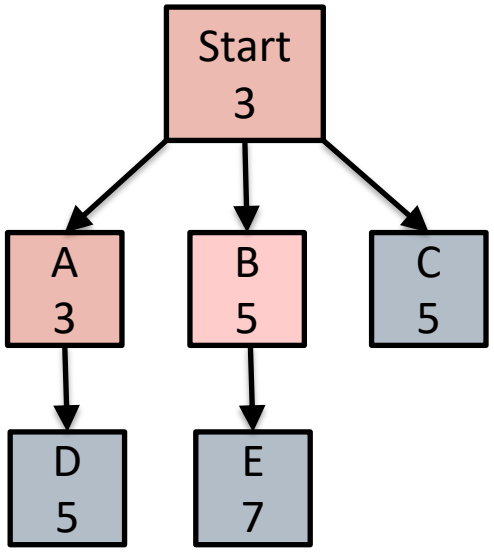
Example: A*



OPEN = [B, C, D]
CLOSED = [Start, A]

D g = 2 h = 3	A g = 1 h = 2		Goal
B g = 1 h = 4	Start g = 0 h = 3		
	C g = 1 h = 4		

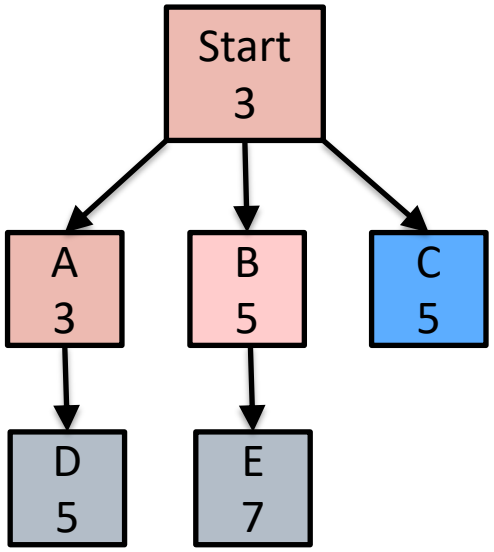
Example: A*



OPEN = [C, D, E]
CLOSED = [Start, A, B]

D g = 2 h = 3	A g = 1 h = 2		Goal
B g = 1 h = 4	Start g = 0 h = 3		
E g = 2 h = 5	C g = 1 h = 4		

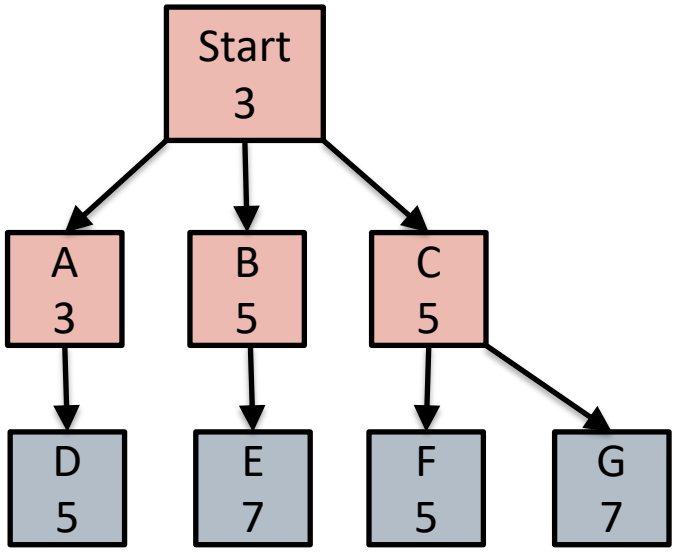
Example: A*



OPEN = [C, D, E]
CLOSED = [Start, A, B]

D g = 2 h = 3	A g = 1 h = 2		Goal
B g = 1 h = 4	Start g = 0 h = 3		
E g = 2 h = 5	C g = 1 h = 4		

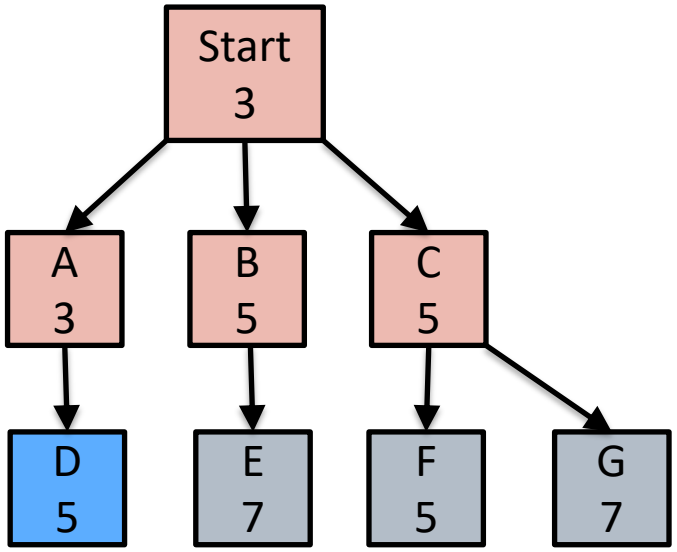
Example: A*



OPEN = [D, E, F, G]
CLOSED = [Start, A, B, C]

D g = 2 h = 3	A g = 1 h = 2		Goal
B g = 1 h = 4	Start g = 0 h = 3		
E g = 2 h = 5	C g = 1 h = 4	F g = 2 h = 3	
	G g = 2 h = 5		

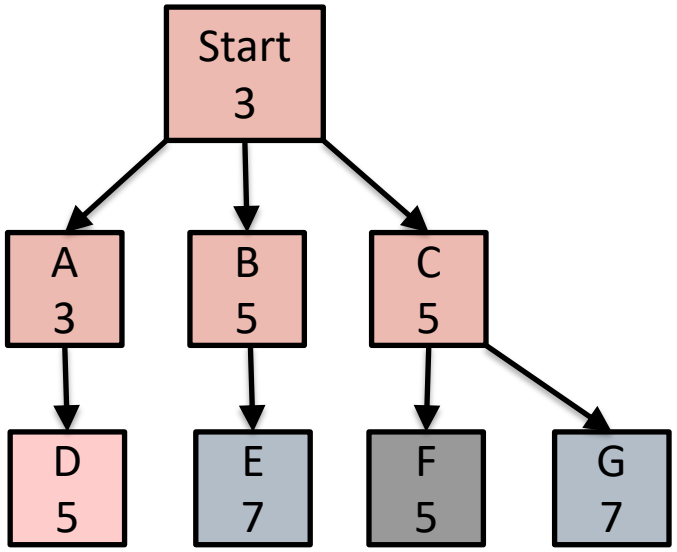
Example: A*



OPEN = [D, E, F, G]
CLOSED = [Start, A, B, C]

D g = 2 h = 3	A g = 1 h = 2		Goal
B g = 1 h = 4	Start g = 0 h = 3		
E g = 2 h = 5	C g = 1 h = 4	F g = 2 h = 3	
	G g = 2 h = 5		

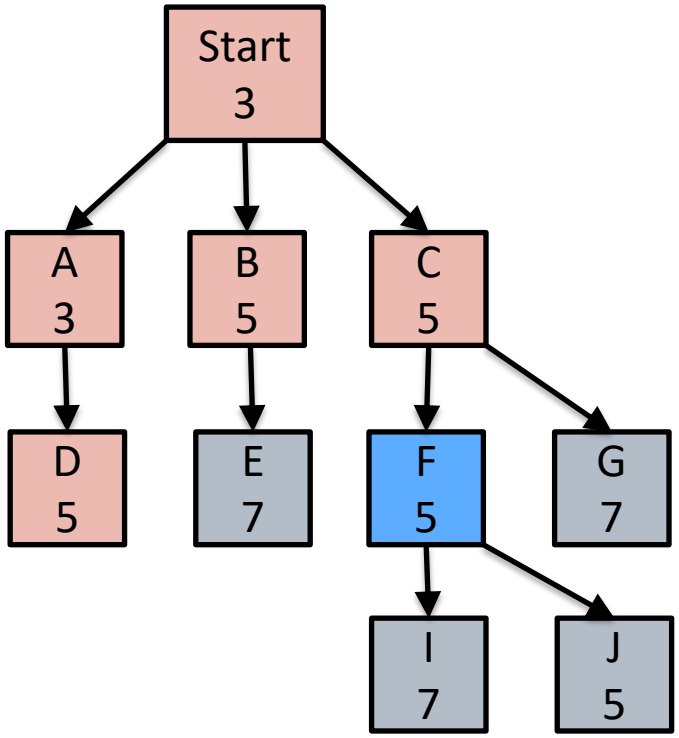
Example: A*



OPEN = [E, F, G]
CLOSED = [Start, A, B, C, D]

D g = 2 h = 3	A g = 1 h = 2		Goal
B g = 1 h = 4	Start g = 0 h = 3		
E g = 2 h = 5	C g = 1 h = 4	F g = 2 h = 3	
	G g = 2 h = 5		

Example: A*

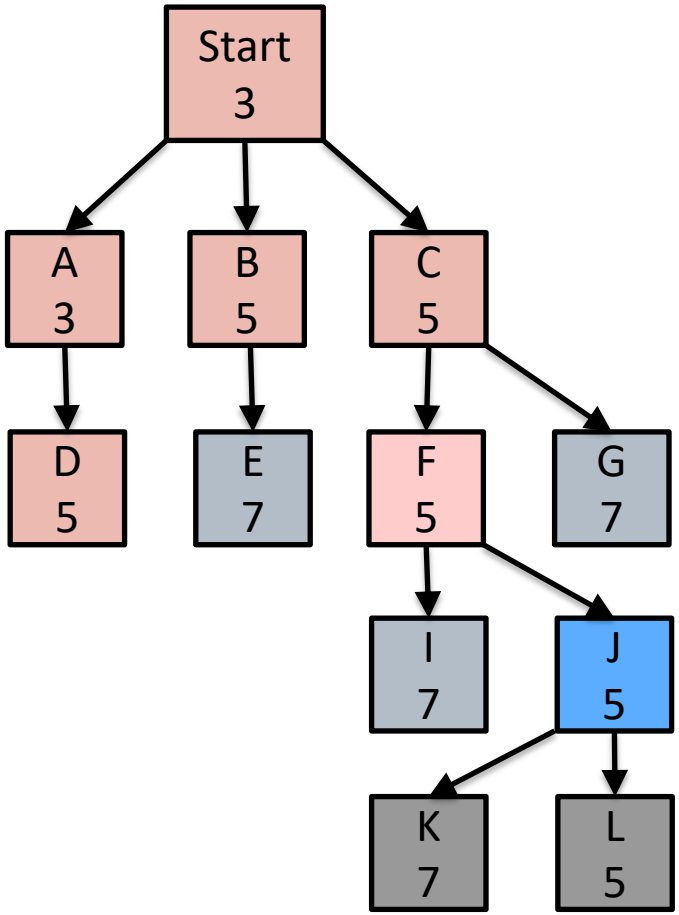


OPEN = [E, G, I, J]

CLOSED = [Start, A, B, C, D, F]

D g = 2 h = 3	A g = 1 h = 2		Goal
B g = 1 h = 4	Start g = 0 h = 3		
E g = 2 h = 5	C g = 1 h = 4	F g = 2 h = 3	J g = 3 h = 2
	G g = 2 h = 5	I g = 3 h = 4	

Example: A*

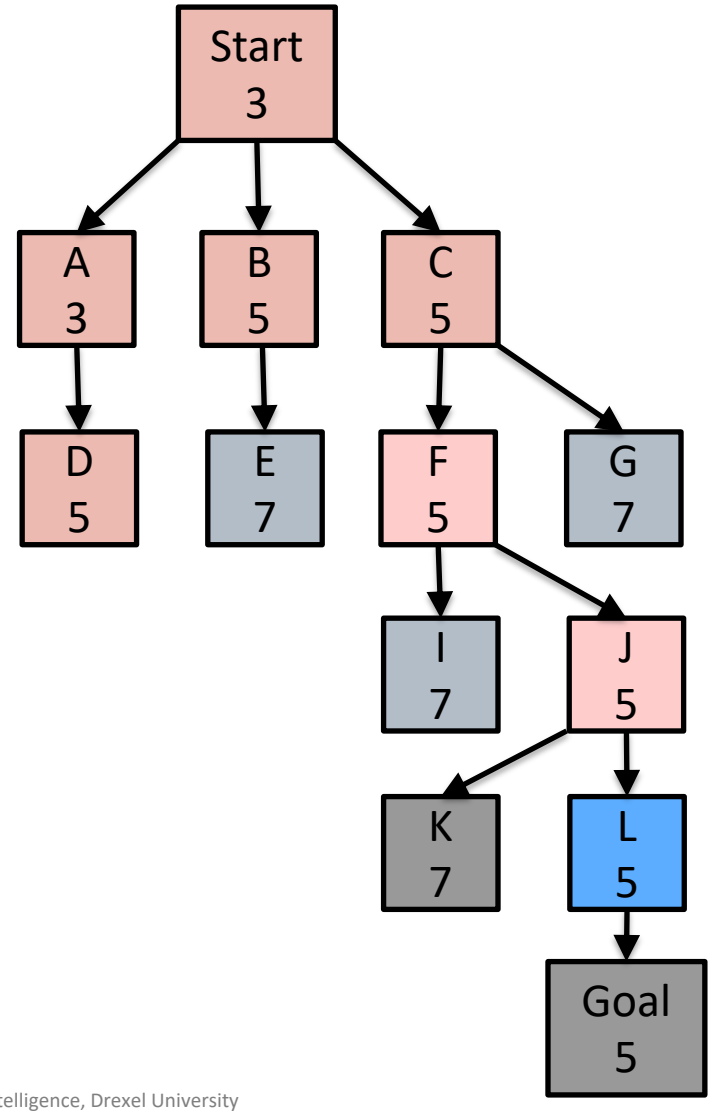


OPEN = [E, G, I, K, L]

CLOSED = [Start, A, B, C, D, F, J]

D g = 2 h = 3	A g = 1 h = 2		Goal
B g = 1 h = 4	Start g = 0 h = 3		L g = 4 h = 1
E g = 2 h = 5	C g = 1 h = 4	F g = 2 h = 3	J g = 3 h = 2
	G g = 2 h = 5	I g = 3 h = 4	K g = 4 h = 3

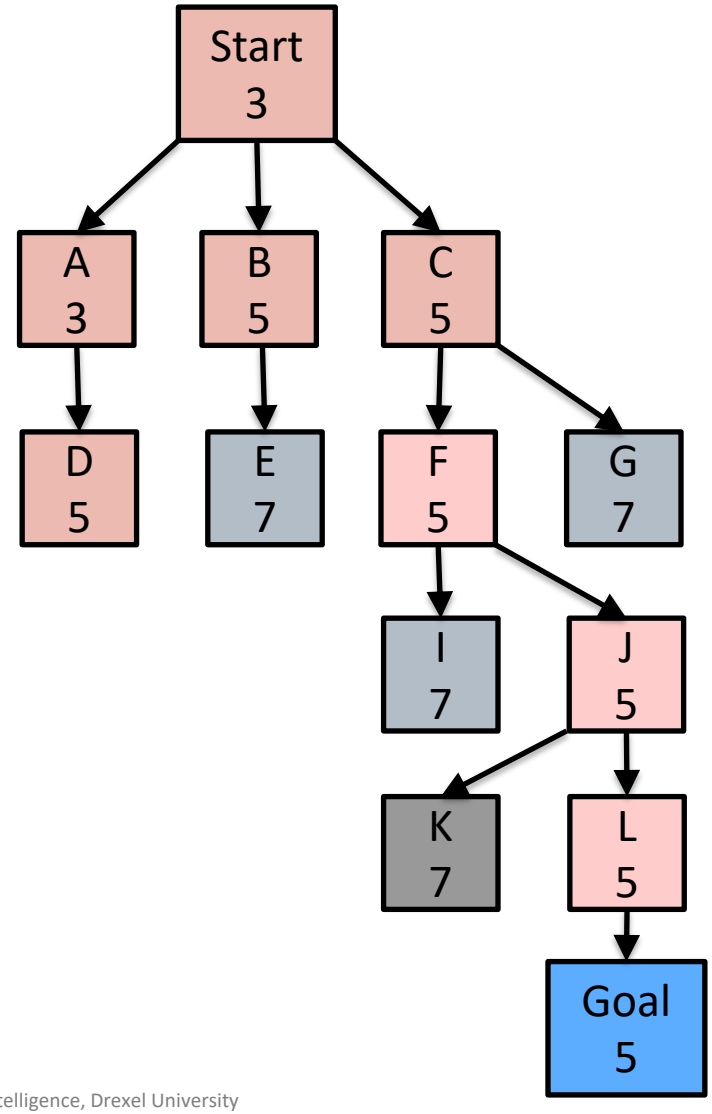
Example: A*



OPEN = [E, G, I, K, Goal]
CLOSED = [Start, A, B, C, D, F, J, L]

D g = 2 h = 3	A g = 1 h = 2		Goal g = 5 h = 0
B g = 1 h = 4	Start g = 0 h = 3		L g = 4 h = 1
E g = 2 h = 5	C g = 1 h = 4	F g = 2 h = 3	J g = 3 h = 2
	G g = 2 h = 5	I g = 3 h = 4	K g = 4 h = 3

Example: A*

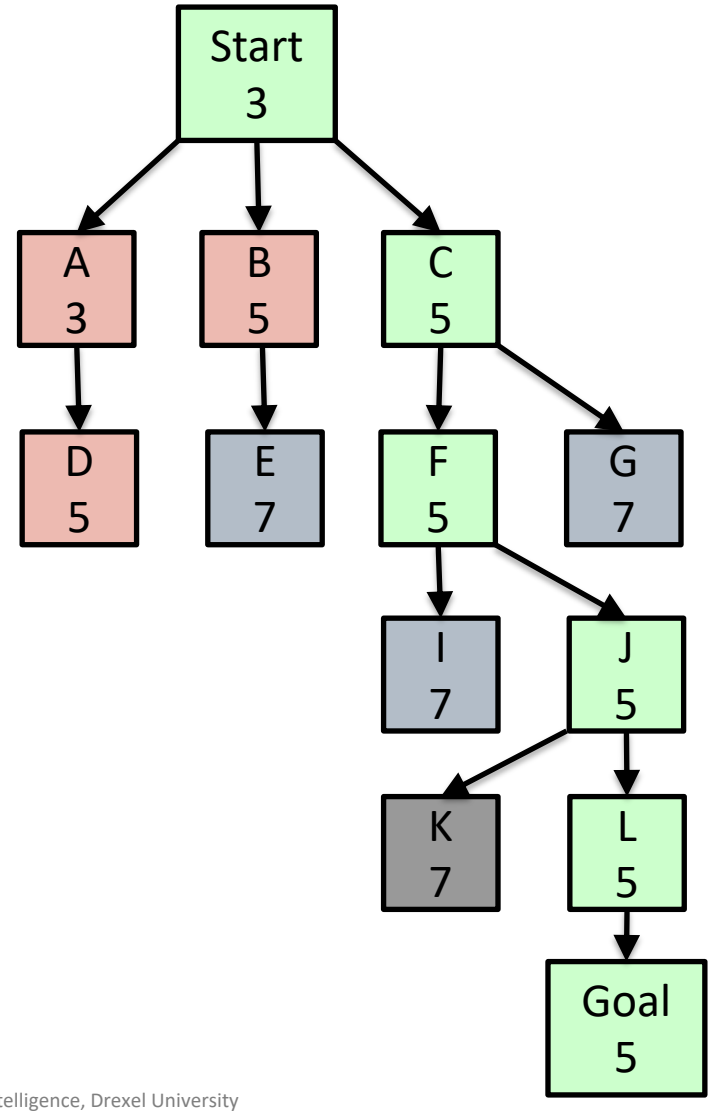


OPEN = [E, G, I, K]

CLOSED = [Start, A, B, C, D, F, J, L, Goal]

D g = 2 h = 3	A g = 1 h = 2		Goal g = 5 h = 0
B g = 1 h = 4	Start g = 0 h = 3		L g = 4 h = 1
E g = 2 h = 5	C g = 1 h = 4	F g = 2 h = 3	J g = 3 h = 2
	G g = 2 h = 5	I g = 3 h = 4	K g = 4 h = 3

Example: A*



OPEN = [E, G, I, K]

CLOSED = [Start, A, B, C, D, F, J, L, Goal]

D g = 2 h = 3	A g = 1 h = 2		Goal g = 5 h = 0
B g = 1 h = 4	Start g = 0 h = 3		L g = 4 h = 1
E g = 2 h = 5	C g = 1 h = 4	F g = 2 h = 3	J g = 3 h = 2
	G g = 2 h = 5	I g = 3 h = 4	K g = 4 h = 3

A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```

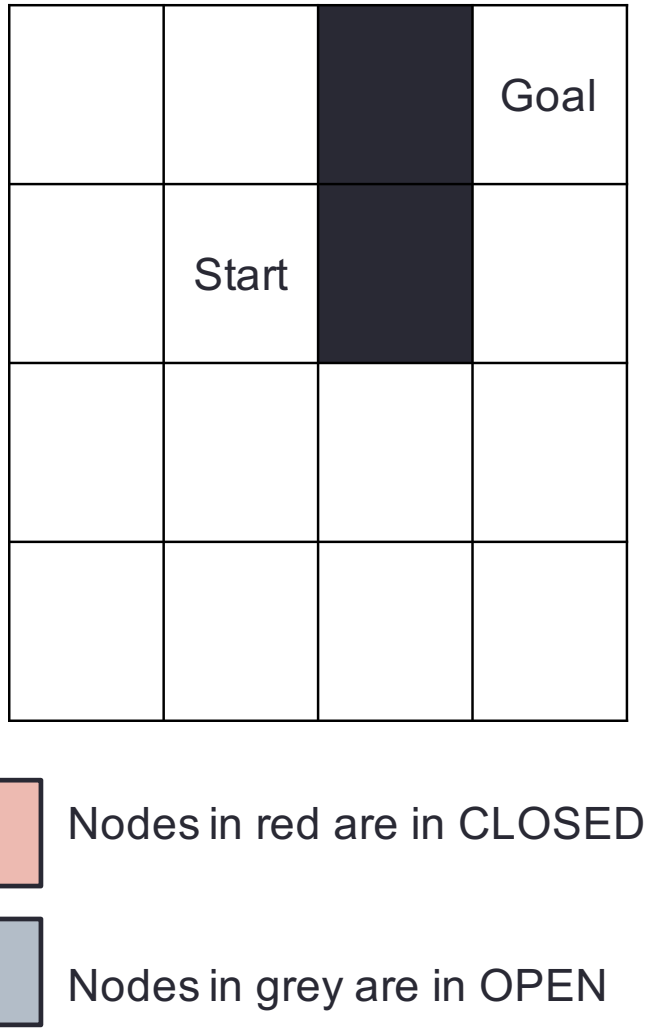
A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```

Differences wrt
Breadth First Search

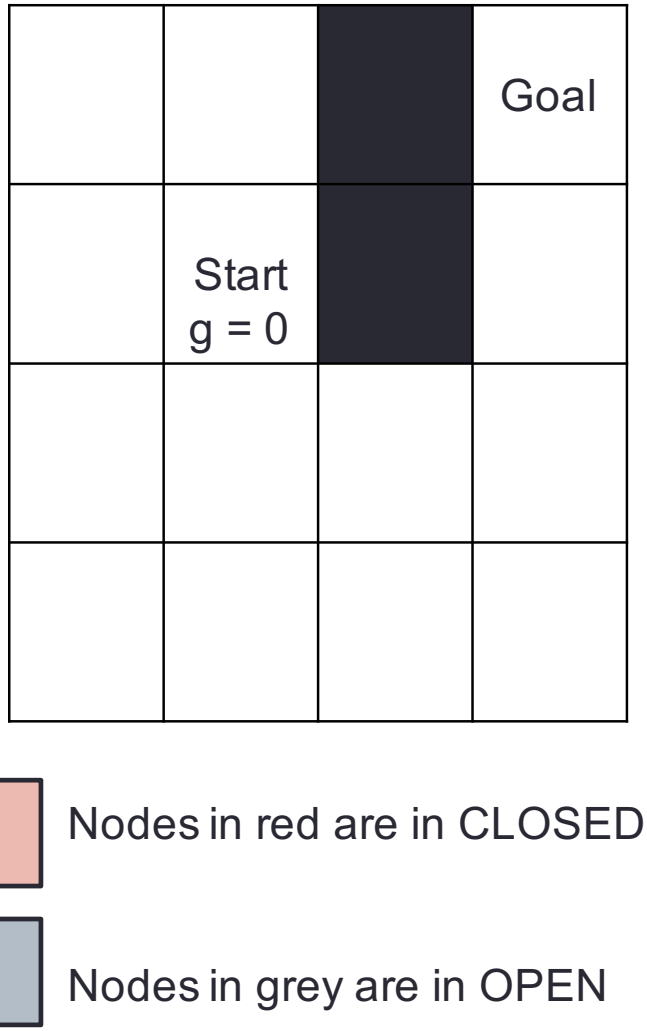
A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```



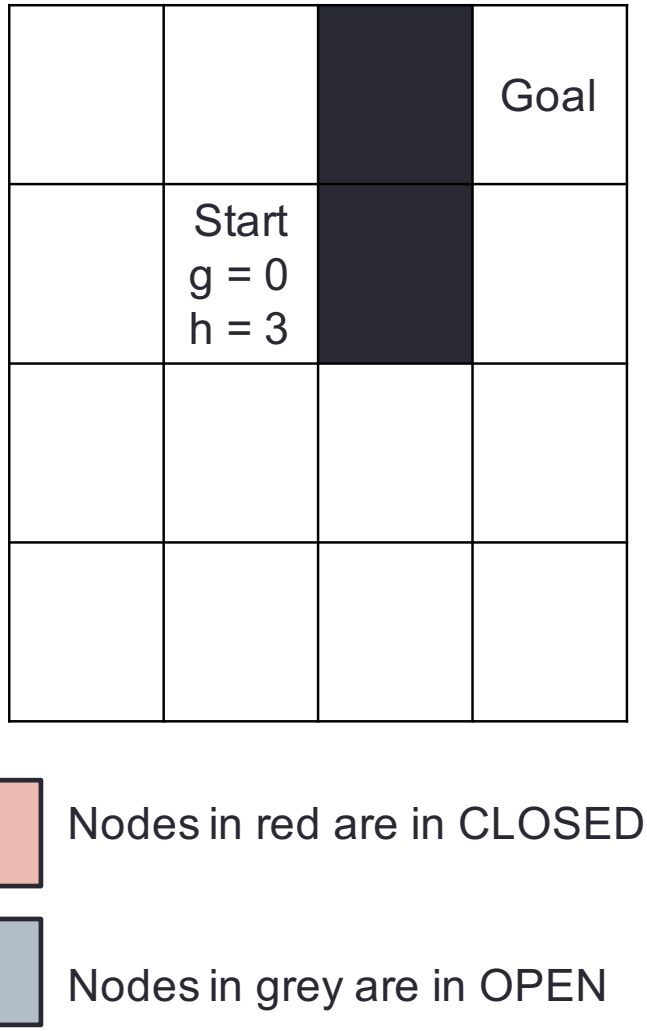
A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```



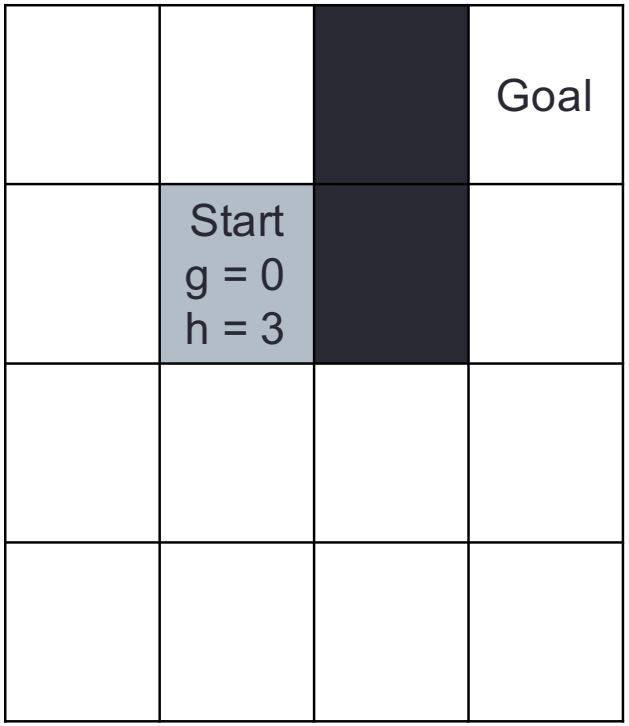
A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```



A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```

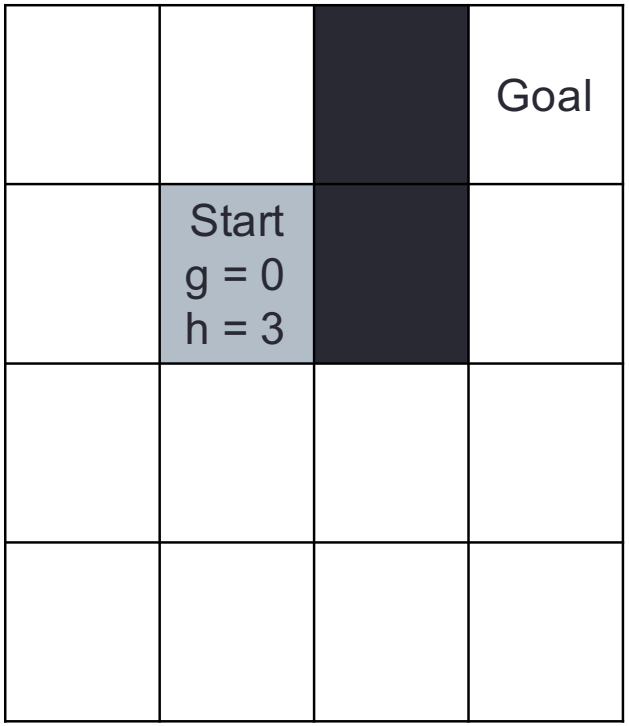


Nodes in red are in CLOSED

Nodes in grey are in OPEN

A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```

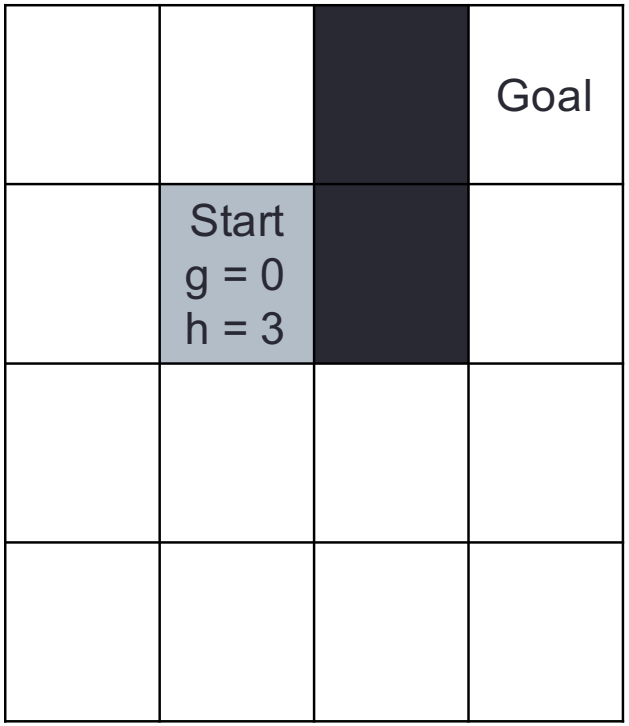


Nodes in red are in CLOSED

Nodes in grey are in OPEN

A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```

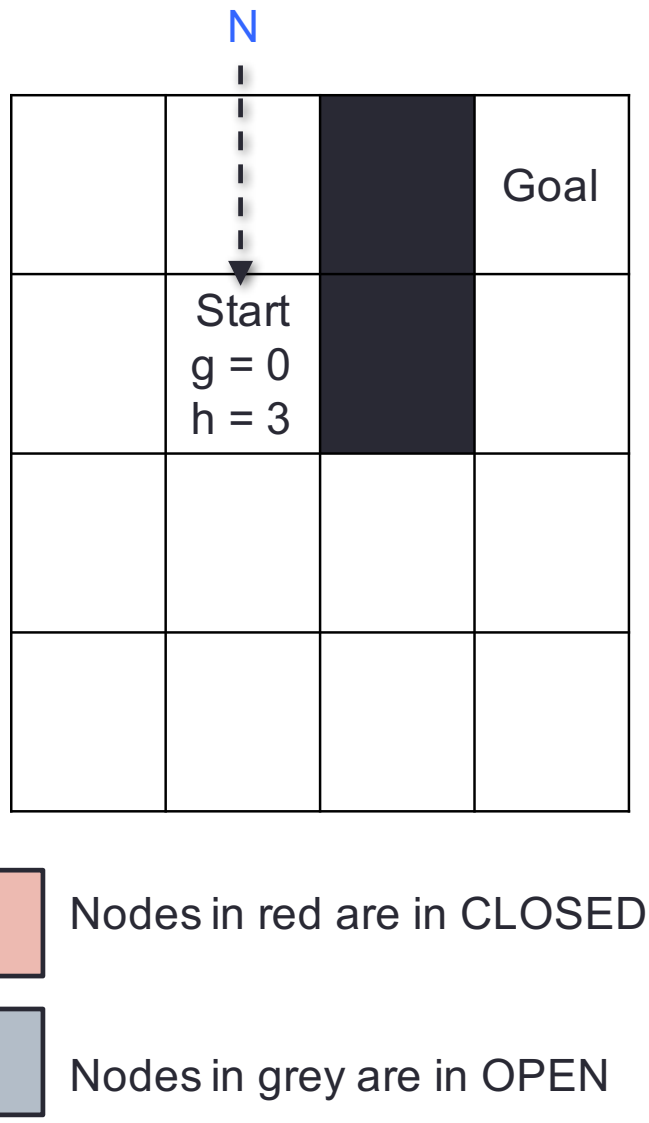


Nodes in red are in CLOSED

Nodes in grey are in OPEN

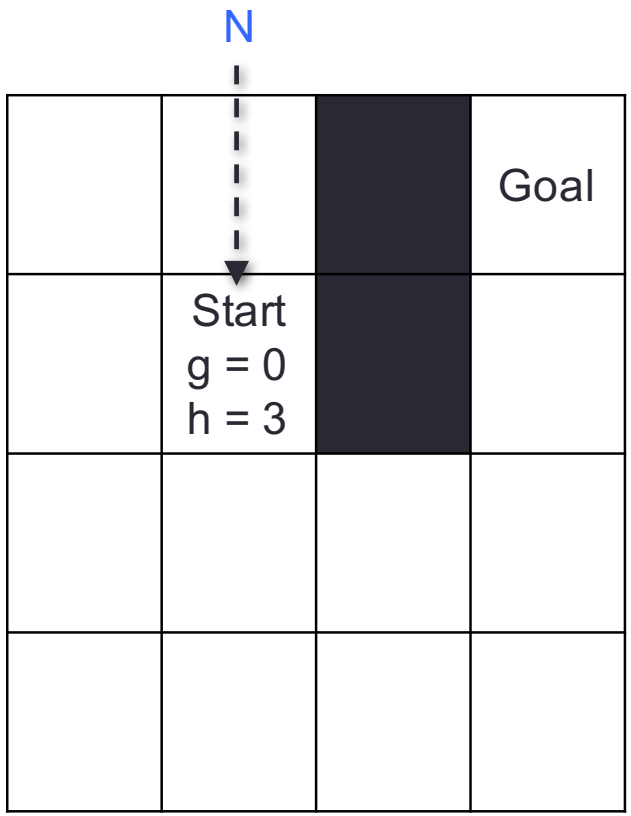
A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```



A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```

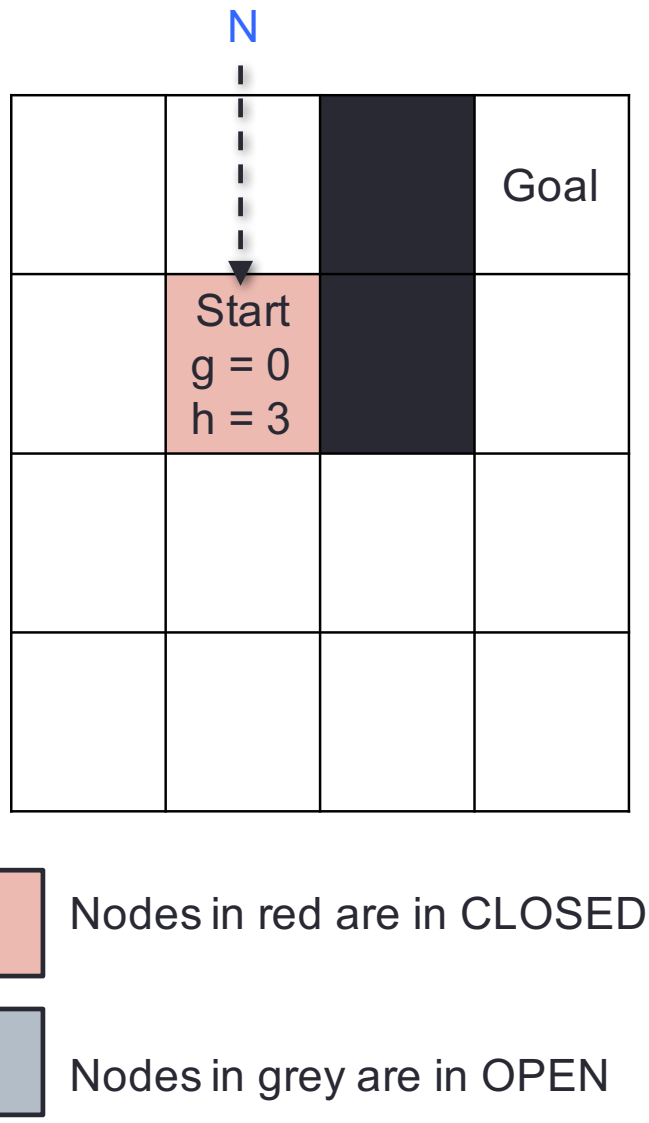


Nodes in red are in CLOSED

Nodes in grey are in OPEN

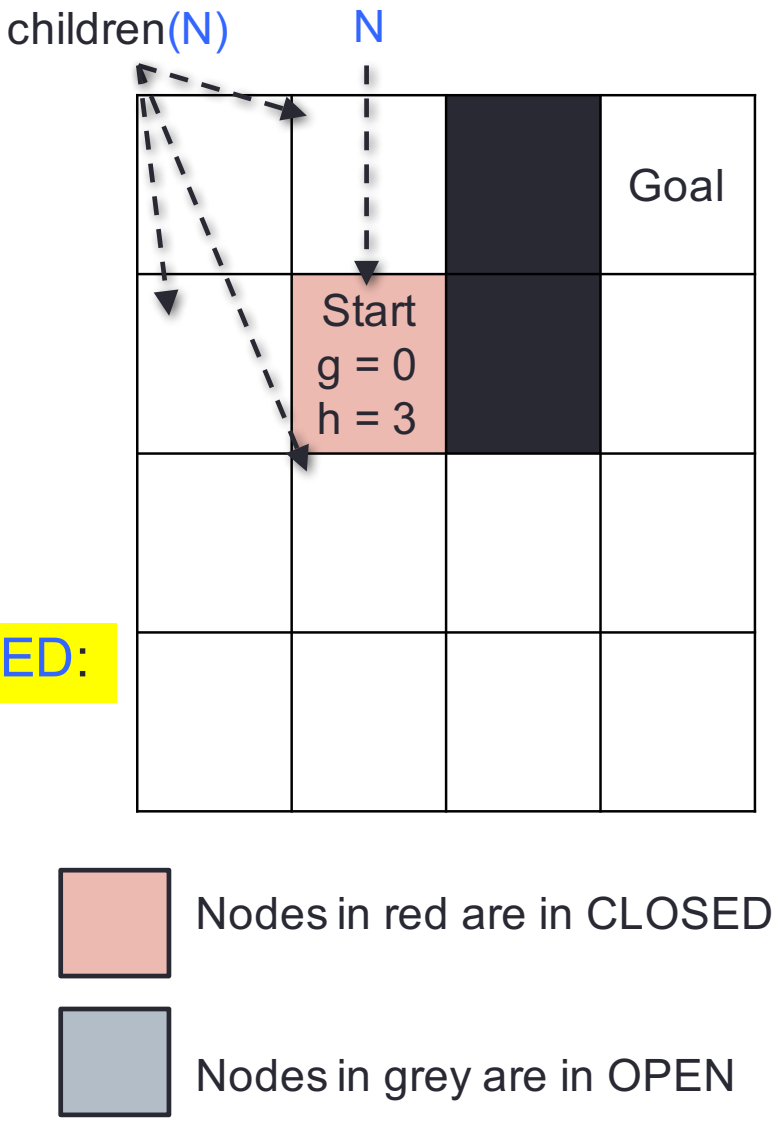
A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```



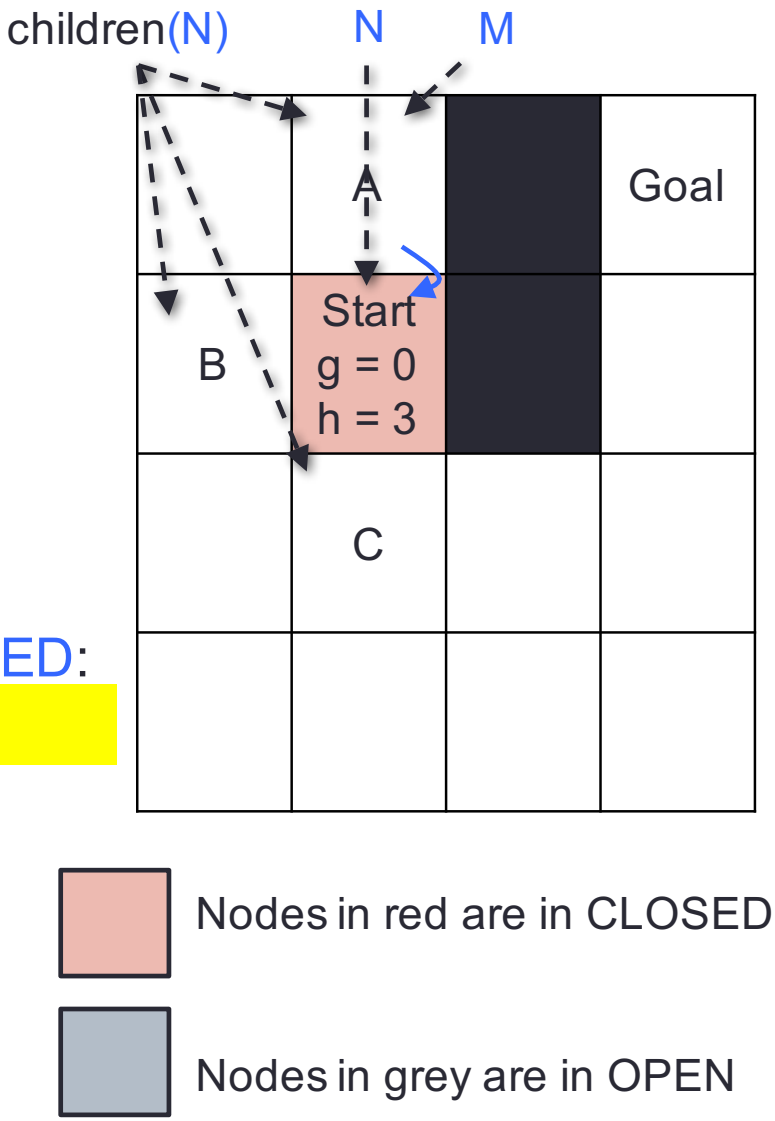
A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```



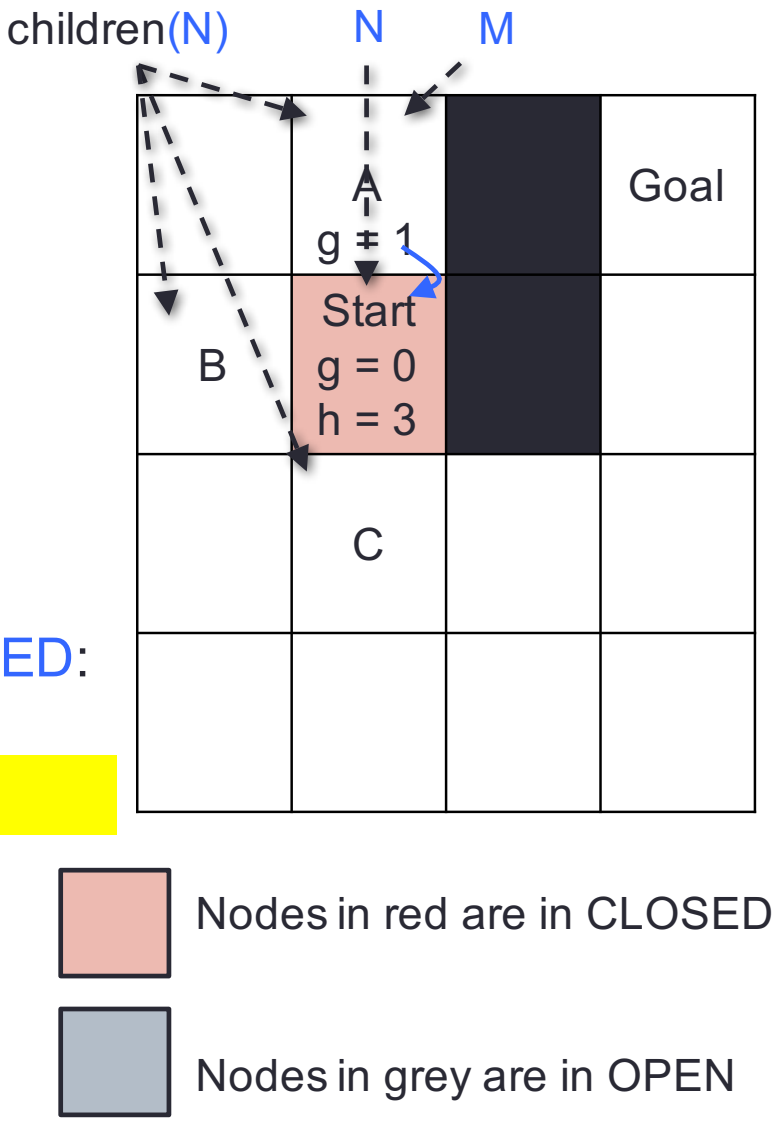
A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```



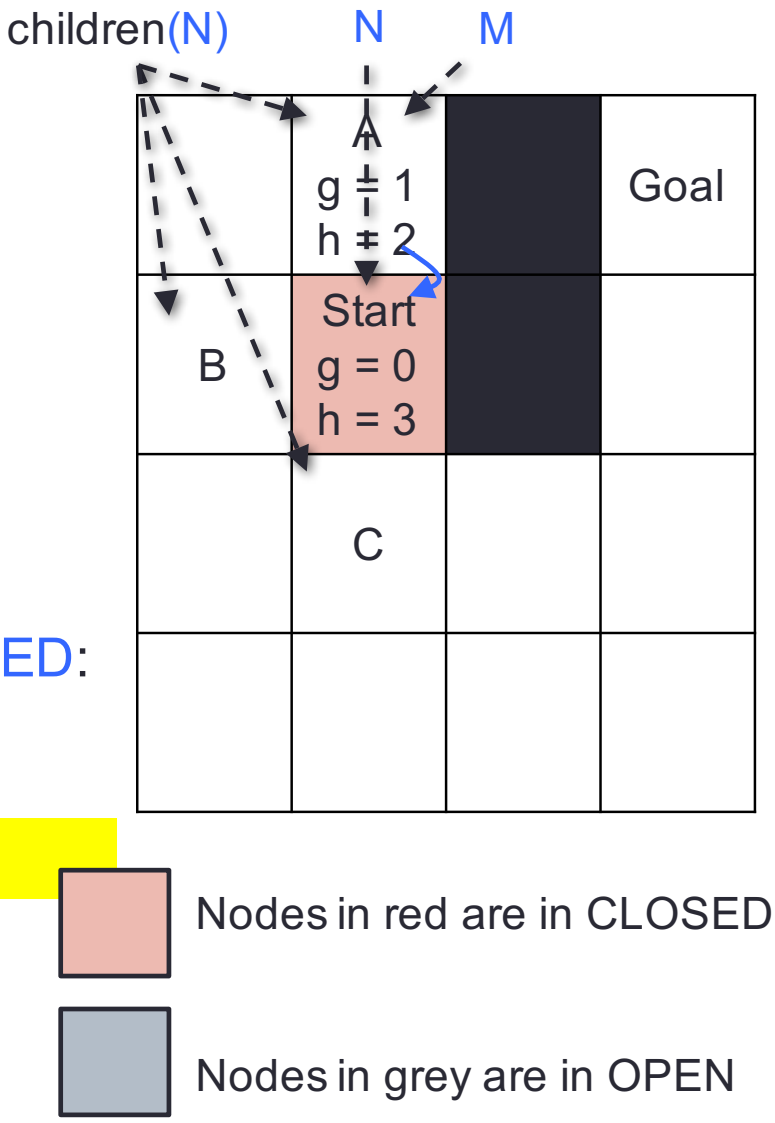
A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```



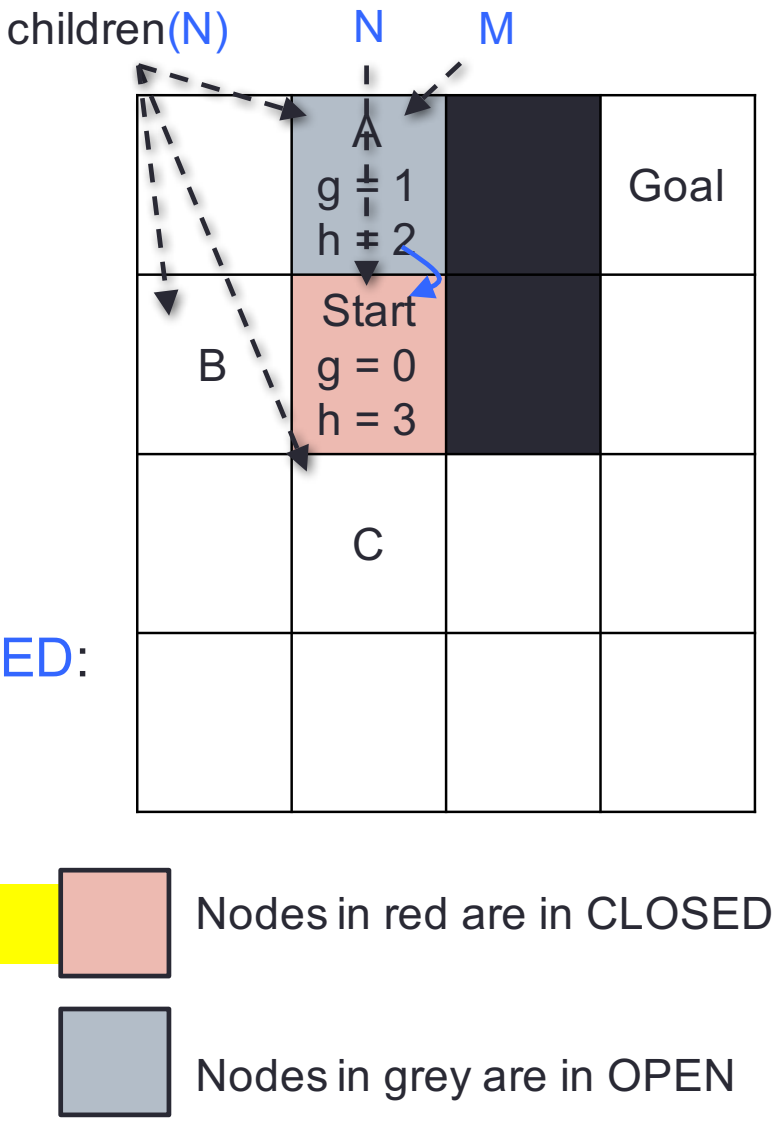
A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```



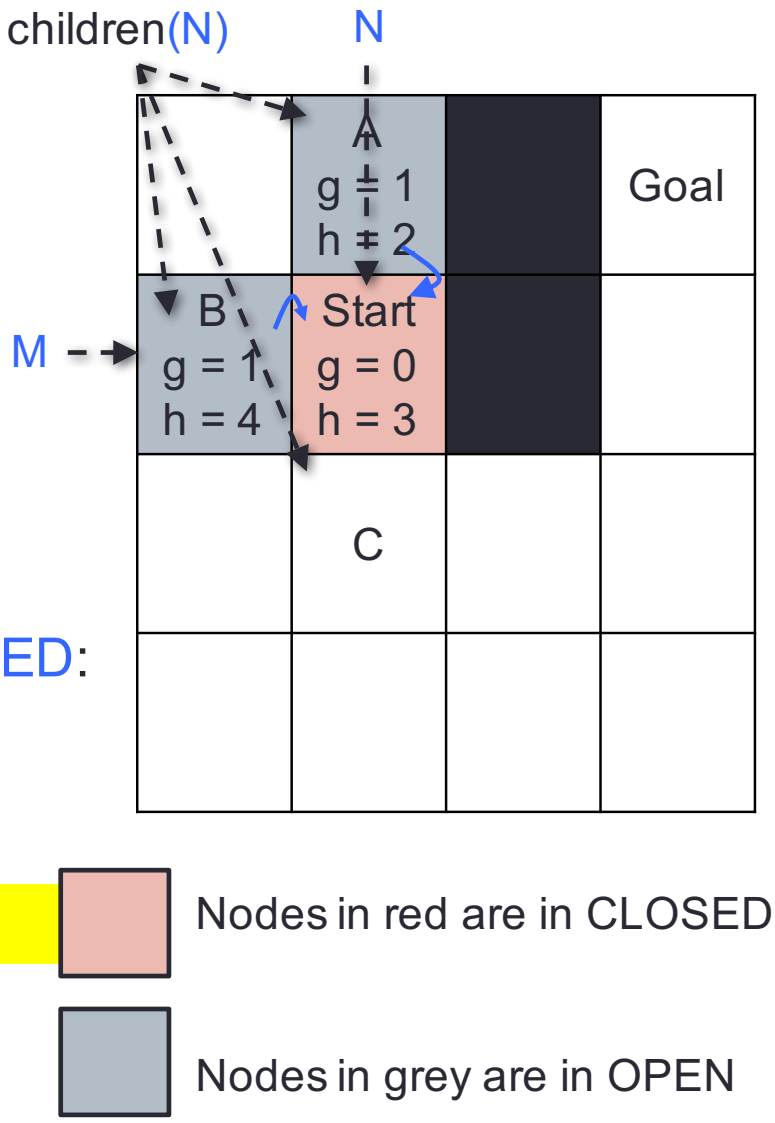
A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```



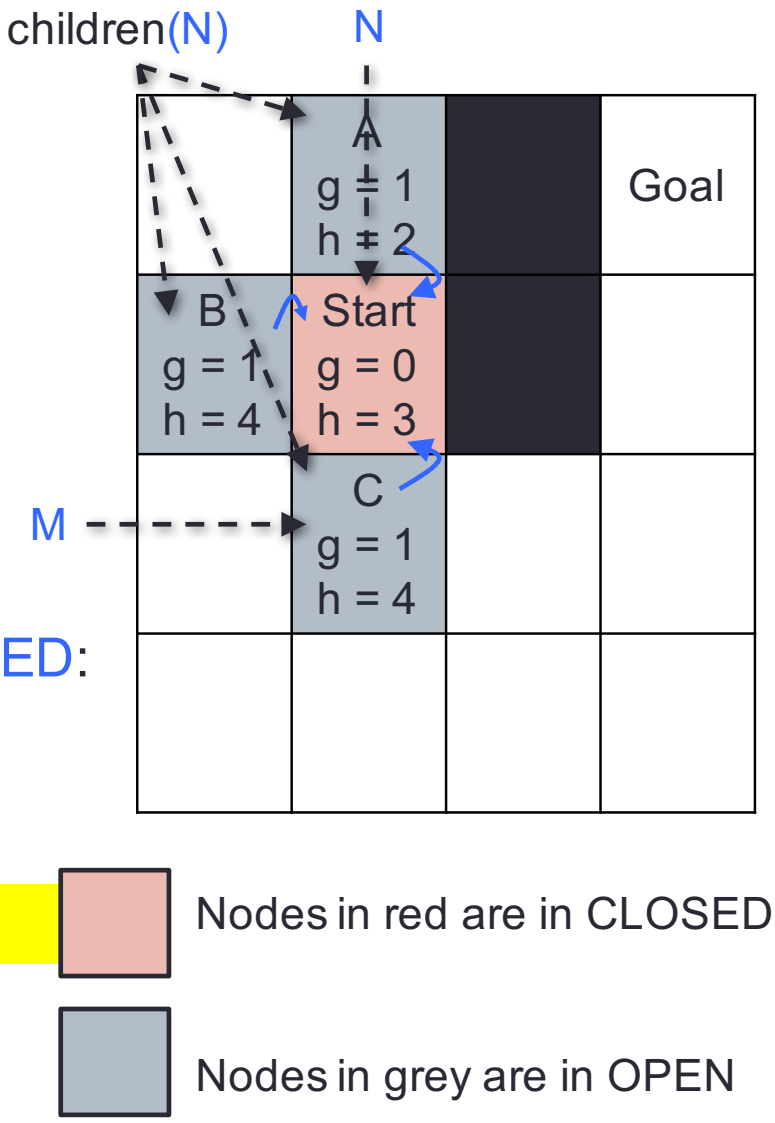
A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```



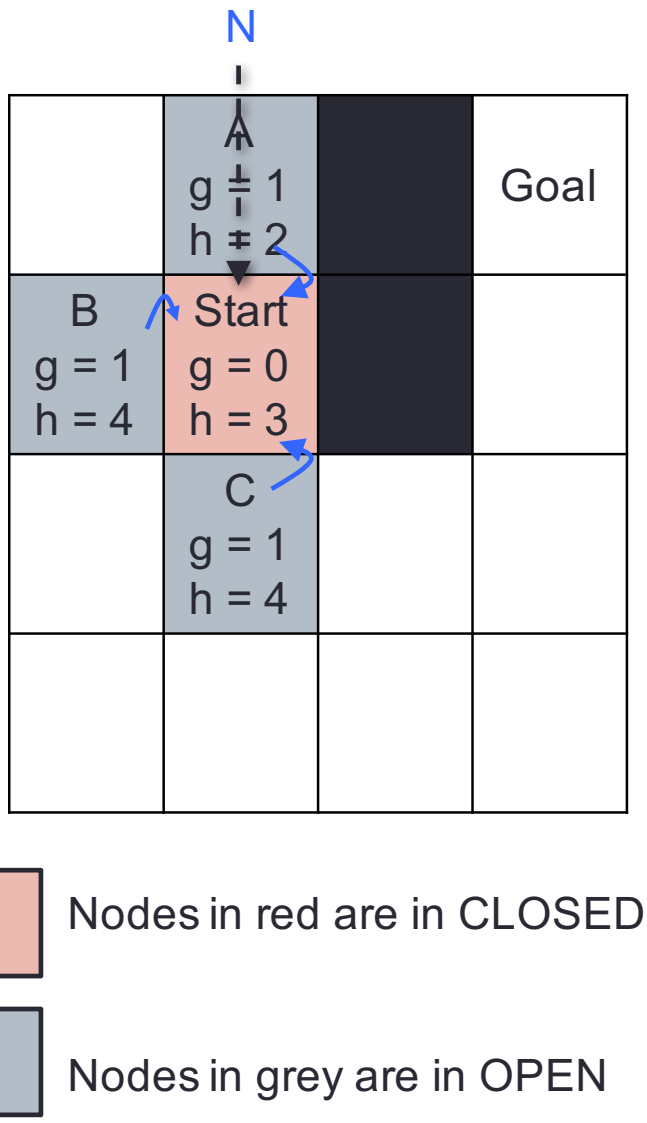
A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```



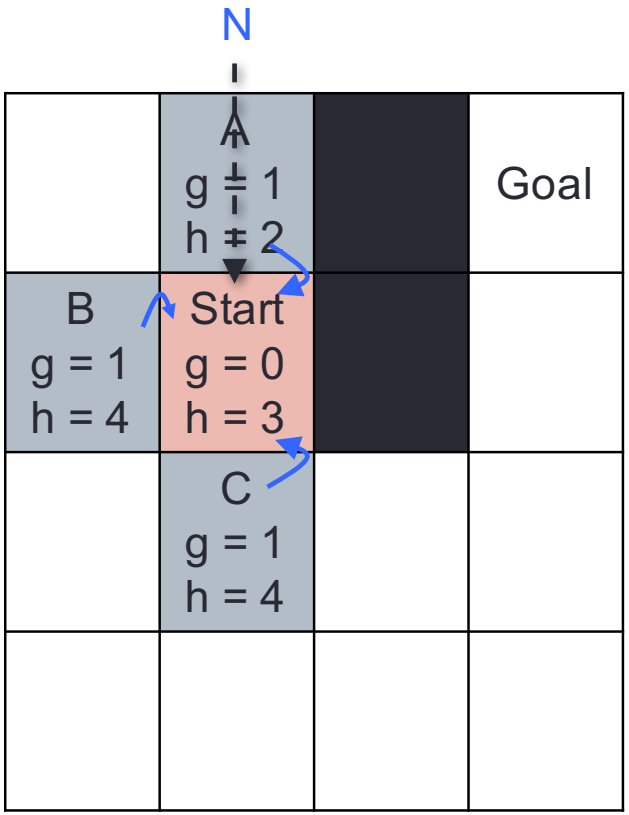
A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```



A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```

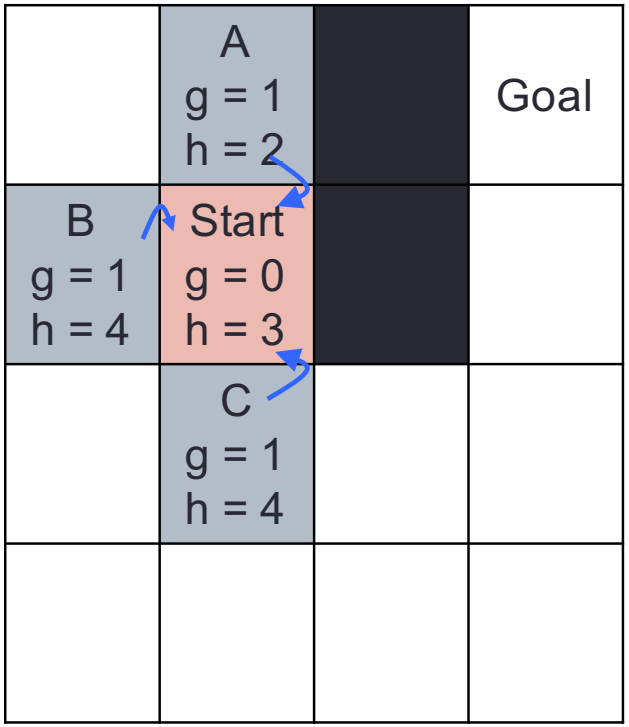


Nodes in red are in CLOSED

Nodes in grey are in OPEN

A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```

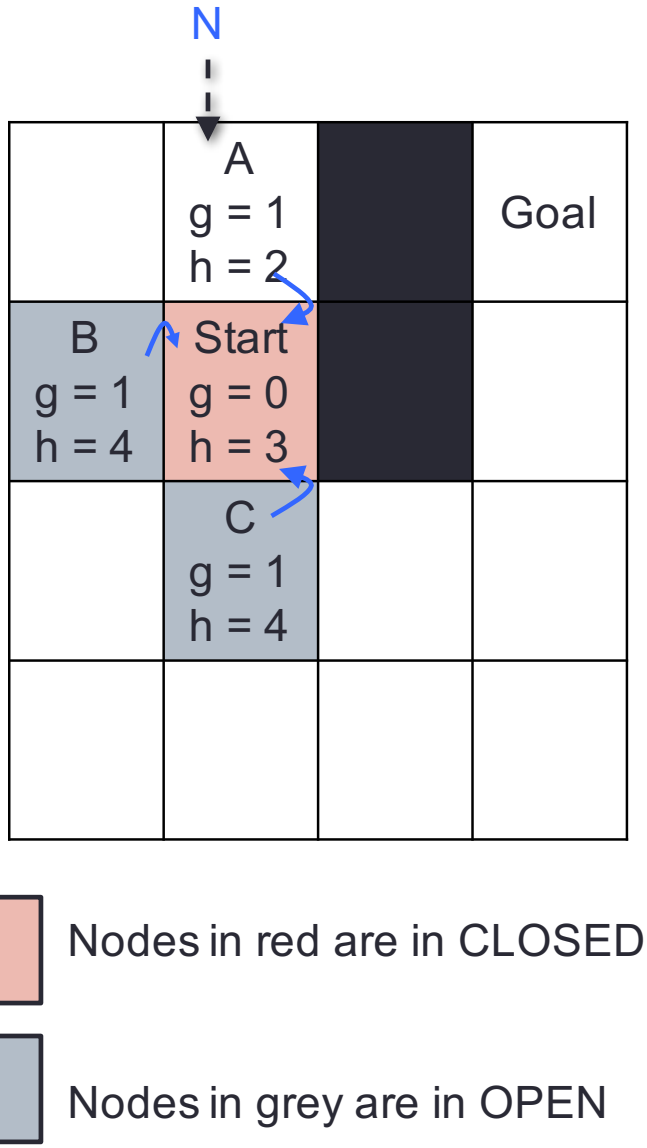


Nodes in red are in CLOSED

Nodes in grey are in OPEN

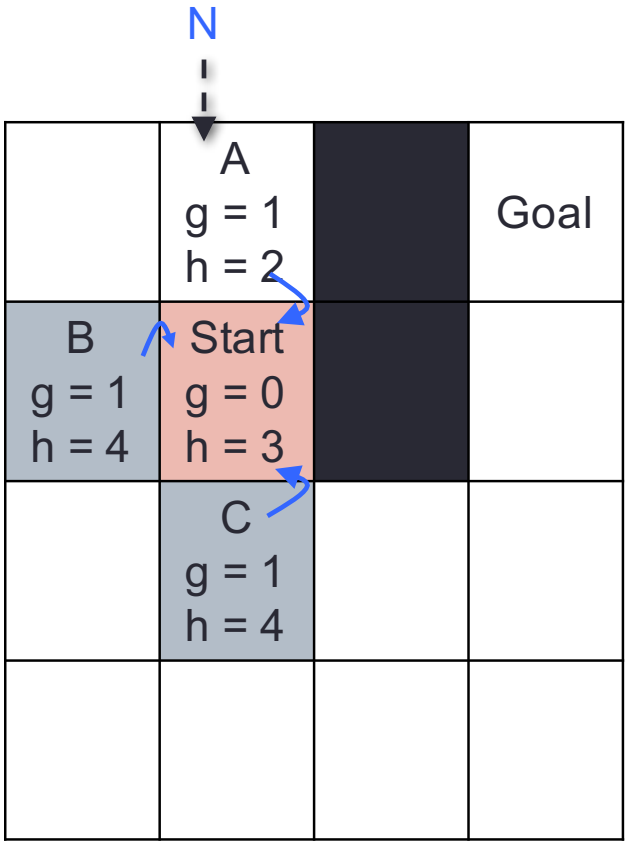
A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```



A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```

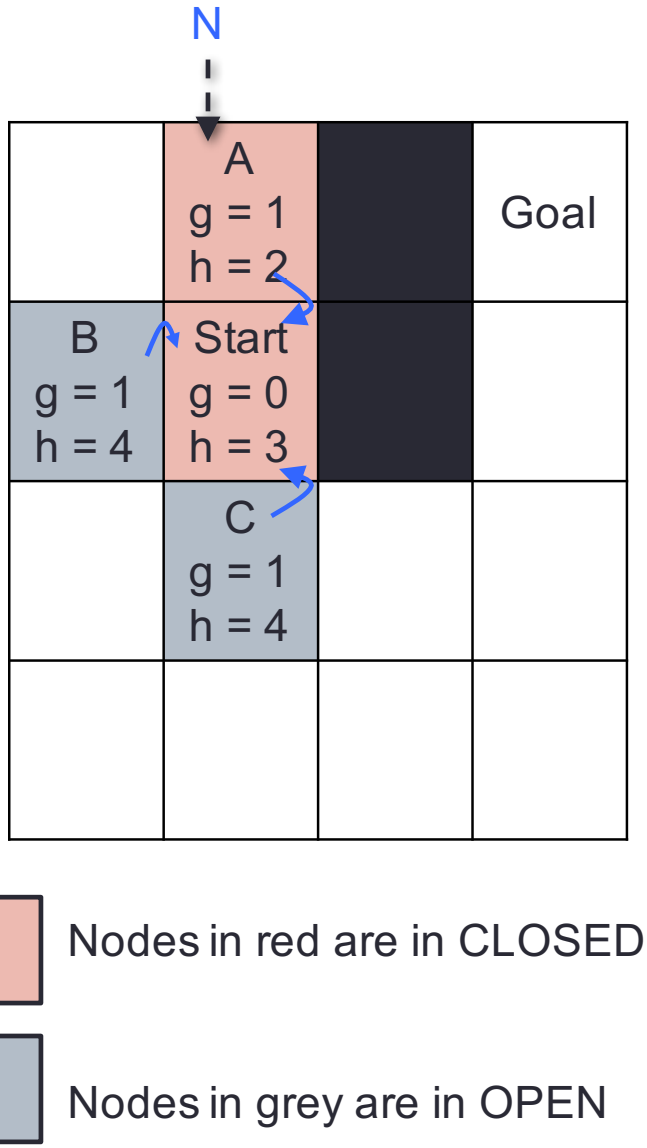


Nodes in red are in CLOSED

Nodes in grey are in OPEN

A*

```
Start.g = 0;
Start.h = heuristic(Start)
OPEN = [Start];
CLOSED = []
WHILE OPEN is not empty
    N = OPEN.removeLowestF()
    IF goal(N) RETURN path to N
    CLOSED.add(N)
    FOR all children M of N not in CLOSED:
        M.parent = N
        M.g = N.g + 1;
        M.h = heuristic(M)
        OPEN.add(M)
    ENDFOR
ENDWHILE
```



Properties of A*

Complete?? Yes, unless there are infinitely many nodes with $f \leq f(G)$

Time?? Exponential in [relative error in h $\times$ length of soln.]

Space?? Keeps all nodes in memory

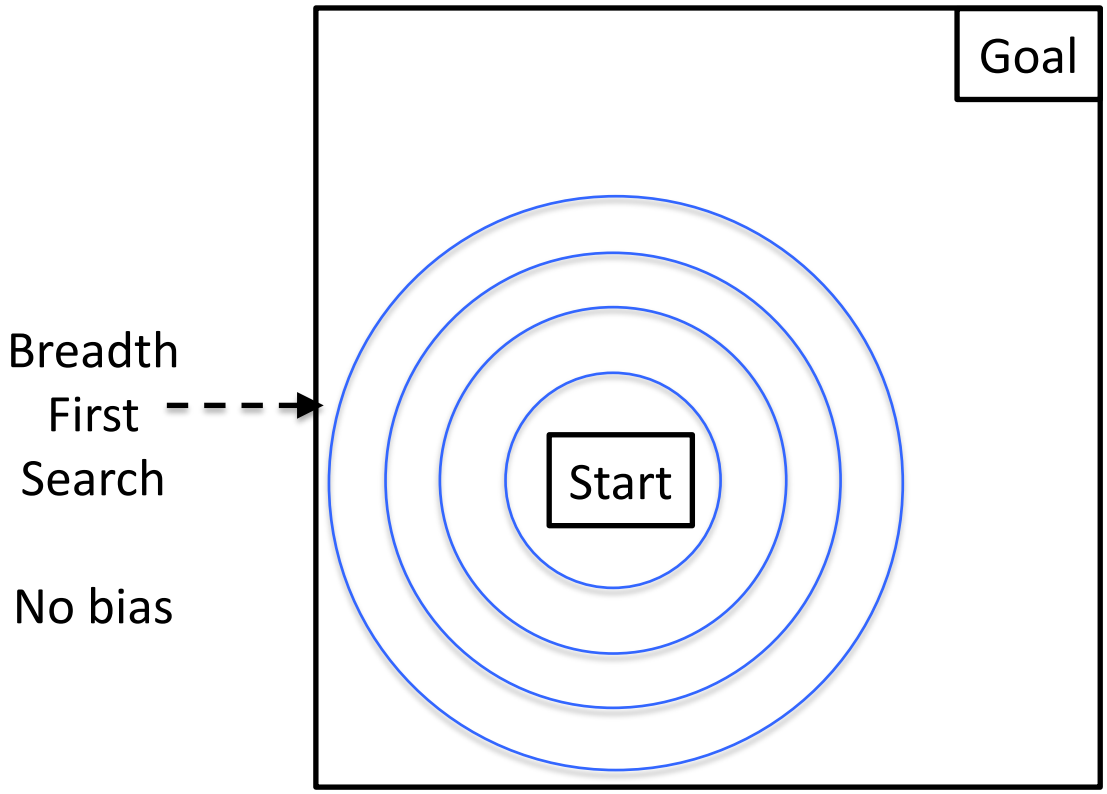
Optimal?? Yes—cannot expand f_{i+1} until f_i is finished

Implementation Notes

- Remember the distinction between **state** and **node**
- State:
  - The configuration of the problem (e.g. coordinates of a robot, positions of the pieces in the 8-puzzle, etc.)
- Node:
  - A state plus: current cost (g), current heuristic (h), parent node, action that got us here form the parent node
  - It is important to remember who was the parent, and which action, so that once the solution is found, we can reconstruct the path

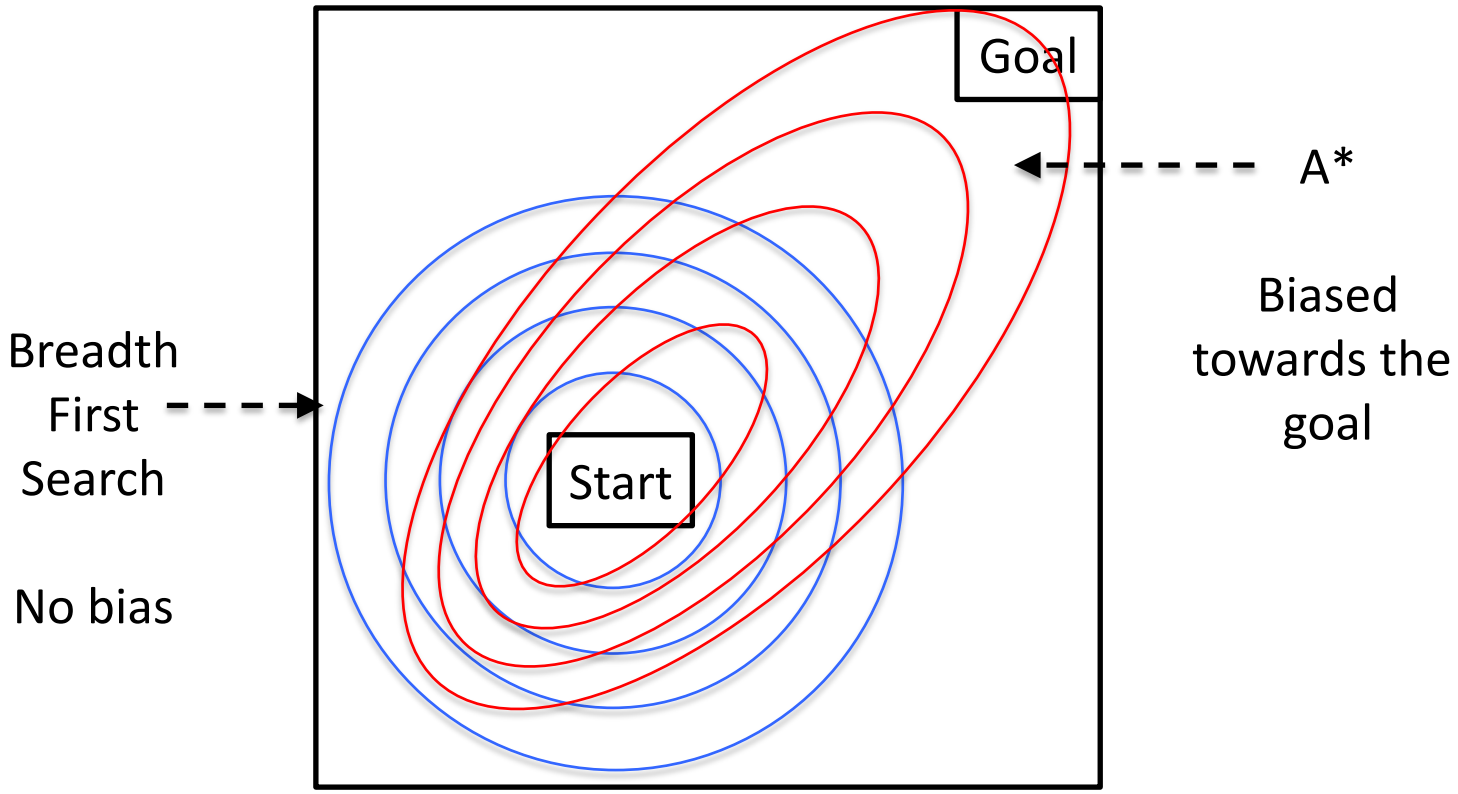
A* Intuition

- The heuristic biases the search of the algorithm towards the goal:



A* Intuition

- The heuristic biases the search of the algorithm towards the goal:



Admissible Heuristics

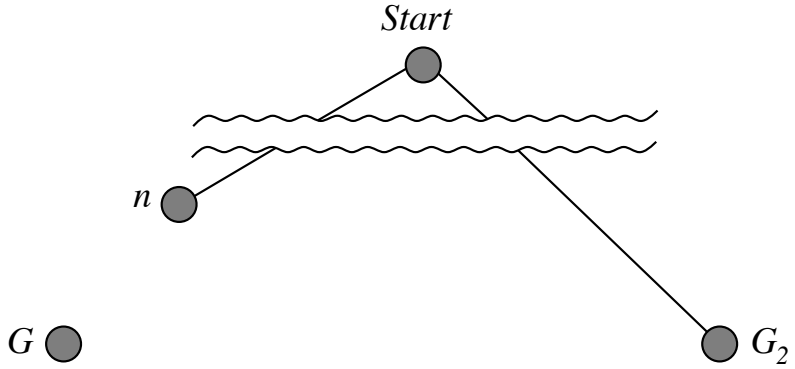
- To ensure optimality, A* requires the heuristic to be **admissible**:

$$h(n) \leq h^*(n)$$ ← Actual cost to the goal

- In other words: the heuristic **must underestimate** the actual remaining cost to the goal.

A* Optimality Proof

Suppose some suboptimal goal G_2 has been generated and is in the queue. Let n be an unexpanded node on a shortest path to an optimal goal G_1.



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

Since $f(G_2) > f(n)$, A* will never select G_2 for expansion

Heuristic Dominance

If $h_2(n) \geq h_1(n)$ for all n (both admissible)
then h_2 dominates h_1 and is better for search

Typical search costs:

$d = 14$ IDS = 3,473,941 nodes
A*$(h_1) = 539$ nodes
A*$(h_2) = 113$ nodes
$d = 24$ IDS $\approx$ 54,000,000,000 nodes
A*$(h_1) = 39{,}135$ nodes
A*$(h_2) = 1{,}641$ nodes

Given any admissible heuristics h_a, h_b,

$$h(n) = \max(h_a(n), h_b(n))$$

is also admissible and dominates h_a, h_b

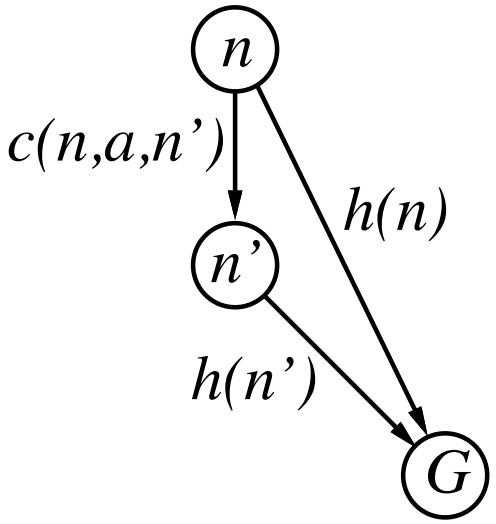
Consistent Heuristics

A heuristic is consistent if

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

If h is consistent, we have

$$\begin{aligned} f(n') &= g(n') + h(n') \\ &= g(n) + c(n, a, n') + h(n') \\ &\geq g(n) + h(n) \\ &= f(n) \end{aligned}$$



I.e., $f(n)$ is nondecreasing along any path.

Consistent heuristics ensure that A* will not try to expand a node more than once (i.e., they make it more efficient). But regardless of whether the heuristic is consistent or not, A* is still guaranteed to find the optimal path if the heuristic is admissible.

Constructing Heuristics by Relaxation

$h_1(n)$ = number of tiles out of place
$h_2(n)$ = Manhattan distance of tiles to proper locations

Admissible heuristics can be derived from the **exact** solution cost of a **relaxed** version of the problem

If the rules of the 8-puzzle are relaxed so that a tile can move **anywhere**, then $h_1(n)$ gives the shortest solution

If the rules are relaxed so that a tile can move to **any adjacent square**, then $h_2(n)$ gives the shortest solution

Key point: the optimal solution cost of a relaxed problem is no greater than the optimal solution cost of the real problem

A* everywhere…

- Even for videogame playing:
  - http://www.youtube.com/watch?v=DlkMs4ZHHr8

Variations of A*

- SMA*: A* with bounded memory usage
- TBA*: A* for real-time domains where we have a bounded time before producing an action
- LRTA*: another real-time version of A* (very simple, and the basis of a whole family of algorithms)
- D*: A* for dynamic domains (problem configuration can change)
- etc.

Coming up…

- We look at some search variations and alternatives, including local search and constraint satisfaction