Heuristic (Informed) Search

(Where we try to choose smartly)

R&N: Chap. 4, Sect. 4.1–3

Recall that the ordering of FRINGE defines the search strategy

Search Algorithm #2

SEARCH#2

1. INSERT(initial-node,FRINGE)
2. Repeat:
   a. If empty(FRINGE) then return failure
   b. N ← REMOVE(FRINGE)
   c. s ← STATE(N)
   d. If GOAL?(s) then return path or goal state
   e. For every state s' in SUCCESSORS(s)
      i. Create a node N' as a successor of N
      ii. INSERT(N',FRINGE)

Best-First Search

- It exploits state description to estimate how "good" each search node is
- An evaluation function f maps each node N of the search tree to a real number $f(N) \geq 0$
  [Traditionally, f(N) is an estimated cost; so, the smaller f(N), the more promising N]
- Best-first search sorts the FRINGE in increasing f
  [Arbitrary order is assumed among nodes with equal f]

Best-First Search

- It exploits state description to estimate how "good" each search node is
- An evaluation function f maps each node N of the search tree to a real number $f(N) \geq 0$
  [Traditionally, f(N
  f(N), the more pro
- Best-first search sorts the FRINGE in increasing f
  [Arbitrary order is assumed among nodes with equal f]

"Best" does not refer to the quality of the generated path
Best-first search does not generate optimal paths in general

How to construct f?

- Typically, f(N) estimates:
  - either the cost of a solution path through N

    Then f(N) = g(N) + h(N), where
    - g(N) is the cost of the path from the initial node to N
    - h(N) is an estimate of the cost of a path from N to a goal node
  - or the cost of a path from N to a goal node

    Then f(N) = h(N) → Greedy best-search

- But there are no limitations on f. Any function of your choice is acceptable.
  But will it help the search algorithm?

How to construct f?

- Typically, f(N) estimates:
  - either the cost of a solution path through N

    Then f(N) = g(N) + h(N), where
    - g(N) is the cost of the path from the initial node to N
    - h(N) is an estimate of the cost of a path from N to a goal node
  - or the cost of a path from N to a goal node

    Then f(N) = h(N)

Heuristic function

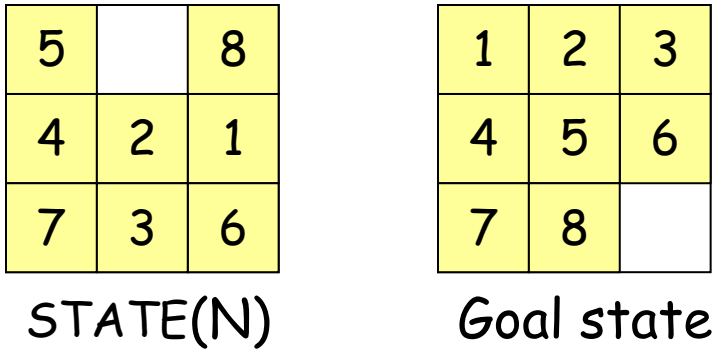
- But there are no limitations on f. Any function of your choice is acceptable.
  But will it help the search algorithm?

Heuristic Function

- The heuristic function h(N) ≥ 0 estimates the cost to go from STATE(N) to a goal state

  Its value is **independent of the current search tree**; it depends only on STATE(N) and the goal test GOAL?

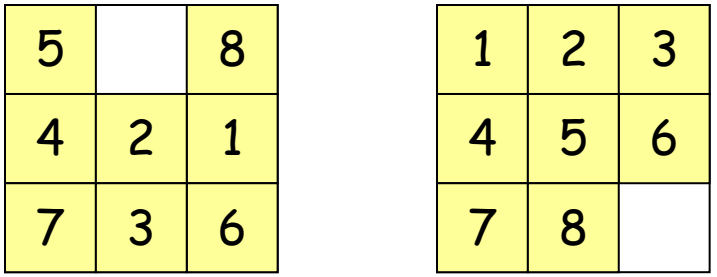
- Example:



$h_1(N)$ = number of misplaced numbered tiles = 6

[Why is it an estimate of the distance to the goal?]

Other Examples



- $h_1(N)$ = number of misplaced numbered tiles = 6
- $h_2(N)$ = sum of the (Manhattan) distance of every numbered tile to its goal position
  = 2 + 3 + 0 + 1 + 3 + 0 + 3 + 1 = 13
- $h_3(N)$ = sum of permutation inversions
  = $n_5 + n_8 + n_4 + n_2 + n_1 + n_7 + n_3 + n_6$
  = 4 + 6 + 3 + 1 + 0 + 2 + 0 + 0
  = 16

8-Puzzle

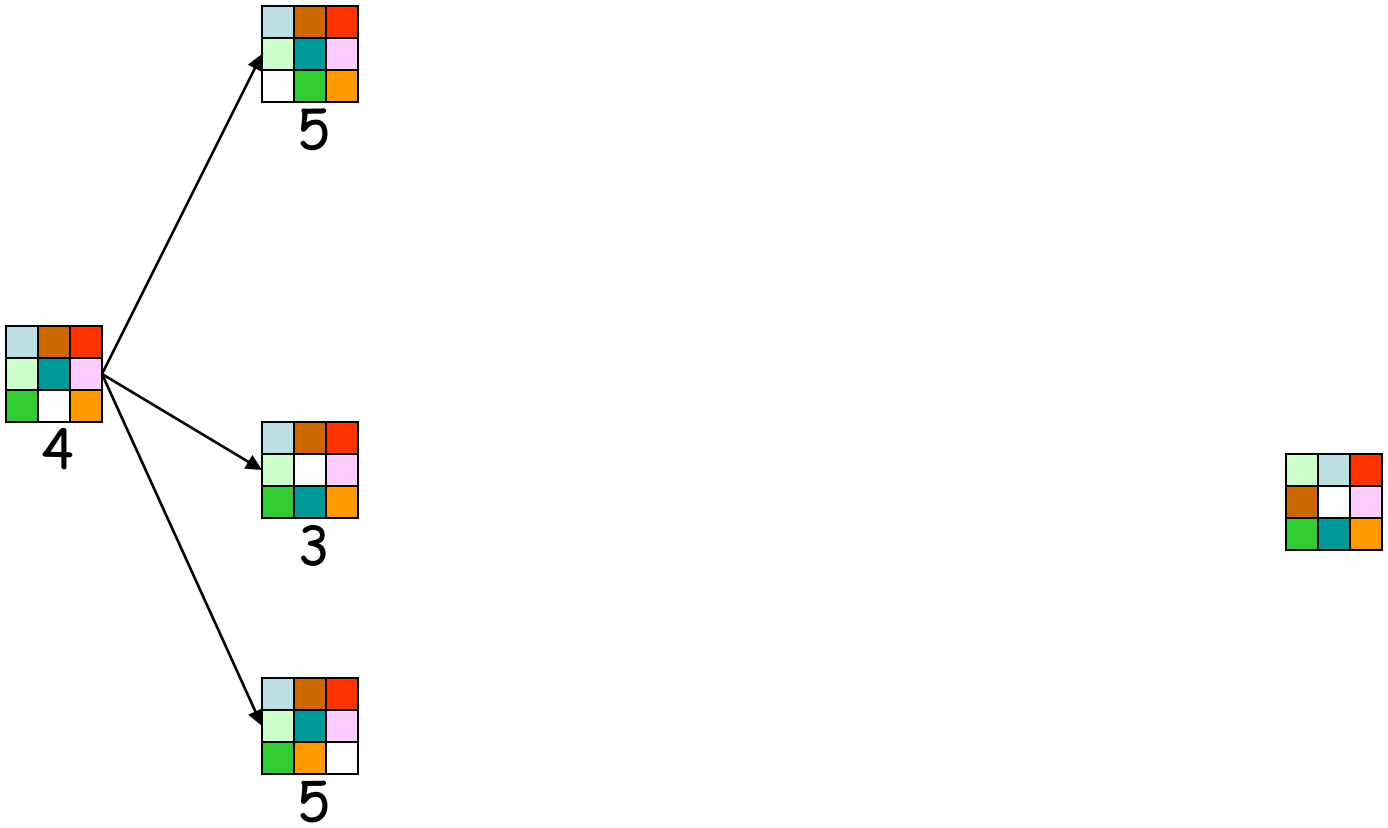
f(N) = h(N) = number of misplaced numbered tiles

4

The white tile is the empty tile

8-Puzzle

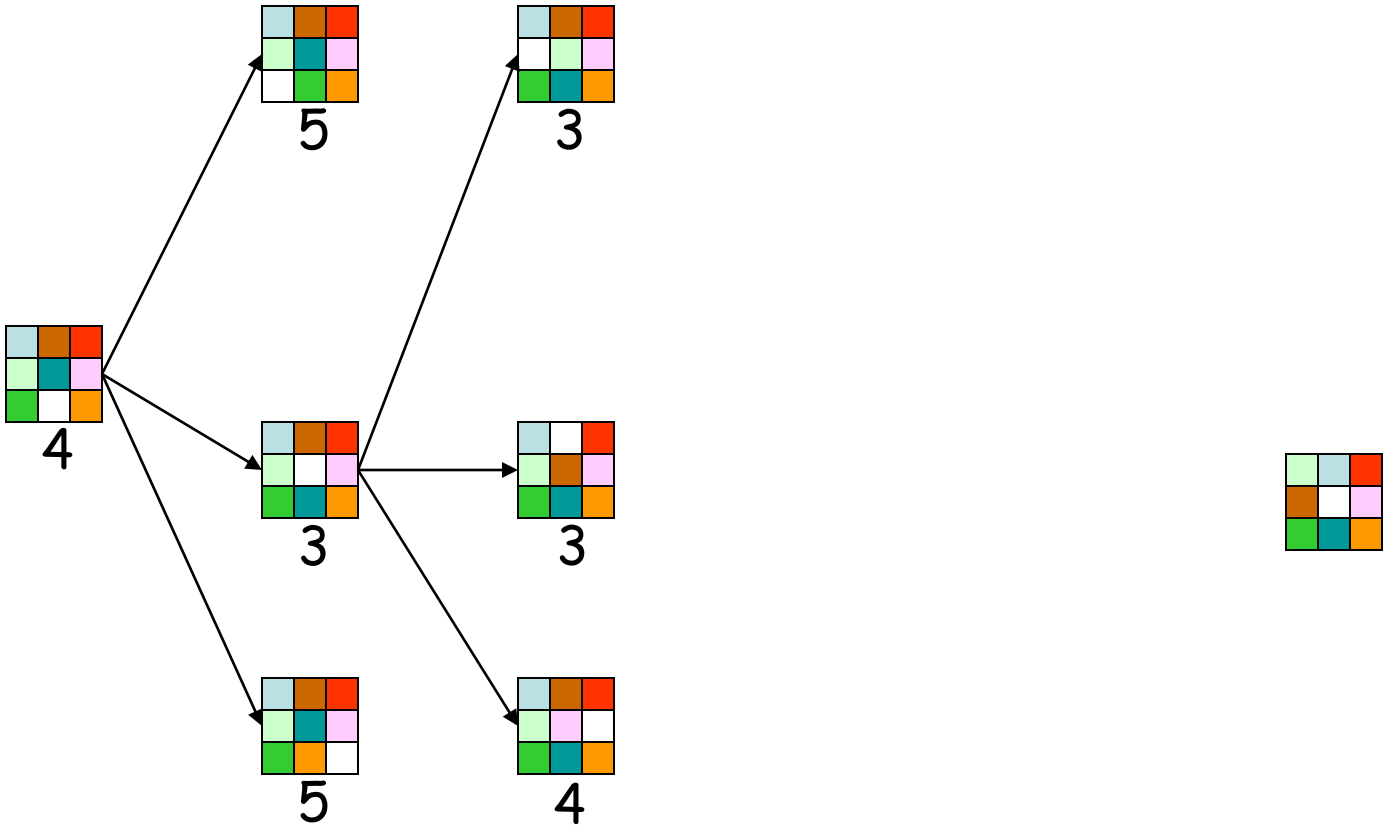
f(N) = h(N) = number of misplaced numbered tiles



The white tile is the empty tile

8-Puzzle

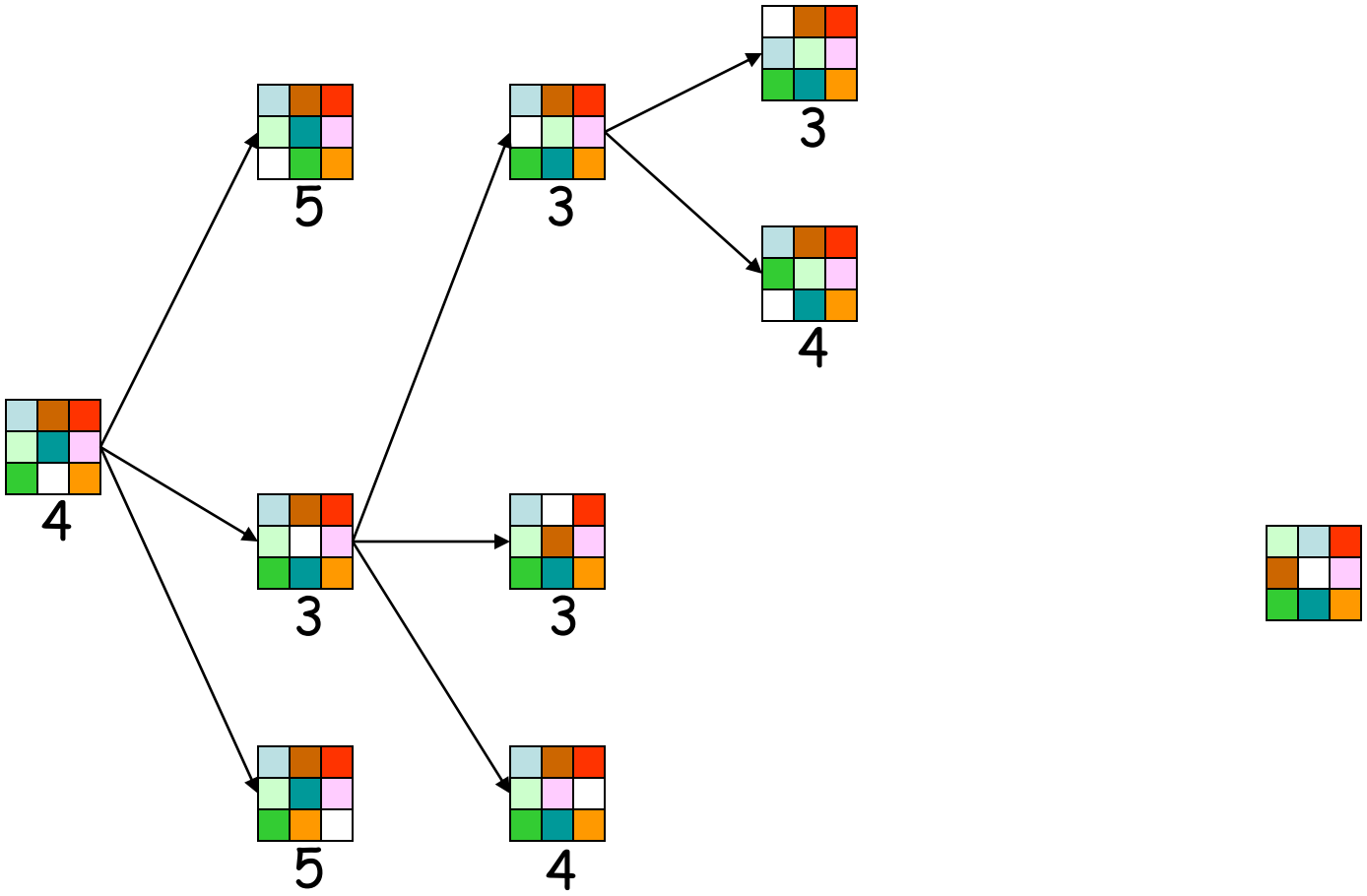
f(N) = h(N) = number of misplaced numbered tiles



The white tile is the empty tile

8-Puzzle

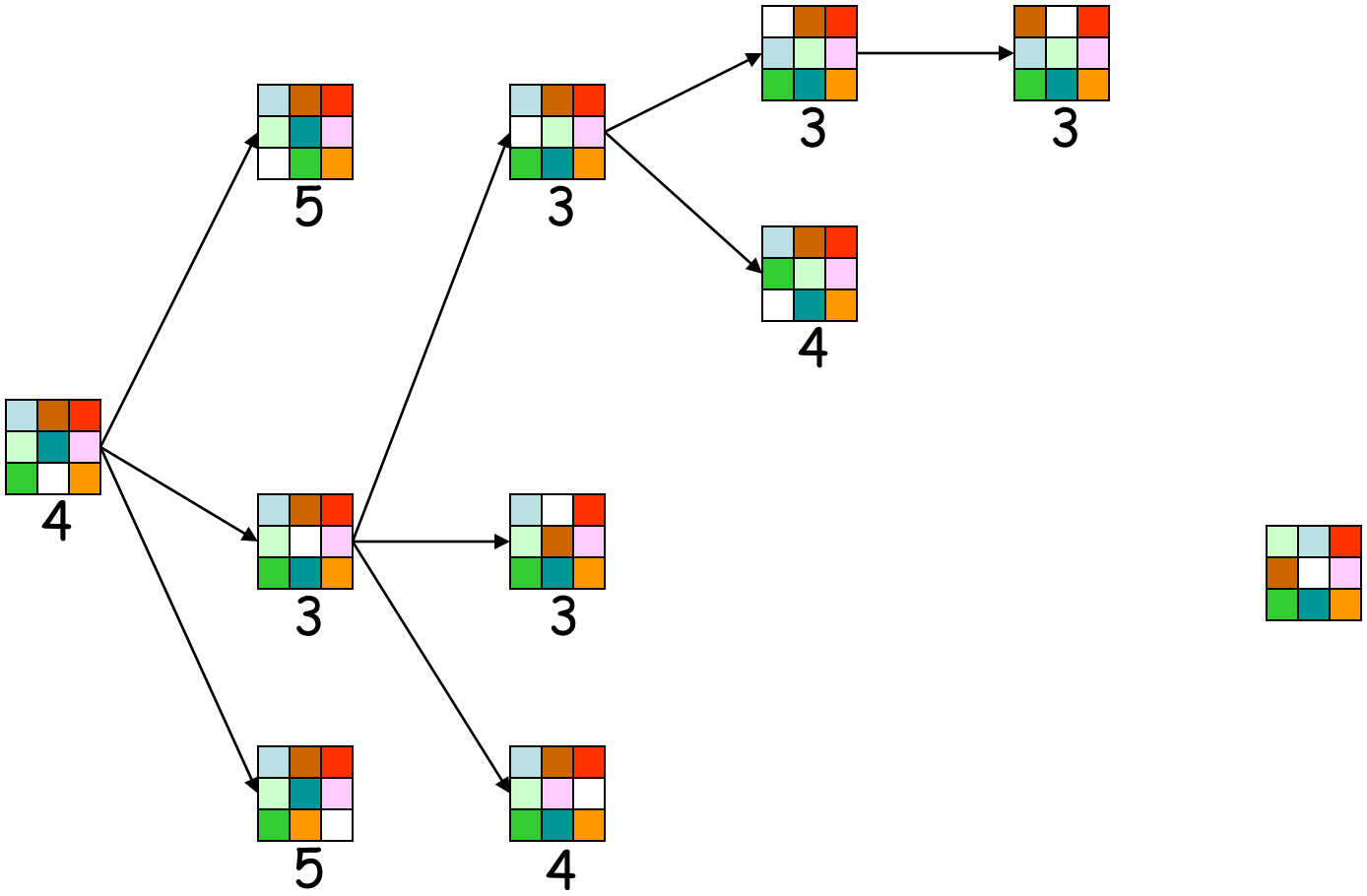
f(N) = h(N) = number of misplaced numbered tiles



The white tile is the empty tile

8-Puzzle

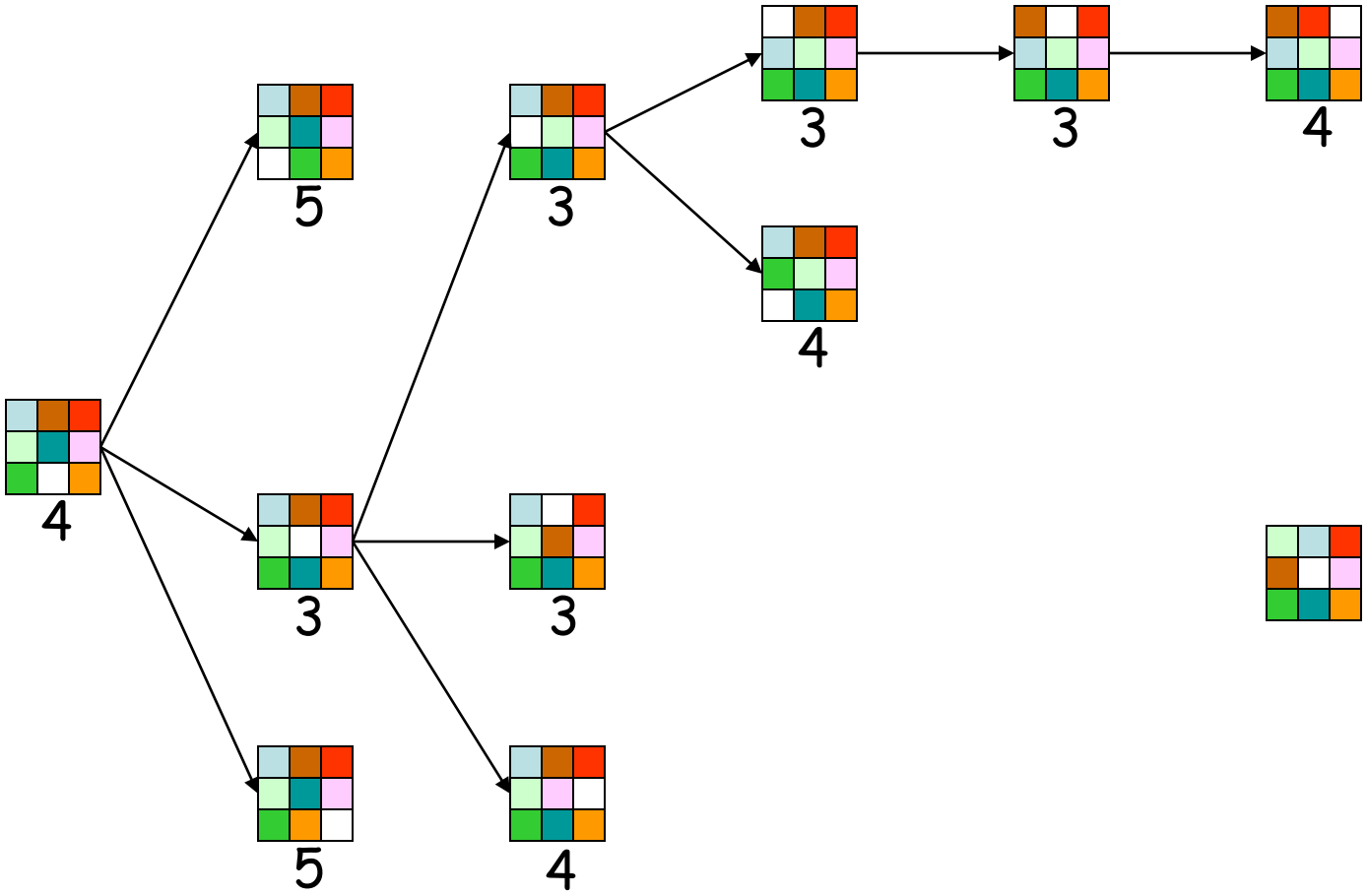
f(N) = h(N) = number of misplaced numbered tiles



The white tile is the empty tile

8-Puzzle

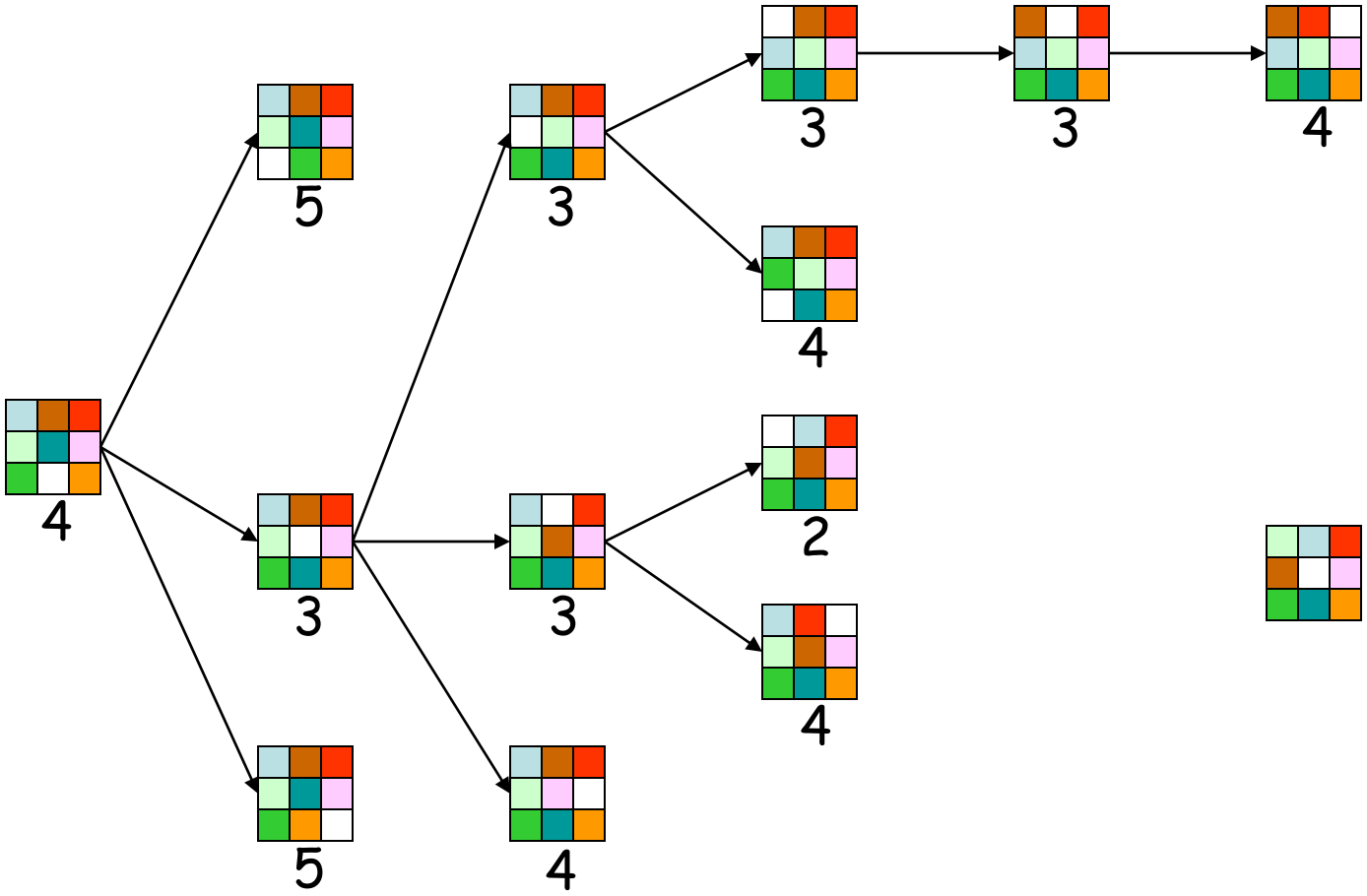
f(N) = h(N) = number of misplaced numbered tiles



The white tile is the empty tile

8-Puzzle

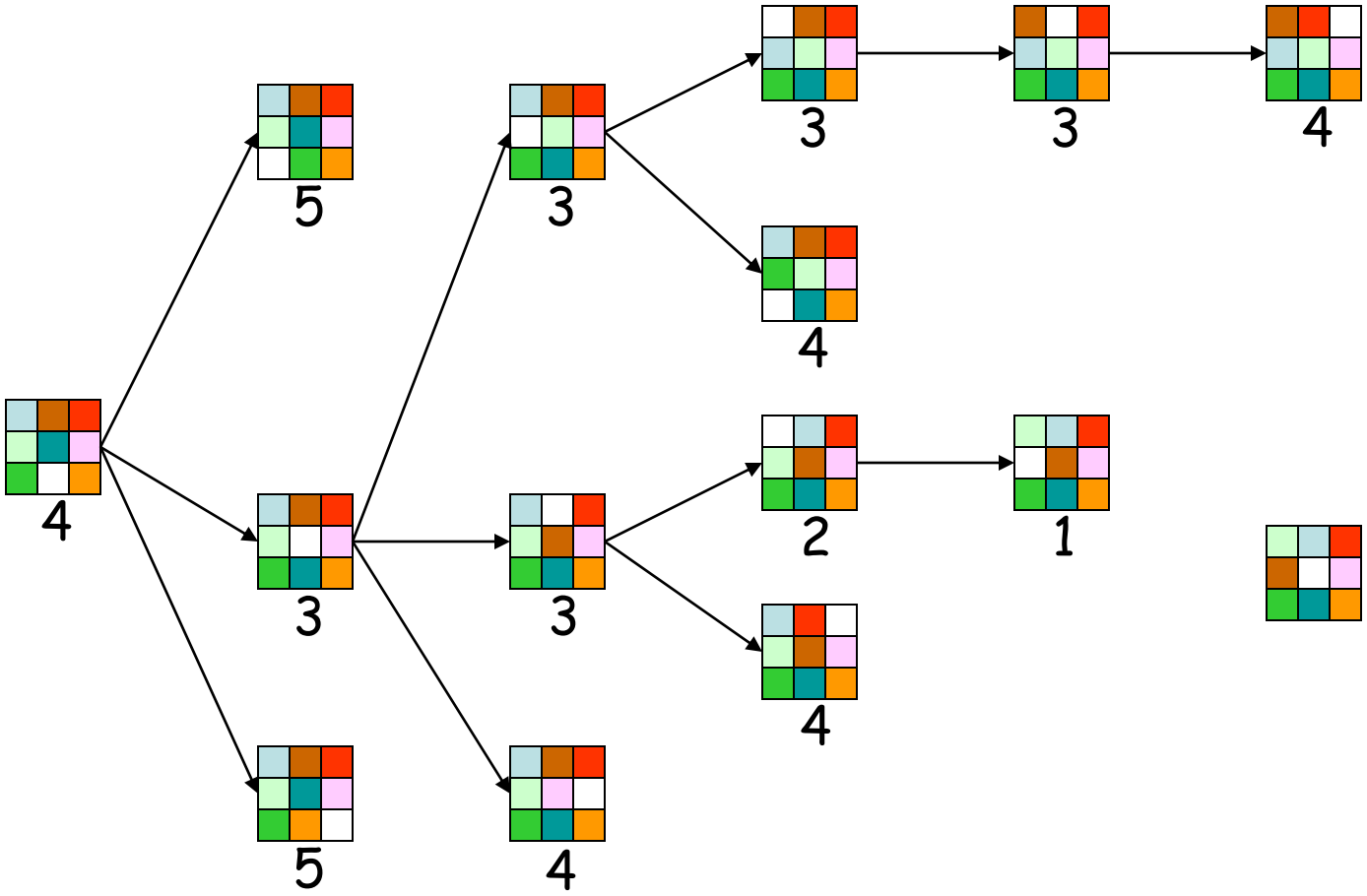
f(N) = h(N) = number of misplaced numbered tiles



The white tile is the empty tile

8-Puzzle

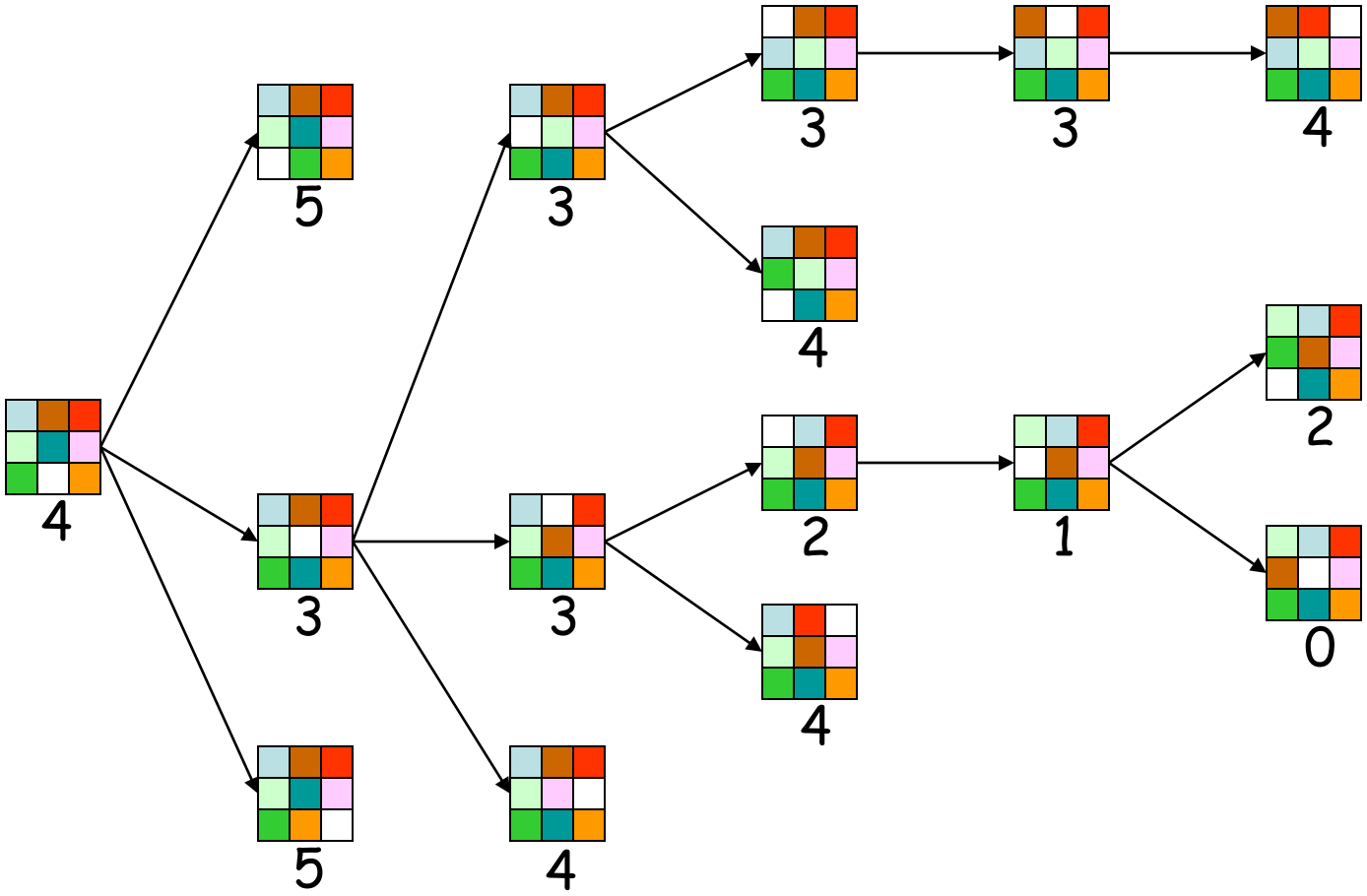
f(N) = h(N) = number of misplaced numbered tiles



The white tile is the empty tile

8-Puzzle

f(N) = h(N) = number of misplaced numbered tiles



The white tile is the empty tile

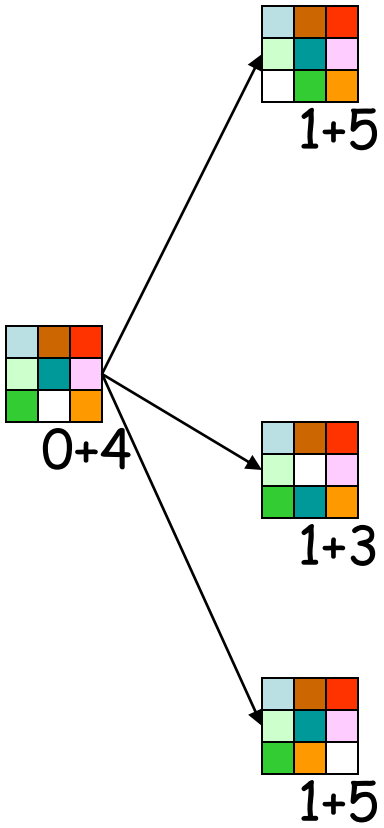
8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced numbered tiles

0+4

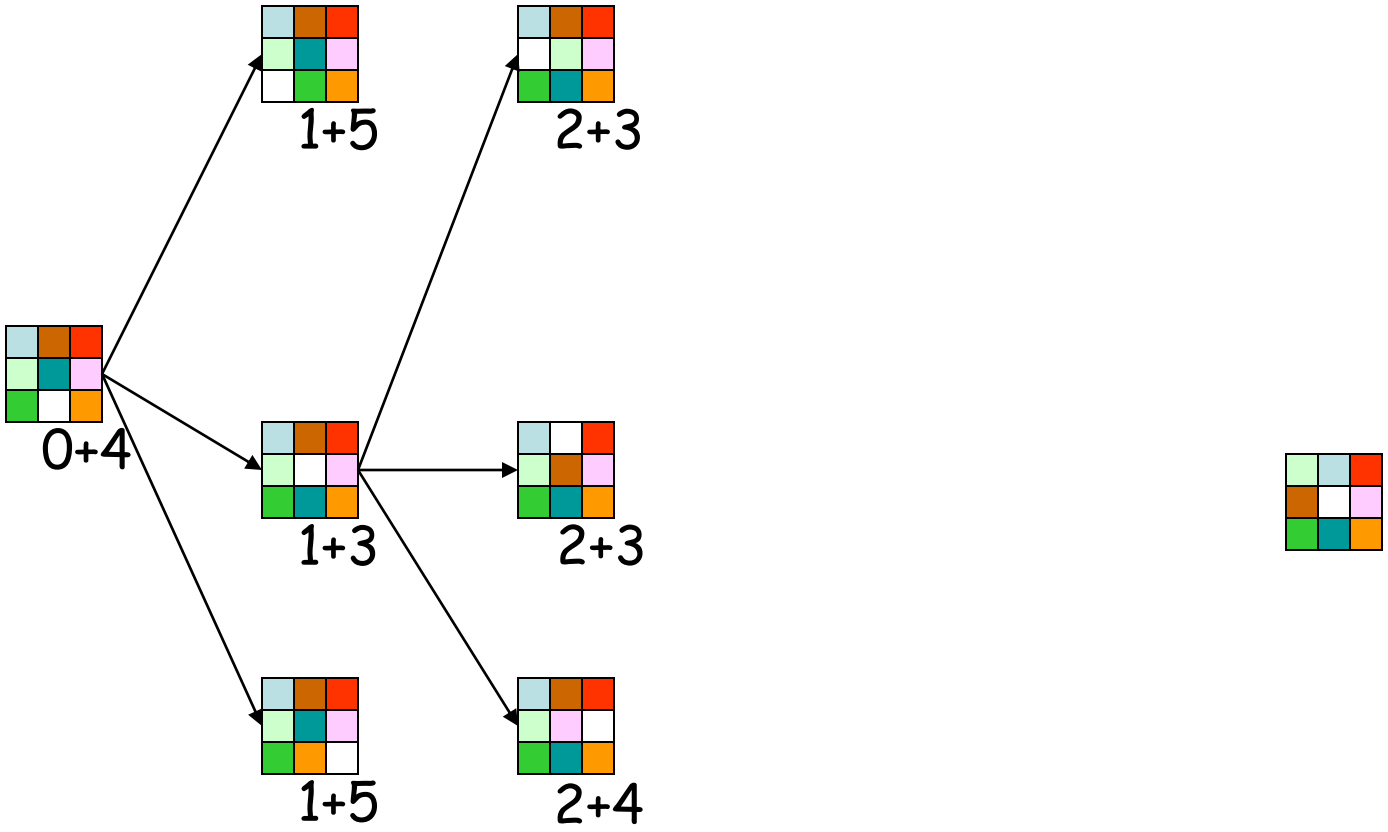
8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced numbered tiles



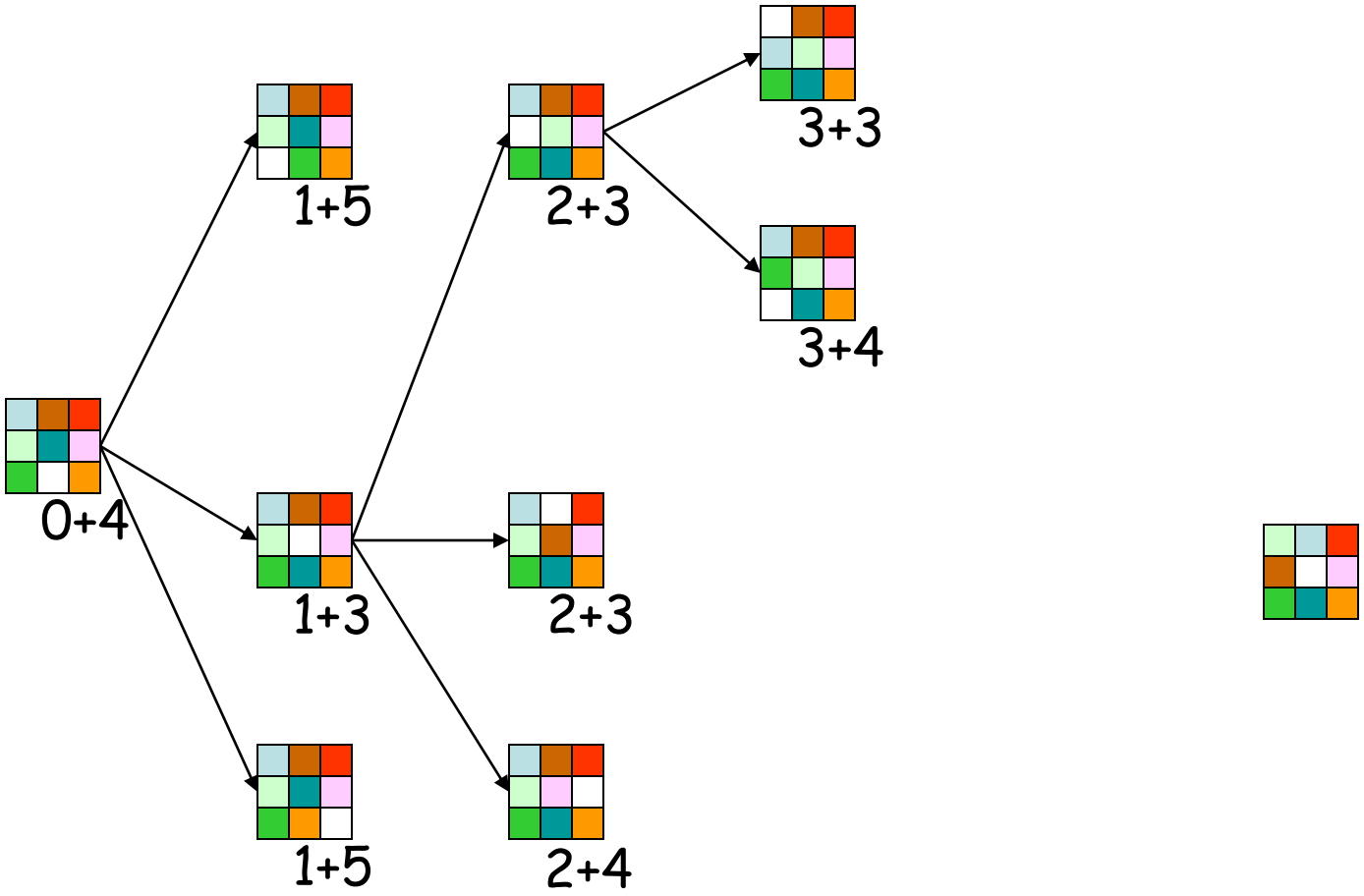
8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced numbered tiles



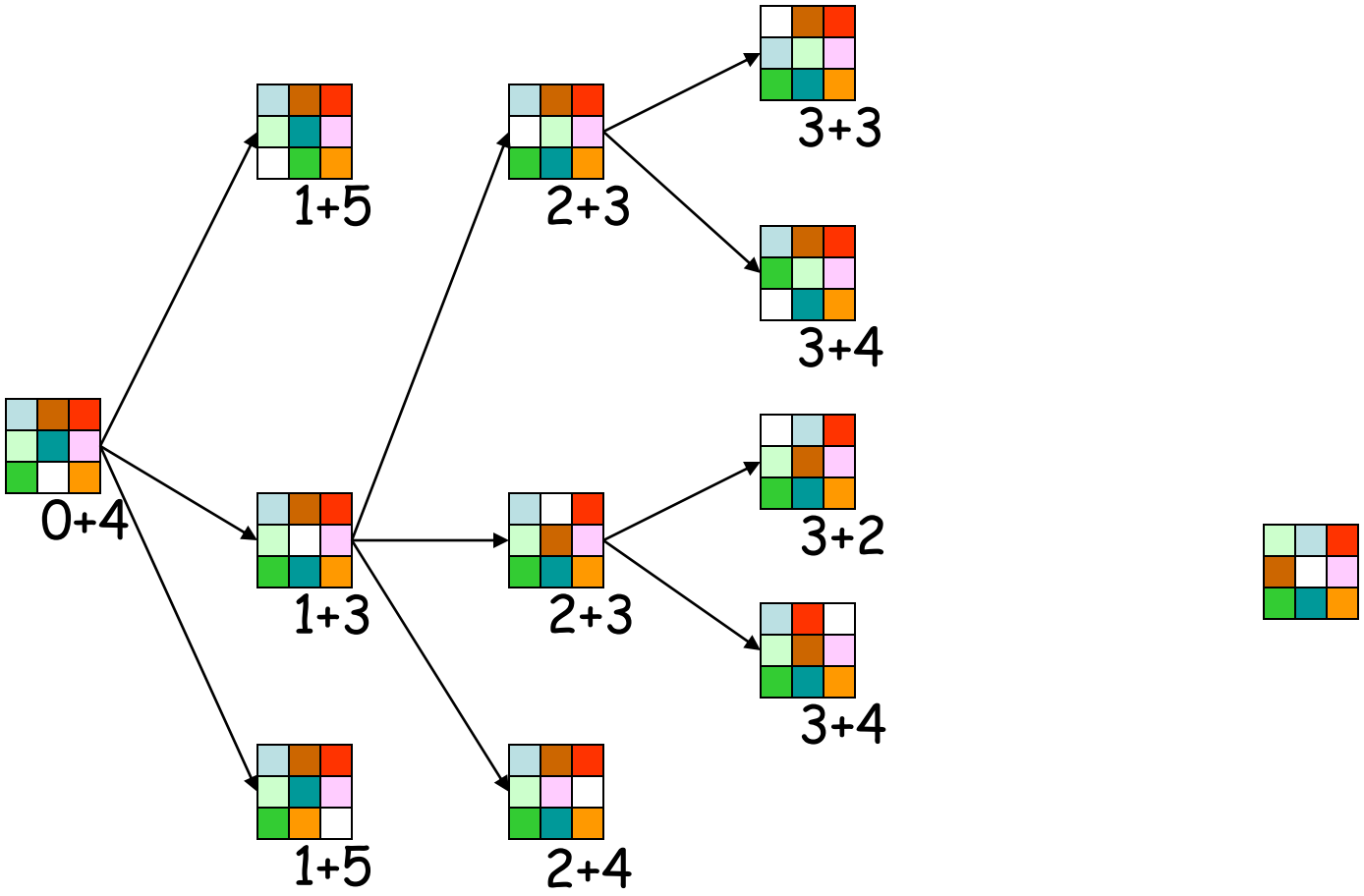
8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced numbered tiles



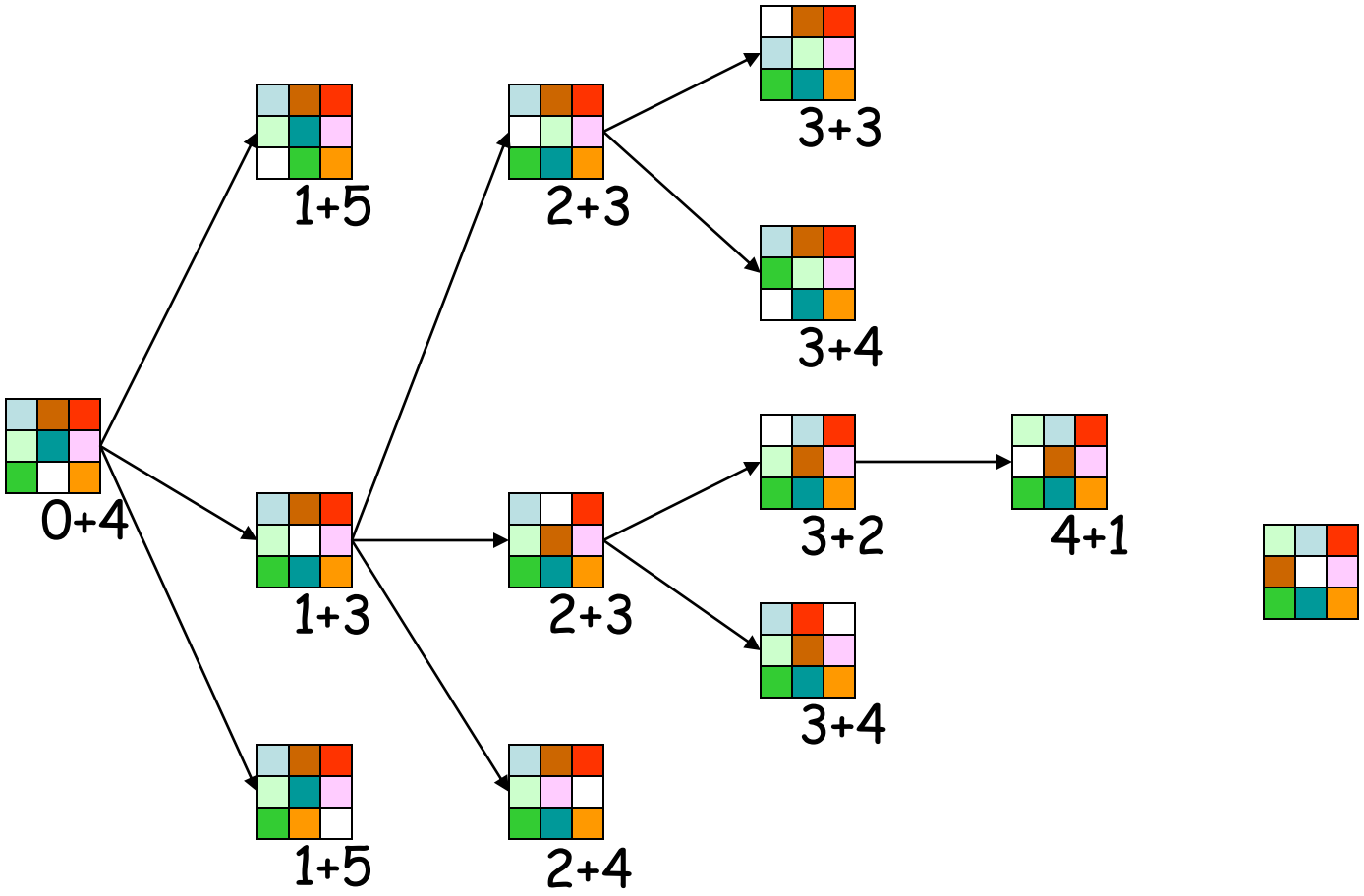
8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced numbered tiles



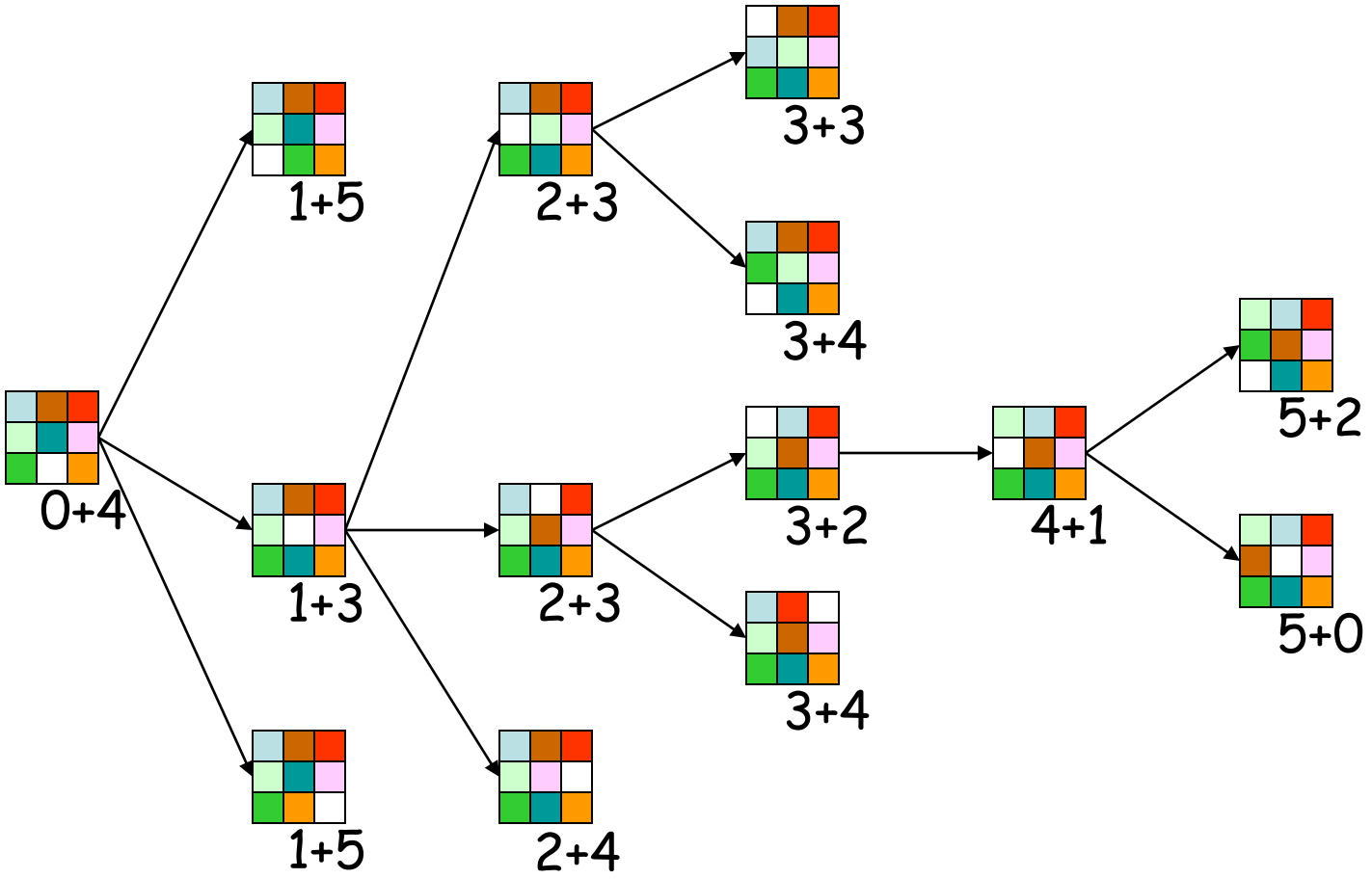
8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced numbered tiles



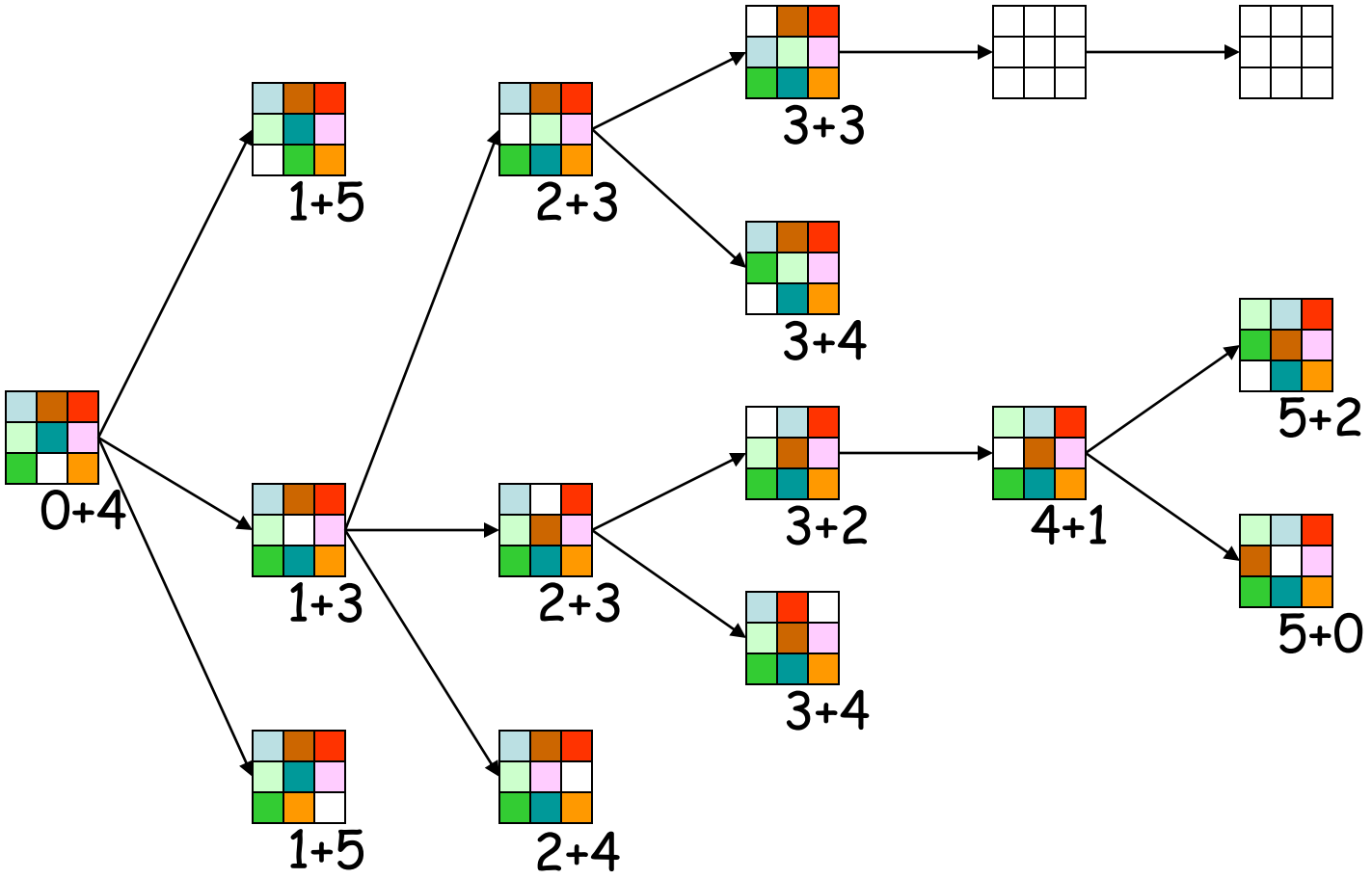
8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced numbered tiles



8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced numbered tiles



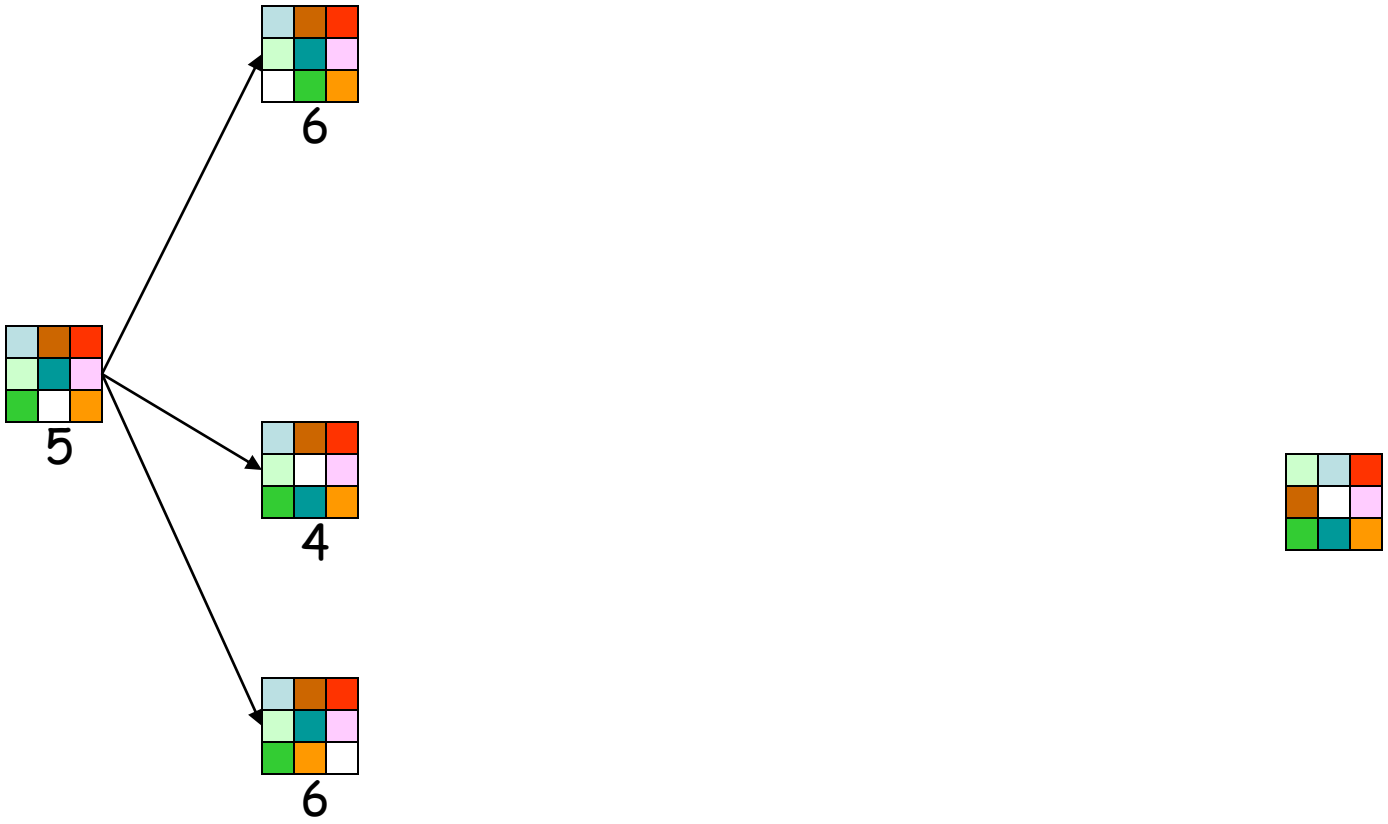
8-Puzzle

f(N) = h(N) = Σ distances of numbered tiles to their goals

5

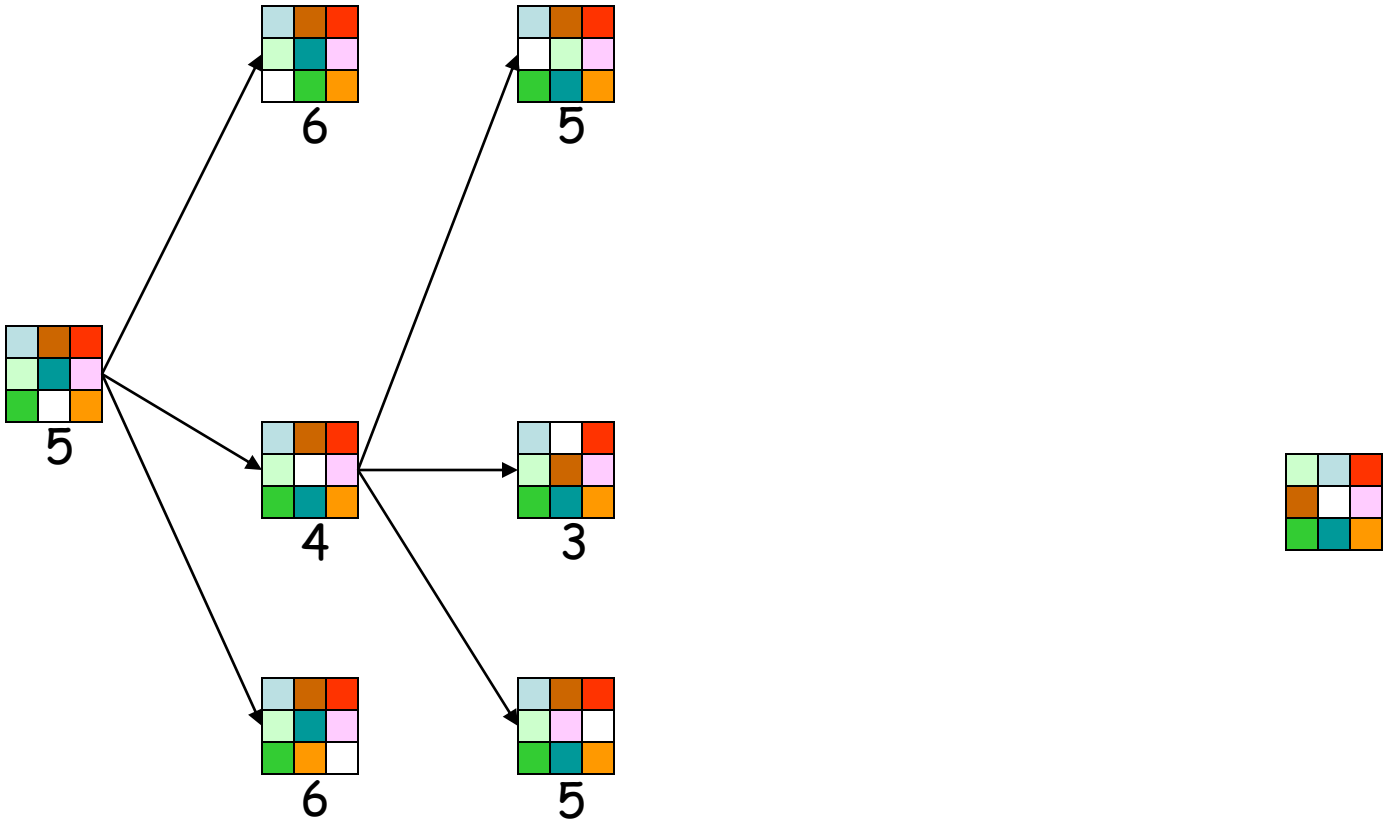
8-Puzzle

f(N) = h(N) = Σ distances of numbered tiles to their goals



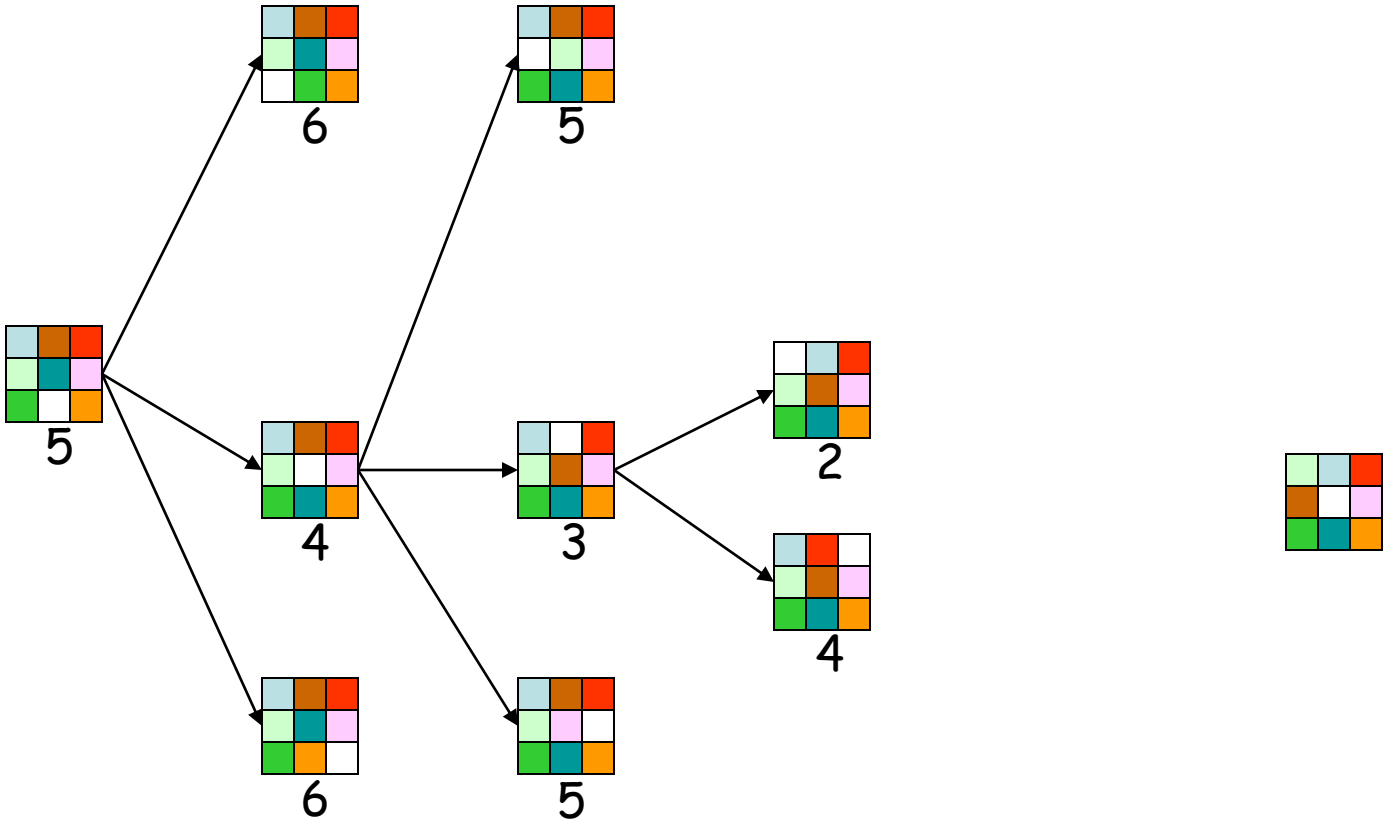
8-Puzzle

f(N) = h(N) = Σ distances of numbered tiles to their goals



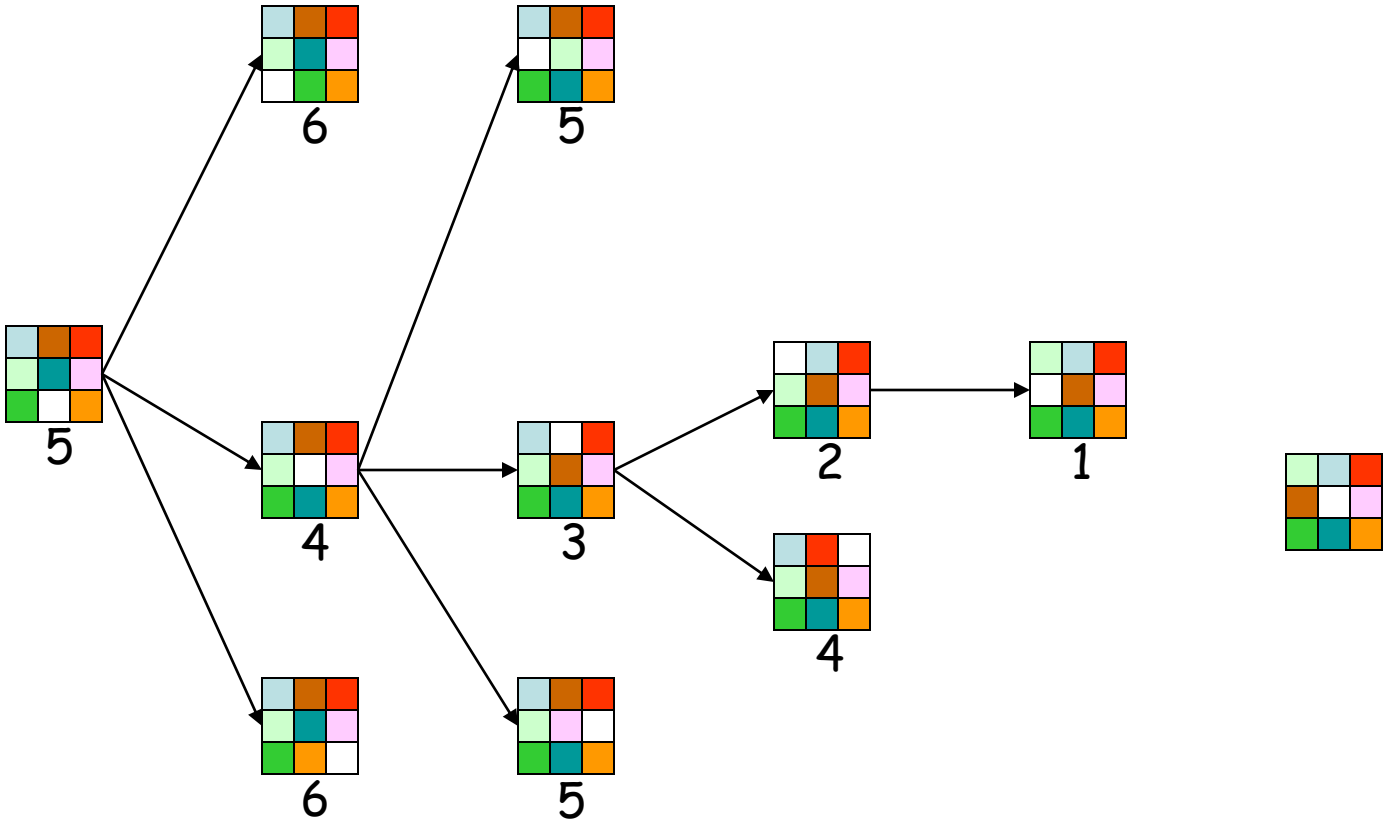
8-Puzzle

f(N) = h(N) = Σ distances of numbered tiles to their goals



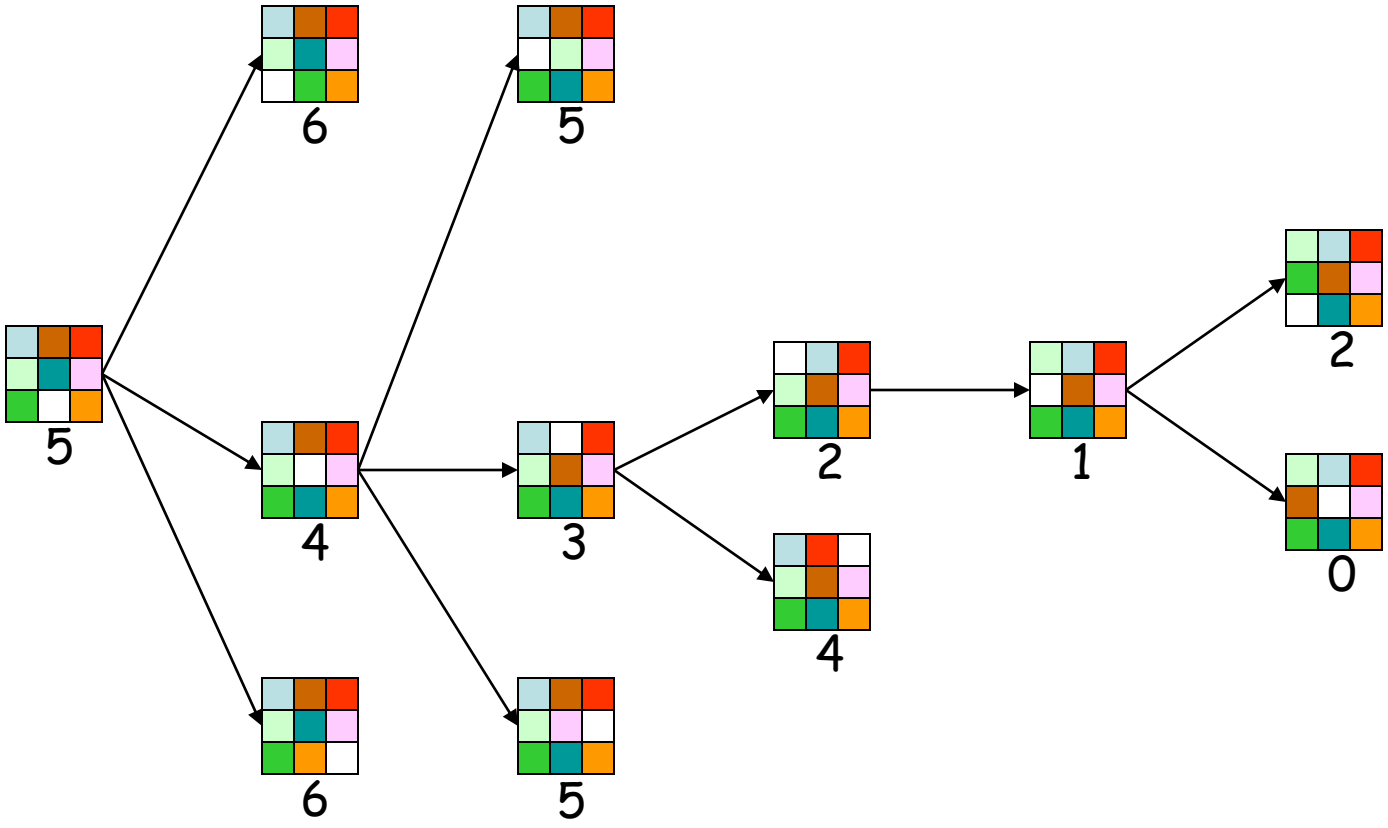
8-Puzzle

f(N) = h(N) = Σ distances of numbered tiles to their goals



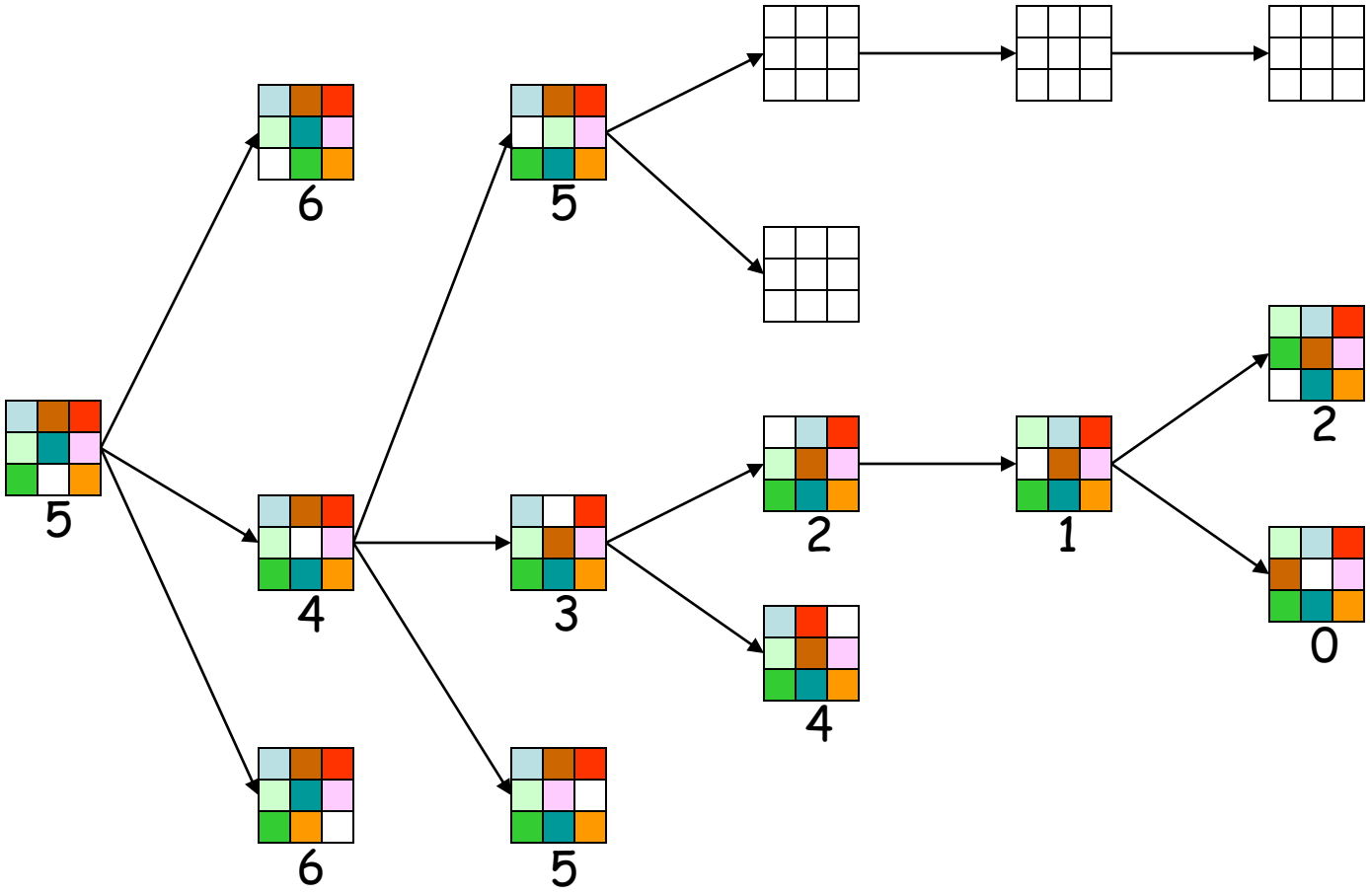
8-Puzzle

f(N) = h(N) = Σ distances of numbered tiles to their goals

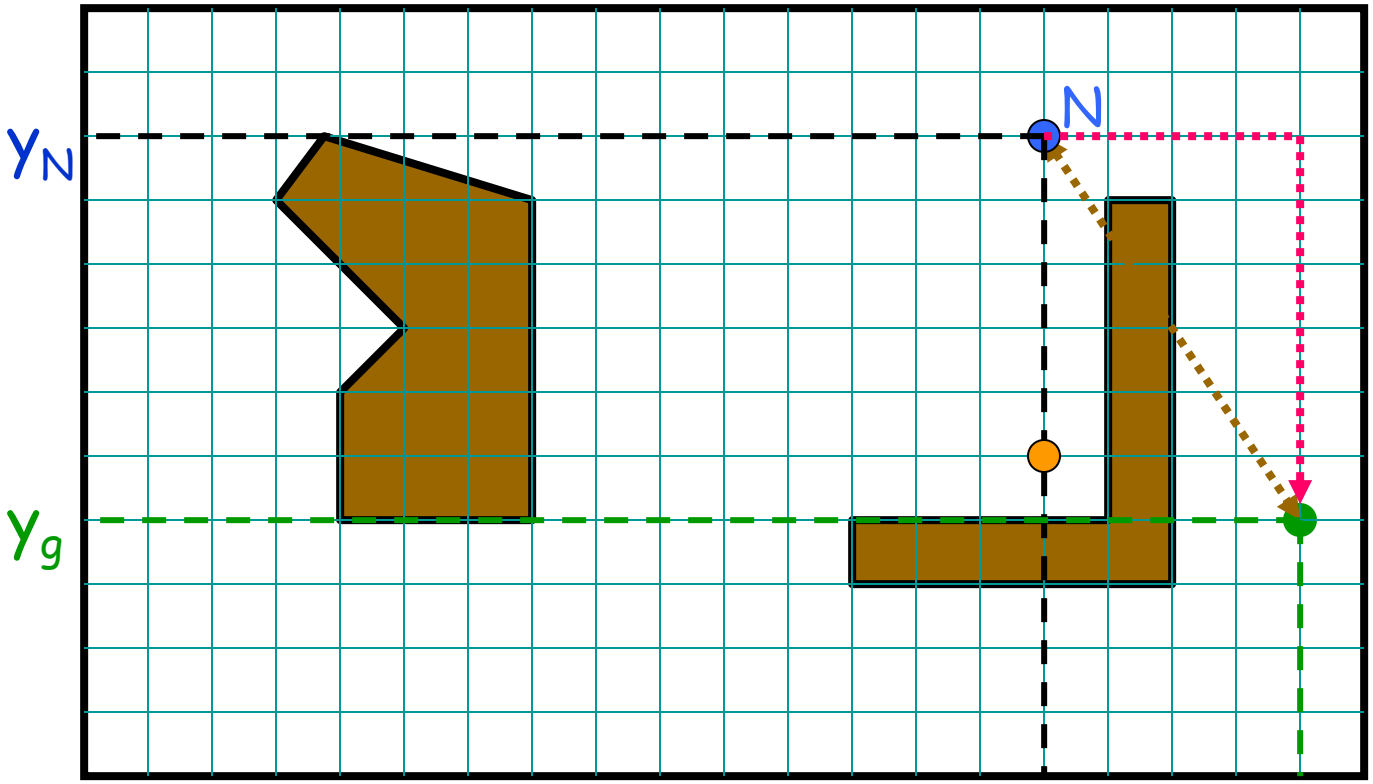


8-Puzzle

f(N) = h(N) = Σ distances of numbered tiles to their goals



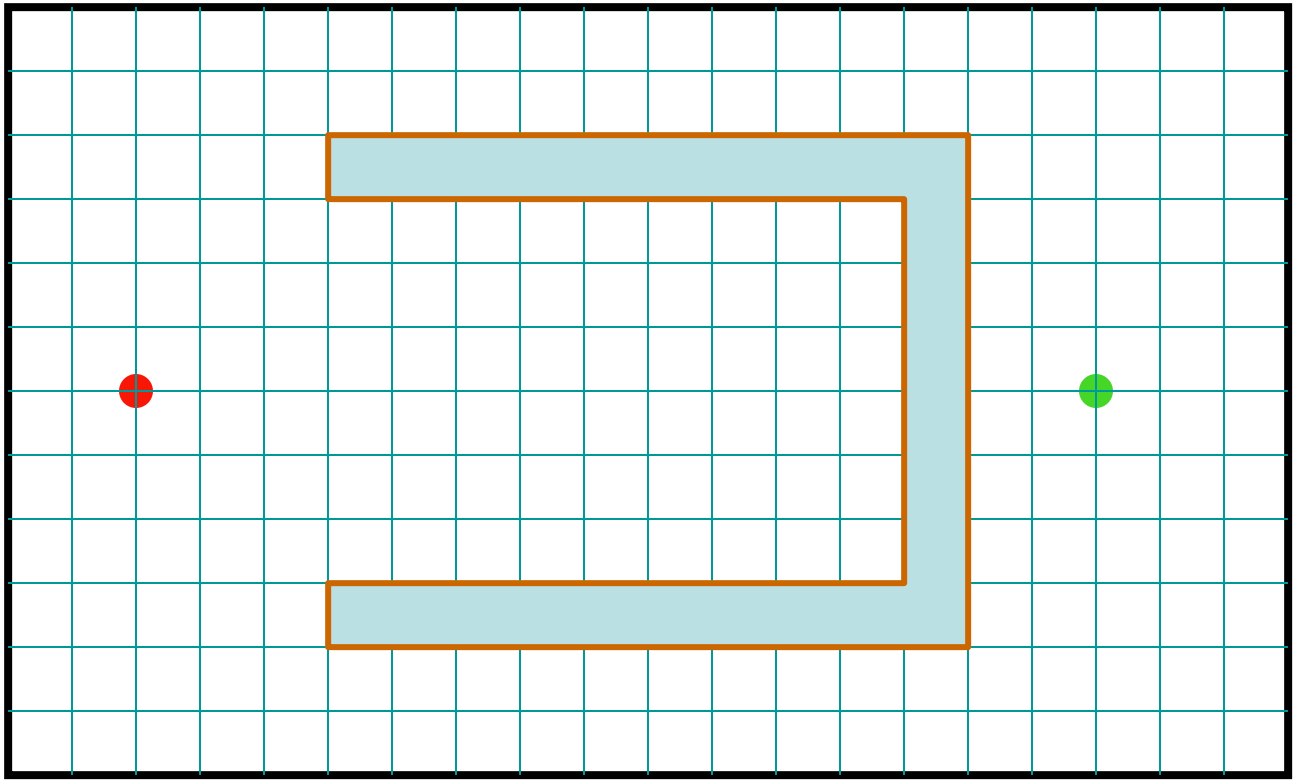
Robot Navigation



$$h_1(N) = \sqrt{(x_N - x_g)^2 + (y_N - y_g)^2}$$ (L_2 or Euclidean distance)

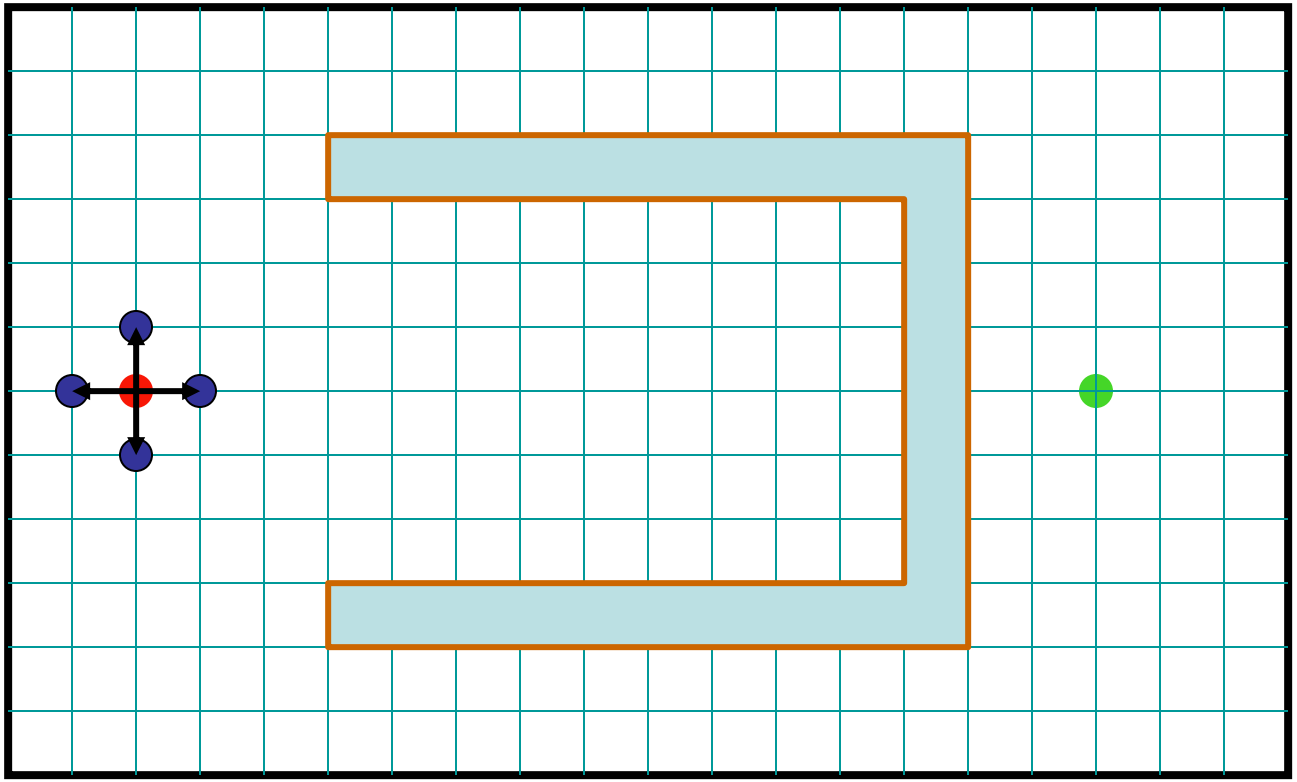
$$h_2(N) = |x_N - x_g| + |y_N - y_g|$$ (L_1 or Manhattan distance)

Best-First $\not\rightarrow$ Efficiency



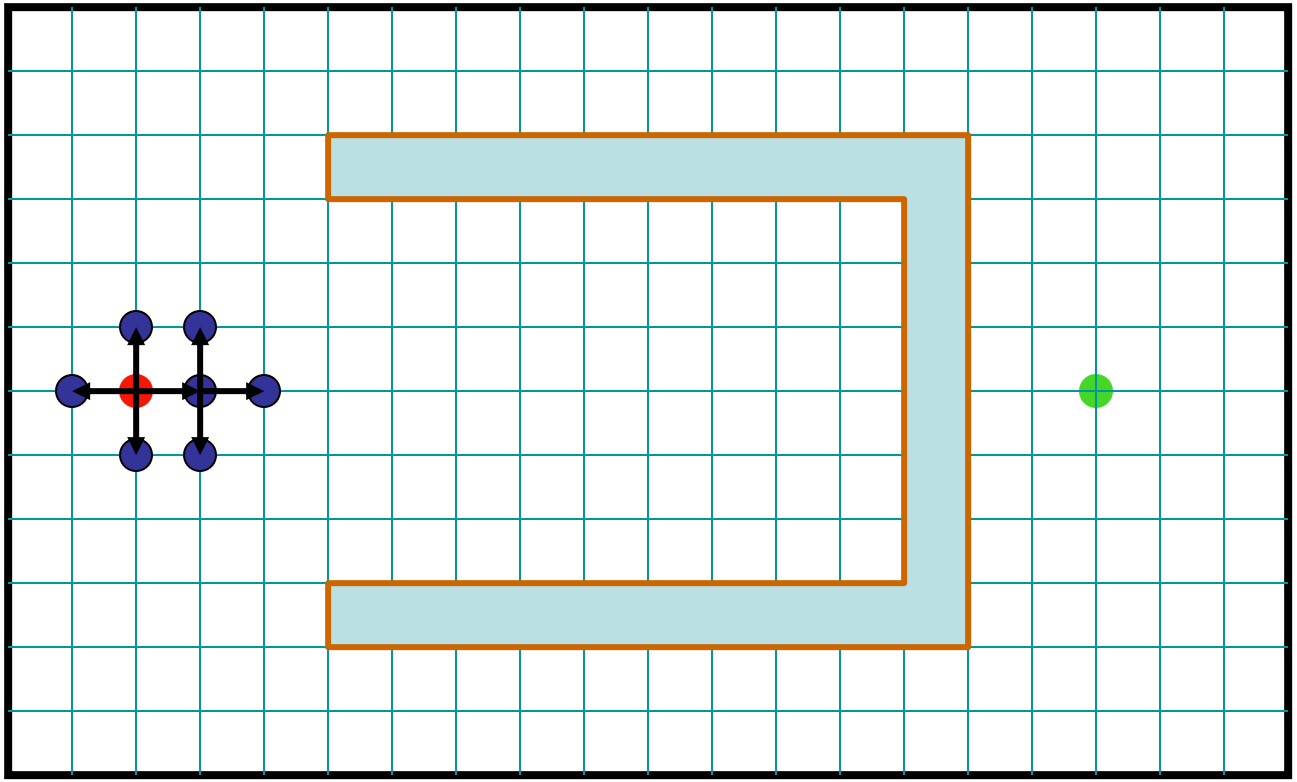
f(N) = h(N) = straight distance to the goal

Best-First $\not\rightarrow$ Efficiency



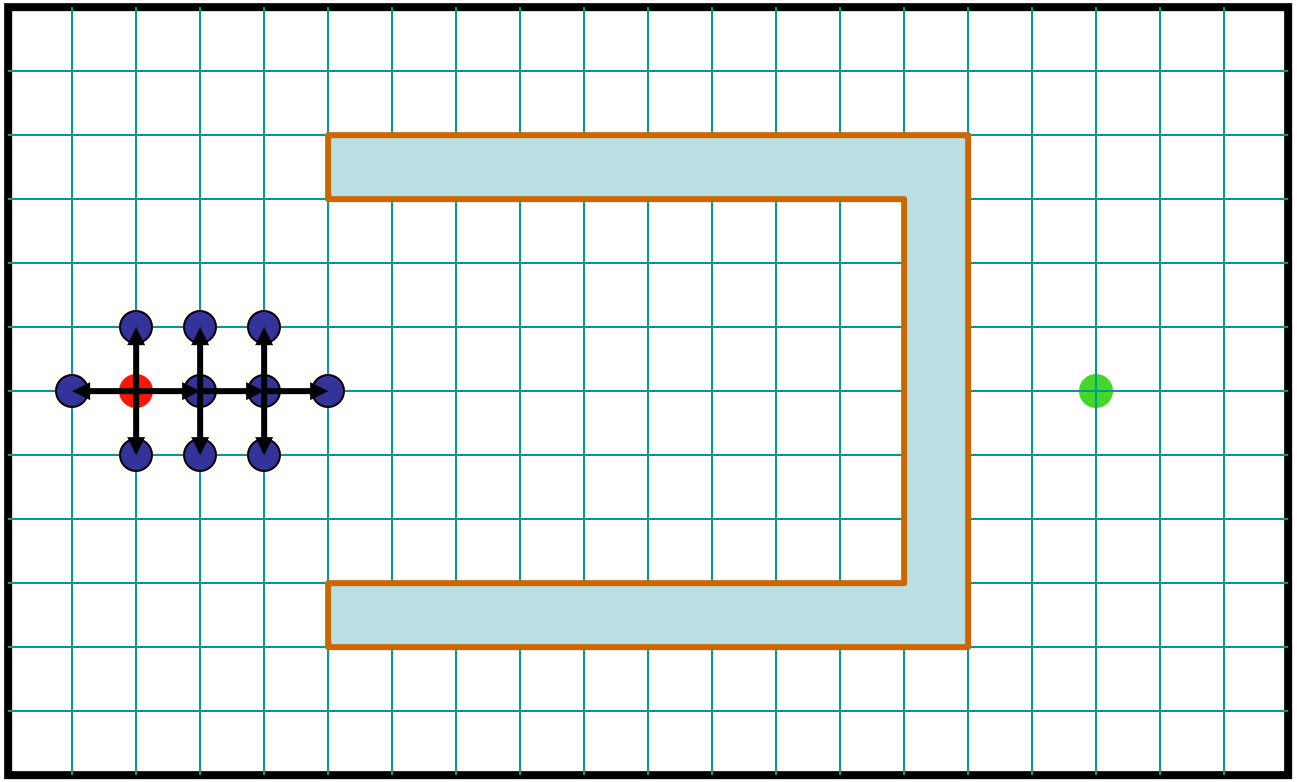
f(N) = h(N) = straight distance to the goal

Best-First $\not\rightarrow$ Efficiency



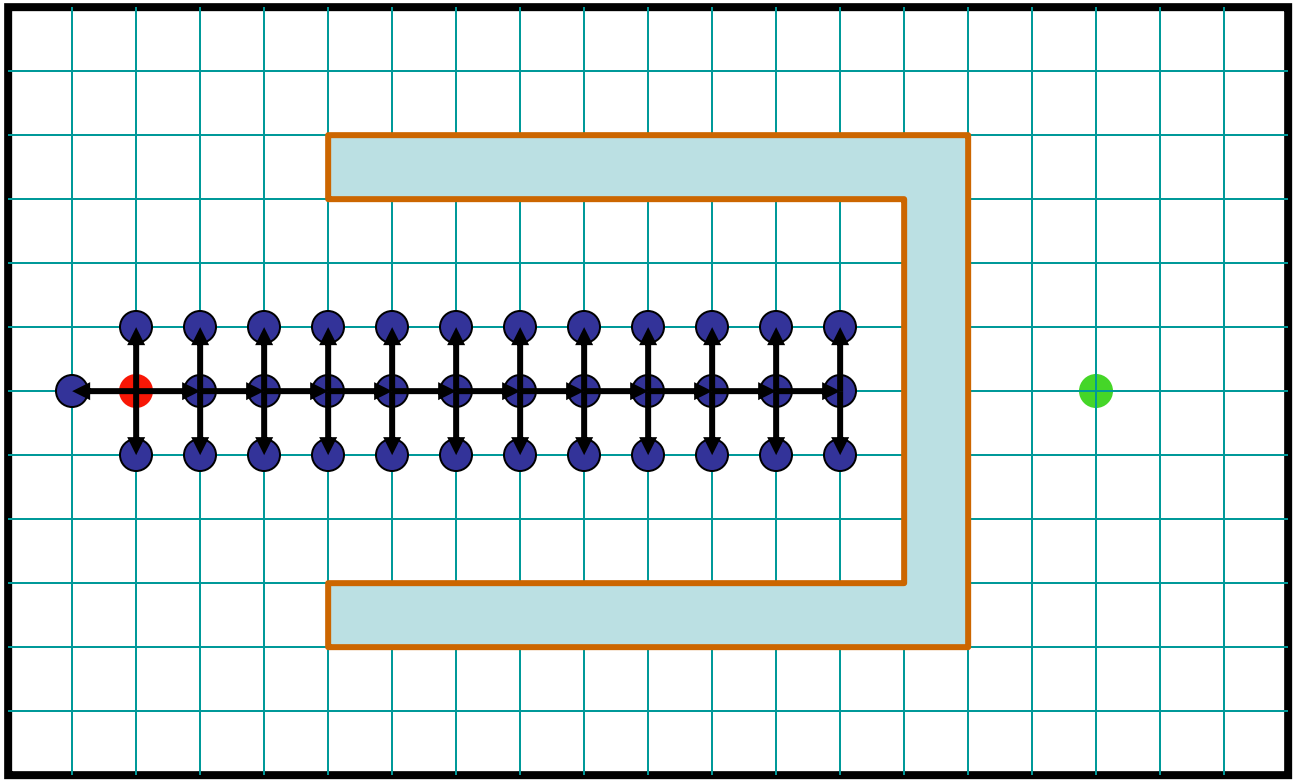
f(N) = h(N) = straight distance to the goal

Best-First $\not\rightarrow$ Efficiency



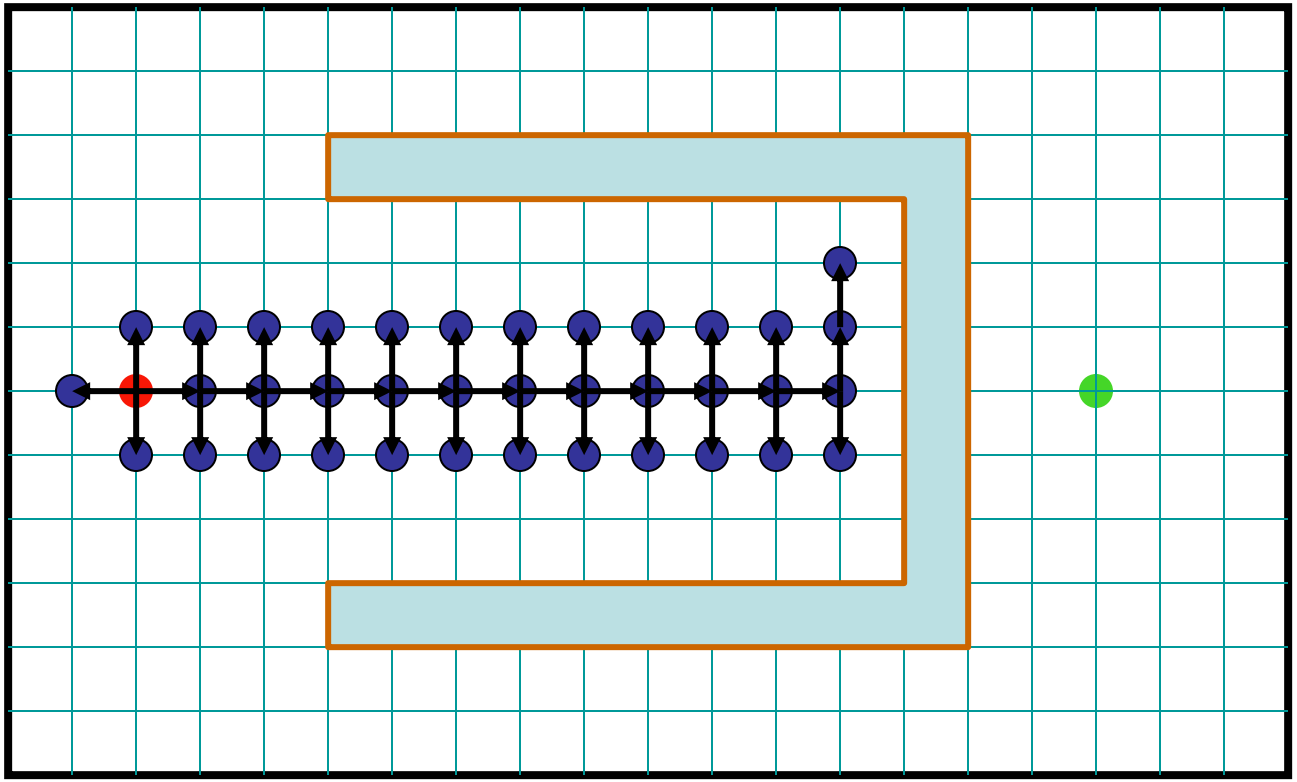
f(N) = h(N) = straight distance to the goal

Best-First $\not\rightarrow$ Efficiency



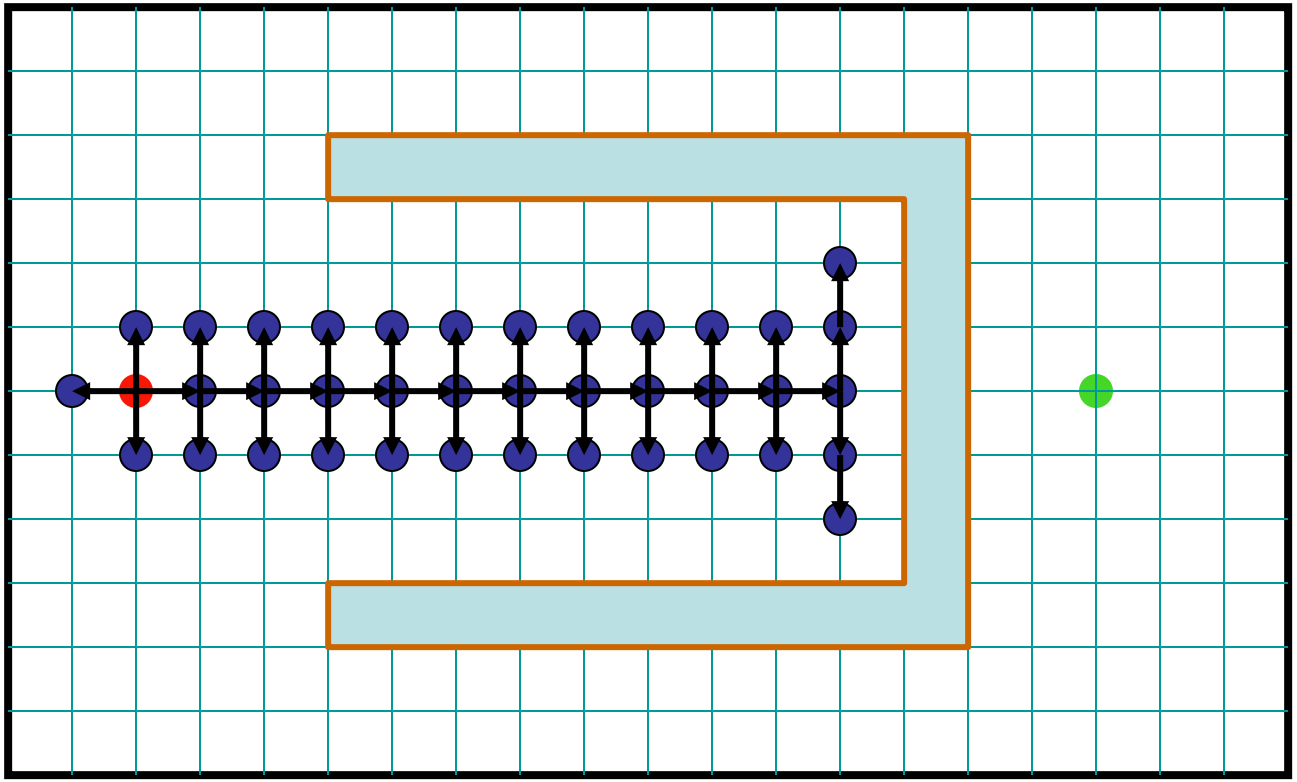
f(N) = h(N) = straight distance to the goal

Best-First $\not\rightarrow$ Efficiency



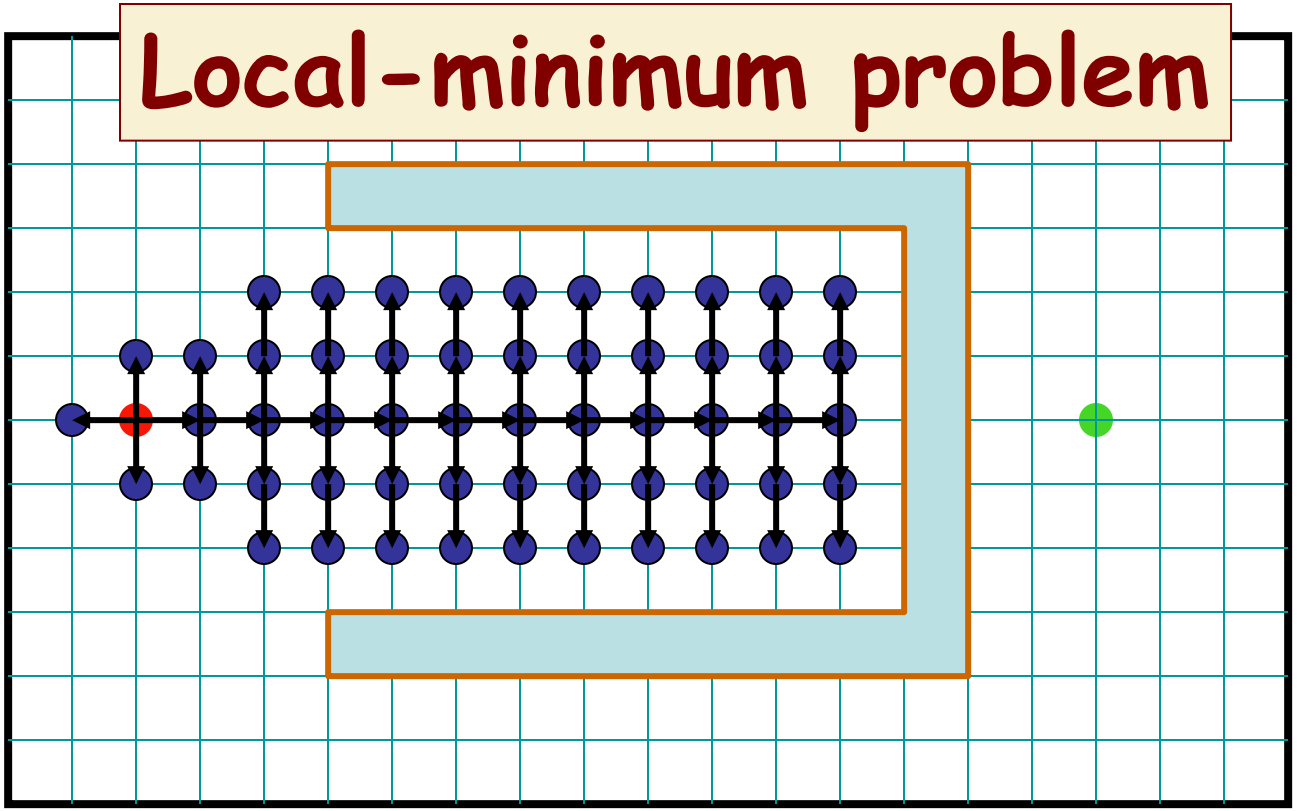
f(N) = h(N) = straight distance to the goal

Best-First $\not\rightarrow$ Efficiency



f(N) = h(N) = straight distance to the goal

Best-First $\not\rightarrow$ Efficiency



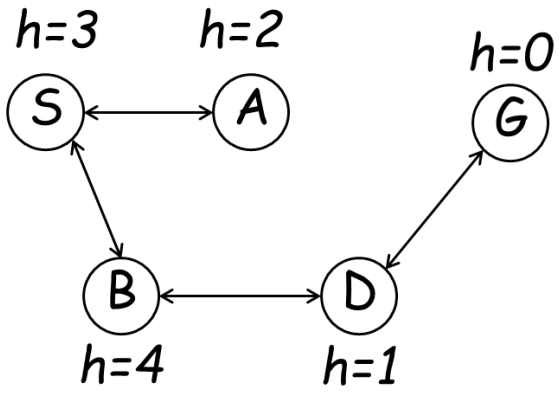
f(N) = h(N) = straight distance to the goal

Greedy best-first search

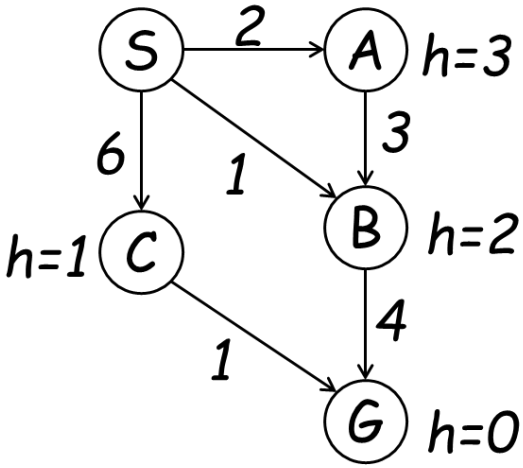
- Evaluation function
  - f(N) = h(N)
- Greedy best-first search expands the node that appears to be closest to goal

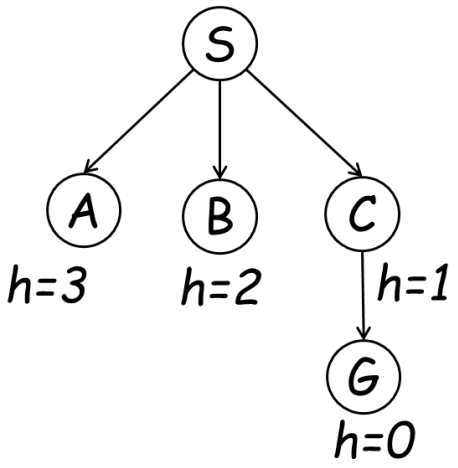
Properties of greedy best-first search

- Complete? No
  - Similar to DFS, only graph search version is complete in finite spaces
  - Infinite loops



- Time and space
  - $O(b^m)$
- Optimal? No





Can we prove anything?

- If the state space is infinite, in general the search is not complete
- If the state space is finite and we do not discard nodes that revisit states, in general the search is not complete
- If the state space is finite and we discard nodes that revisit states, the search is complete, but in general is not optimal

Admissible Heuristic

- Let h*(N) be the cost of the optimal path from N to a goal node
- The heuristic function h(N) is admissible if:

$$\forall N: 0 \leq h(N) \leq h^*(N)$$

- An admissible heuristic function is always optimistic !

Admissible Heuristic

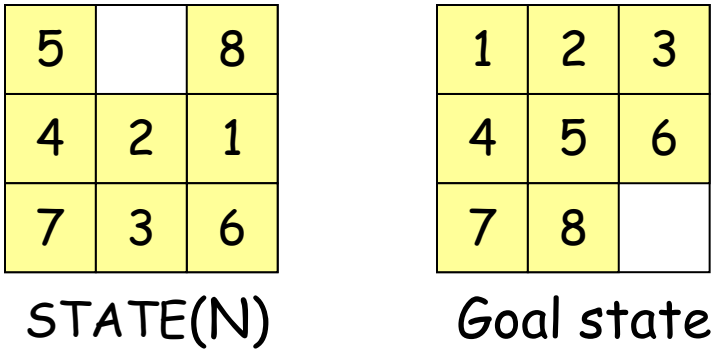
- Let h*(N) be the cost of the optimal path from N to a goal node
- The heuristic function h(N) is admissible if:

$$\forall N: 0 \leq h(N) \leq h^*(N)$$

- An admissible heuristic function is always optimistic !

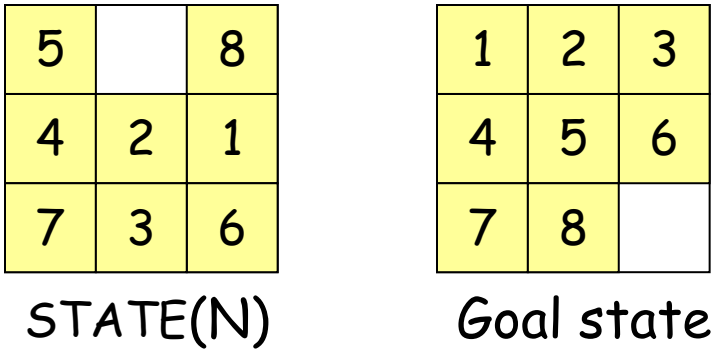
G is a goal node ➔ h(G) = 0

8-Puzzle Heuristics



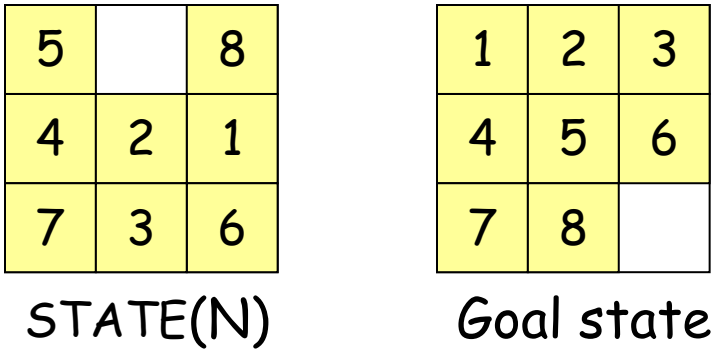
- $h_1(N)$ = number of misplaced tiles = 6 is ???

8-Puzzle Heuristics



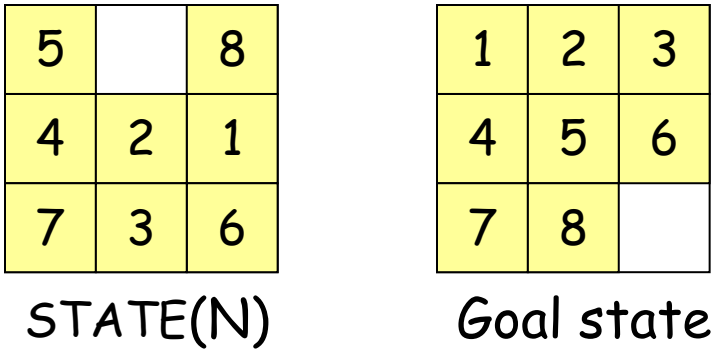
- $h_1(N)$ = number of misplaced tiles = 6
  is admissible
- $h_2(N)$ = sum of the (Manhattan) distances of every tile to its goal position
  = 2 + 3 + 0 + 1 + 3 + 0 + 3 + 1 = 13
  is ???

8-Puzzle Heuristics



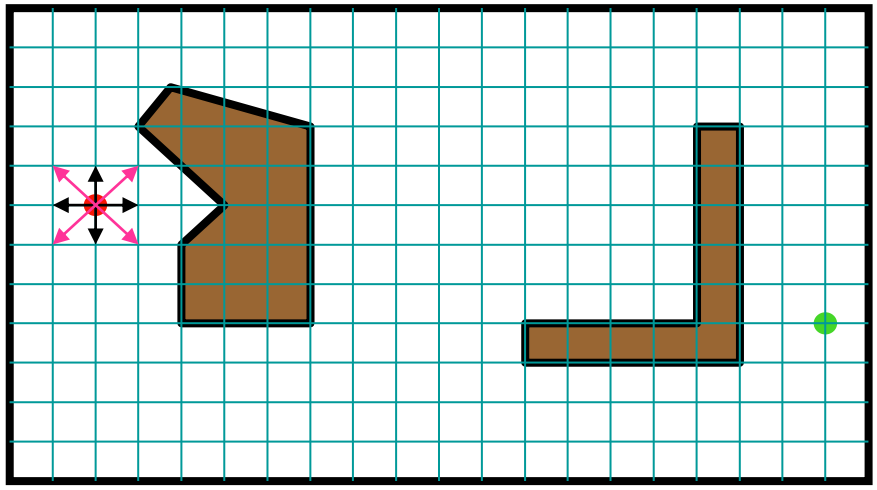
- $h_1(N)$ = number of misplaced tiles = 6
  is admissible
- $h_2(N)$ = sum of the (Manhattan) distances of every tile to its goal position
  = 2 + 3 + 0 + 1 + 3 + 0 + 3 + 1 = 13
  is admissible
- $h_3(N)$ = sum of permutation inversions
  = 4 + 6 + 3 + 1 + 0 + 2 + 0 + 0 = 16
  is ???

8-Puzzle Heuristics



- $h_1(N)$ = number of misplaced tiles = 6
  is admissible
- $h_2(N)$ = sum of the (Manhattan) distances of every tile to its goal position
  = 2 + 3 + 0 + 1 + 3 + 0 + 3 + 1 = 13
  is admissible
- $h_3(N)$ = sum of permutation inversions
  = 4 + 6 + 3 + 1 + 0 + 2 + 0 + 0 = 16
  is not admissible

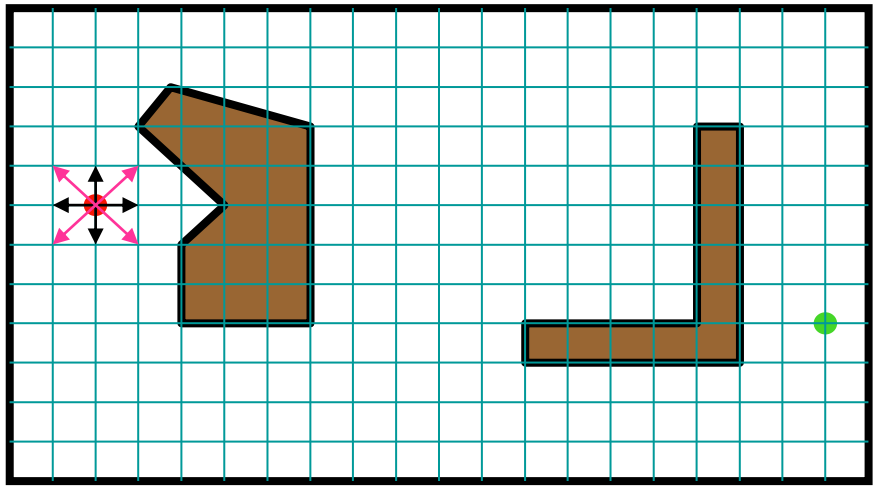
Robot Navigation Heuristics



Cost of one horizontal/vertical step = 1
Cost of one diagonal step = $\sqrt{2}$

$h_1(N) = \sqrt{(x_N - x_g)^2 + (y_N - y_g)^2}$ is admissible

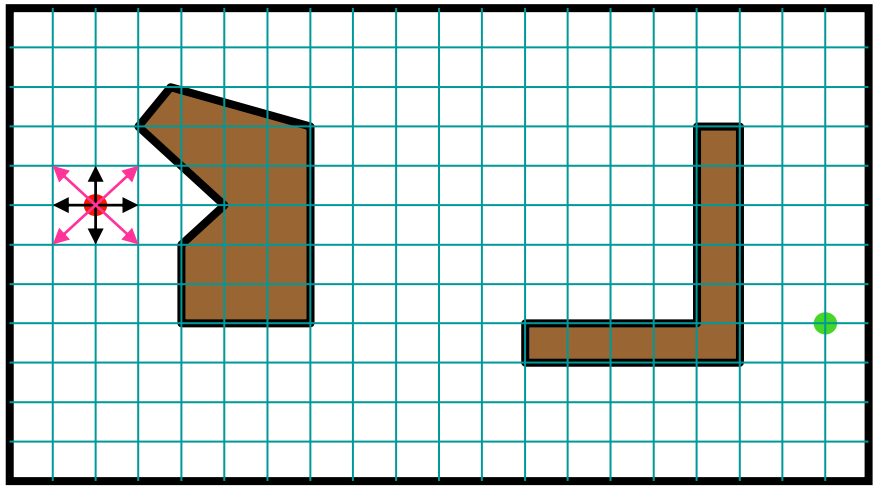
Robot Navigation Heuristics



Cost of one horizontal/vertical step = 1
Cost of one diagonal step = $\sqrt{2}$

$h_2(N) = |x_N - x_g| + |y_N - y_g|$ is ???

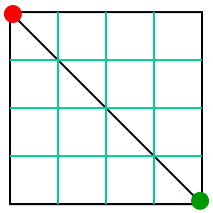
Robot Navigation Heuristics



Cost of one horizontal/vertical step = 1
Cost of one diagonal step = $\sqrt{2}$

$h_2(N) = |x_N - x_g| + |y_N - y_g|$ is admissible if moving along diagonals is not allowed, and not admissible otherwise

$h^*(I) = 4\sqrt{2}$
$h_2(I) = 8$



How to create an admissible h?

- An admissible heuristic can usually be seen as the cost of an optimal solution to a relaxed problem (one obtained by removing constraints)
- In robot navigation:
  - The Manhattan distance corresponds to removing the obstacles
  - The Euclidean distance corresponds to removing both the obstacles and the constraint that the robot moves on a grid
- More on this topic later

A* Search
(most popular algorithm in AI)

1) f(N) = g(N) + h(N), where:
   - g(N) = cost of best path found so far to N
   - h(N) = **admissible** heuristic function

2) for all arcs: $c(N,N') \geq \varepsilon > 0$

3) SEARCH#2 algorithm is used

➔ Best-first search is then called *A* search*

Result #1

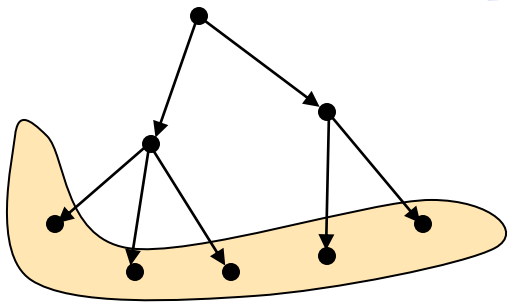
A* is complete and optimal

[This result holds if nodes revisiting states are not discarded]

[TREE-SEARCH version]

Proof (1/2)

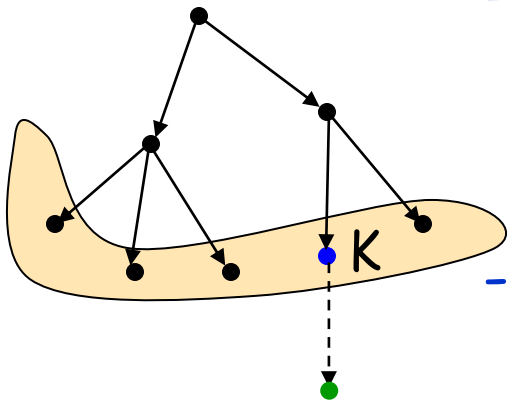
1) If a solution exists, A* terminates and returns a solution



- For each node N on the fringe,
  $f(N) = g(N)+h(N) \geq g(N) \geq d(N) \times \varepsilon$,
  where d(N) is the depth of N in the tree

Proof (1/2)

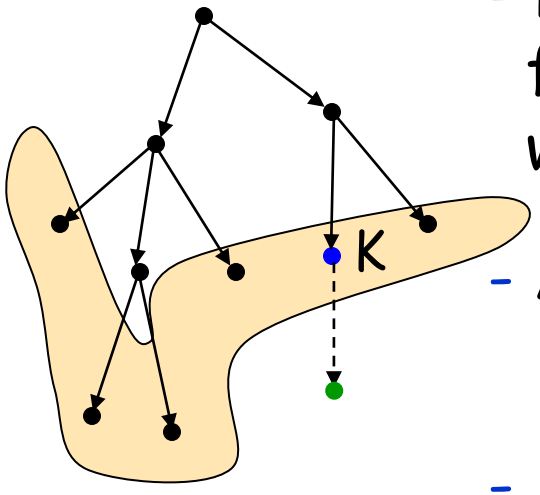
1) If a solution exists, A* terminates and returns a solution



- For each node N on the fringe,
  $f(N) = g(N)+h(N) \geq g(N) \geq d(N)\times\varepsilon$,
  where d(N) is the depth of N in the tree
- As long as A* hasn't terminated, a node K on the fringe lies on a solution path

Proof (1/2)

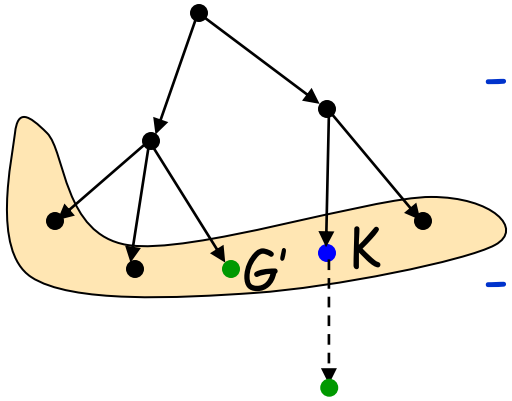
1) If a solution exists, A* terminates and returns a solution



- For each node N on the fringe, $f(N) = g(N)+h(N) \geq g(N) \geq d(N) \times \varepsilon$, where d(N) is the depth of N in the tree
- As long as A* hasn't terminated, a node K on the fringe lies on a solution path
- Since each node expansion increases the length of one path, K will eventually be selected for expansion, unless a solution is found along another path

Proof (2/2)

2) Whenever A* chooses to expand a goal node, the path to this node is optimal



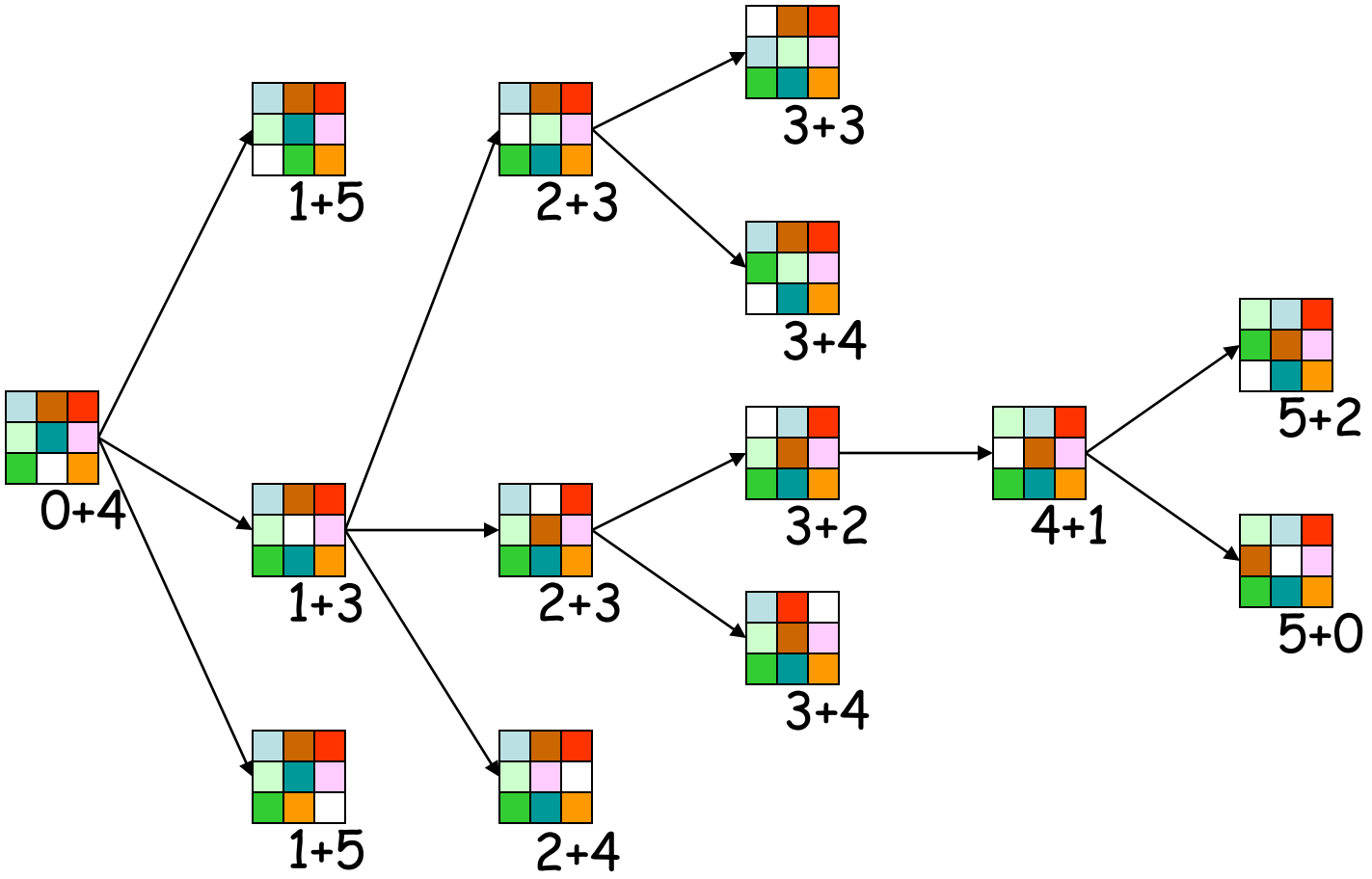
- C^*= cost of the optimal solution path
- G': non-optimal goal node in the fringe
  $f(G') = g(G') + h(G') = g(G') > C^*$
- A node K in the fringe lies on an optimal path:
  $f(K) = g(K) + h(K) \leq C^*$
- So, G' will not be selected for expansion

Time Limit Issue

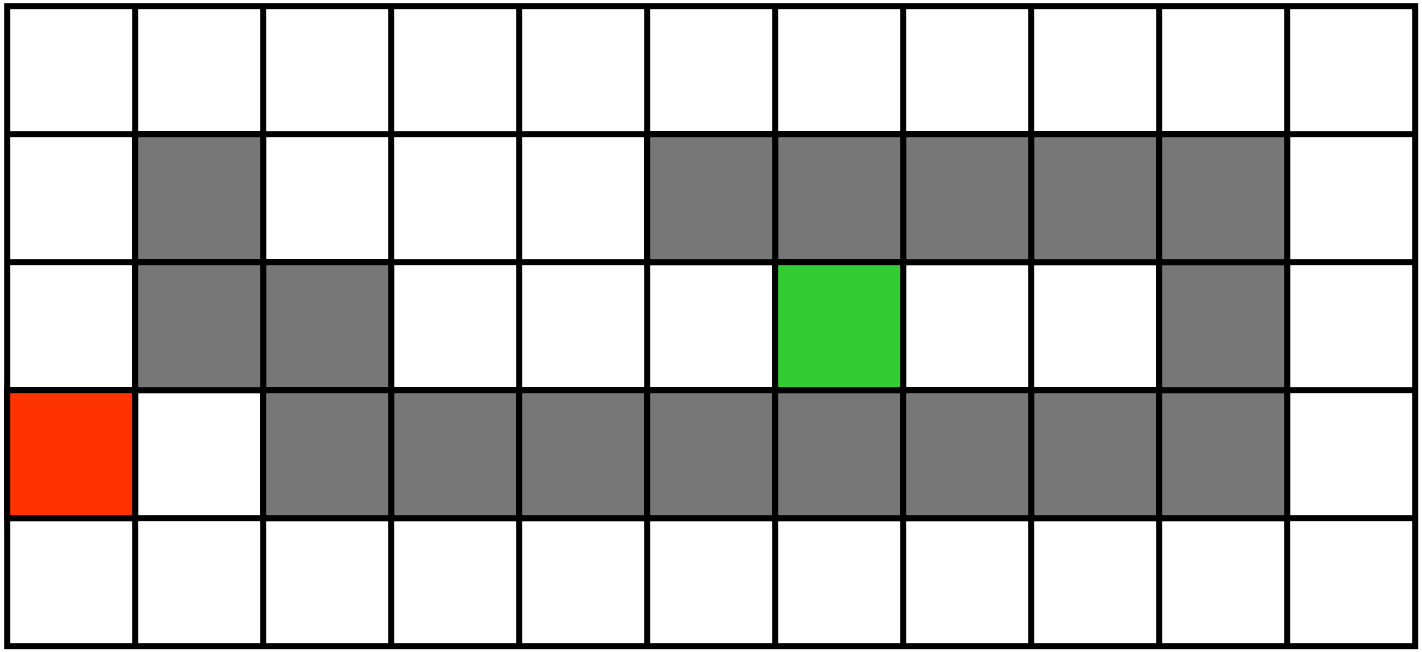
- When a problem has no solution, A* runs for ever if the state space is infinite or states can be revisited an arbitrary number of times. In other cases, it may take a huge amount of time to terminate
- So, in practice, A* is given a time limit. If it has not found a solution within this limit, it stops. Then there is no way to know if the problem has no solution, or if more time was needed to find it
- When AI systems are "small" and solving a single search problem at a time, this is not too much of a concern.
- When AI systems become larger, they solve many search problems concurrently, some with no solution. **What should be the time limit for each of them?**
  More on this in the lecture on Motion Planning ...

8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles



Robot Navigation



Robot Navigation

f(N) = h(N), with h(N) = Manhattan distance to the goal
(not A*)

8	7	6	5	4	3	2	3	4	5	6
7		5	4	3						5
6			3	2	1	0	1	2		4
7	6									5
8	7	6	5	4	3	2	3	4	5	6

Robot Navigation

f(N) = h(N), with h(N) = Manhattan distance to the goal
(not A*)

8	7	6	5	4	3	2	3	4	5	6
7		5	4	3						5
6			3	2	1	0	1	2		4
7	6									5
8	7	6	5	4	3	2	3	4	5	6

Robot Navigation

f(N) = h(N), with h(N) = Manhattan distance to the goal
(not A*)

8	7	6	5	4	3	2	3	4	5	6
7		5	4	3						5
6			3	2	1	0	1	2		4
7	6									5
8	7	6	5	4	3	2	3	4	5	6

Robot Navigation

f(N) = h(N), with h(N) = Manhattan distance to the goal
(not A*)

8	7	6	5	4	3	2	3	4	5	6
7		5	4	3						5
6			3	2	1	0	1	2		4
7	6									5
8	7	6	5	4	3	2	3	4	5	6

Robot Navigation

f(N) = h(N), with h(N) = Manhattan distance to the goal
(not A*)

8	7	6	5	4	3	2	3	4	5	6
7		5	4	3						5
6			3	2	1	0	1	2		4
7	6									5
8	7	6	5	4	3	2	3	4	5	6

Robot Navigation

f(N) = h(N), with h(N) = Manhattan distance to the goal
(not A*)

8	7	6	5	4	3	2	3	4	5	6
7		5	4	3						5
6			3	2	1	0	1	2		4
7	6									5
8	7	6	5	4	3	2	3	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8	7	6	5	4	3	2	3	4	5	6
7		5	4	3						5
6			3	2	1	0	1	2		4
7	6									5
8	7	6	5	4	3	2	3	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8	7	6	5	4	3	2	3	4	5	6
7		5	4	3						5
6			3	2	1	0	1	2		4
7+0	6									5
8	7	6	5	4	3	2	3	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8	7	6	5	4	3	2	3	4	5	6
7		5	4	3						5
6+1			3	2	1	0	1	2		4
7+0	6+1									5
8+1	7	6	5	4	3	2	3	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8	7	6	5	4	3	2	3	4	5	6
7+2		5	4	3						5
6+1			3	2	1	0	1	2		4
7+0	6+1									5
8+1	7	6	5	4	3	2	3	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8	7	6	5	4	3	2	3	4	5	6
7+2		5	4	3						5
6+1			3	2	1	0	1	2		4
7+0	6+1									5
8+1	7+2	6	5	4	3	2	3	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8+3	7	6	5	4	3	2	3	4	5	6
7+2		5	4	3						5
6+1			3	2	1	0	1	2		4
7+0	6+1									5
8+1	7+2	6	5	4	3	2	3	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8+3	7	6	5	4	3	2	3	4	5	6
7+2		5	4	3						5
6+1			3	2	1	0	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5	4	3	2	3	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8+3	7	6	5	4	3	2	3	4	5	6
7+2		5	4	3						5
6+1			3	2	1	0	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4	3	2	3	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8+3	7	6	5	4	3	2	3	4	5	6
7+2		5	4	3						5
6+1			3	2	1	0	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3	2	3	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8+3	7	6	5	4	3	2	3	4	5	6
7+2		5	4	3						5
6+1			3	2	1	0	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3+6	2	3	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8+3	7	6	5	4	3	2	3	4	5	6
7+2		5	4	3						5
6+1			3	2	1	0	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3+6	2+7	3	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8+3	7	6	5	4	3	2	3	4	5	6
7+2		5	4	3						5
6+1			3	2	1	0	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3+6	2+7	3+8	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8+3	7	6	5	4	3	2	3	4	5	6
7+2		5	4	3						5
6+1			3	2	1	0	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3+6	2+7	3+8	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8+3	7+4	6	5	4	3	2	3	4	5	6
7+2		5	4	3						5
6+1			3	2	1	0	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3+6	2+7	3+8	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8+3	7+4	6+5	5	4	3	2	3	4	5	6
7+2		5	4	3						5
6+1			3	2	1	0	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3+6	2+7	3+8	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8+3	7+4	6+3	5+6	4	3	2	3	4	5	6
7+2		5+6	4	3						5
6+1			3	2	1	0	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3+6	2+7	3+8	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8+3	7+4	6+3	5+6	4+7	3	2	3	4	5	6
7+2		5+6	4+7	3						5
6+1			3	2	1	0	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3+6	2+7	3+8	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8+3	7+4	6+3	5+6	4+7	3+8	2	3	4	5	6
7+2		5+6	4+7	3+8						5
6+1			3	2	1	0	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3+6	2+7	3+8	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8+3	7+4	6+3	5+6	4+7	3+8	2+9	3	4	5	6
7+2		5+6	4+7	3+8						5
6+1			3	2	1	0	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3+6	2+7	3+8	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8+3	7+4	6+3	5+6	4+7	3+8	2+9	3+10	4	5	6
7+2		5+6	4+7	3+8						5
6+1			3	2	1	0	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3+6	2+7	3+8	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8+3	7+4	6+3	5+6	4+7	3+8	2+9	3+10	4	5	6
7+2		5+6	4+7	3+8						5
6+1			3	2+9	1	0	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3+6	2+7	3+8	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8+3	7+4	6+3	5+6	4+7	3+8	2+9	3+10	4	5	6
7+2		5+6	4+7	3+8						5
6+1			3	2+9	1+10	0	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3+6	2+7	3+8	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8+3	7+4	6+3	5+6	4+7	3+8	2+9	3+10	4	5	6
7+2		5+6	4+7	3+8						5
6+1			3	2+9	1+10	0+11	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3+6	2+7	3+8	4	5	6

Robot Navigation

f(N) = g(N)+h(N), with h(N) = Manhattan distance to goal (A*)

8+3	7+4	6+3	5+6	4+7	3+8	2+9	3+10	4	5	6
7+2		5+6	4+7	3+8						5
6+1			3	2+9	1+10	0+11	1	2		4
7+0	6+1									5
8+1	7+2	6+3	5+4	4+5	3+6	2+7	3+8	4	5	6

Best-First Search

- An evaluation function f maps each node N of the search tree to a real number $f(N) \geq 0$
- Best-first search sorts the FRINGE in increasing f

A* Search

1) f(N) = g(N) + h(N), where:
   - g(N) = cost of best path found so far to N
   - h(N) = **admissible** heuristic function
2) for all arcs: $c(N,N') \geq \varepsilon > 0$
3) SEARCH#2 algorithm is used

➔ Best-first search is then called A* search

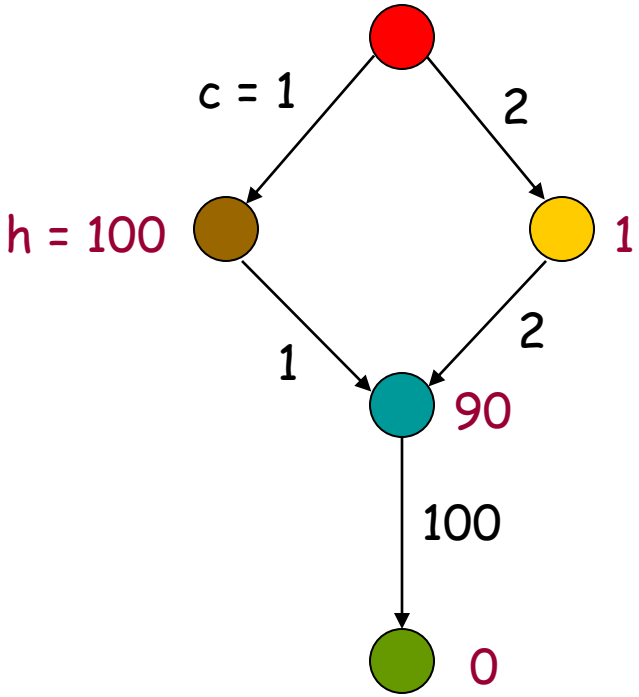
Result #1

A* is complete and optimal

[This result holds if nodes revisiting states are not discarded]

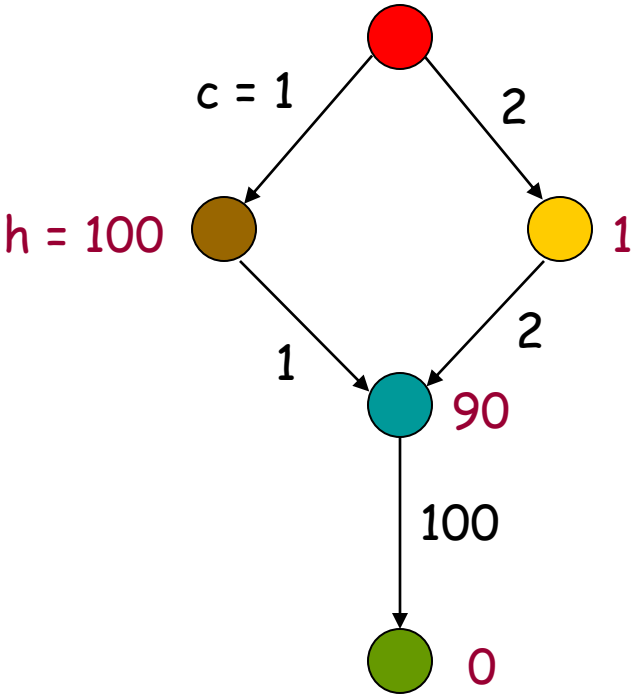
[TREE-SEARCH version]

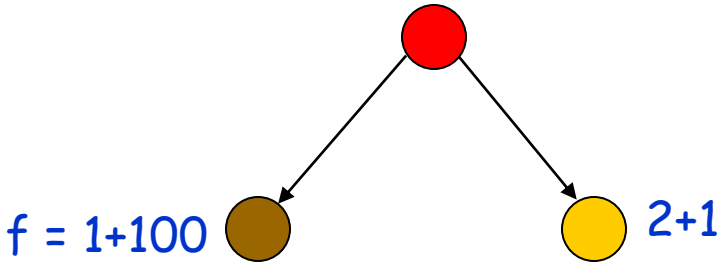
What to do with revisited states?



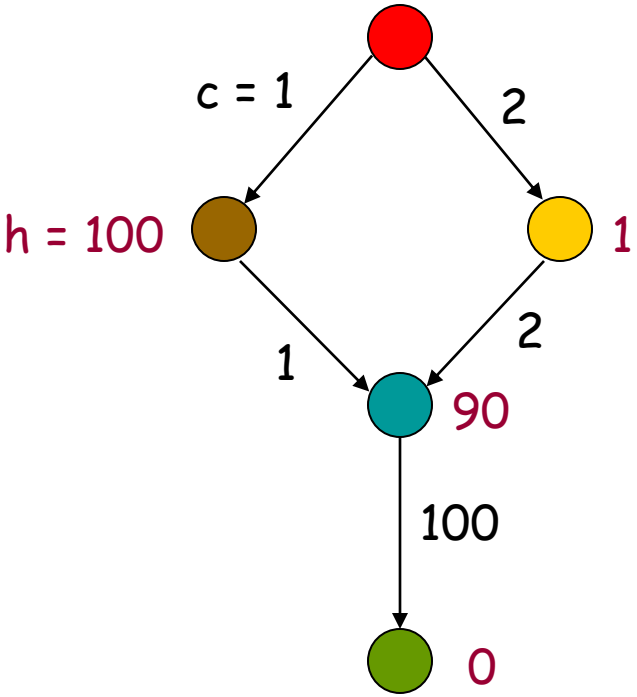
The heuristic h is clearly admissible

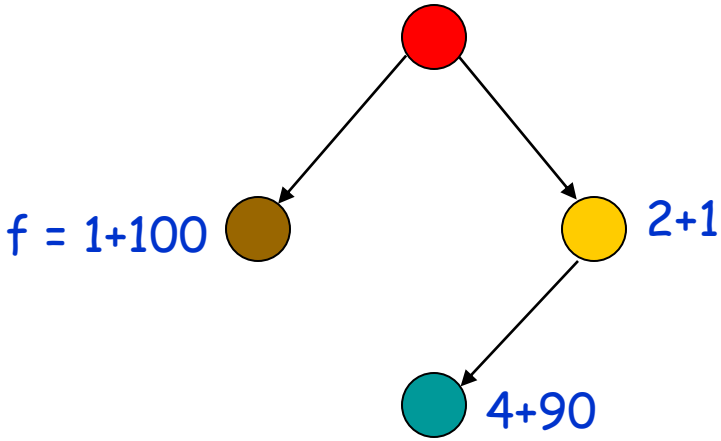
What to do with revisited states?



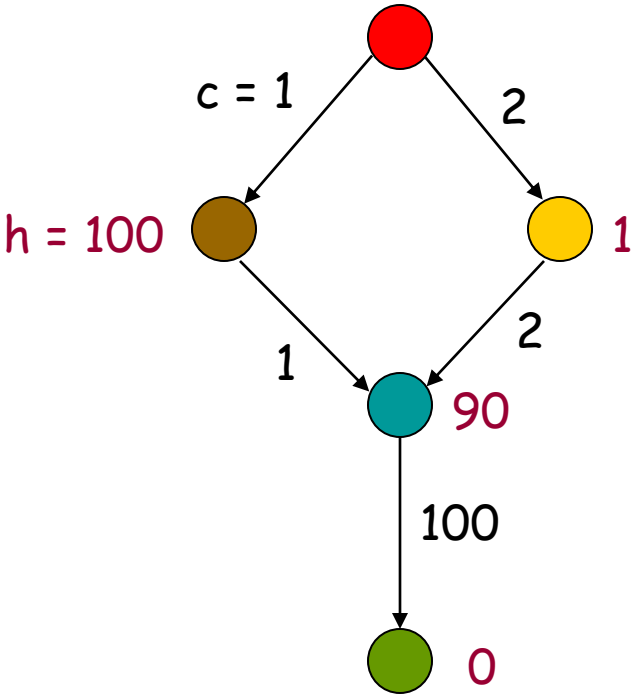


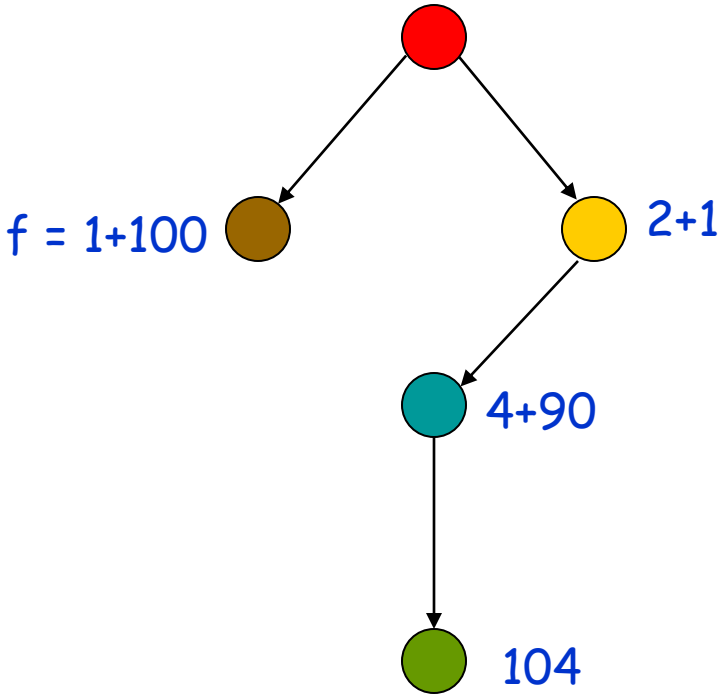
What to do with revisited states?



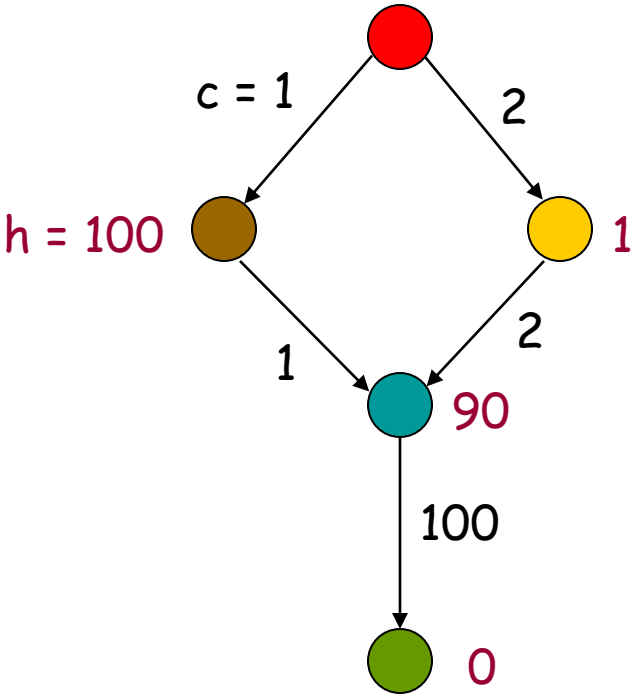


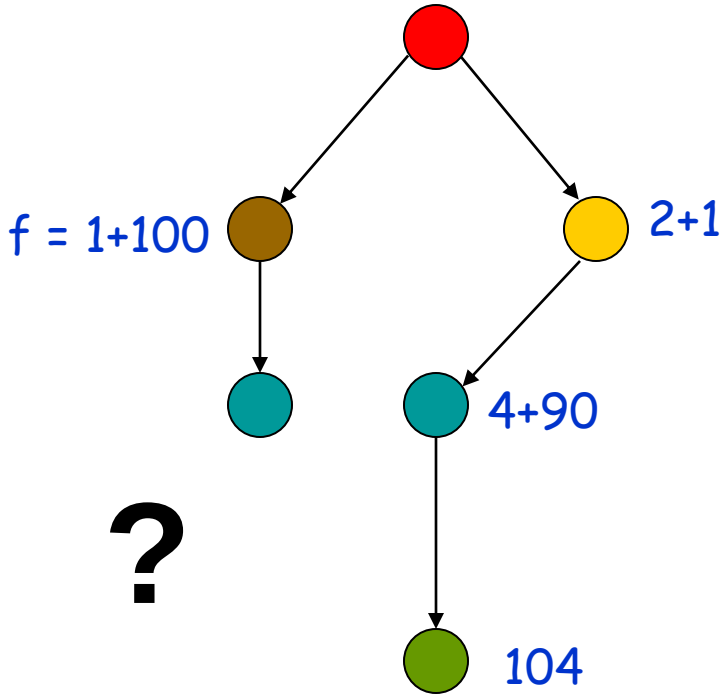
What to do with revisited states?



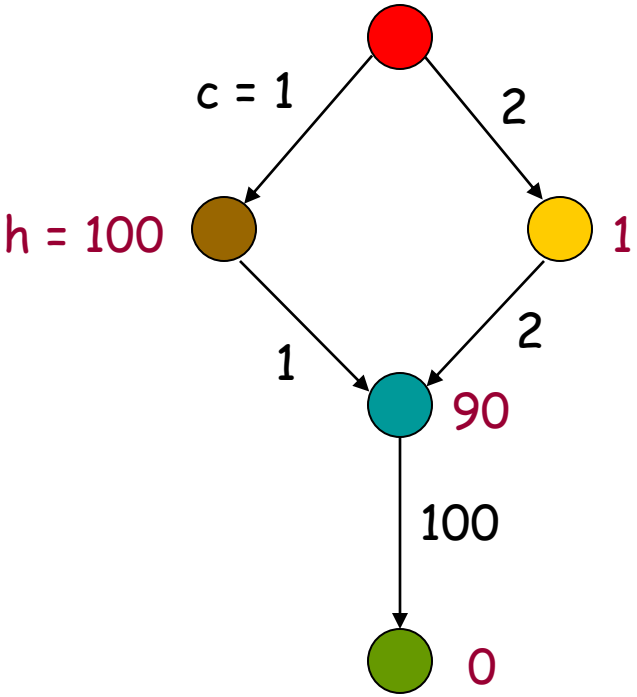


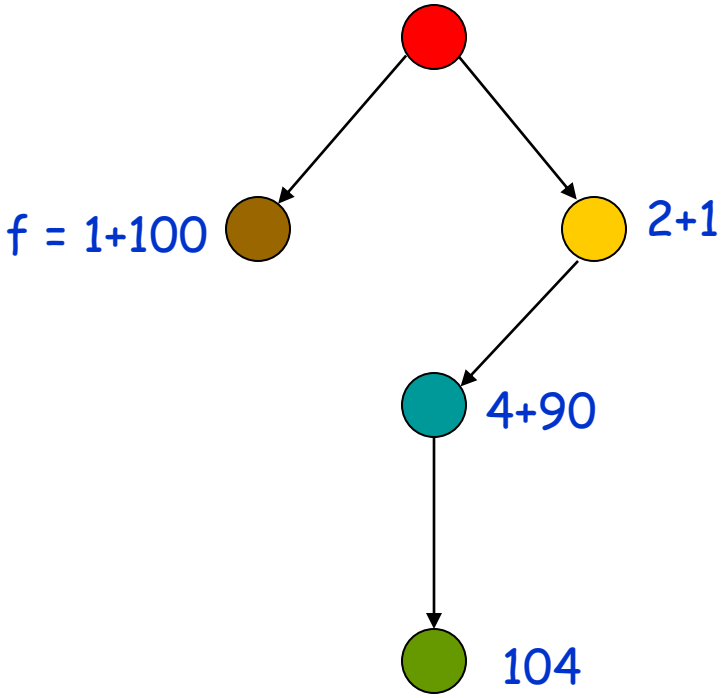
What to do with revisited states?





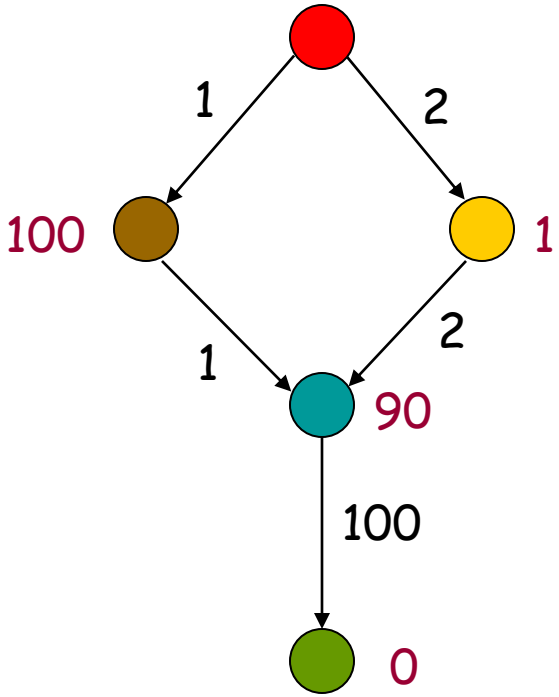
What to do with revisited states?

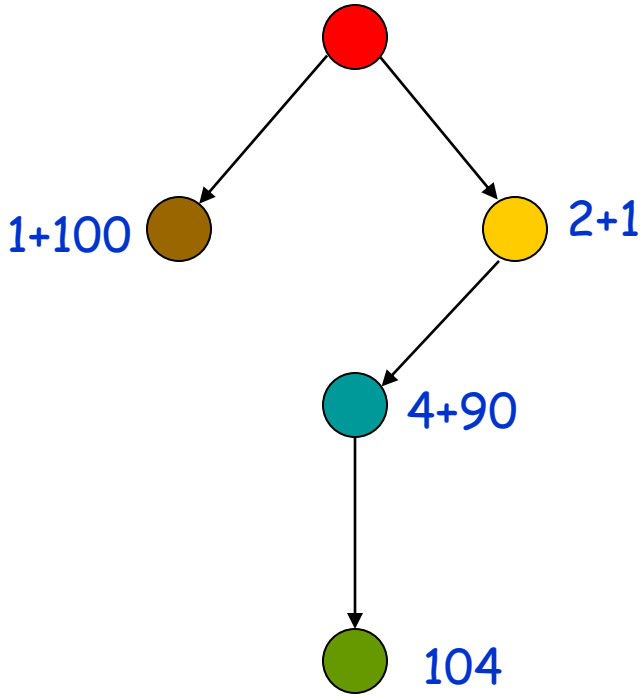




If we discard this new node, then the search algorithm expands the goal node next and returns a non-optimal solution

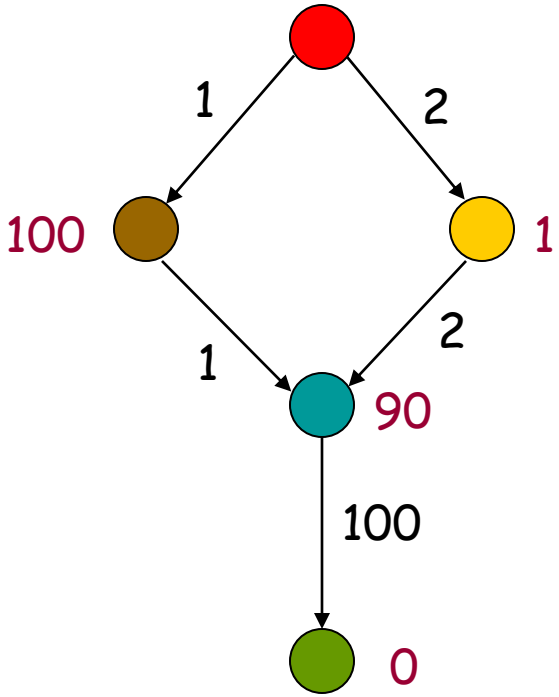
What to do with revisited states?

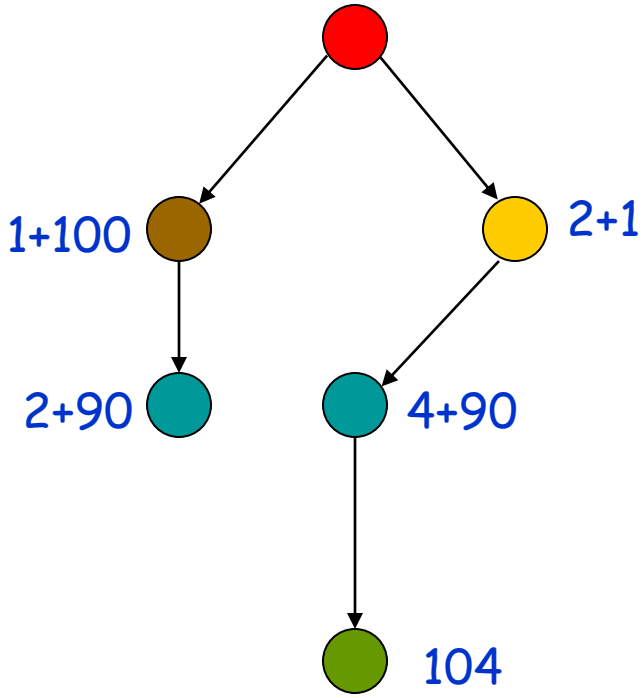




Instead, if we do not discard nodes revisiting states, the search terminates with an optimal solution

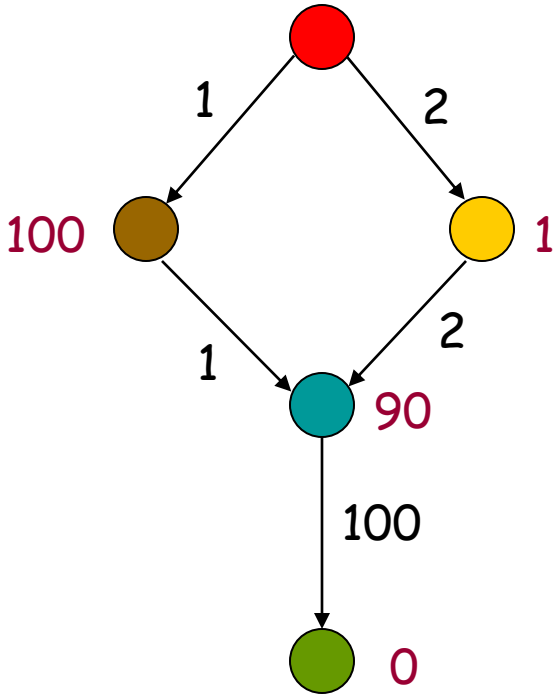
What to do with revisited states?

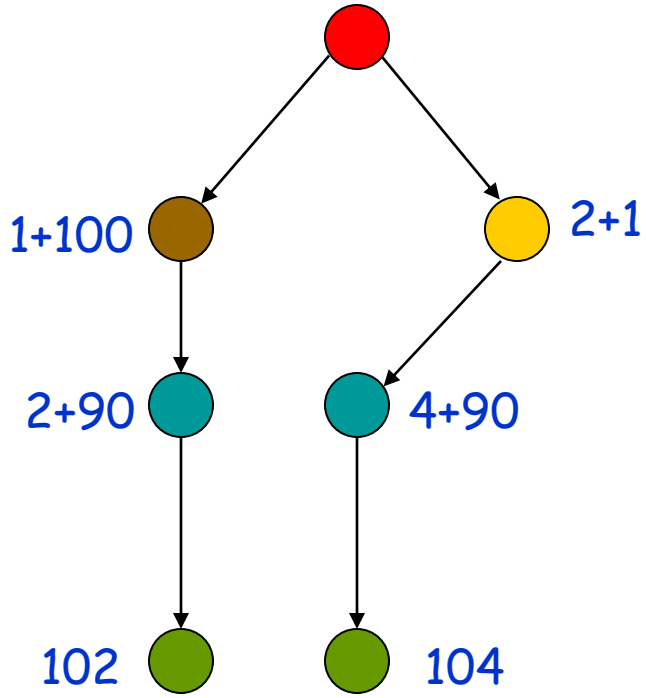




Instead, if we do not discard nodes revisiting states, the search terminates with an optimal solution

What to do with revisited states?

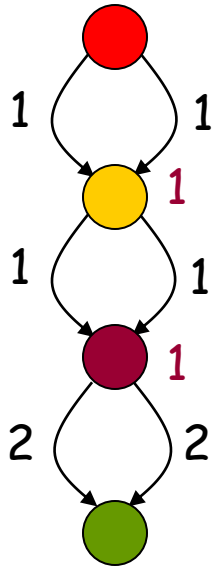




Instead, if we do not discard nodes revisiting states, the search terminates with an optimal solution

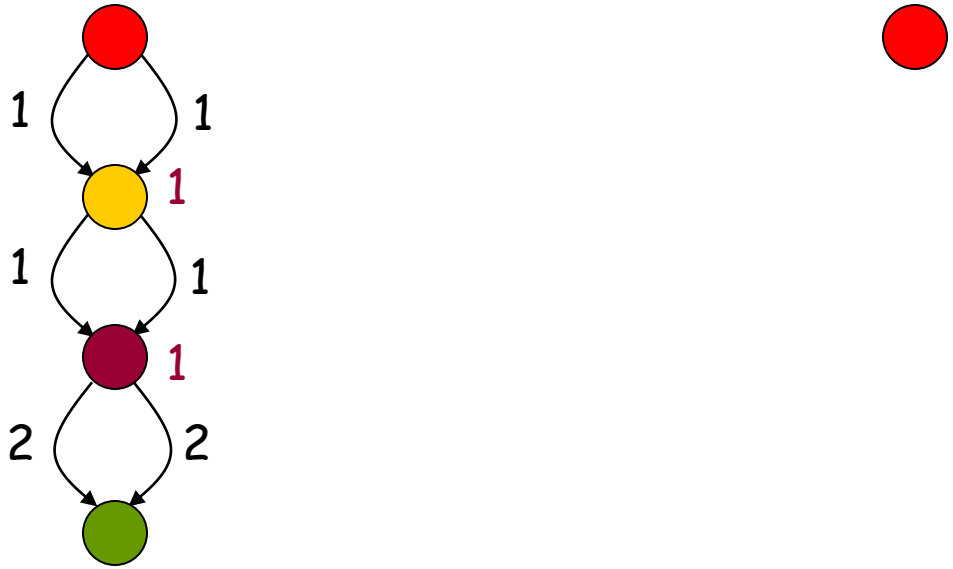
But ...

If we do not discard nodes revisiting states, the size of the search tree can be exponential in the number of visited states



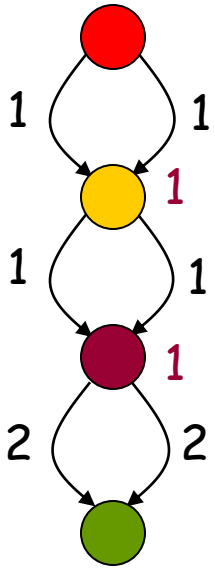
But ...

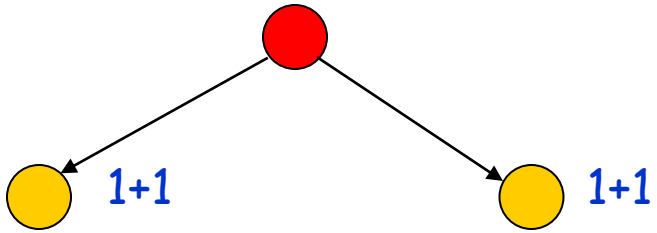
If we do not discard nodes revisiting states, the size of the search tree can be exponential in the number of visited states



But ...

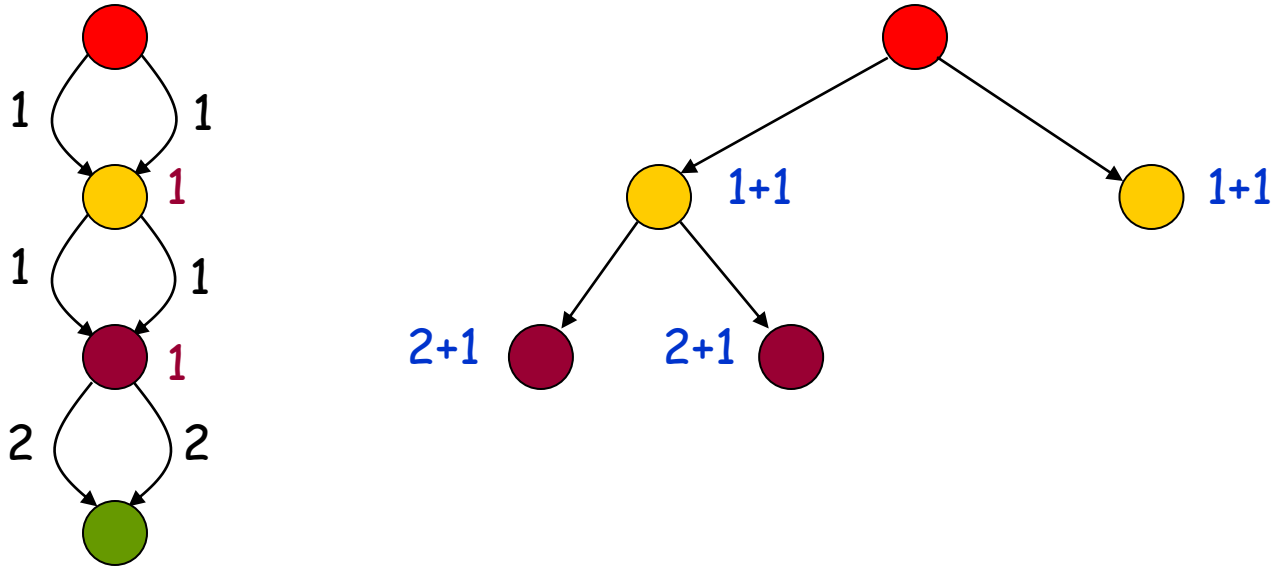
If we do not discard nodes revisiting states, the size of the search tree can be exponential in the number of visited states





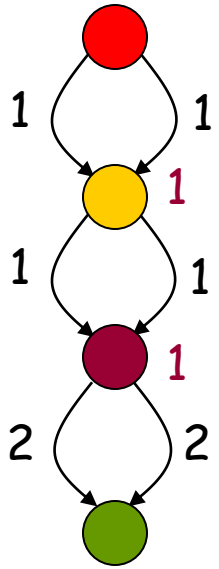
But ...

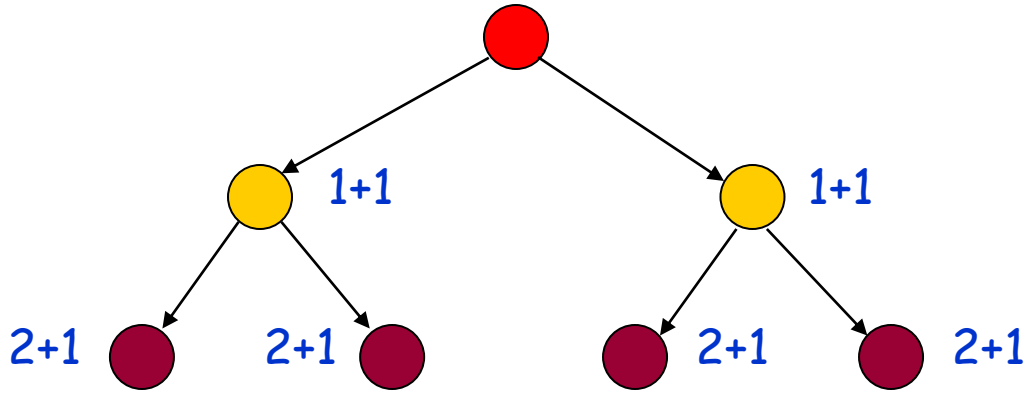
If we do not discard nodes revisiting states, the size of the search tree can be exponential in the number of visited states



But ...

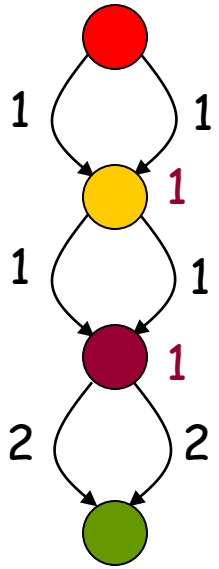
If we do not discard nodes revisiting states, the size of the search tree can be exponential in the number of visited states

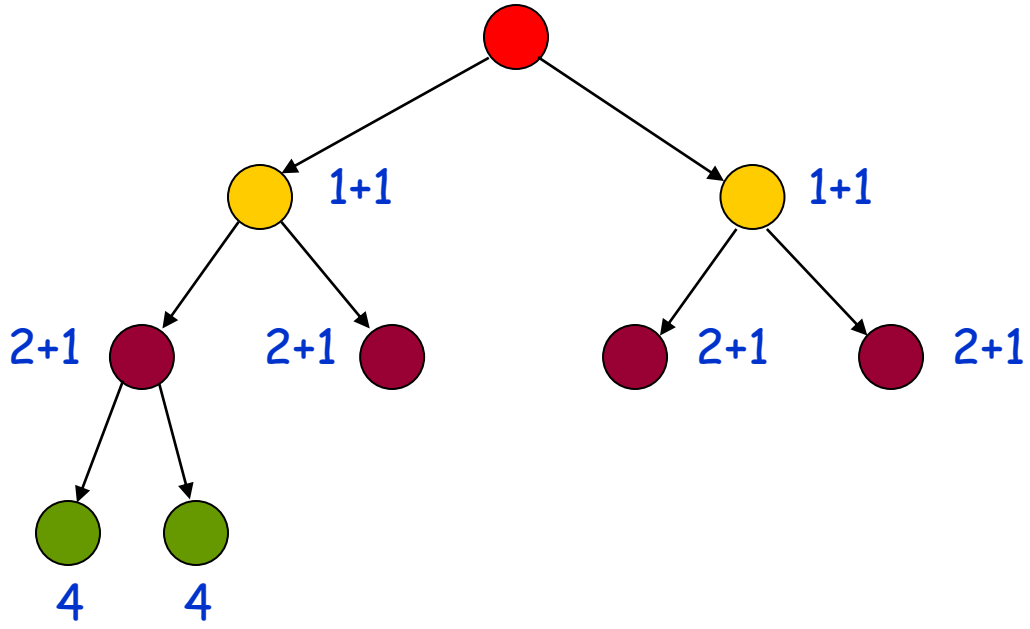




But ...

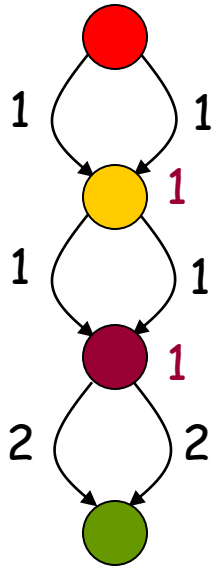
If we do not discard nodes revisiting states, the size of the search tree can be exponential in the number of visited states

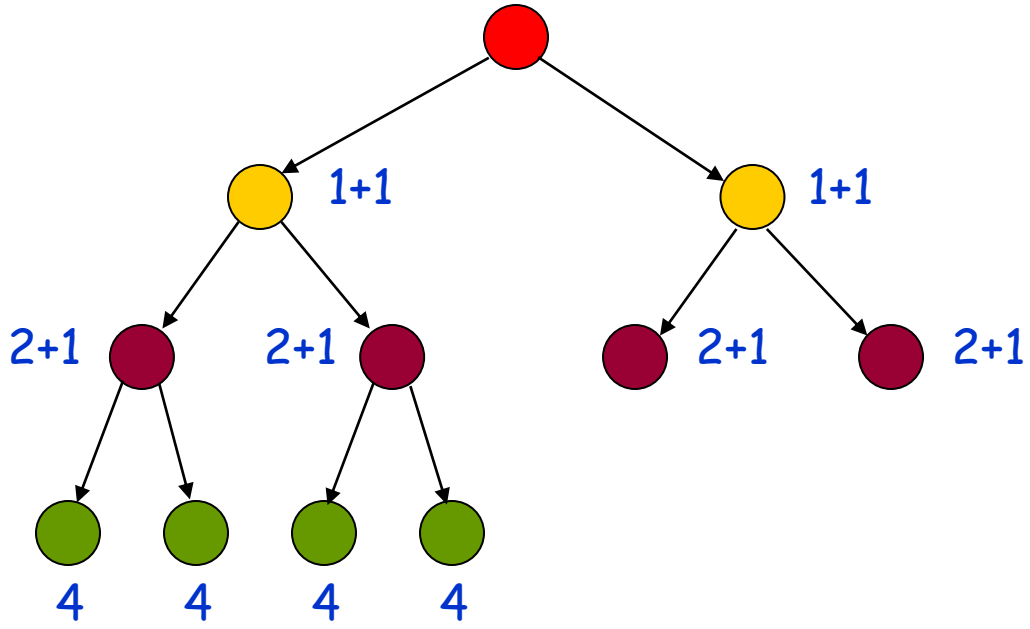




But ...

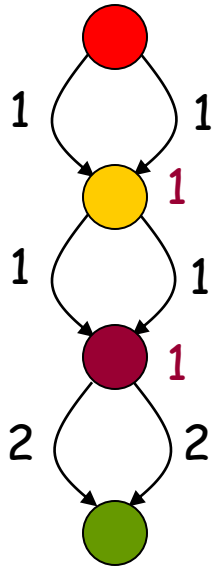
If we do not discard nodes revisiting states, the size of the search tree can be exponential in the number of visited states

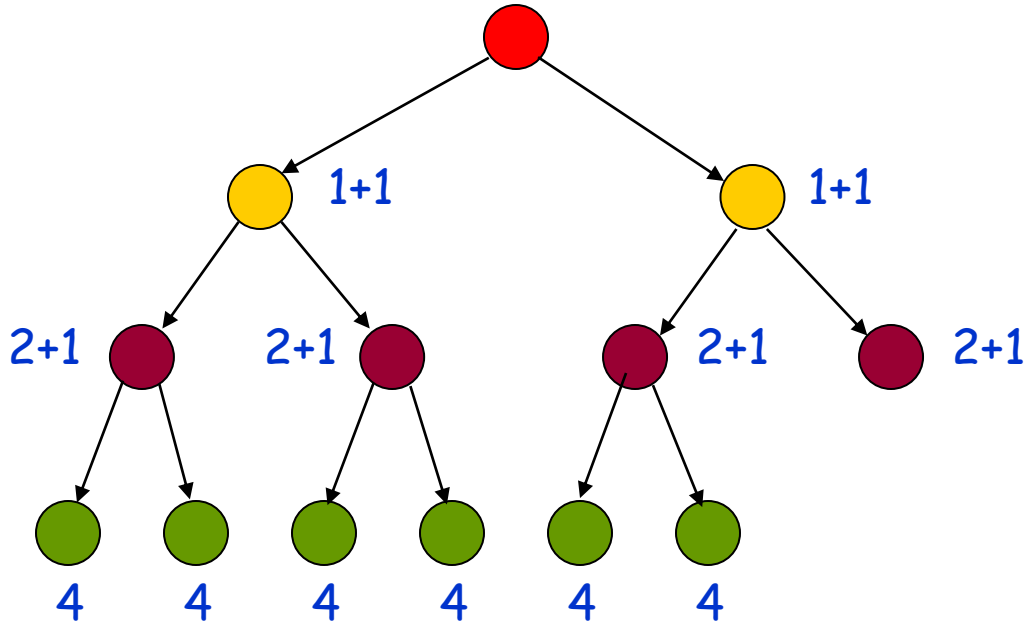




But ...

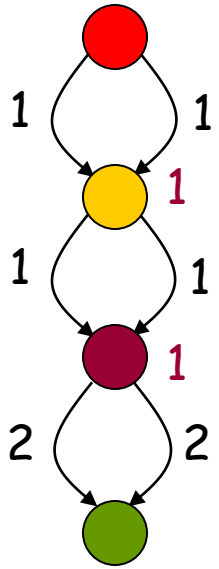
If we do not discard nodes revisiting states, the size of the search tree can be exponential in the number of visited states

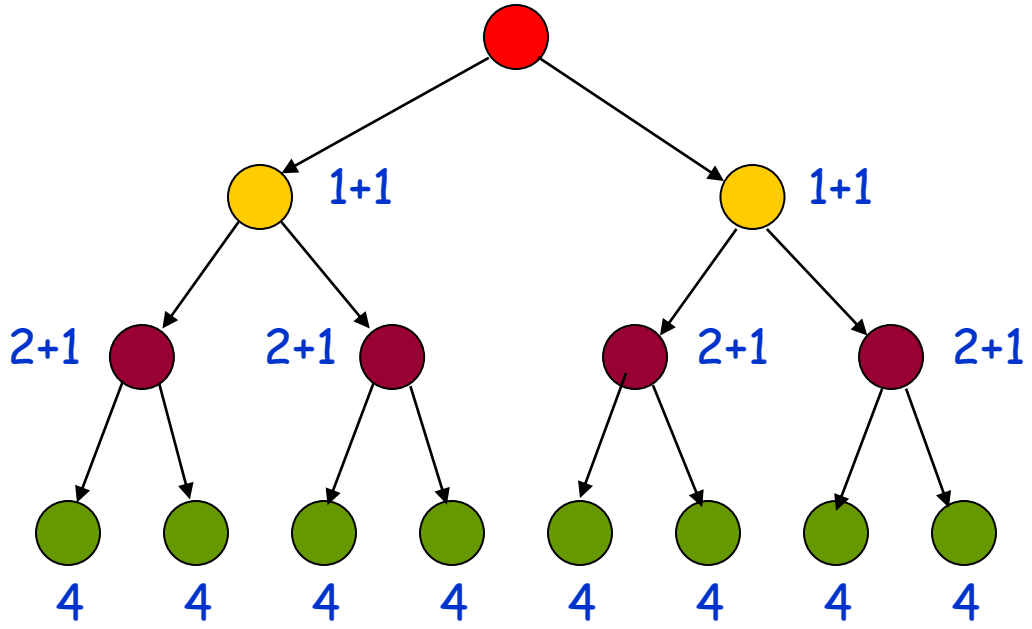




But ...

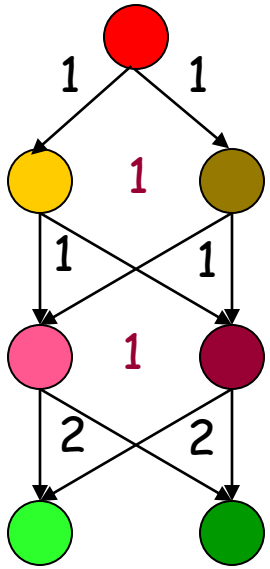
If we do not discard nodes revisiting states, the size of the search tree can be exponential in the number of visited states



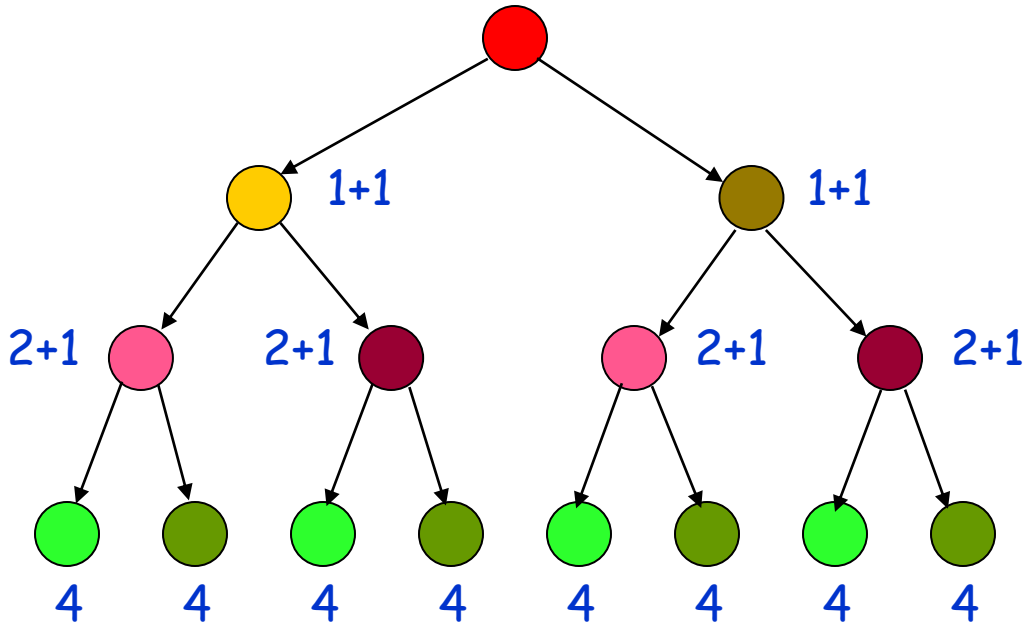


But ...

If we do not discard nodes revisiting states, the size of the search tree can be exponential in the number of visited states



2n+1 states



$O(2^n)$ nodes

- It is not harmful to discard a node revisiting a state if the cost of the new path to this state is ≥ cost of the previous path
  [so, in particular, one can discard a node if it re-visits a state already visited by one of its ancestors]

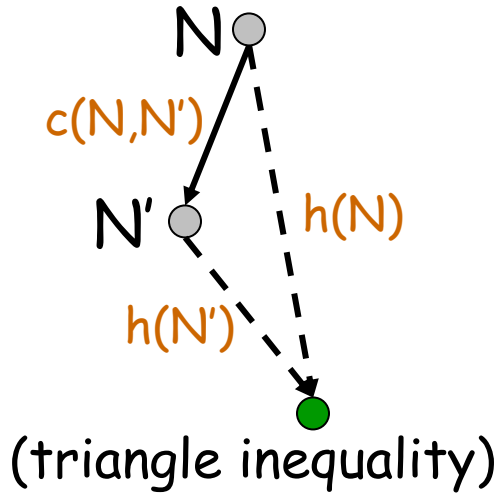
- A* remains optimal, but states can still be re-visited multiple times
  [the size of the search tree can still be exponential in the number of visited states]

- Fortunately, for a large family of admissible heuristics – consistent heuristics – there is a much more efficient way to handle revisited states

Consistent Heuristic

An admissible heuristic h is consistent (or monotone) if for each node N and each child N' of N:

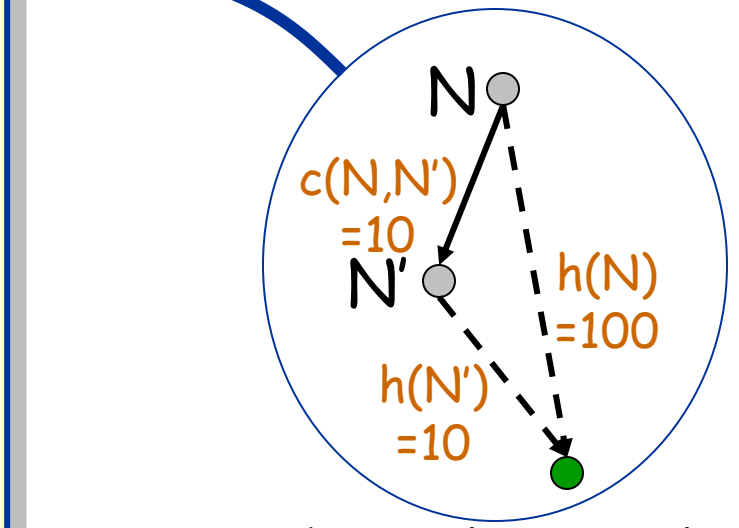
$$h(N) \leq c(N,N') + h(N')$$



→ Intuition: a consistent heuristics becomes more precise as we get deeper in the search tree

Consistency Violation

If h tells that N is 100 units from the goal, then moving from N along an arc costing 10 units should **not** lead to a node N' that h estimates to be 10 units away from the goal



(triangle inequality)

Consistent Heuristic
(alternative definition)

A heuristic h is consistent (or monotone) if

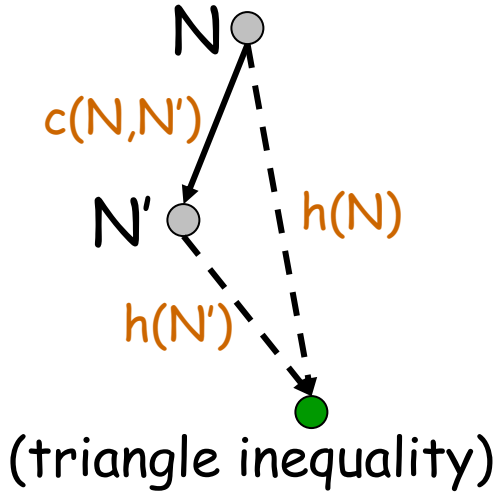
1) for each node N and each child N′ of N:

$h(N) \leq c(N,N') + h(N')$

2) for each goal node G:

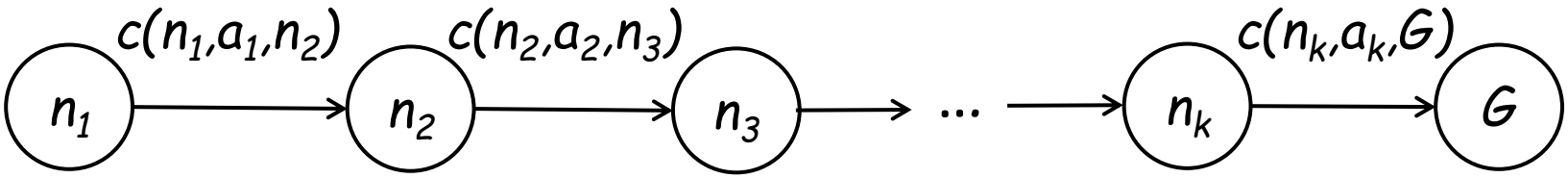
$h(G) = 0$

A consistent heuristic is also admissible



Admissibility and Consistency

- A consistent heuristic is also admissible



$$h(n_1) \le c(n_1,a_1,n_2) + h(n_2)$$
$$\le c(n_1,a_1,n_2) + c(n_2,a_2,n_3) + h(n_3)$$
$$\ldots$$
$$\le c(n_1,a_1,n_2) + \ldots + c(n_k,a_k,G) + h(G)$$
$$\le c(n_1,a_1,n_2) + \ldots + c(n_k,a_k,G)$$
$$\le \text{cost of each path from } n_1 \text{ to goal}$$

Admissibility and Consistency

- A consistent heuristic is also admissible
- An admissible heuristic may not be consistent, but many admissible heuristics are consistent

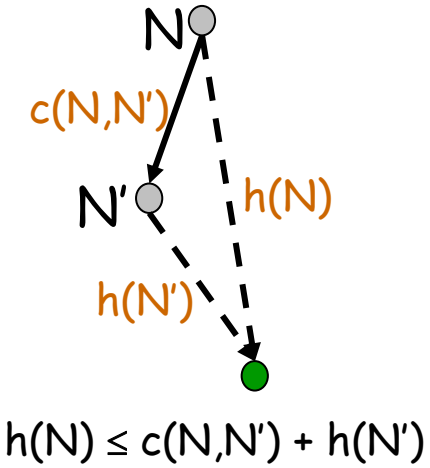
8-Puzzle

5		8
4	2	1
7	3	6

STATE(N)

1	2	3
4	5	6
7	8	

goal

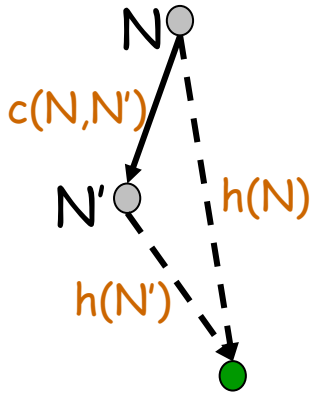


$h(N) \leq c(N,N') + h(N')$

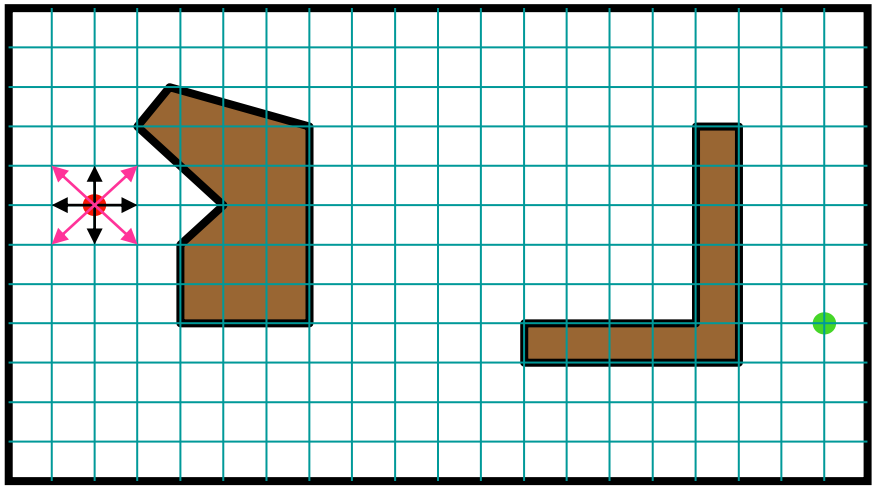
- $h_1(N)$ = number of misplaced tiles
- $h_2(N)$ = sum of the (Manhattan) distances of every tile to its goal position

are both consistent (why?)

Robot Navigation



$h(N) \leq c(N,N') + h(N')$



Cost of one horizontal/vertical step = 1
Cost of one diagonal step = $\sqrt{2}$

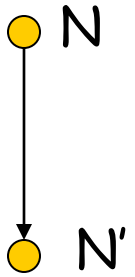
$h_1(N) = \sqrt{(x_N - x_g)^2 + (y_N - y_g)^2}$ is consistent

$h_2(N) = |x_N - x_g| + |y_N - y_g|$ is consistent if moving along diagonals is not allowed, and not consistent otherwise

Result #2

- If h is consistent, then whenever A* expands a node, it has already found an optimal path to this node's state
- (In other words If h is consistent, A* using GRAPH-SEARCH is optimal)

Proof (1/2)



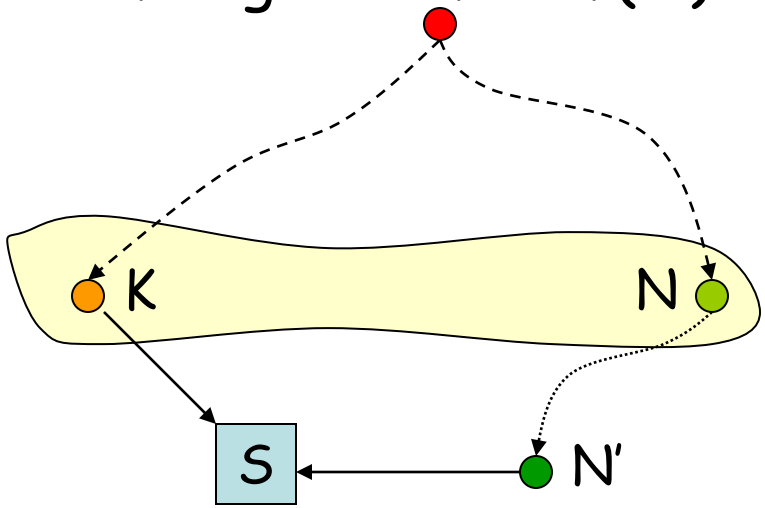
1) Consider a node N and its child N'
Since h is consistent: $h(N) \leq c(N,N')+h(N')$

$f(N) = g(N)+h(N) \leq g(N)+c(N,N')+h(N') = f(N')$

So, f is non-decreasing along any path

Proof (2/2)

2) If a node K is selected for expansion, then any other node N in the fringe verifies $f(N) \geq f(K)$



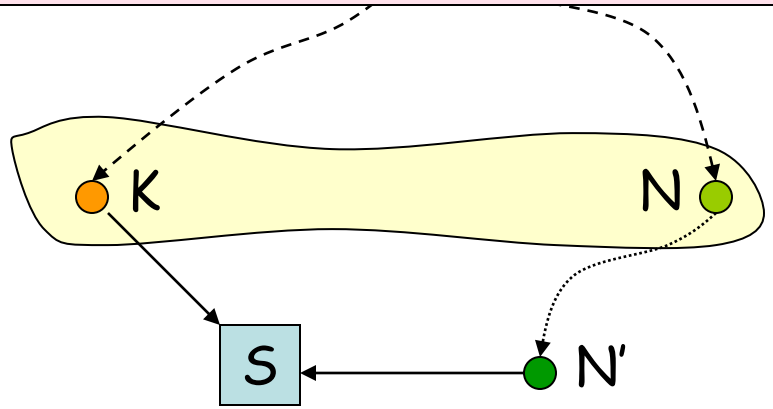
If one node N lies on another path to the state of K, the cost of this other path is no smaller than that of the path to K:

$f(N') \geq f(N) \geq f(K)$ and $h(N') = h(K)$

So, $g(N') \geq g(K)$

Result #2

If h is consistent, then whenever A* expands a node, it has already found an optimal path to this node's state

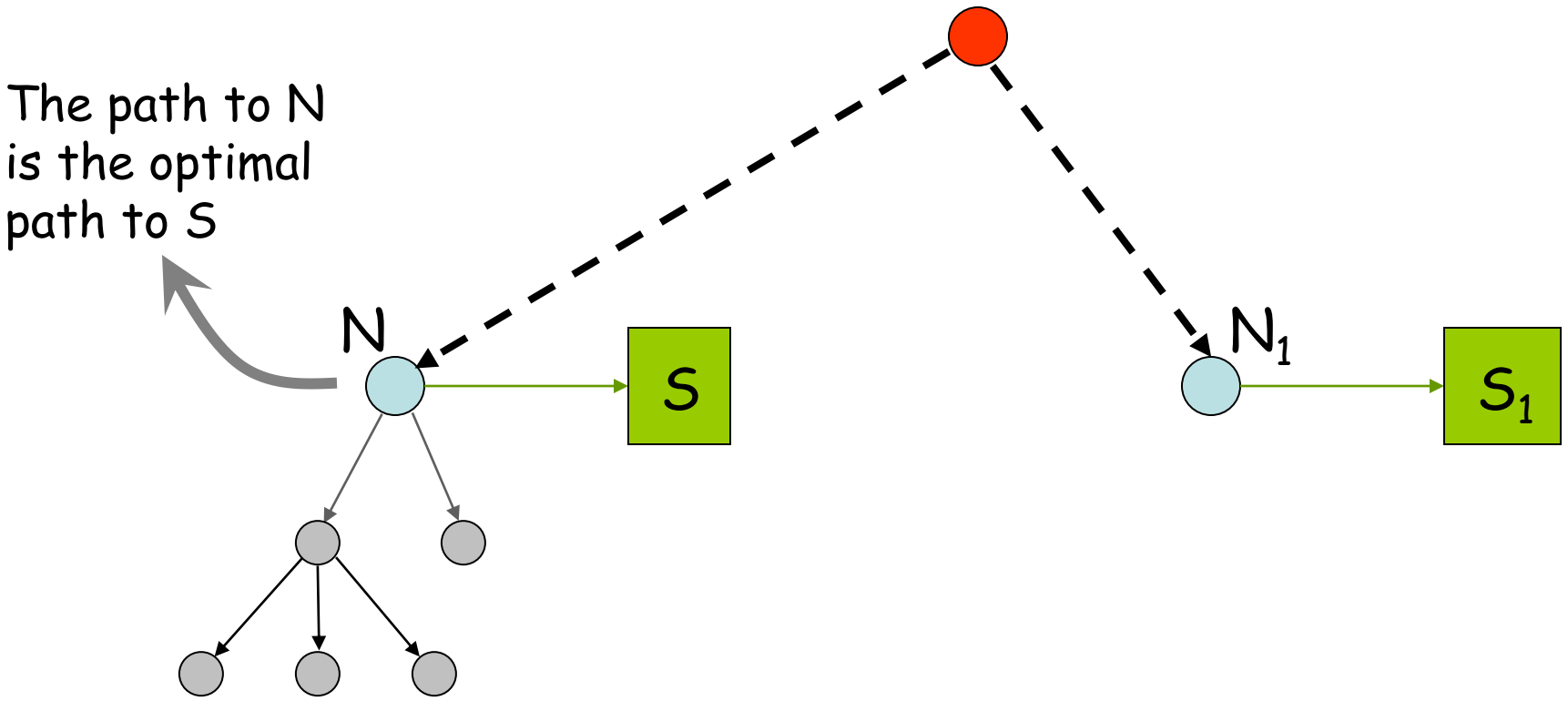


If one node N lies on another path to the state of K, the cost of this other path is no smaller than that of the path to K:

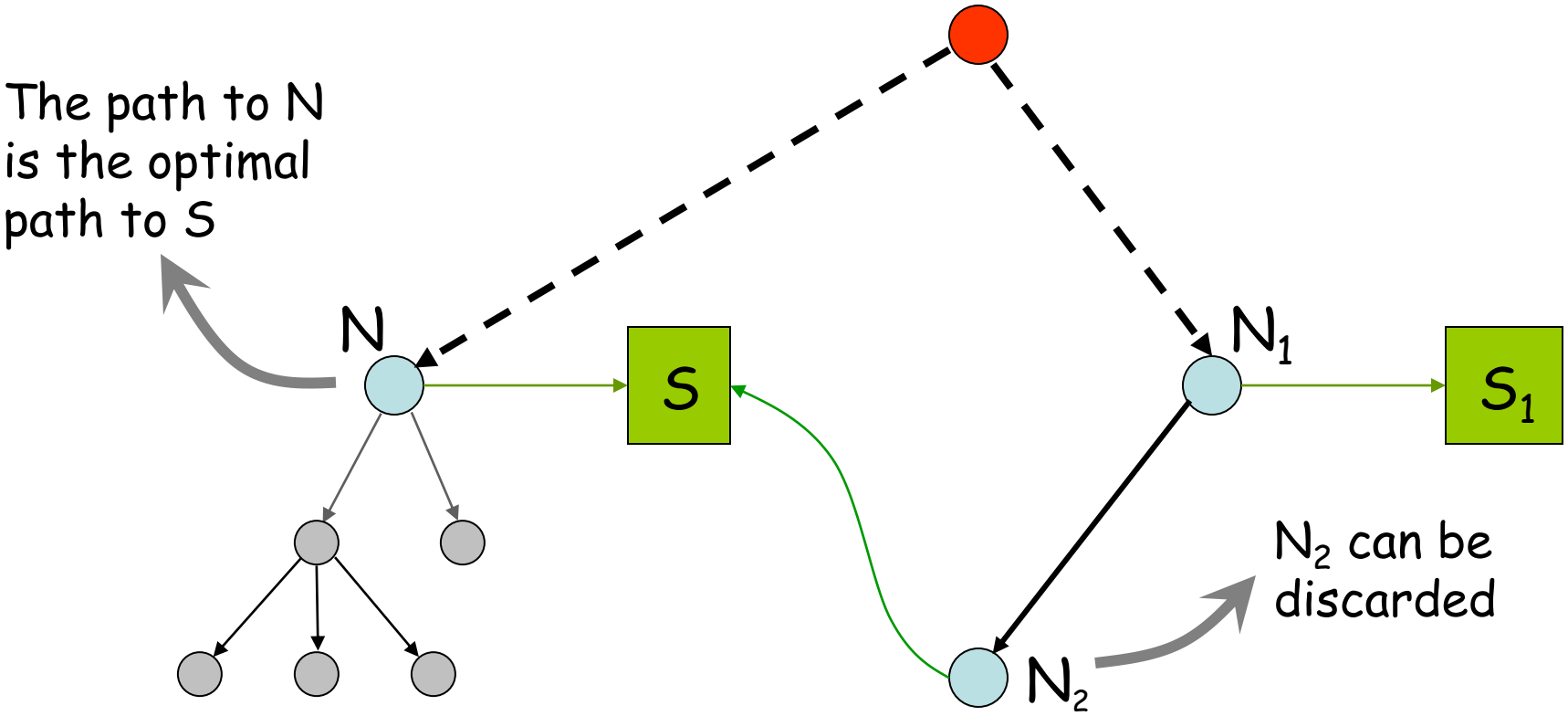
$f(N') \geq f(N) \geq f(K)$ and $h(N') = h(K)$

So, $g(N') \geq g(K)$

Implication of Result #2



Implication of Result #2

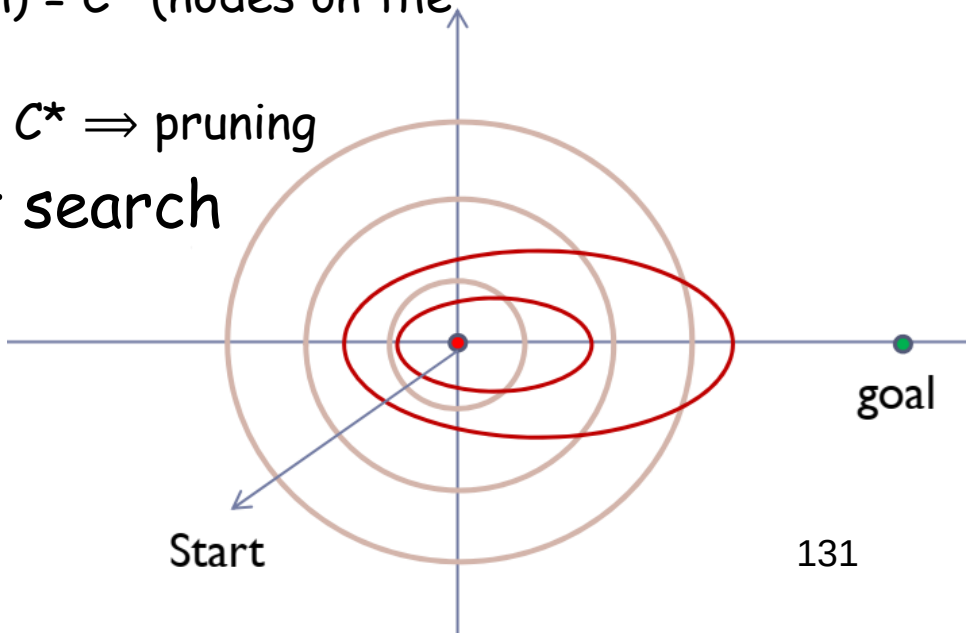


Revisited States with Consistent Heuristic

- When a node is expanded, store its state into CLOSED (Explored)
- When a new node N is generated:
  - If STATE(N) is in CLOSED (Explored), discard N
  - If there exists a node N' in the fringe such that STATE(N') = STATE(N), discard the node – N or N' – with the largest f (or, equivalently, g)

Contours in the state space

- A* (using GRAPH-SEARCH) expands nodes in order of increasing f value
- Gradually adds "f-contours" of nodes
  - Contour i has all nodes with $f = f_i$ where $f_i < f_{i+1}$
  - A* expands all nodes with $f(n) < C^*$
  - A* expands some nodes with $f(n) = C^*$ (nodes on the goal contour)
  - A* expands no nodes with $f(n) > C^* \Longrightarrow$ pruning
- A* search vs. uniform cost search



Properties of A*

- Complete? Yes
  - If nodes with $f \leq f(G) = C^*$ are finite and step cost$\geq \varepsilon > 0$ and b is finite
- Optimal? Yes
- Time?
  - Exponential
    - But, with a smaller branching factor (depends on $h^* - h$)
  - Polynomial when $|h(x) - h^*(x)| \leq O(\log h^*(x))$
  - A* is optimally efficient for any given consistent heuristic
    - No optimal algorithm of this type is guaranteed to expand fewer nodes than A*
- Space?
  - Keeps all leaf and/or explored nodes in memory

Is A* with some consistent heuristic all that we need?

No !

There are **very dumb** consistent heuristic functions

For example: $h \equiv 0$

- It is consistent (hence, admissible) !
- *A** with $h \equiv 0$ is uniform-cost search
- Breadth-first and uniform-cost are particular cases of *A**

Heuristic Accuracy

Let h_1 and h_2 be two consistent heuristics such that for all nodes N:

$$h_1(N) \leq h_2(N)$$

h_2 is said to be more accurate (or more informed) than h_1

5		8
4	2	1
7	3	6

STATE(N)

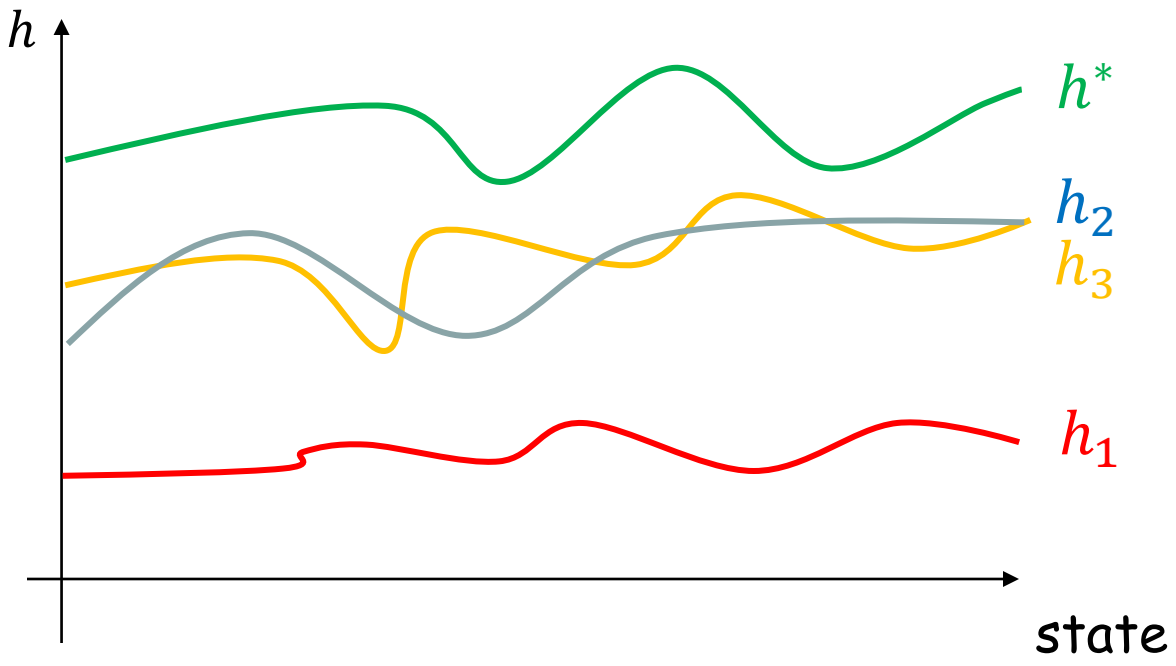
1	2	3
4	5	6
7	8	

Goal state

- $h_1(N)$ = number of misplaced tiles
- $h_2(N)$ = sum of distances of every tile to its goal position

- h_2 is more accurate than h_1

Heuristic Accuracy



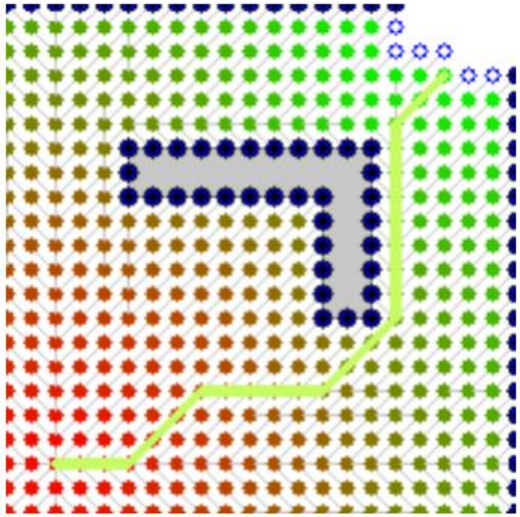
Result #3

- Let h_2 be more accurate than h_1
- Let A_1^* be A^* using h_1
  and A_2^* be A^* using h_2
- Whenever a solution exists, all the nodes expanded by A_2^*, except possibly for some nodes such that
  $f_1(N) = f_2(N) = C^*$ (cost of optimal solution)
  are also expanded by A_1^*

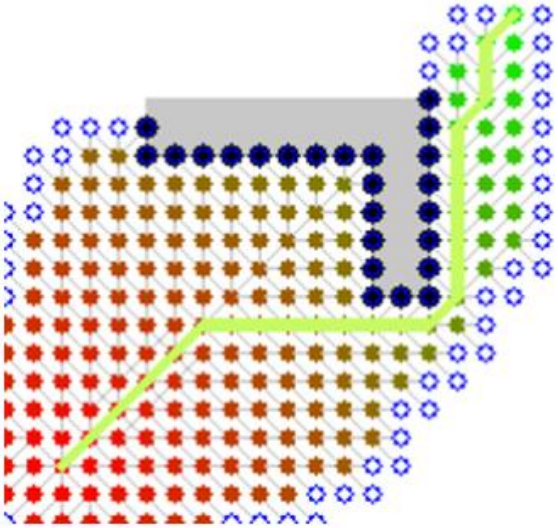
Proof

- C^* = h*(initial-node) [cost of optimal solution]
- Every node N such that $f(N) < C^*$ is eventually expanded. No node N such that $f(N) > C^*$ is ever expanded
- Every node N such that $h(N) < C^* - g(N)$ is eventually expanded. So, every node N such that $h_2(N) < C^* - g(N)$ is expanded by A_2^*. Since $h_1(N) \leq h_2(N)$, N is also expanded by A_1^*
- If there are several nodes N such that $f_1(N) = f_2(N) = C^*$ (such nodes include the optimal goal nodes, if there exists a solution), A_1^* and A_2^* may or may not expand them in the same order (until one goal node is expanded)

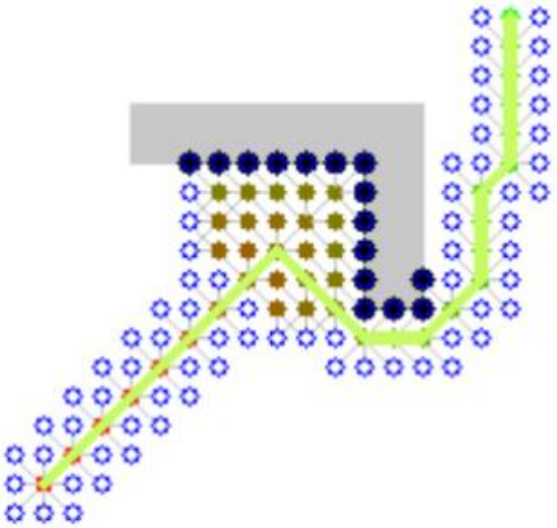
A* vs. UCS: Robot navigation example



$g(n) < C^*$



$g(n)+h(n) < C^*$



$g(n)+5h(n) < C^*$

Effective Branching Factor

- It is used as a measure the effectiveness of a heuristic
- Let n be the total number of nodes expanded by A* for a particular problem and d the depth of the solution
- The effective branching factor b* is defined by $n = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d$

Experimental Results

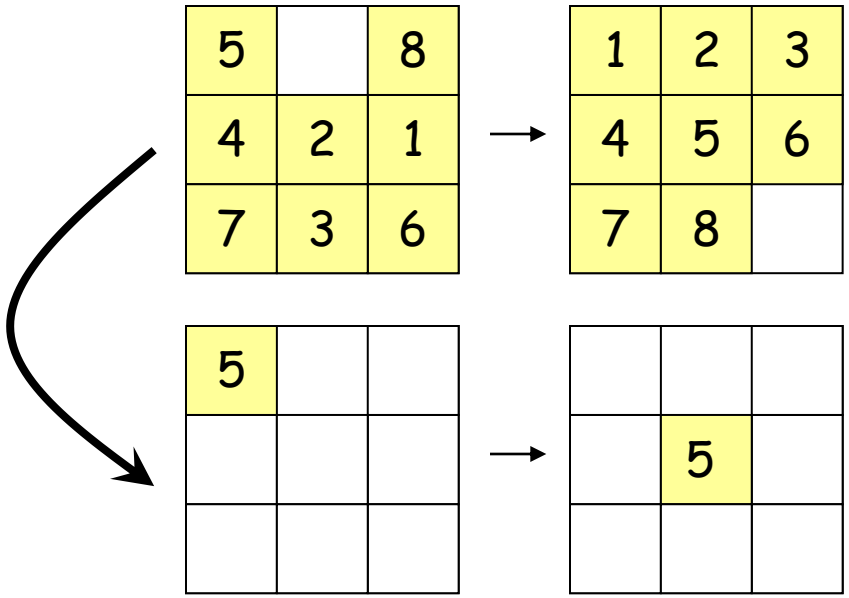
(see R&N for details)

- 8-puzzle with:
  - h_1 = number of misplaced tiles
  - h_2 = sum of distances of tiles to their goal positions
- Random generation of many problem instances
- Average effective branching factors (number of expanded nodes):

d	IDS	A_1^*	A_2^*
2	2.45	1.79	1.79
6	2.73	1.34	1.30
12	2.78 (3,644,035)	1.42 (227)	1.24 (73)
16	--	1.45	1.25
20	--	1.47	1.27
24	--	1.48 (39,135)	1.26 (1,641)

How to create good heuristics?

- By solving relaxed problems at each node
- In the 8-puzzle, the sum of the distances of each tile to its goal position (h_2) corresponds to solving 8 simple problems:



d_i is the length of the shortest path to move tile i to its goal position, ignoring the other tiles, e.g., $d_5 = 2$

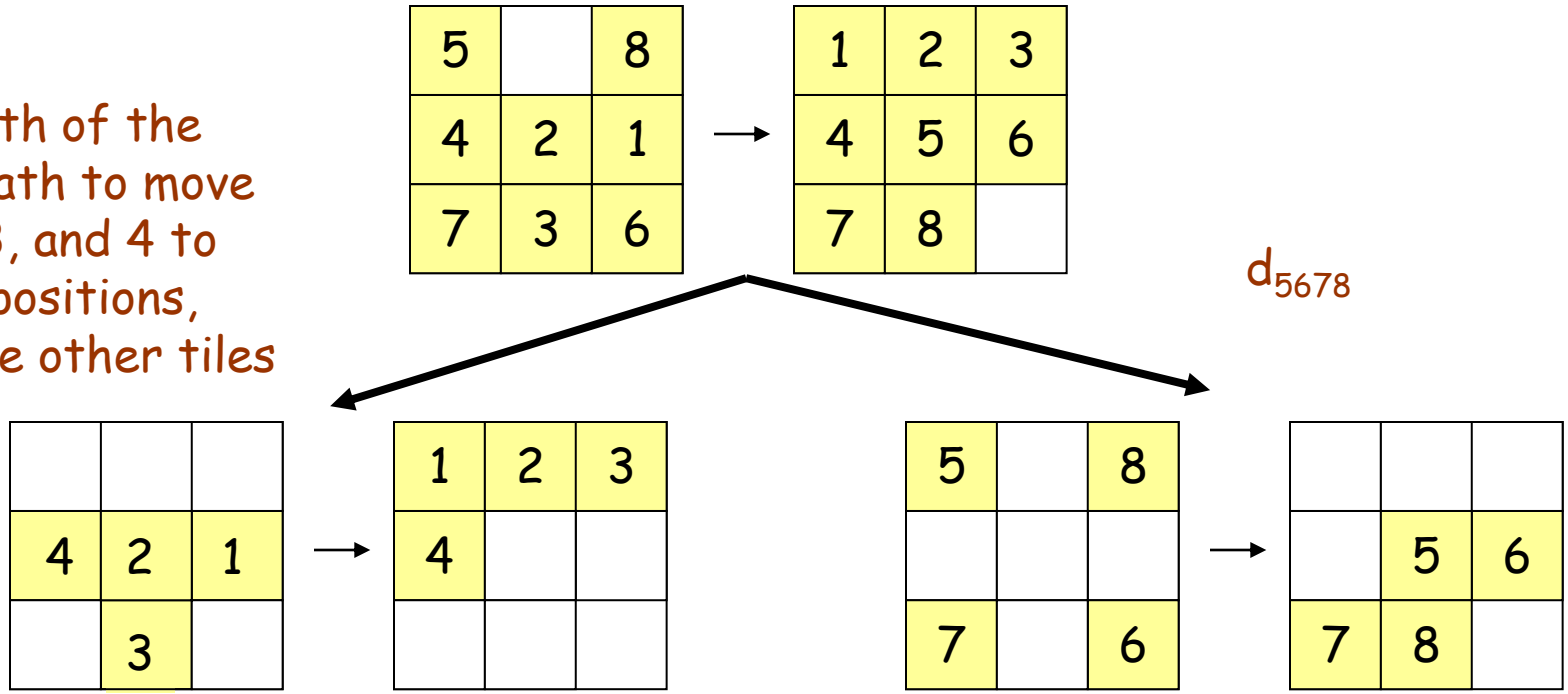
$h_2 = \Sigma_{i=1,\ldots 8}\ d_i$

- It ignores negative interactions among tiles

Can we do better?

- For example, we could consider two more complex relaxed problems:

d_{1234} = length of the shortest path to move tiles 1, 2, 3, and 4 to their goal positions, ignoring the other tiles

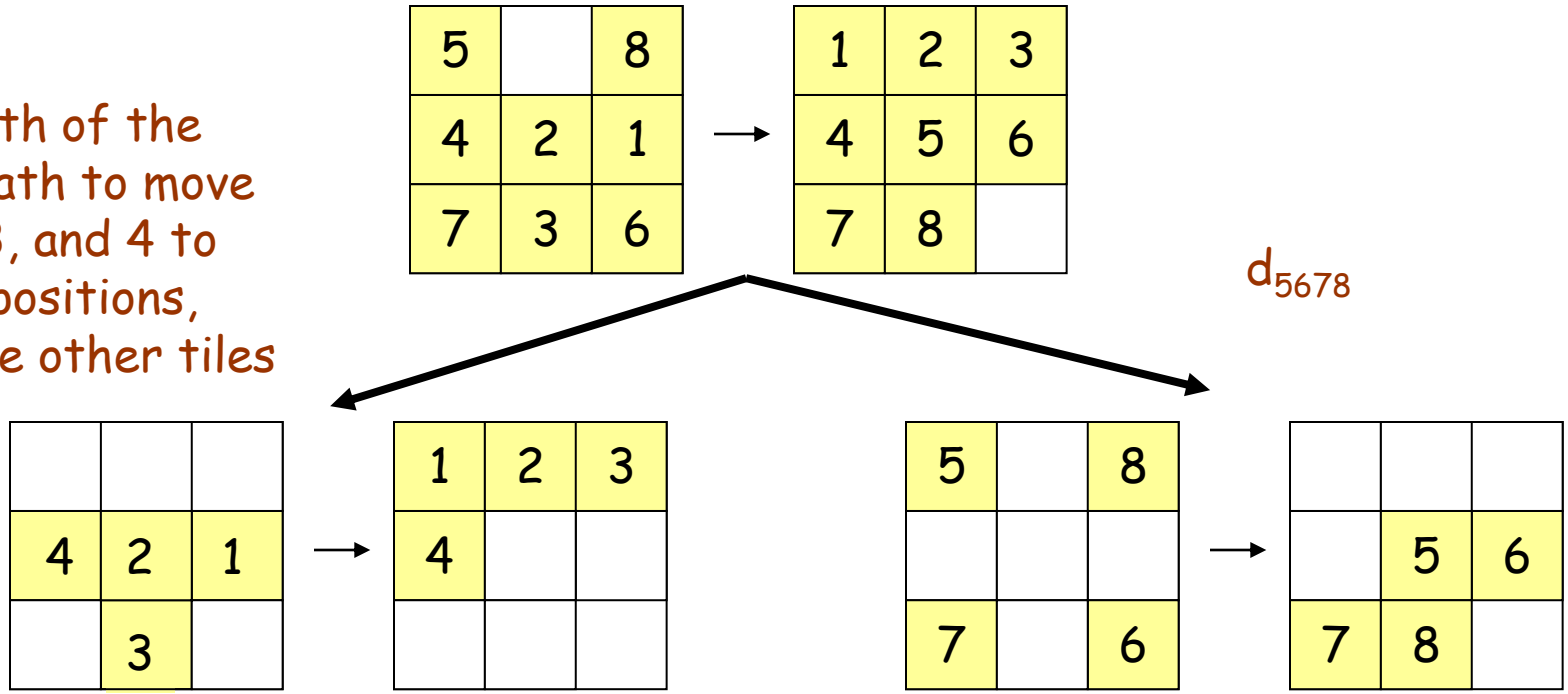


- → $h = d_{1234} + d_{5678}$ [disjoint pattern heuristic]

Can we do better?

- For example, we could consider two more complex relaxed problems:

d_{1234} = length of the shortest path to move tiles 1, 2, 3, and 4 to their goal positions, ignoring the other tiles

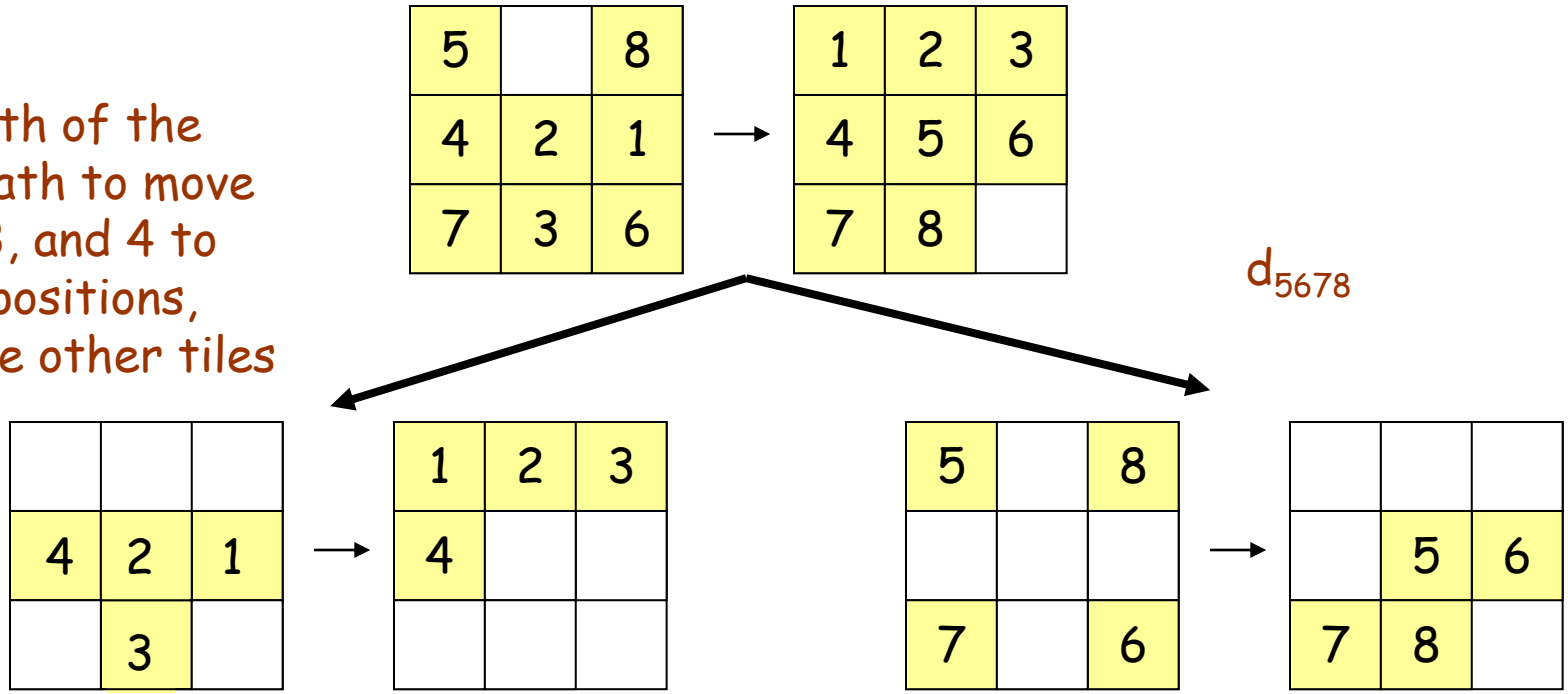


- → $h = d_{1234} + d_{5678}$ [disjoint pattern heuristic]
- How to compute d_{1234} and d_{5678}?

Can we do better?

- For example, we could consider two more complex relaxed problems:

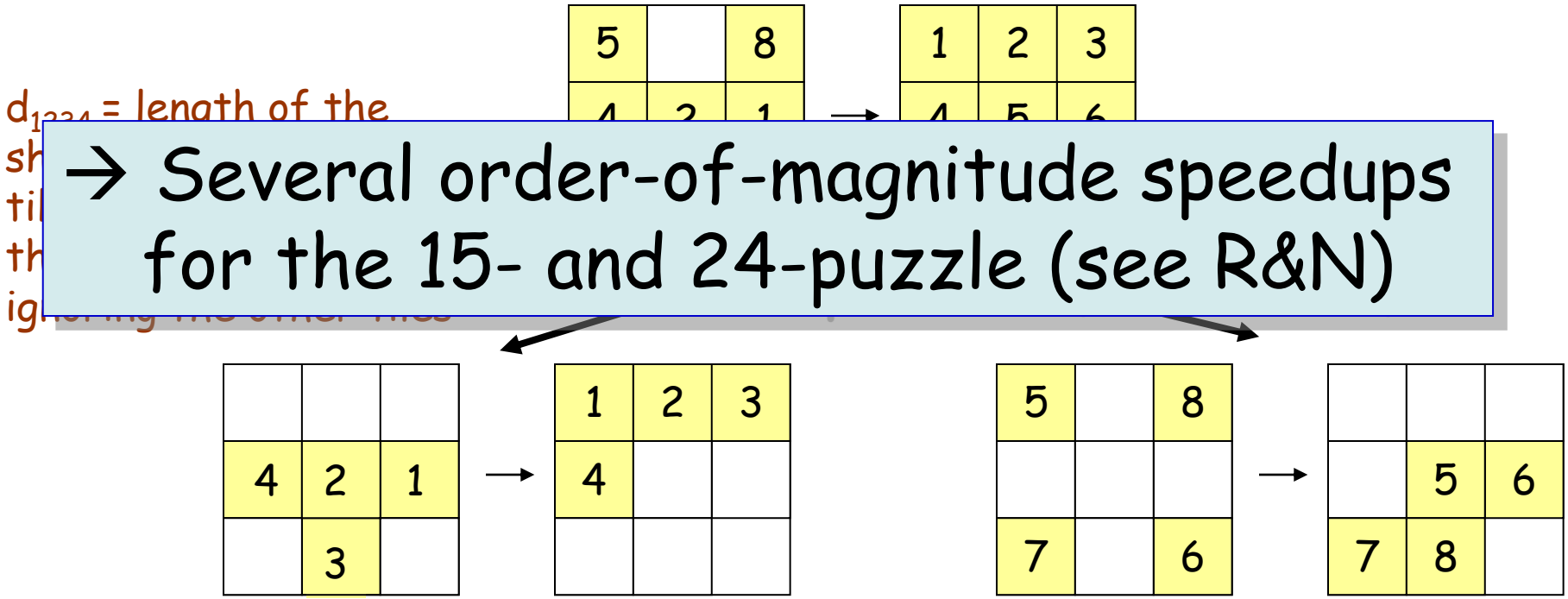
d_{1234} = length of the shortest path to move tiles 1, 2, 3, and 4 to their goal positions, ignoring the other tiles



- → $h = d_{1234} + d_{5678}$ [disjoint pattern heuristic]
- These distances are pre-computed and stored
  [Each requires generating a tree of 3,024 nodes/states (breadth-first search)]

Can we do better?

- For example, we could consider two more complex relaxed problems:



- → $h = d_{1234} + d_{5678}$ [disjoint pattern heuristic]
- These distances are pre-computed and stored
  [Each requires generating a tree of 3,024 nodes/states (breadth-first search)]

Learning heuristics from experience

- "Experience" here means solving lots of 8-puzzles, for instance.
- Each optimal solution to an 8-puzzle problem provides examples from which h(n) can be learned
- From these examples, a learning algorithm can be used to construct a function h(n)
- Features of state (instead of raw state description)
  - 8-puzzle
    - Number of misplaced tiles
    - Number of pairs of adjacent tiles that are not adjacent in the goal state
  - Linear Combination of features

$$h(n)=w_1X_1(n)+w_2X_2(n)$$

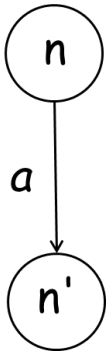
On Completeness and Optimality

- A* with a consistent heuristic function has nice properties: completeness, optimality, no need to revisit states
- Theoretical completeness does not mean "practical" completeness if you must wait too long to get a solution (remember the time limit issue)
- So, if one can't design an accurate consistent heuristic, it may be better to settle for a non-admissible heuristic that "works well in practice", even through completeness and optimality are no longer guaranteed

Memory-bounded heuristic search

Monotonicity of f()

- When h is consistent then f is non-decreasing along any path
- When h is admissible but not consistent:



$$f(n') = \max\big(g(n') + h(n'), f(n)\big)$$

$$if \quad g(n') + h(n') \geq f(n) \rightarrow f(n') = g(n') + h(n')$$

$$if \quad g(n') + h(n') < f(n) \rightarrow f(n') = f(n) = g(n) + h(n)$$

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

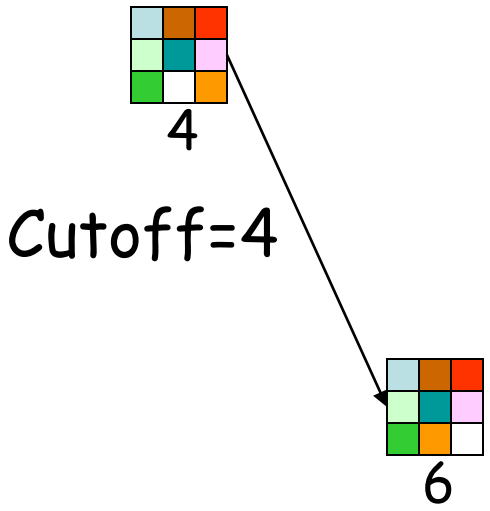
$$h(n') = h(n) - c(n, a, n')$$

Iterative Deepening A* (IDA*)

- Idea: Reduce memory requirement of A* by applying cutoff on values of f
- Consistent heuristic function h
- Algorithm IDA*:
  1. Initialize cutoff to f(initial-node)
  2. Repeat:
     a. Perform depth-first search by expanding all nodes N such that f(N) ≤ cutoff
     b. Reset cutoff to smallest value f of non-expanded (leaf) nodes

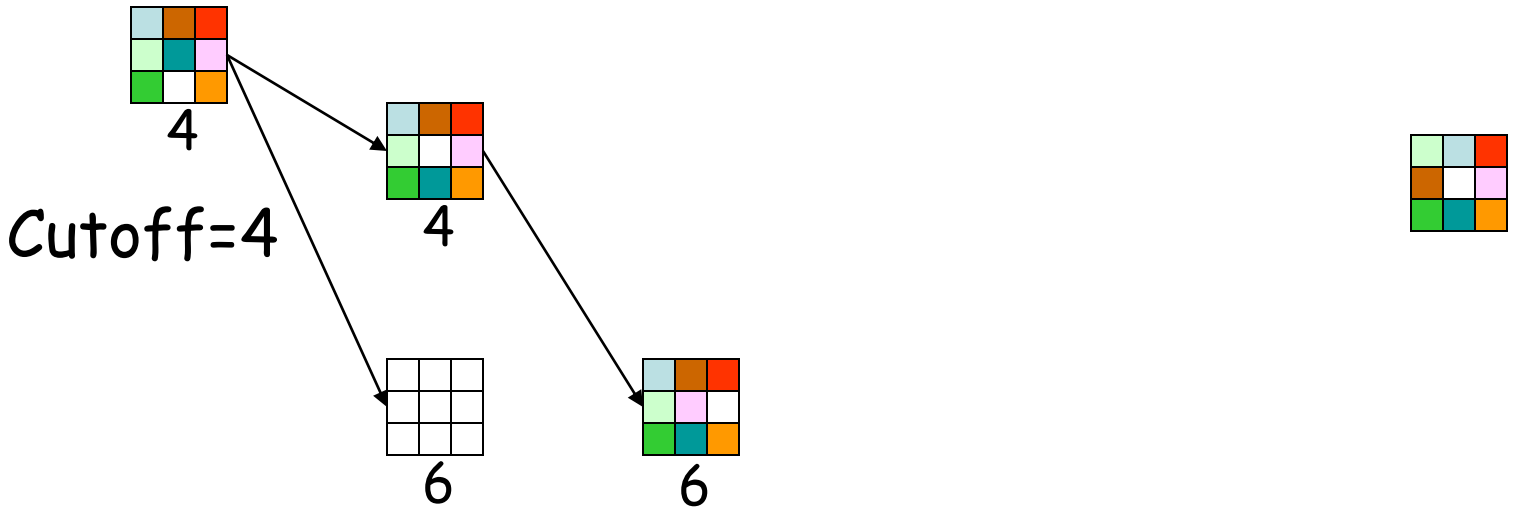
8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles



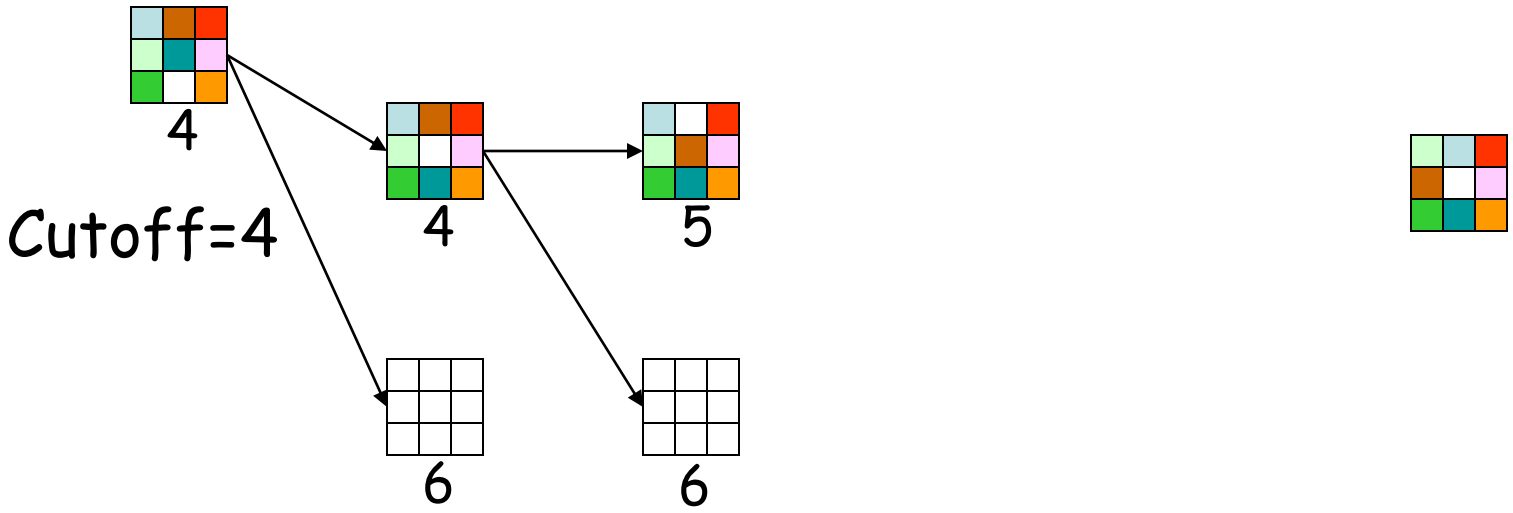
8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles



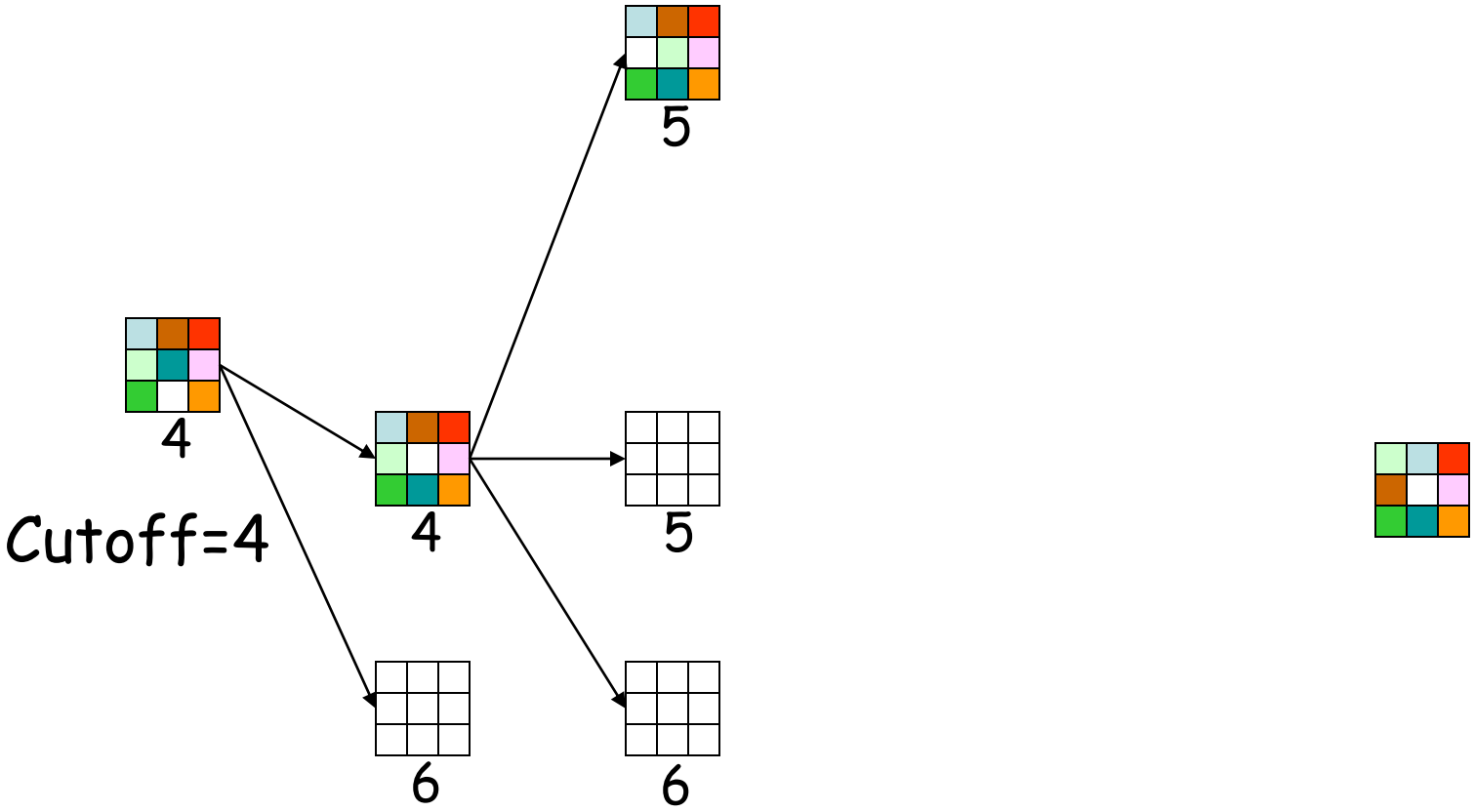
8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles



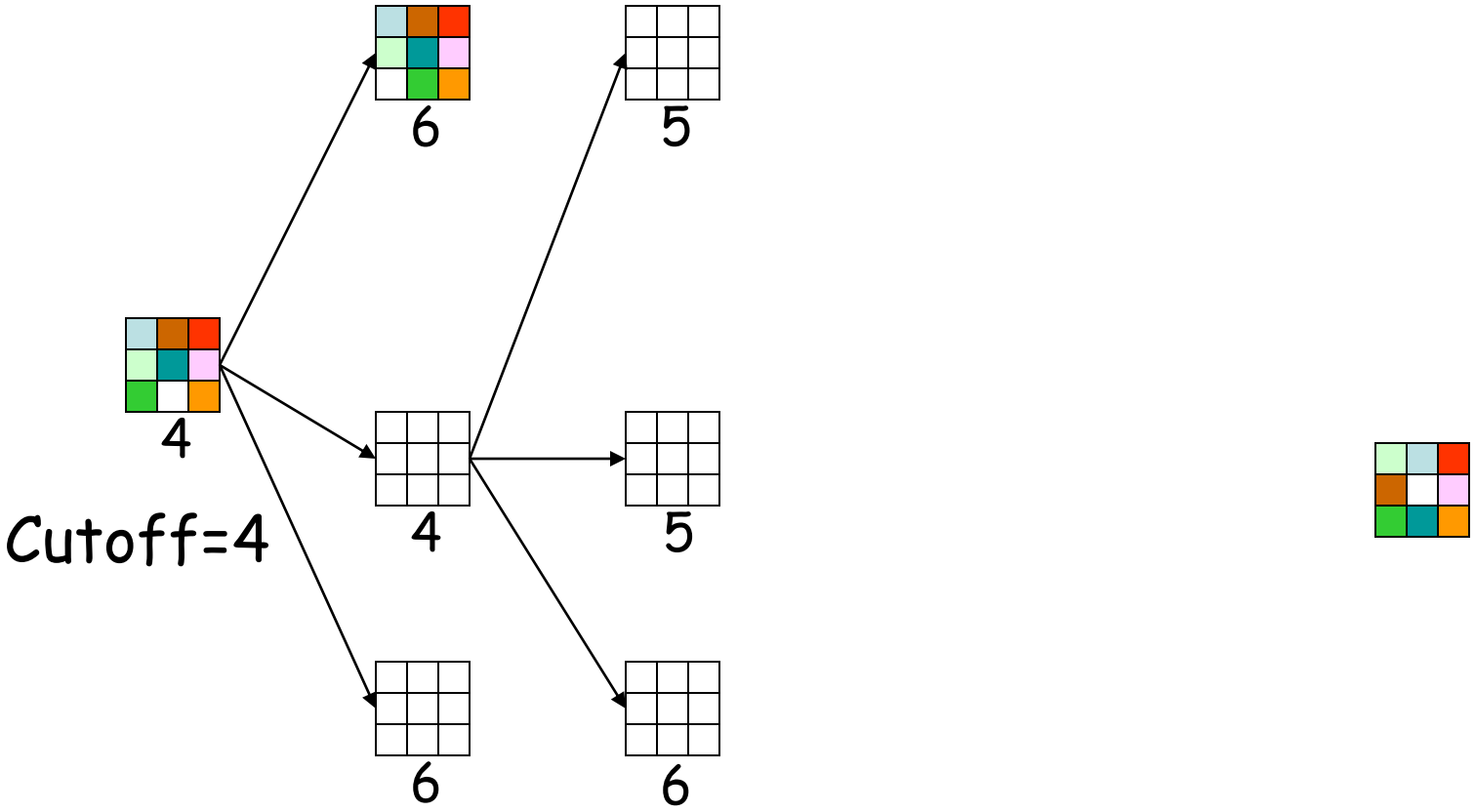
8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles



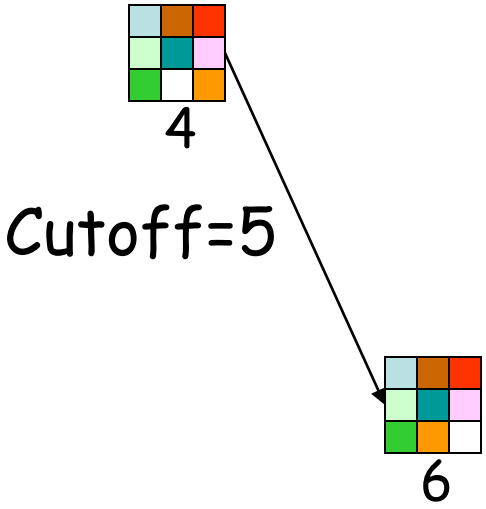
8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles



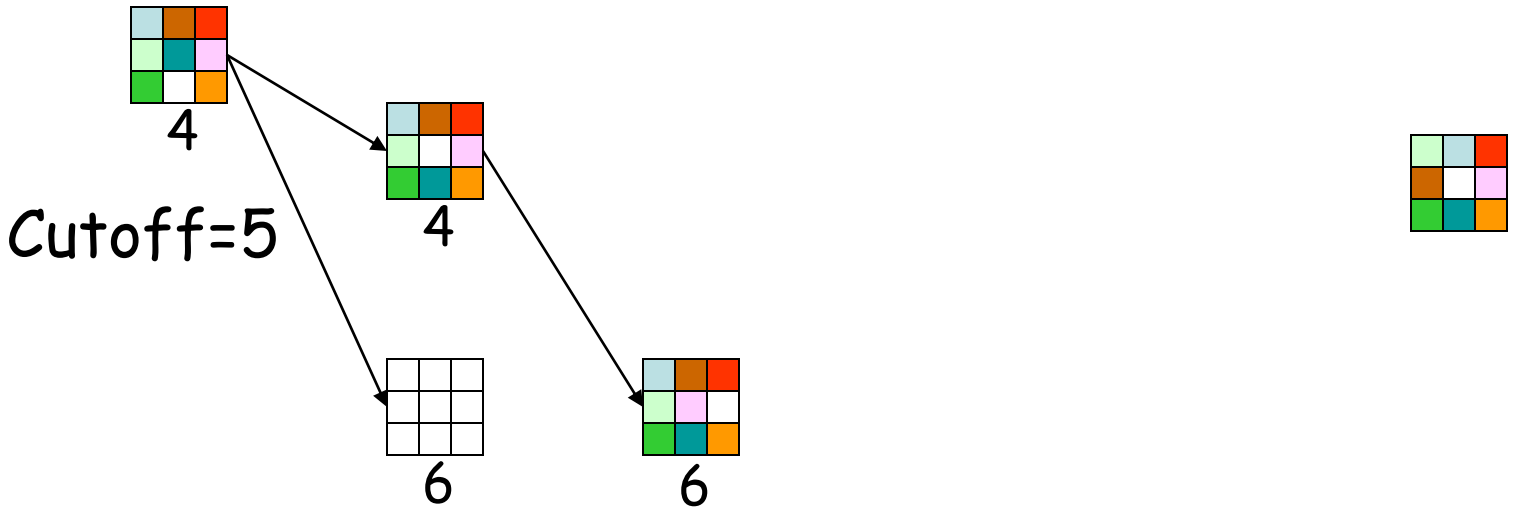
8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles



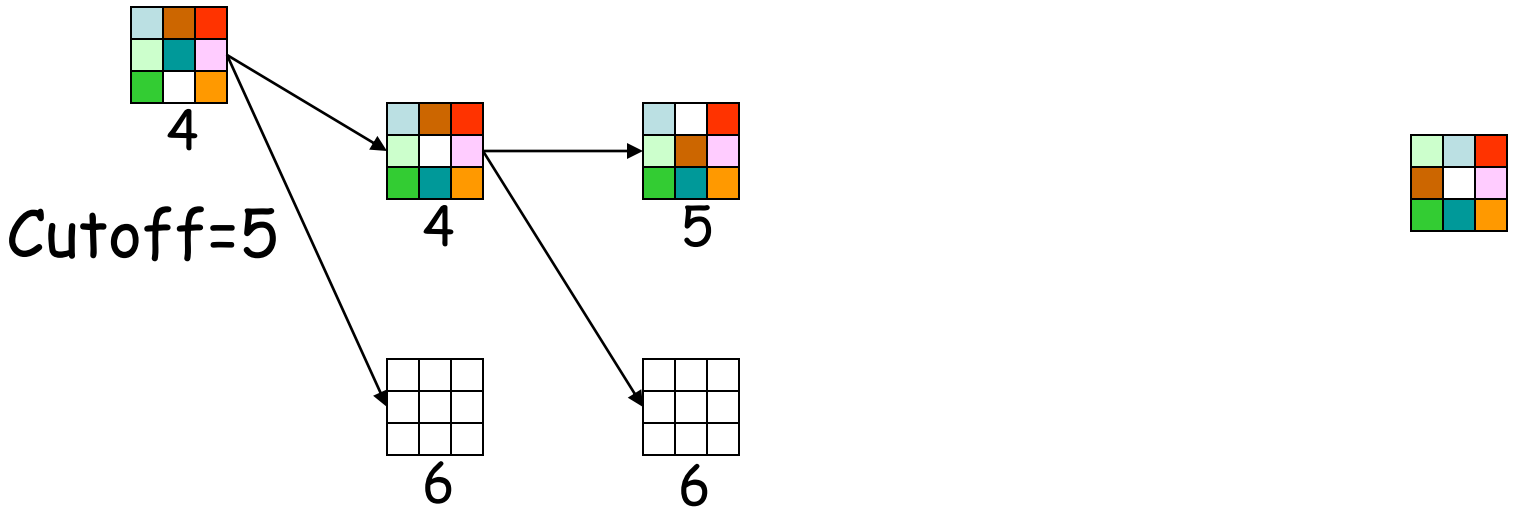
8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles



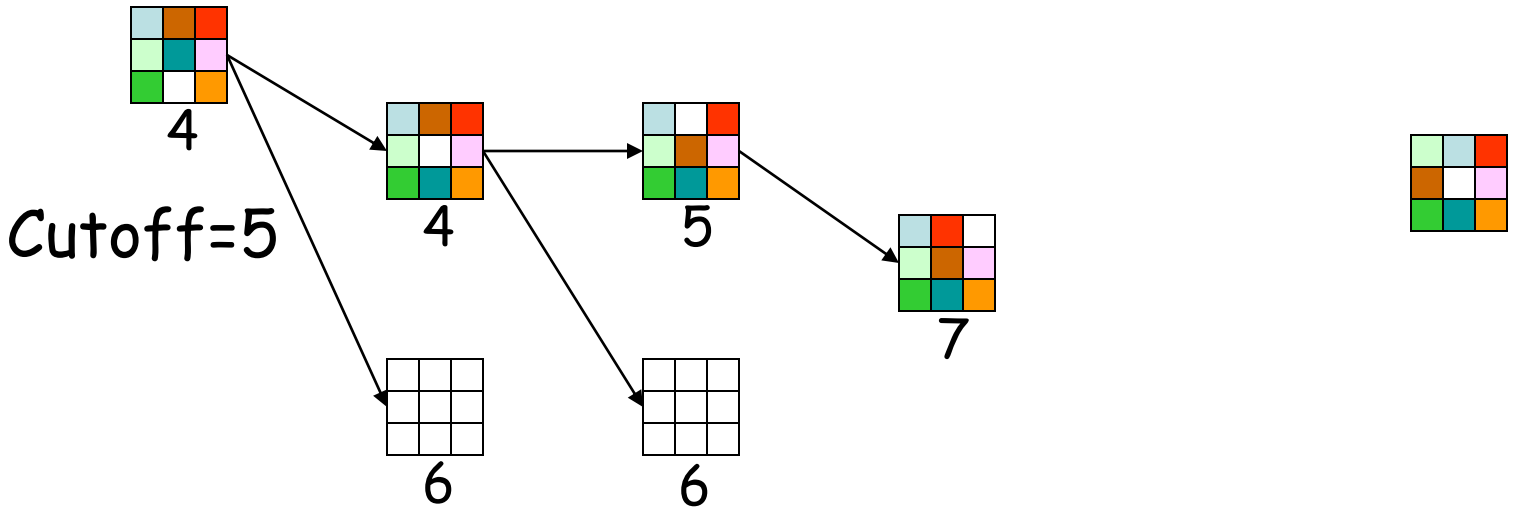
8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles



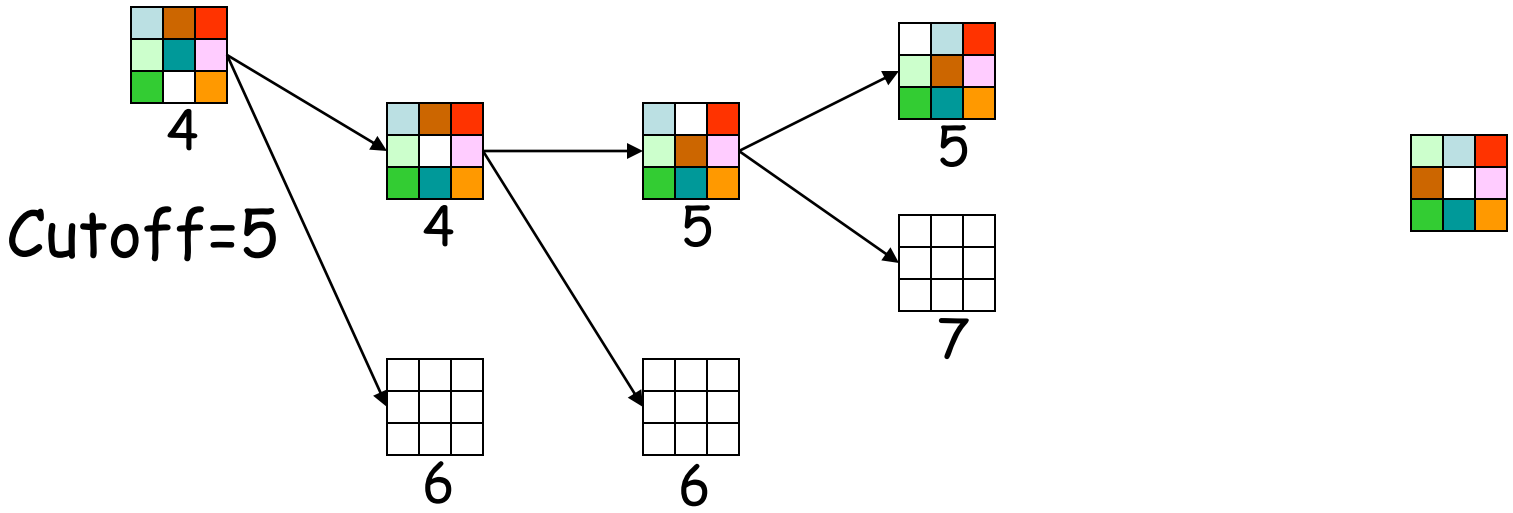
8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles



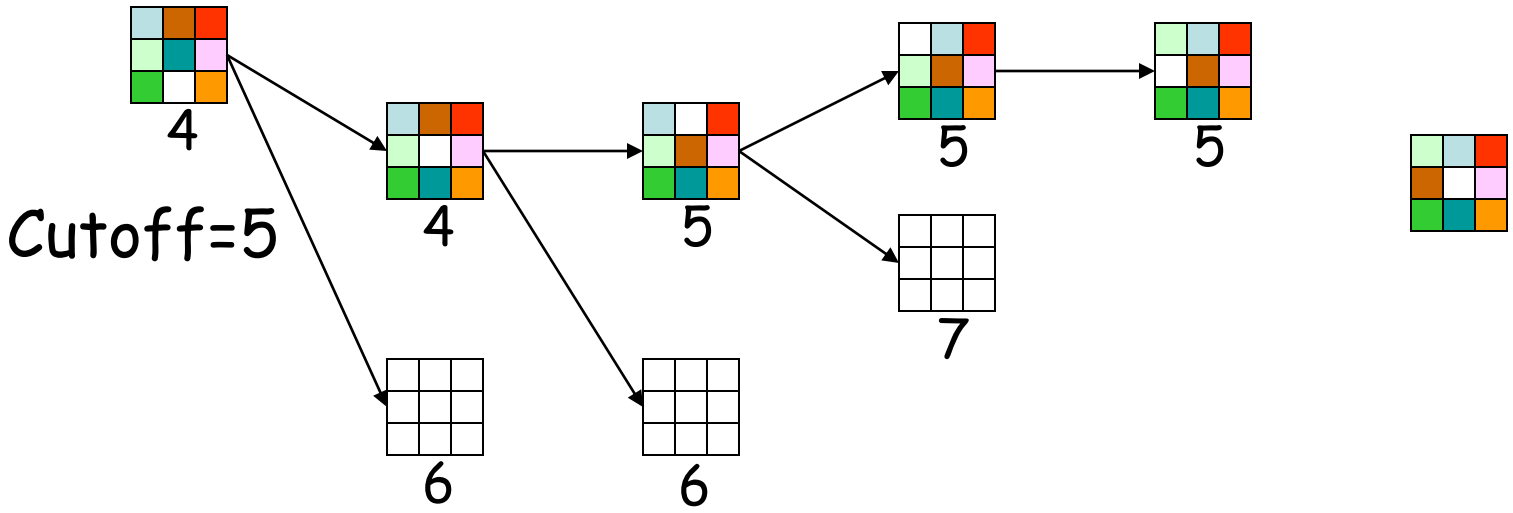
8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles



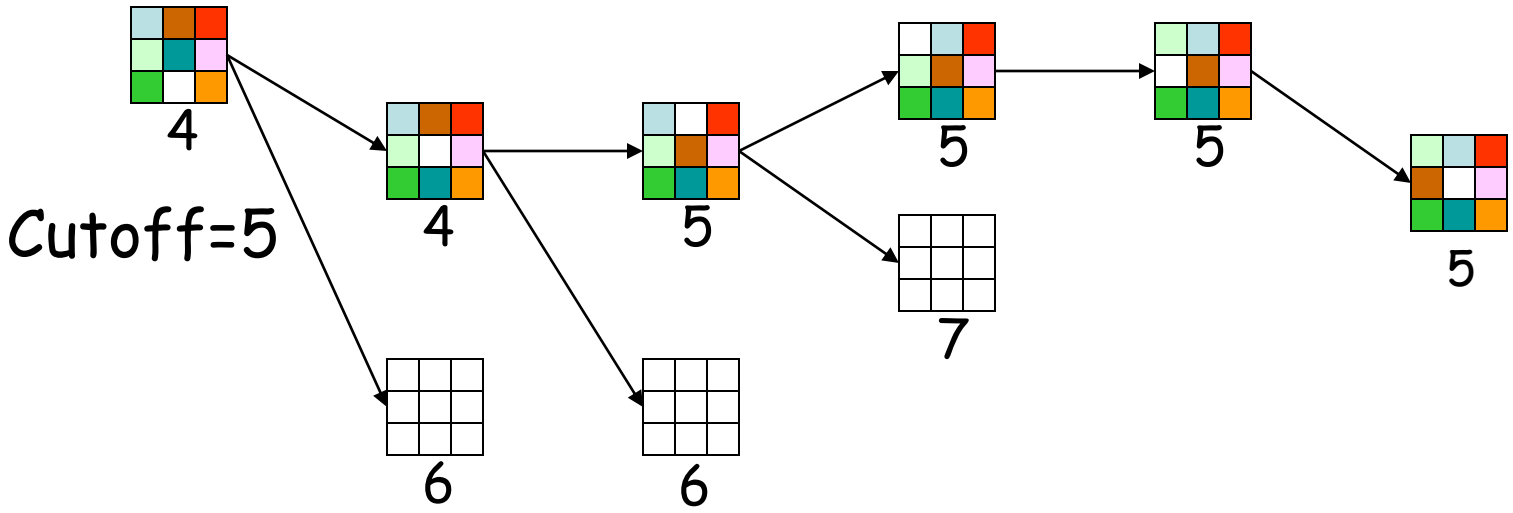
8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles



8-Puzzle

f(N) = g(N) + h(N)
with h(N) = number of misplaced tiles



Advantages/Drawbacks of IDA*

- Advantages:
  - Still complete and optimal
  - Requires less memory than A*
  - Avoid the overhead to sort the fringe
- Drawbacks:
  - Can't avoid revisiting states not on the current path
  - Available memory is poorly used

Recursive best-first search (RBFS)

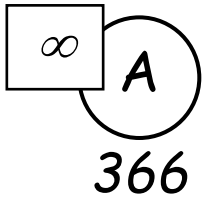
- Its structure is similar to that of a recursive depth-first search
- Rather than continuing indefinitely down the current path, it uses the f-limit variable to keep track of the f-value of the best alternative path available from any ancestor of the current node.
- If the current node exceeds this limit, the recursion unwinds back to the alternative path then replaces the f-value of each node along the path with a backed-up value (the best f-value of its children).

Recursive best-first search (RBFS)

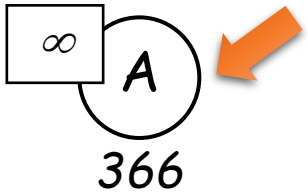
```
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure
    return RBFS(problem, MAKE-NODE(problem.INITIAL-STATE), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit
  if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
  successors ← [ ]
  for each action in problem.ACTIONS(node.STATE) do
      add CHILD-NODE(problem, node, action) into successors
  if successors is empty then return failure, ∞
  for each s in successors do /* update f with value from previous search, if any */
      s.f ← max(s.g + s.h, node.f))
  loop do
      best ← the lowest f-value node in successors
      if best.f > f_limit then return failure, best.f
      alternative ← the second-lowest f-value among successors
      result, best.f ← RBFS(problem, best, min(f_limit, alternative))
      if result ≠ failure then return result
```

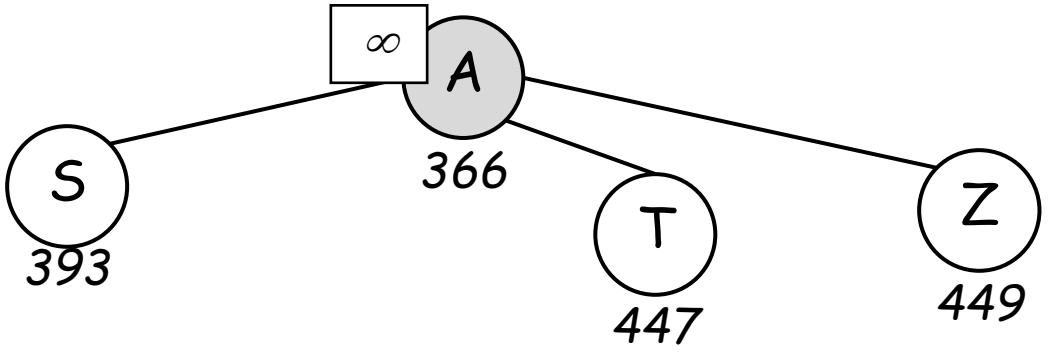
RBFS- Example



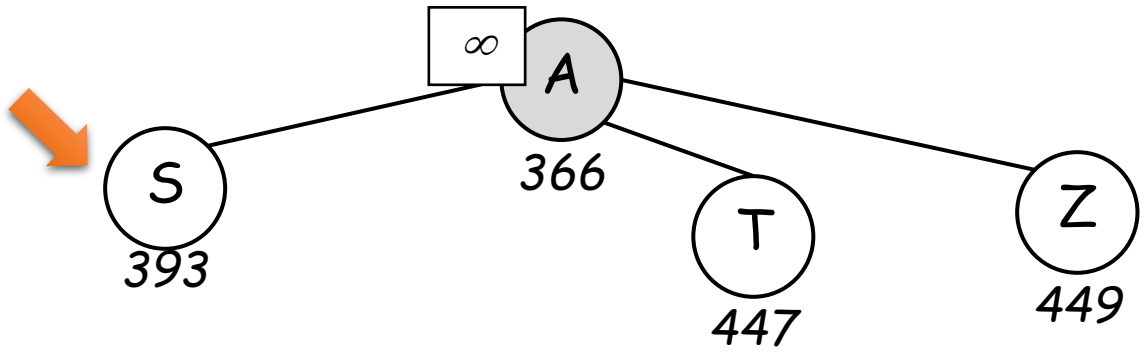
RBFS- Example



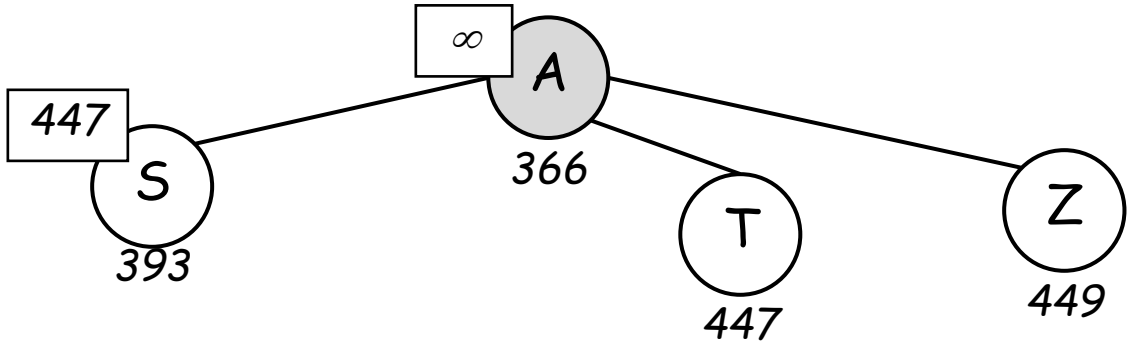
RBFS- Example



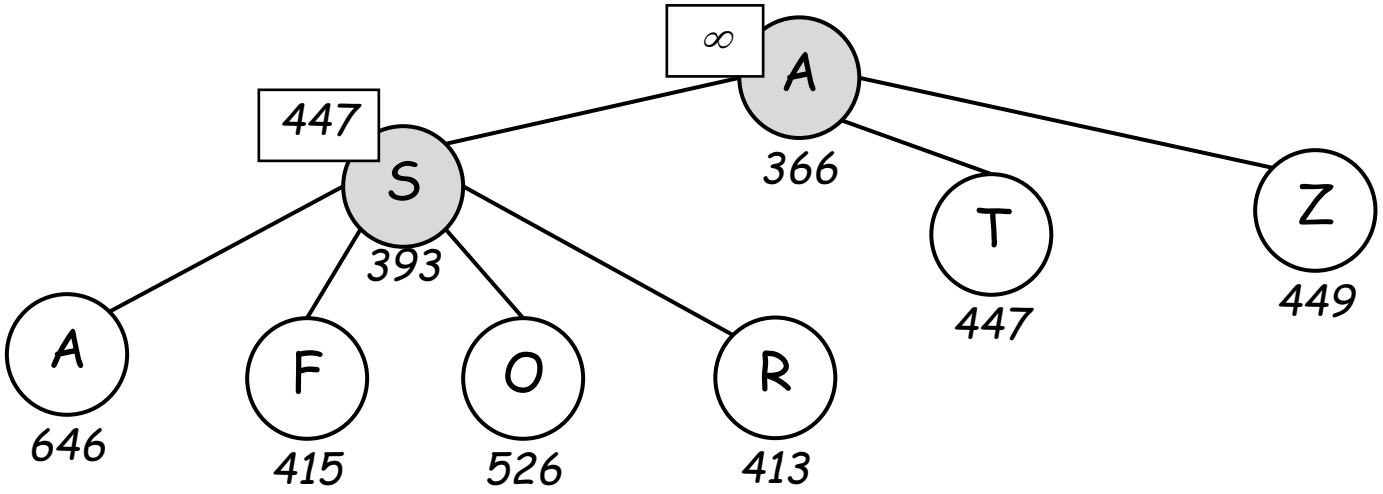
RBFS- Example



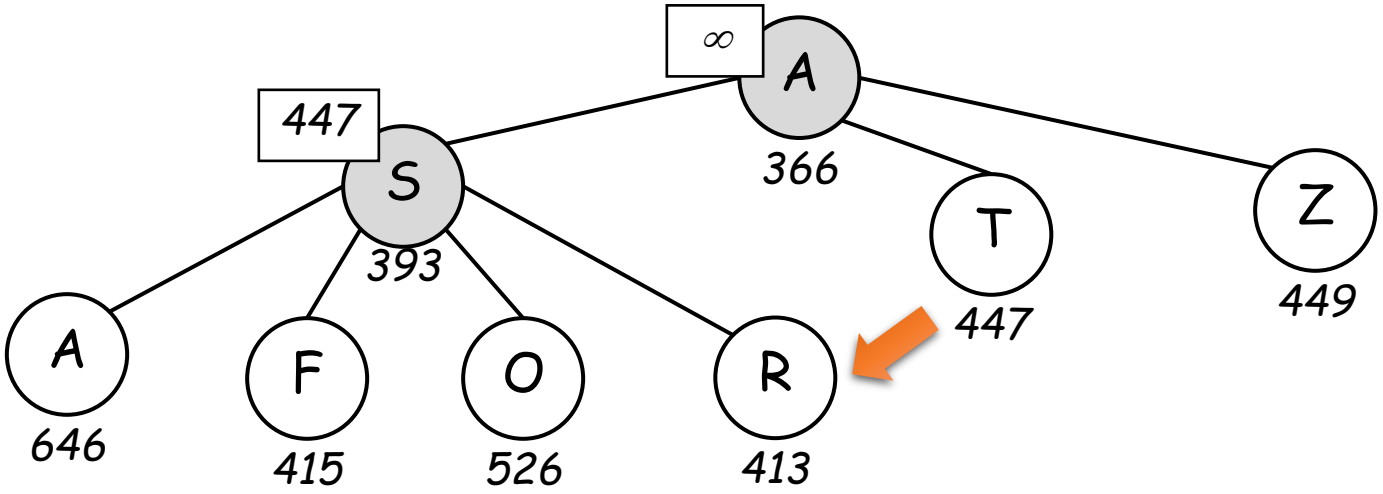
RBFS- Example



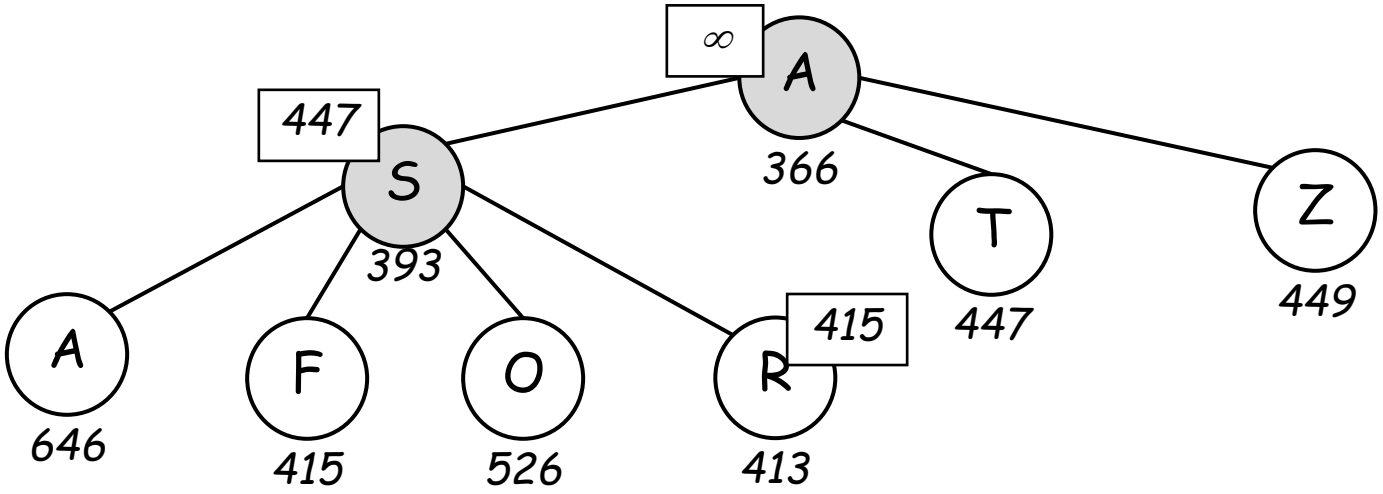
RBFS- Example



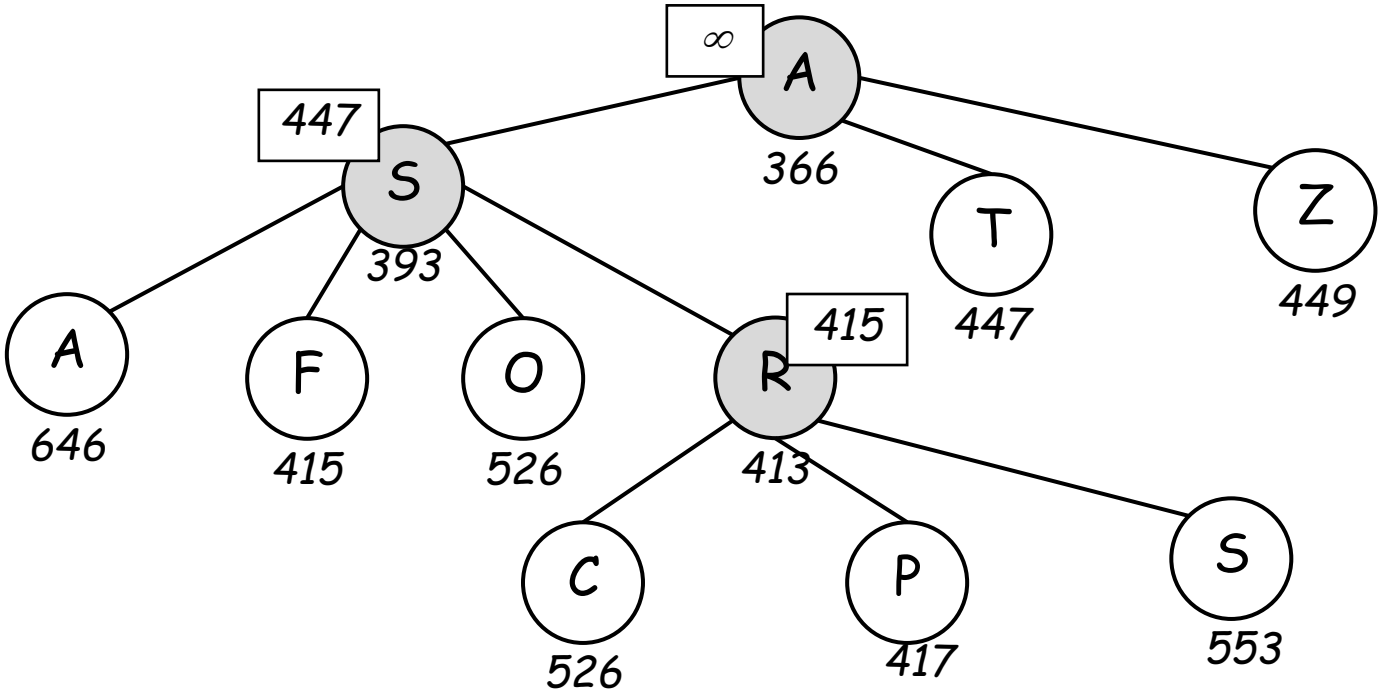
RBFS- Example



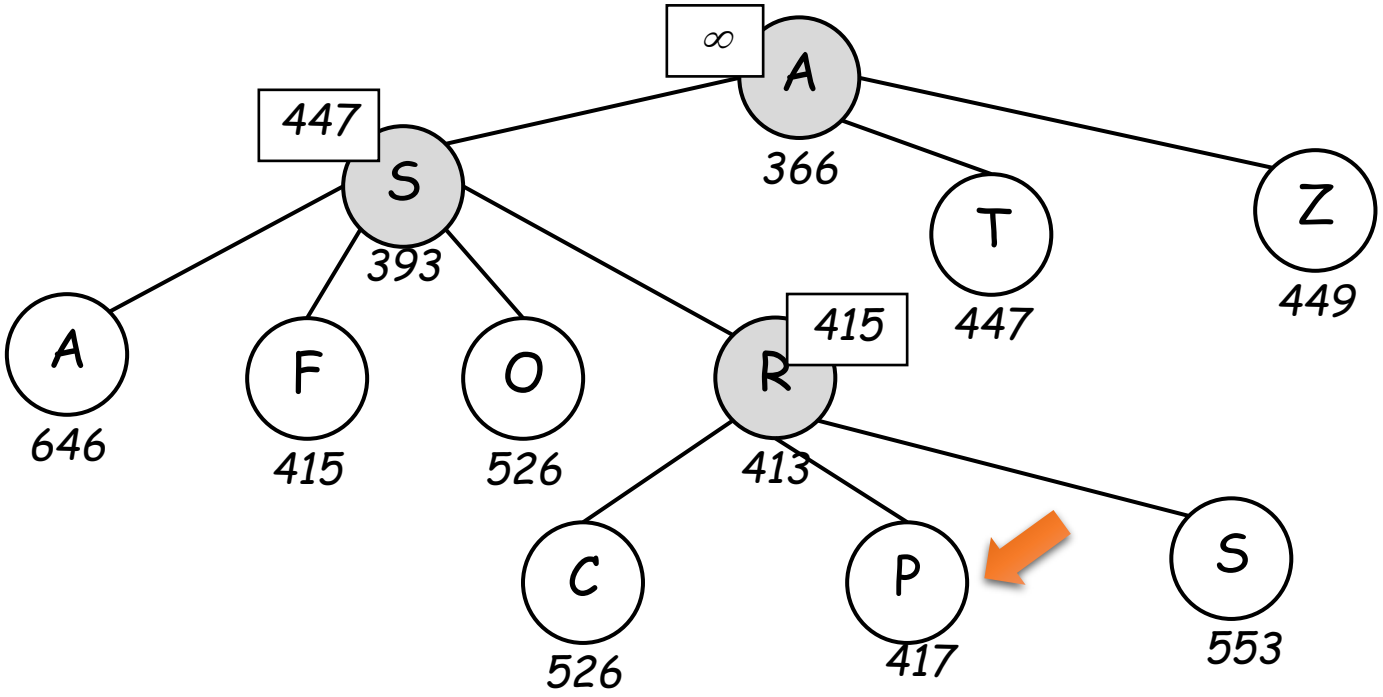
RBFS- Example



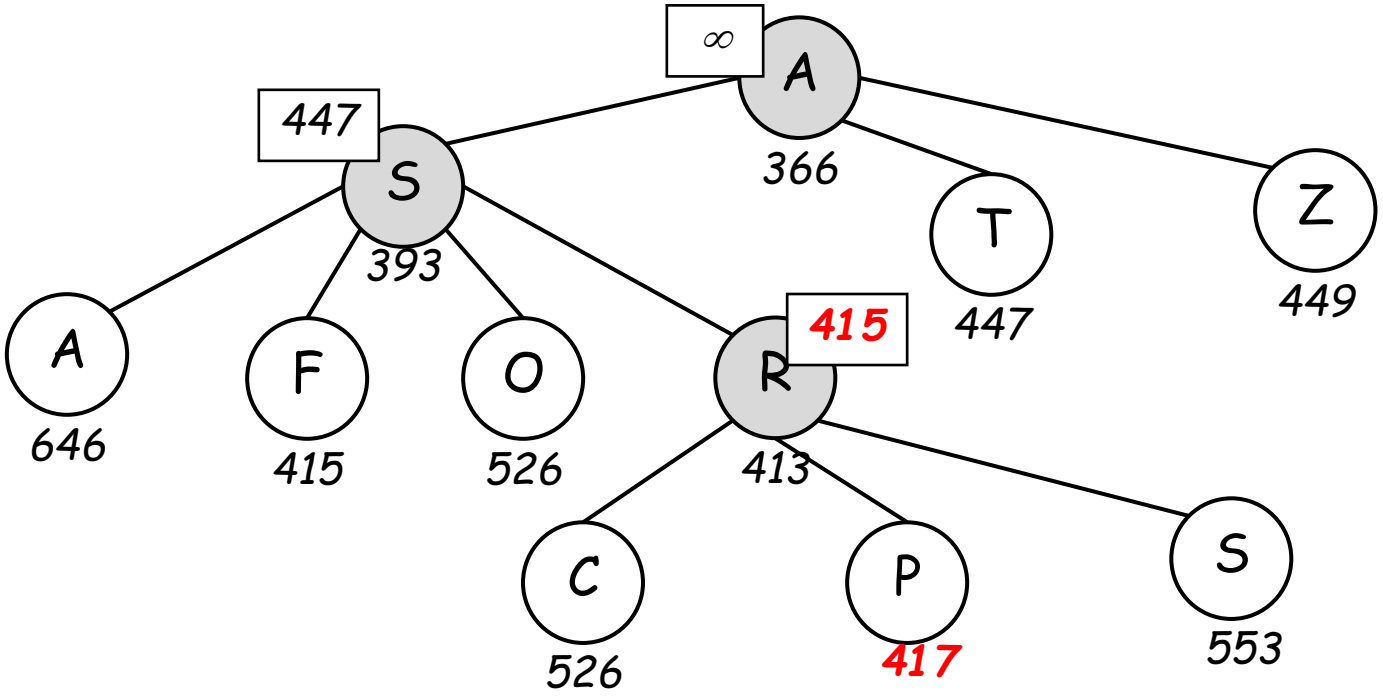
RBFS- Example



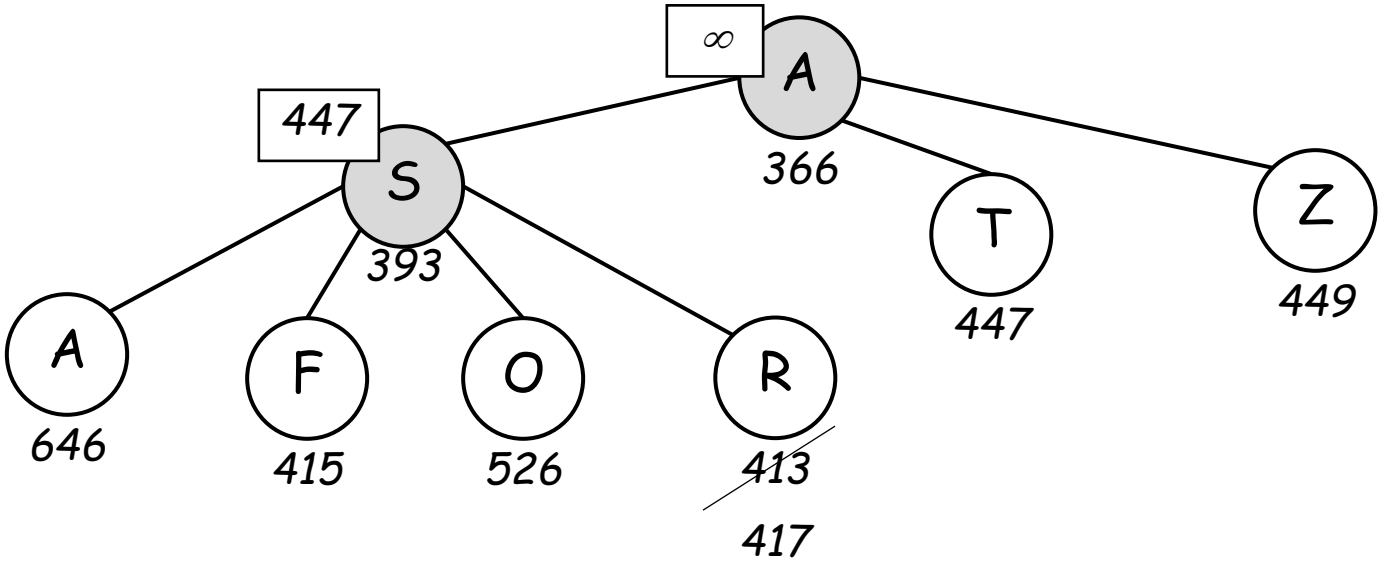
RBFS- Example



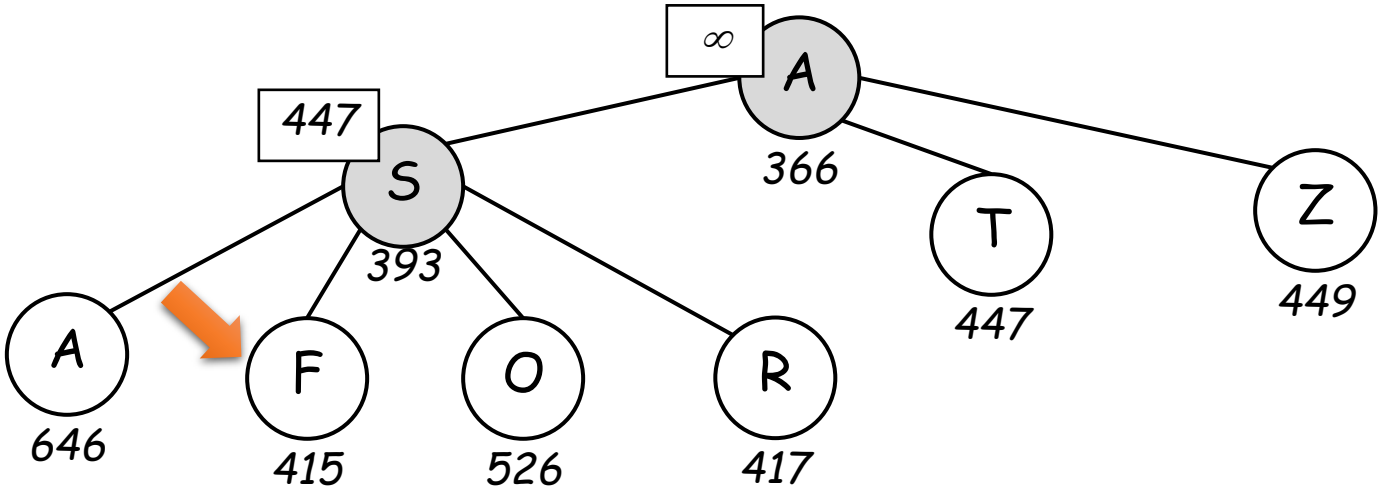
RBFS- Example



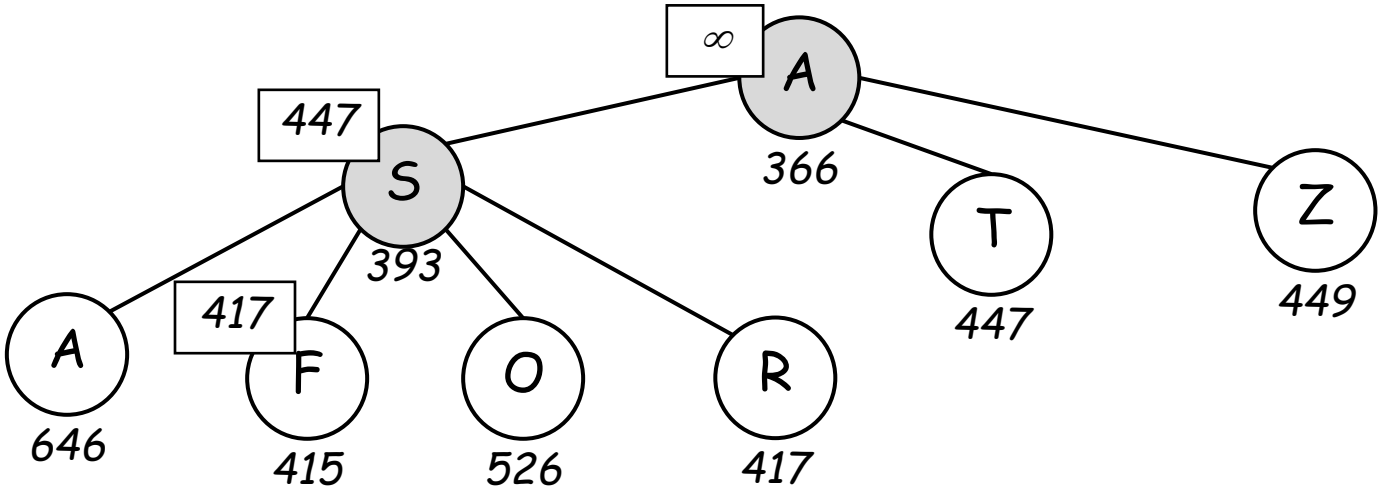
RBFS- Example



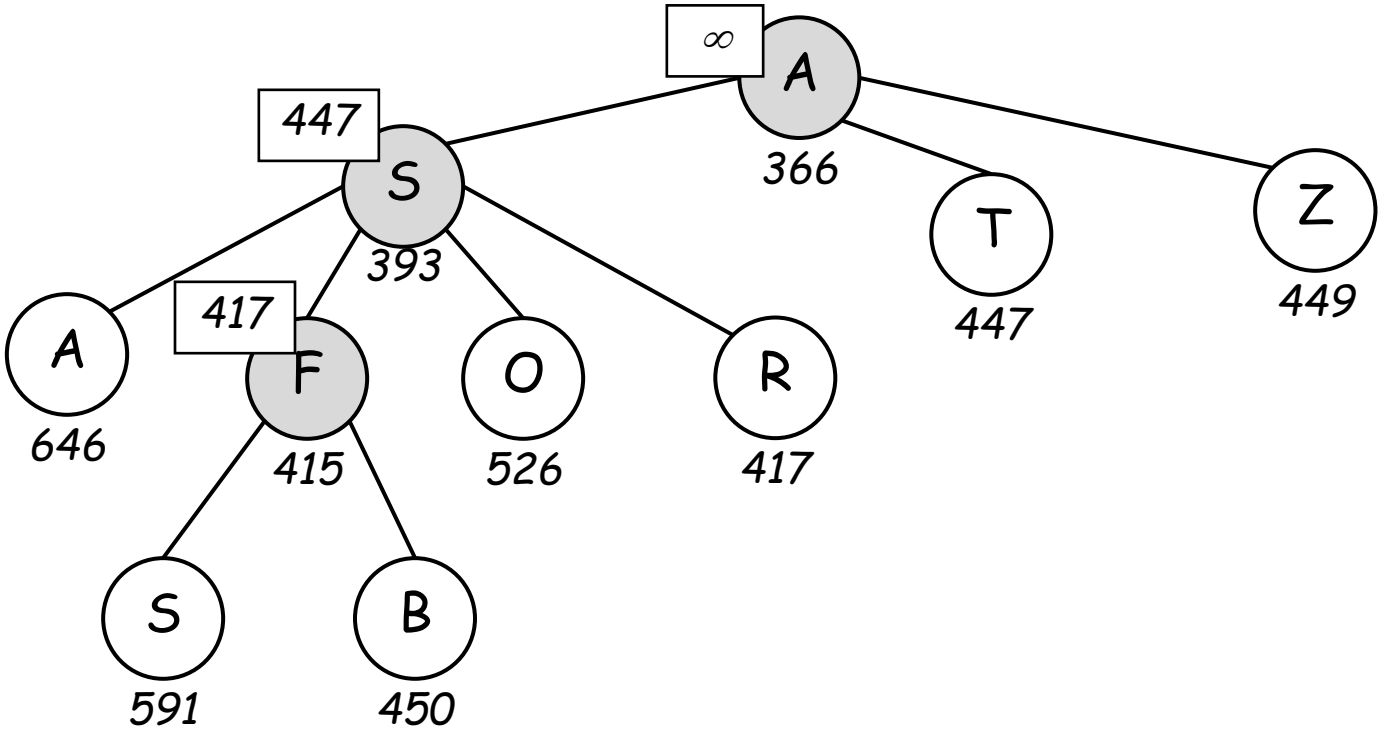
RBFS- Example



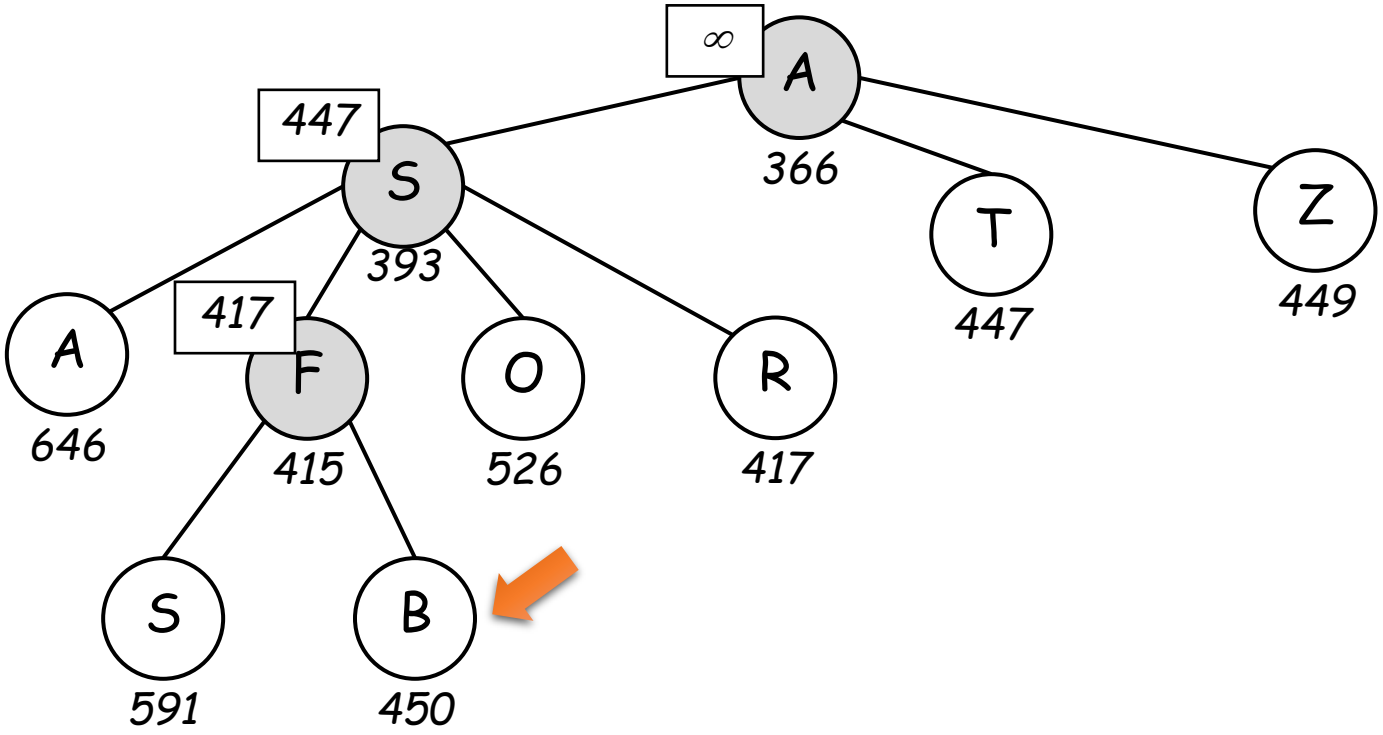
RBFS- Example



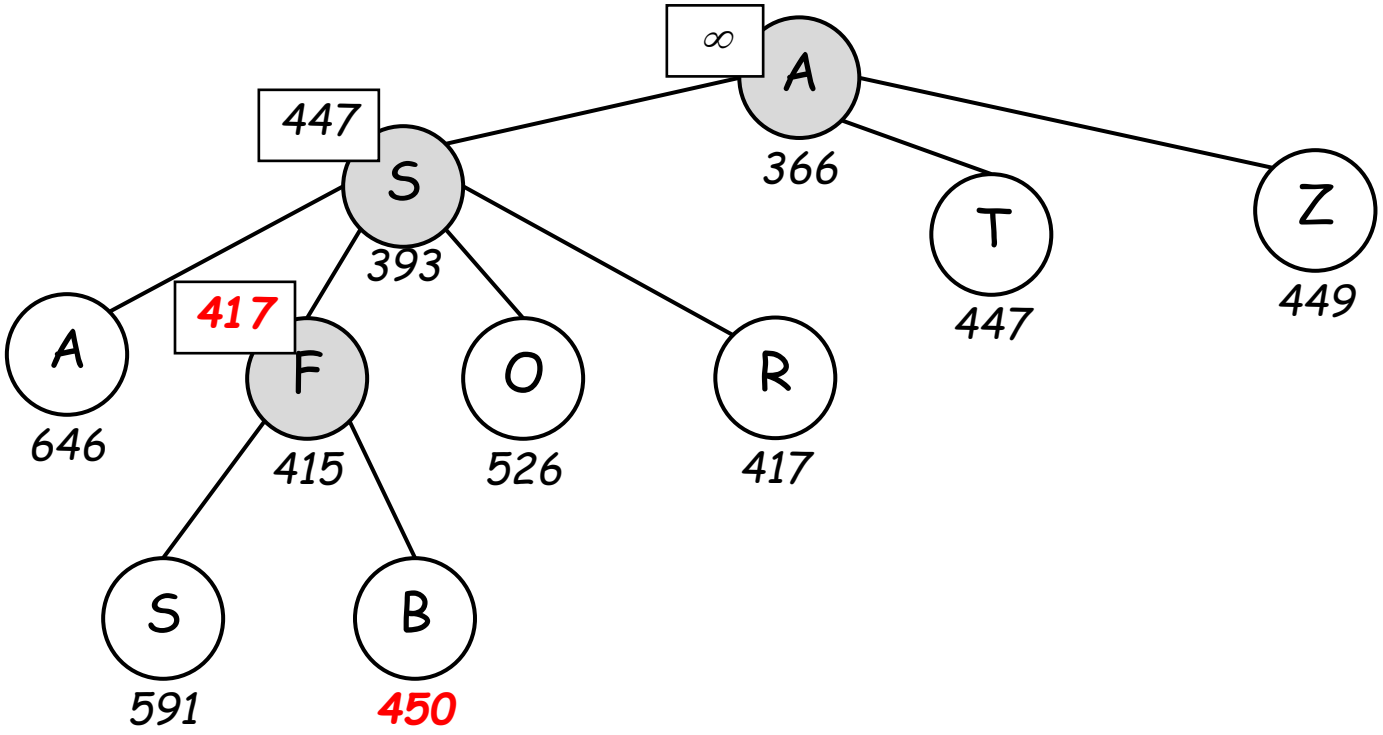
RBFS- Example



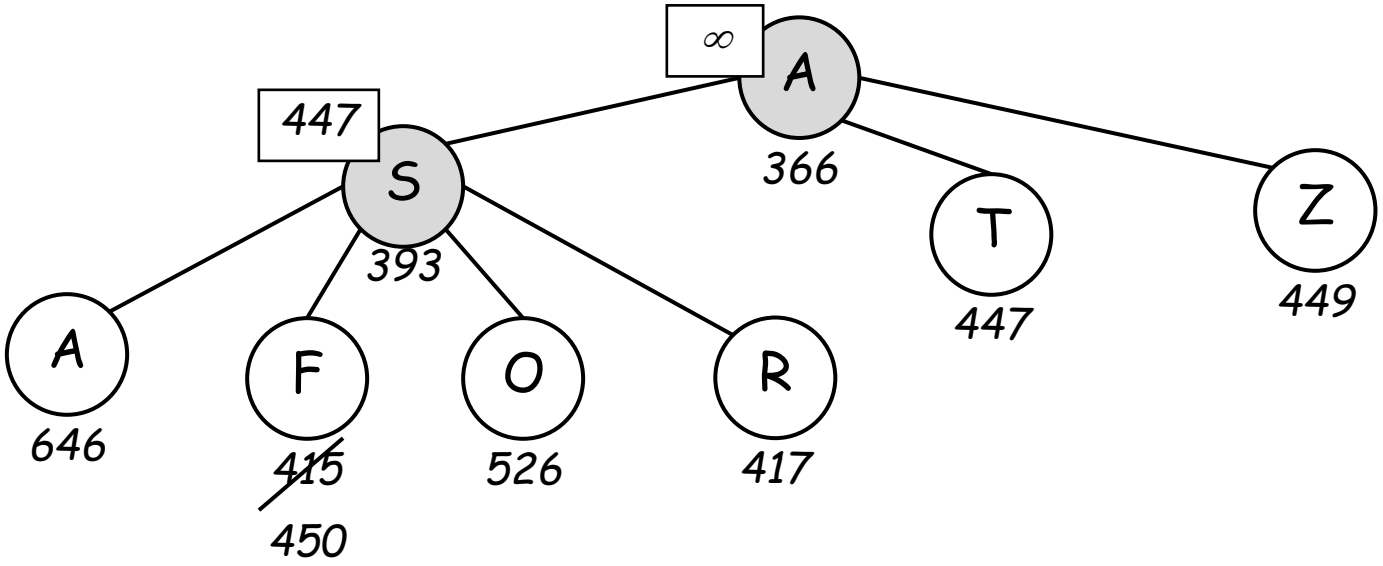
RBFS- Example



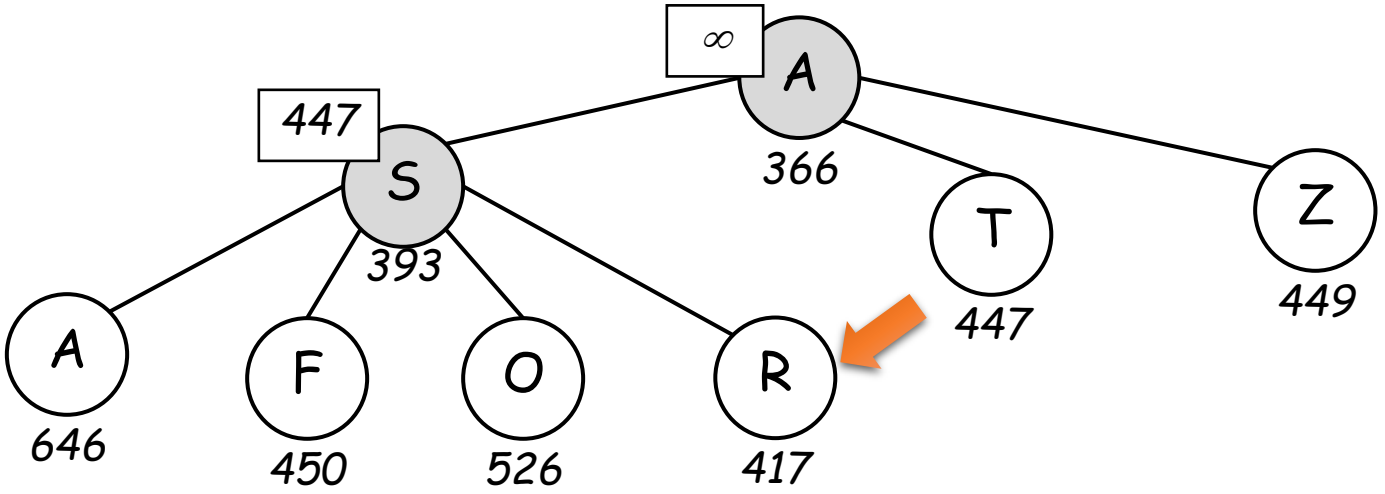
RBFS- Example



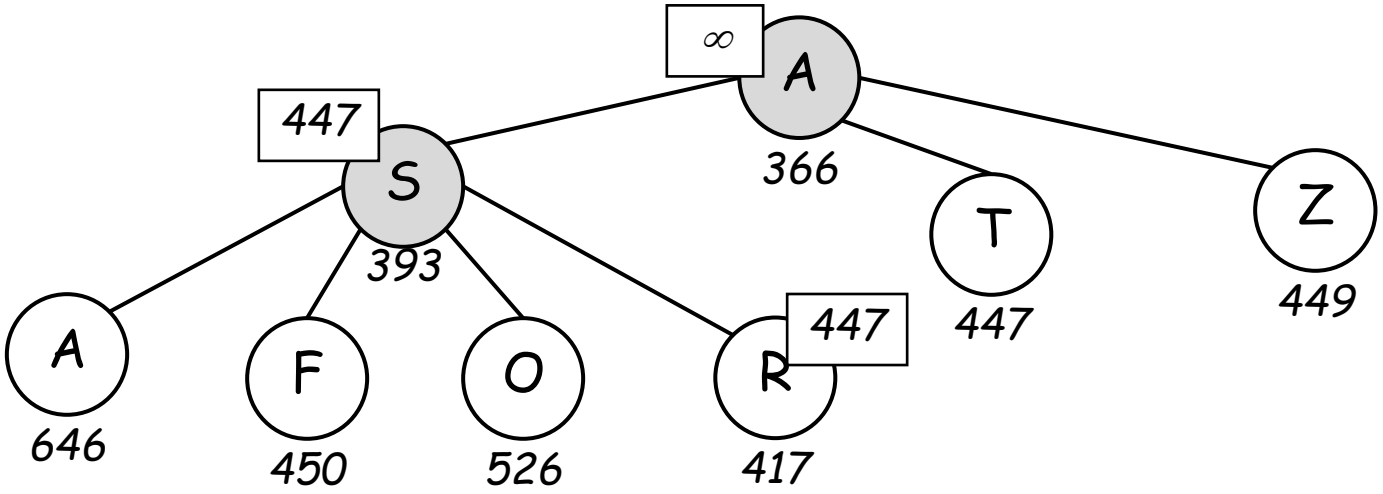
RBFS- Example



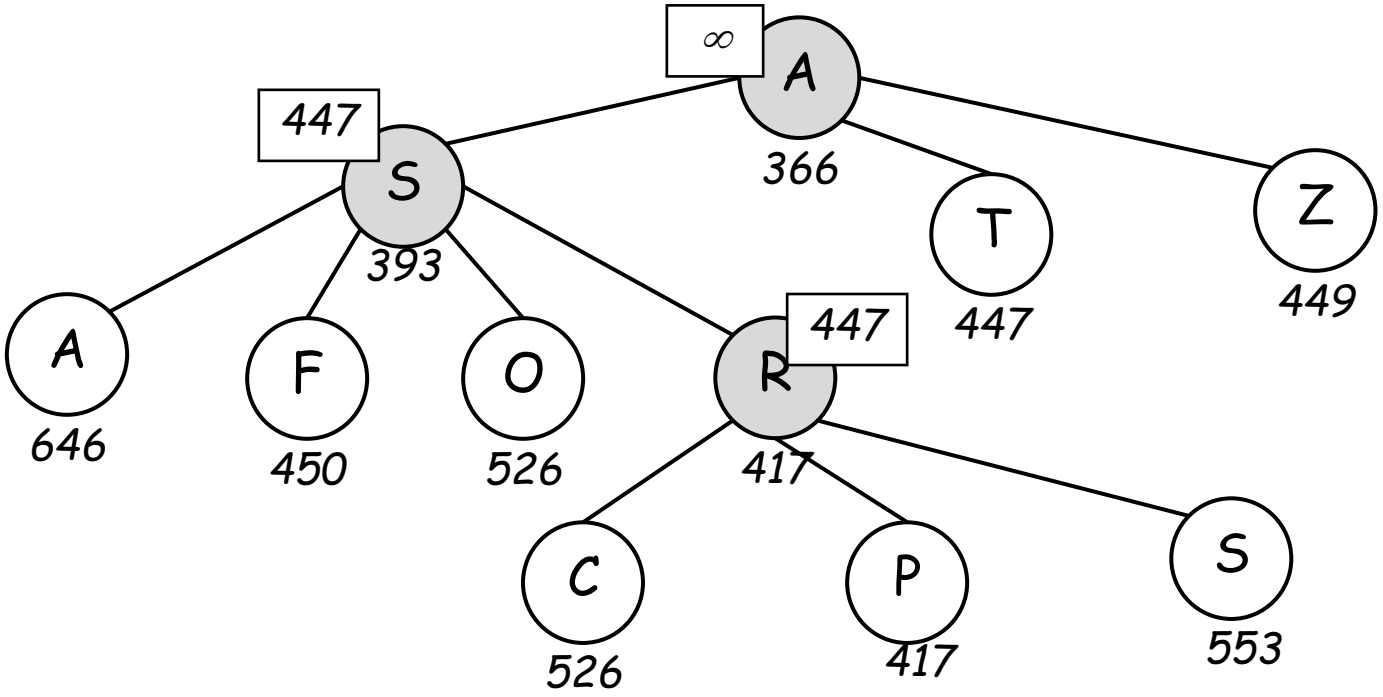
RBFS- Example



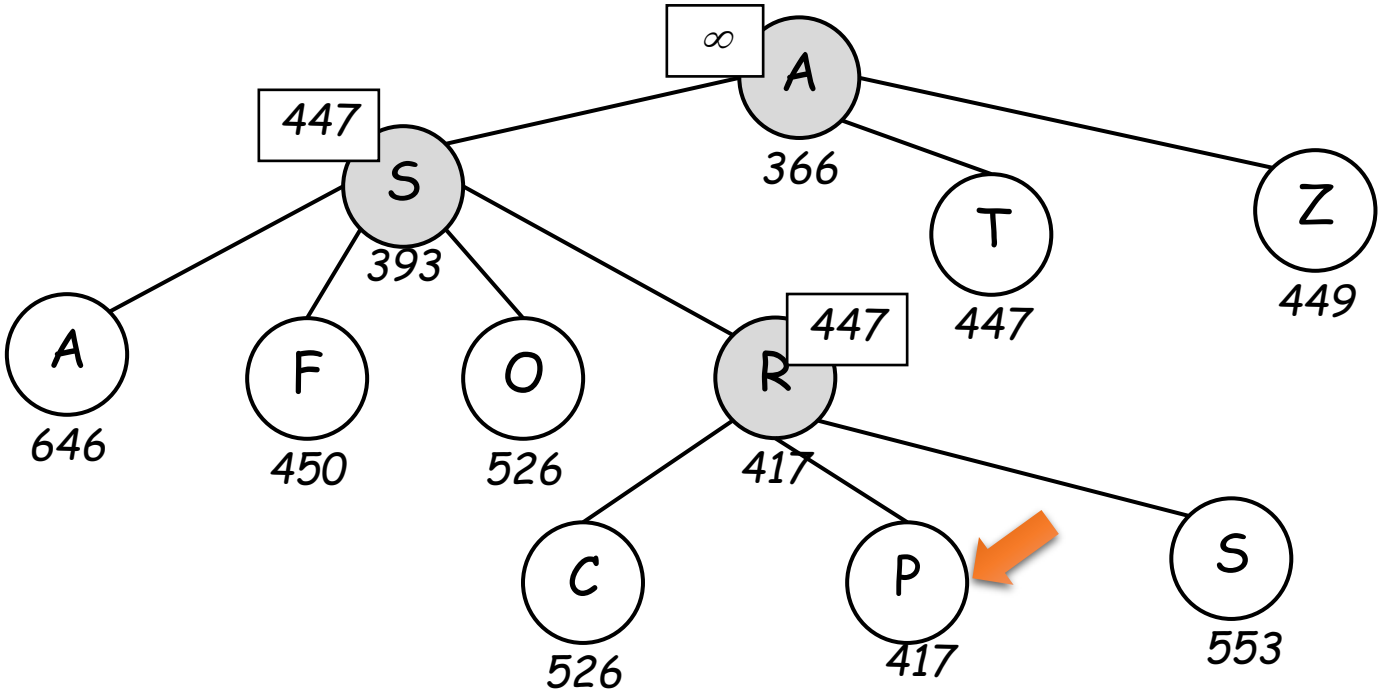
RBFS- Example



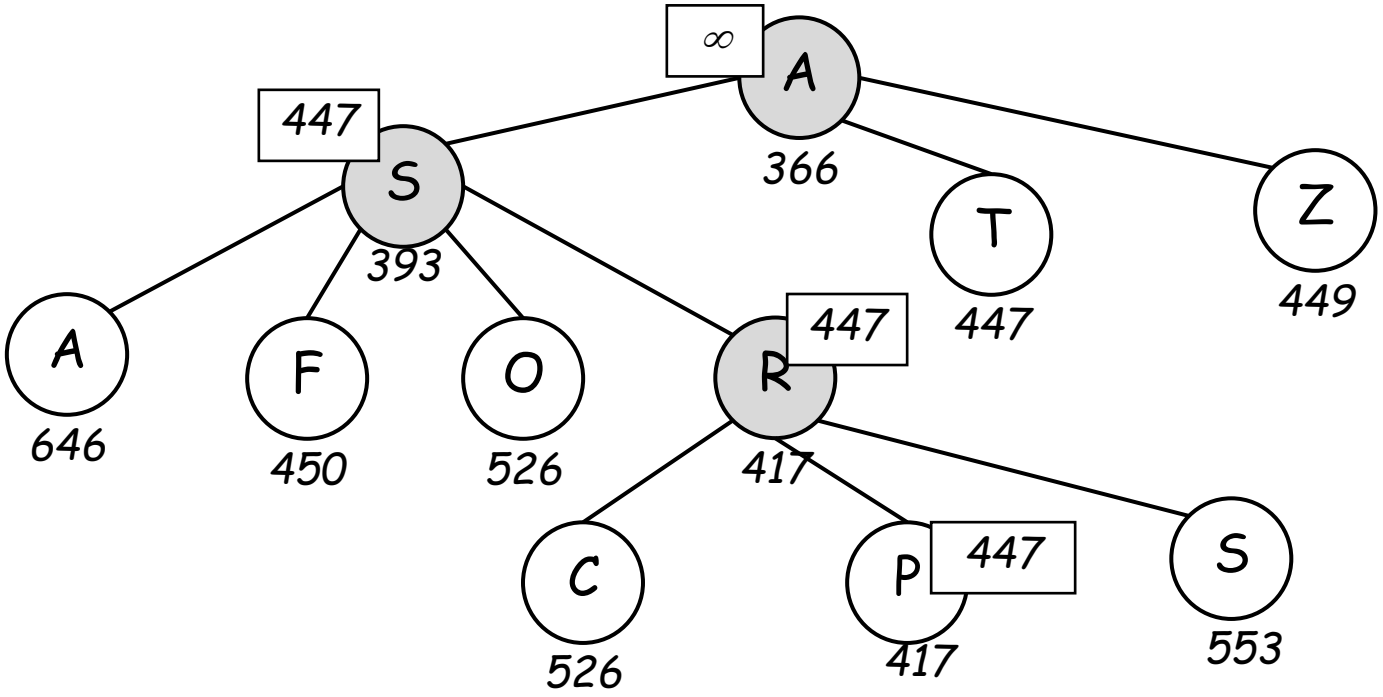
RBFS- Example



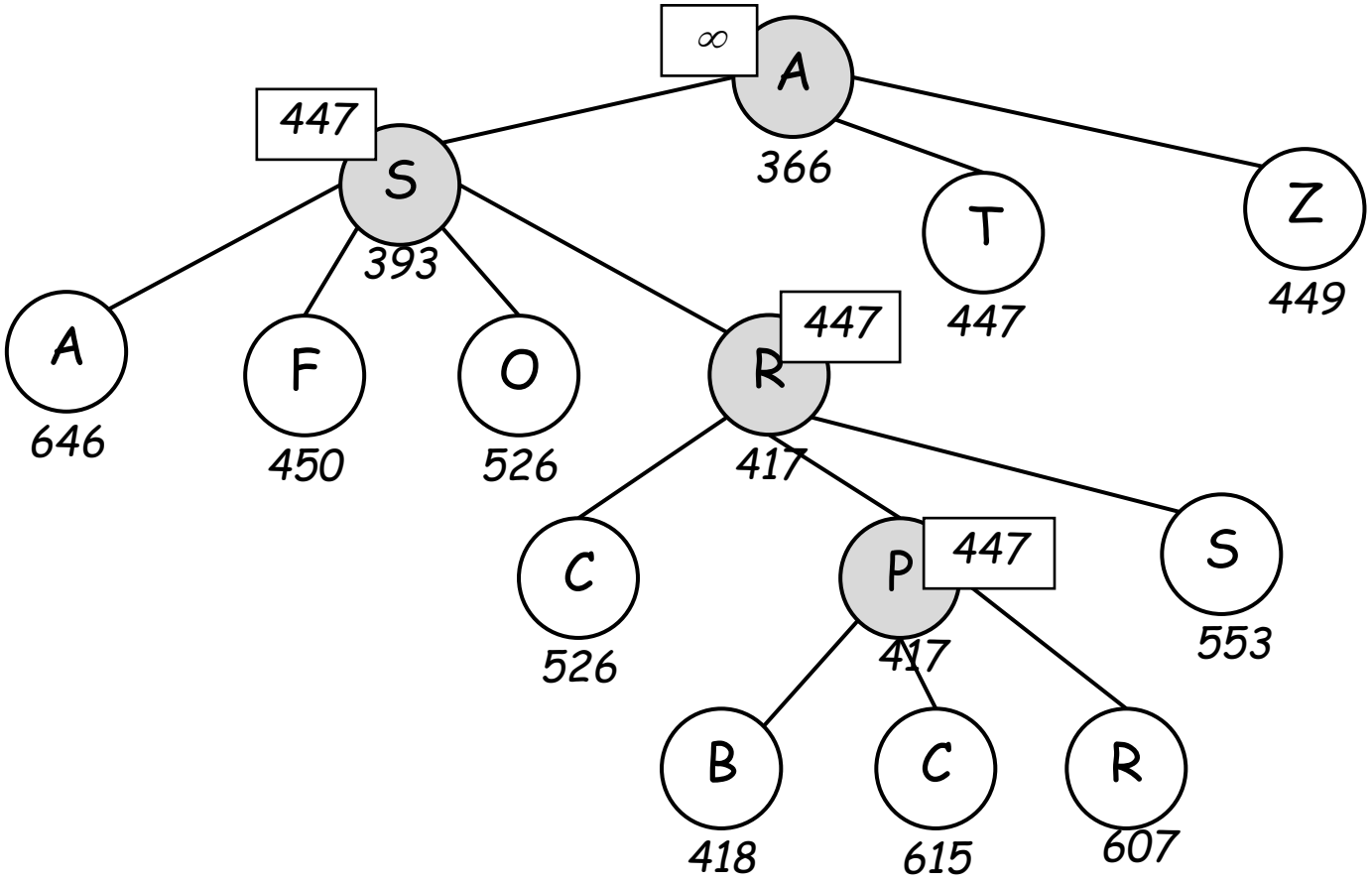
RBFS- Example



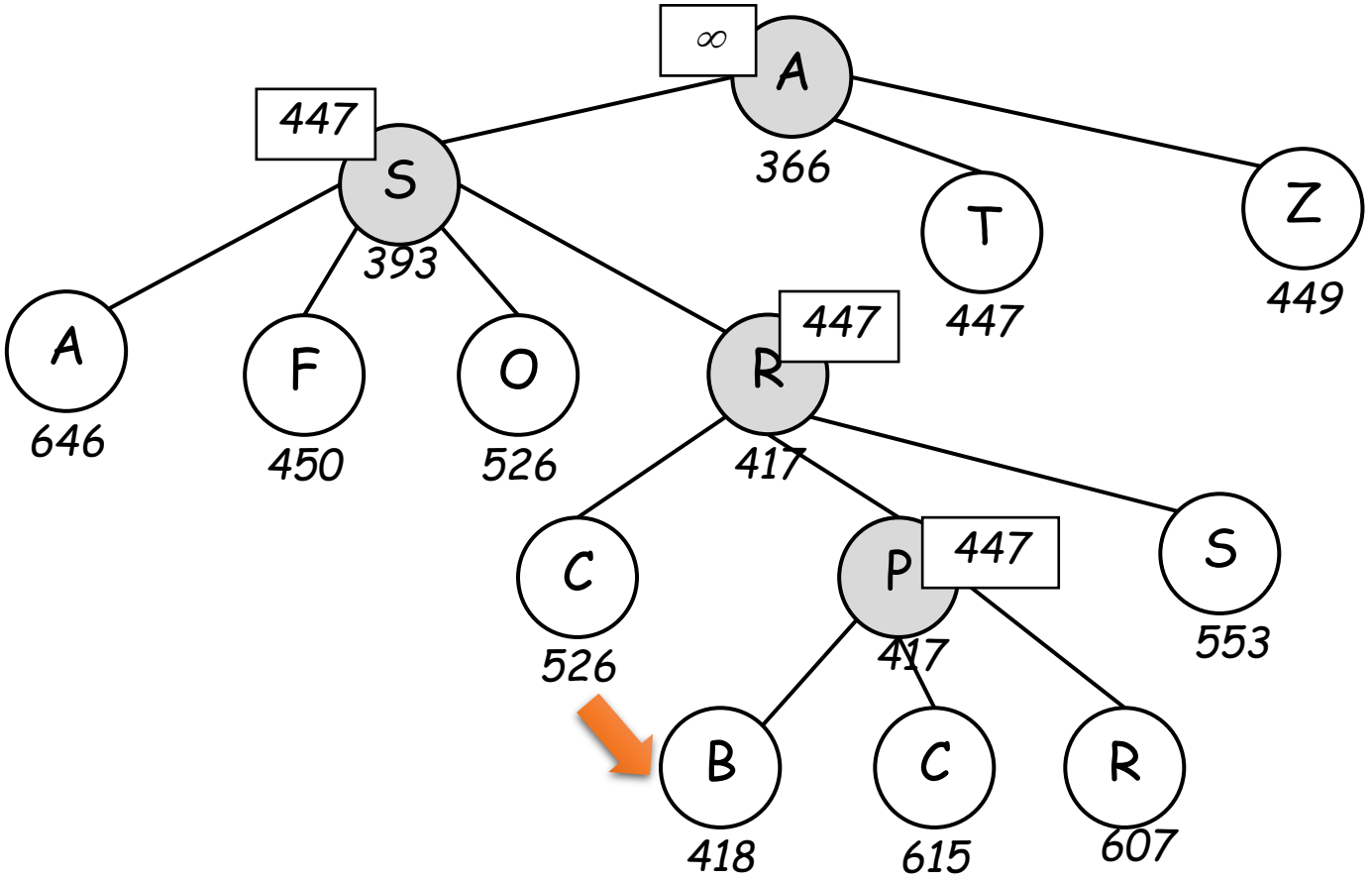
RBFS- Example



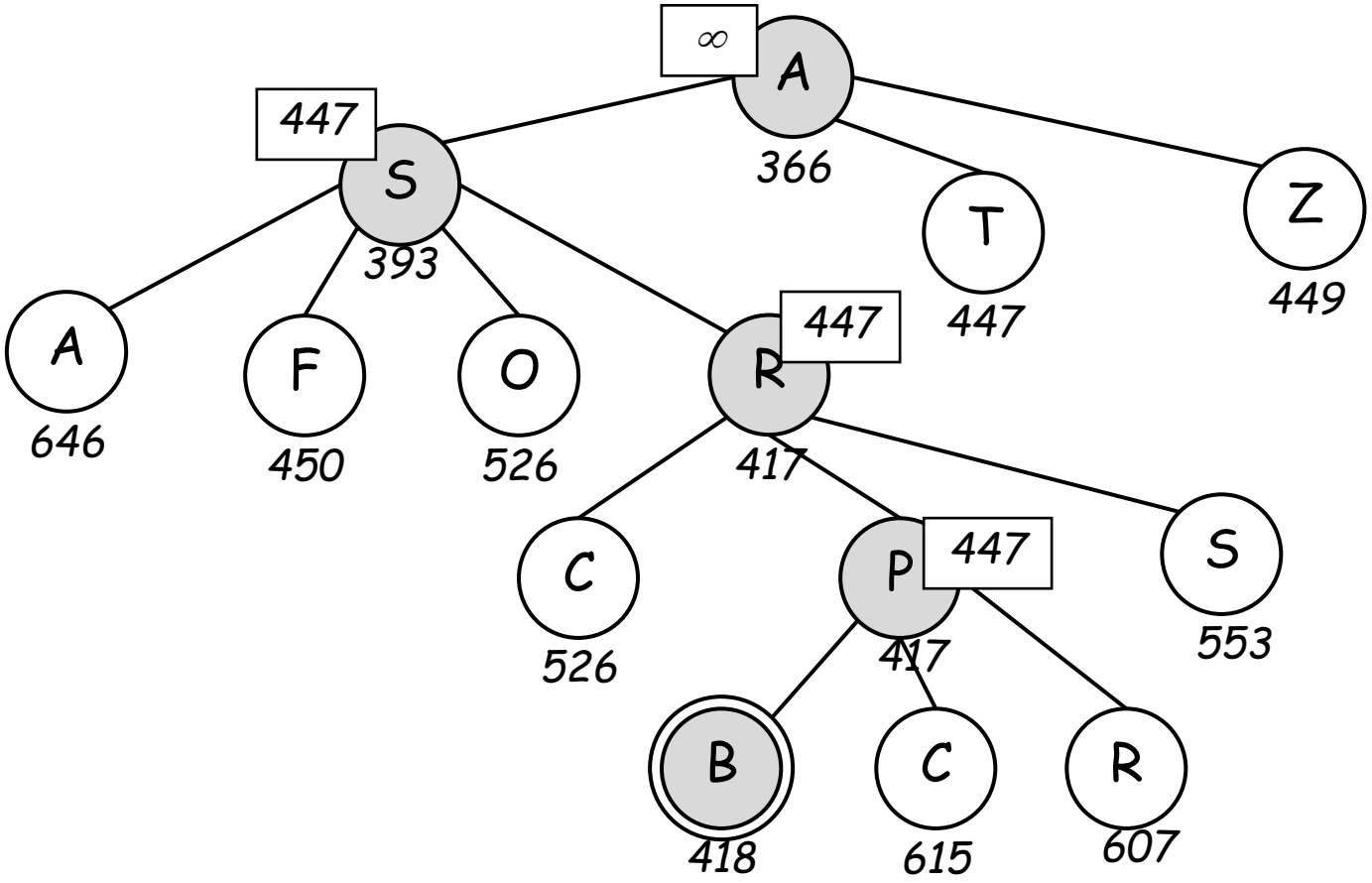
RBFS- Example



RBFS- Example



RBFS- Example



Properties of RBFS

- Complete and optimal
  - If h is admissible
- Time?
  - Depends both on the accuracy of the heuristic function and on how often the best path changes as nodes are expanded.
- Space?
  - Linear in the depth of the deepest optimal solution

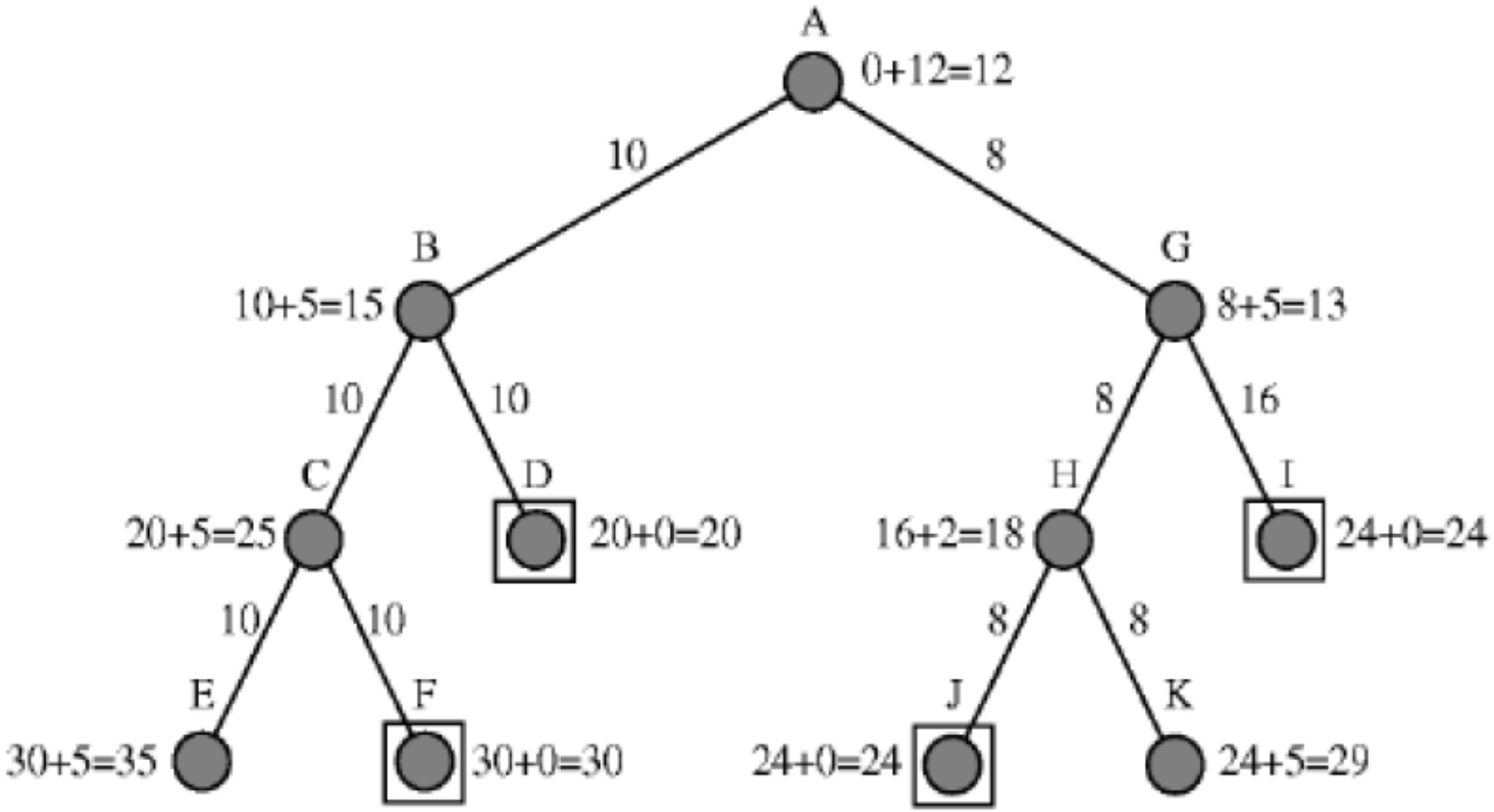
→ IDA* and RBFS suffer from using too little memory

SMA*

- We want to use all available memory
- There are two algorithms that do this:
  - MA* (Memory-bounded A*)
  - SMA* (Simplified MA*)
- SMA*
  - Proceeds just like A*, expanding the best leaf until memory is full
  - Then drops the worst leaf node
    - The one with the highest f-value
    - If two nodes are the worst, then drop the oldest one
  - Like RBFS, SMA* then backs up the value of the forgotten node to its parent

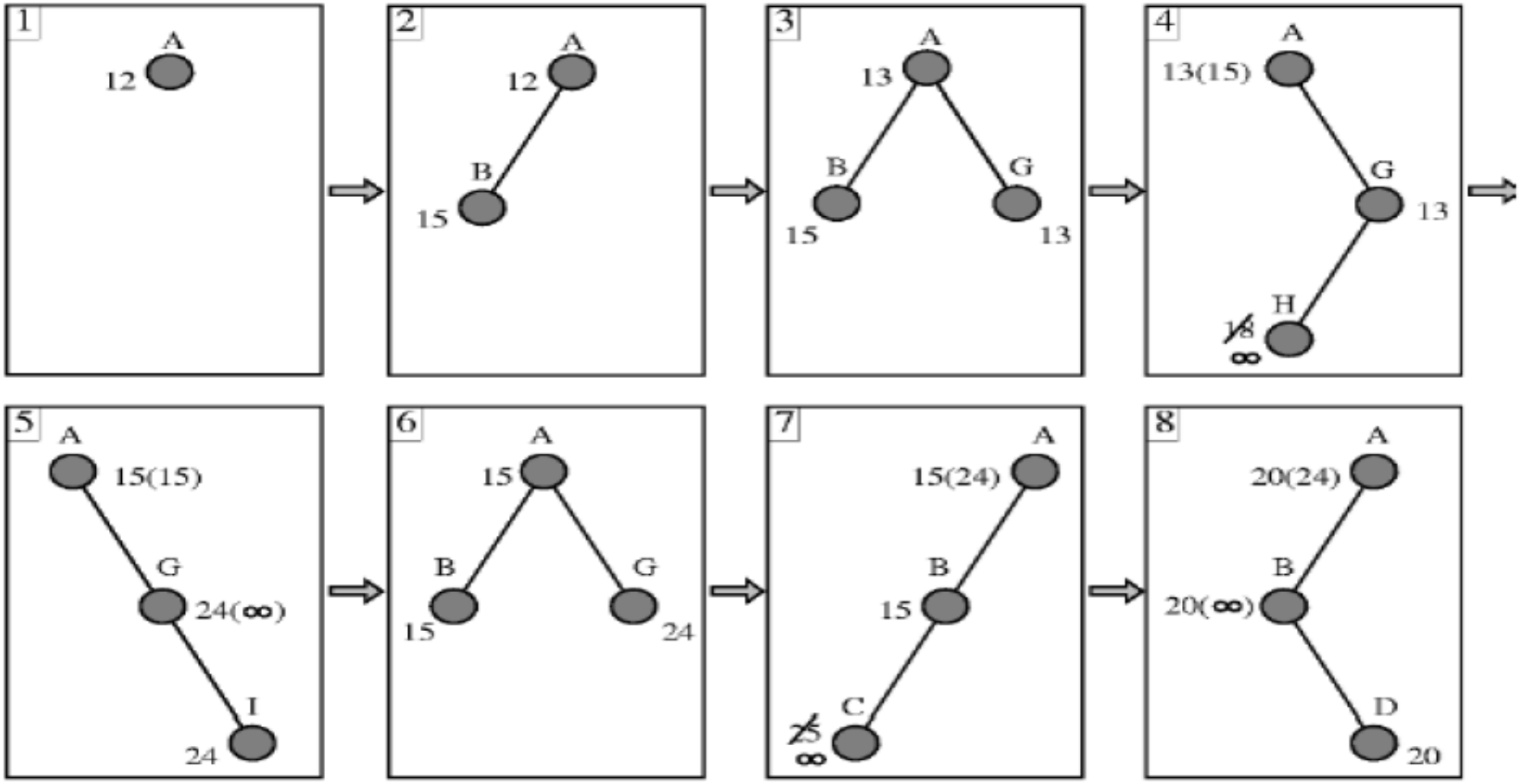
SMA*- Example

- Memory with a capacity of 3 nodes



SMA*- Example

- Memory with a capacity of 3 nodes



Properties of SMA*

- Complete
  - If there is any reachable solution
    - if d, the depth of the shallowest goal node, is less than the memory size (expressed in nodes)
- Optimal
  - If any optimal solution is reachable
    - Otherwise, it returns the best reachable solution
- In practical terms, SMA* is a fairly robust choice for finding optimal solutions
  - On very hard problems, however, it will often be the case that SMA* is forced to switch back and forth continually among many candidate solution paths, only a small subset of which can fit in memory. (This resembles the problem of thrashing in disk paging systems.)