3010

ARTIFICIAL INTELLIGENCE

Lecture 3 A* search

Masashi Shimbo

2019-05-15

Today's agenda

- Heuristic evaluation function
- The A* algorithm

Heuristic search ヒューリスティック探索

Heuristic search: Motivation ヒューリスティック探索: 動機

Even if we have some “knowledge” about a given problem, Dijkstra’s algorithm doesn’t have a means to take advantage of it.

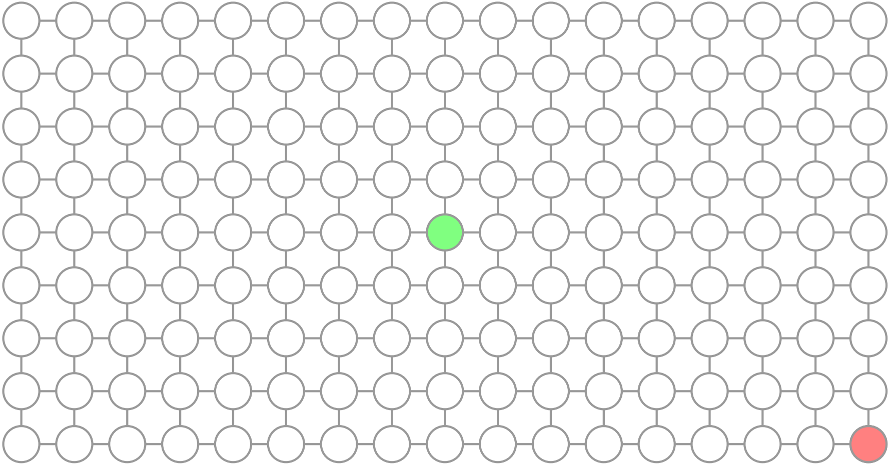
与えられた問題に対するなんらかの事前知識があっても, ダイクストラ法では活用することができない

Here is a motivating example…

たとえば…

Even if you know the goal state is located at the lower right corner… 目標

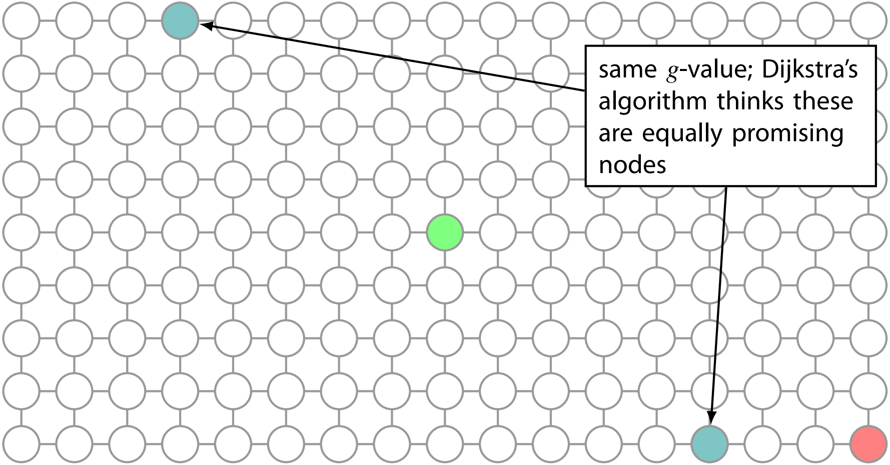
節点が右下隅にあることがわかっていても…



Dijkstra's algorithm does not take this information into account ダイクスト

Even if you know the goal state is located at the lower right corner… 目標

節点が右下隅にあることがわかっていても…



Dijkstra's algorithm does not take this information into account ダイクスト

Heuristic search ヒューリスティック探索

Also called “informed” search 「情報付き」探索とも呼ばれる

Search using domain- (problem-)specific **knowledge**, or **heuristics**

解きたい問題に関する「知識」/「ヒューリスティック」を活用した探索

But what kind of “knowledge”?

では, どんな「知識」が活用できる?

Heuristic evaluation function $h(v)$

It is assumed that the knowledge about the problem is given as a form of **heuristic evaluation function** $h(v)$

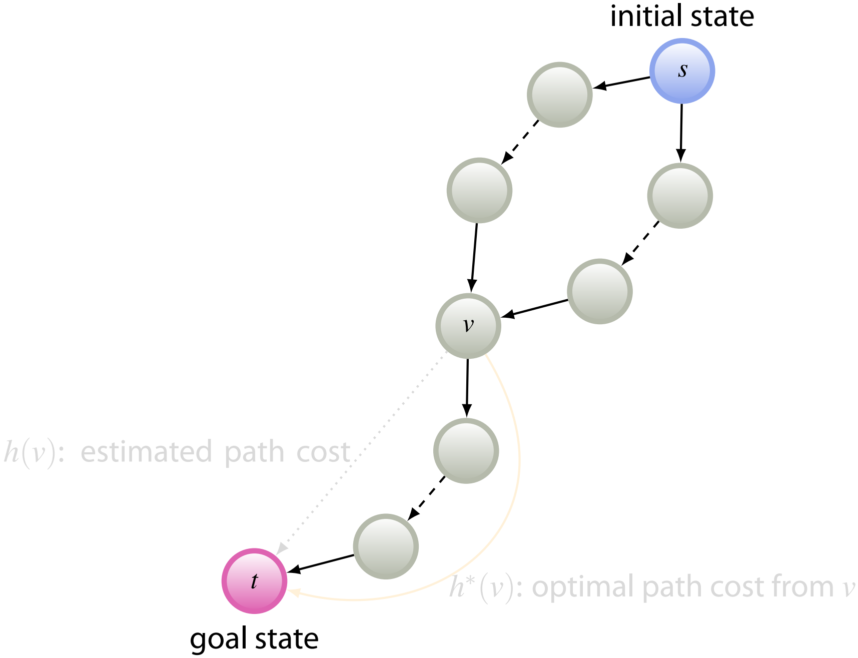
問題に関する知識は「ヒューリスティック評価関数」$h(v)$ という形で与えられると仮定する

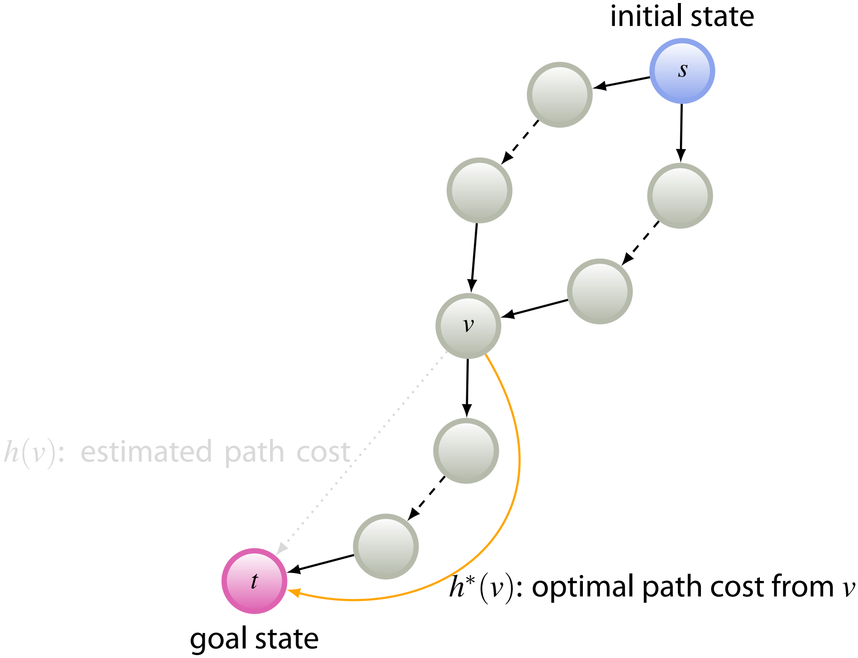
Sometimes simply called a **heuristic function**, or **heuristic**

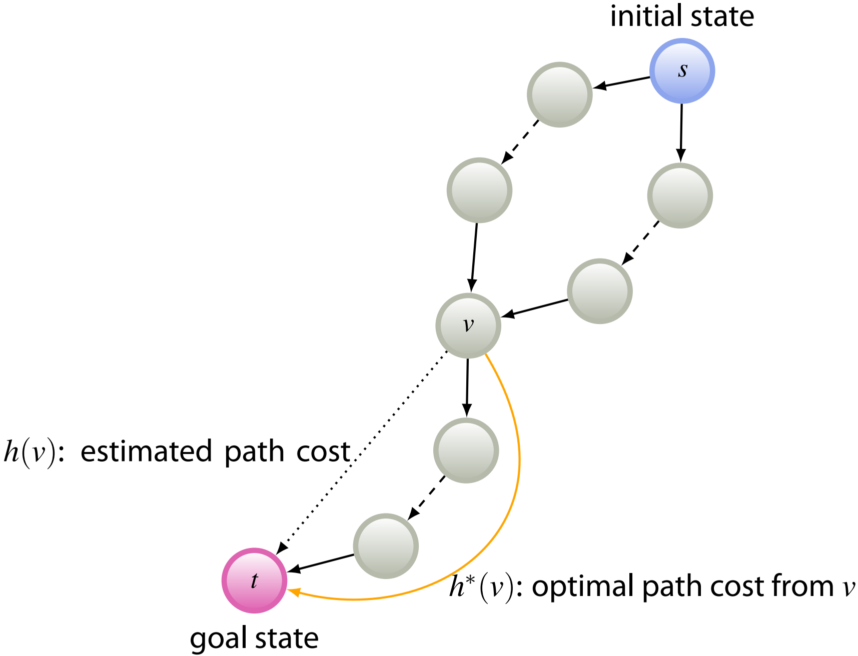
単に「ヒューリスティック関数」とか「ヒューリスティック」と呼ばれることもある

$h(v)$ = **estimated** cost of the cheapest path from node v to a terminal node

$h(v)$ の意味は「節点 v から一番近い (= コストが低い) 目標節点までの経路コストの**見積もり**」


initial state
s
v
h(v): estimated path cost
t
h*(v): optimal path cost from v
goal state


initial state
s
v
h(v): estimated path cost
t
goal state
h*(v): optimal path cost from v


initial state
s
v
h(v): estimated path cost
t
goal state
h*(v): optimal path cost from v

Admissible heuristic 適格なヒューリスティック関数

An important class of heuristic function

A heuristic function h is said to be **admissible**

$$\Updownarrow$$

For every node v, $h(v)$ never overestimates the actual cheapest cost $h^*(v)$ to reach a goal from v

$$\Updownarrow$$

$$h(v) \leq h^*(v) \quad \text{for every node } v.$$

$$\Updownarrow$$

h gives optimistic estimates of actual cost h^*.

Admissible heuristics and the A* algorithm

適格なヒューリスティック関数と A*

Why admissible heuristics matter?

なぜヒューリスティック関数の適格性が重要か?

… Because the A* algorithm (described later) is admissible (= guaranteed to find a shortest path) if the used heuristic function h is admissible.

用いるヒューリスティック関数 h が適格なら A* は適格 (最短経路を発見することが保証される)

How do we build an admissible heuristic?

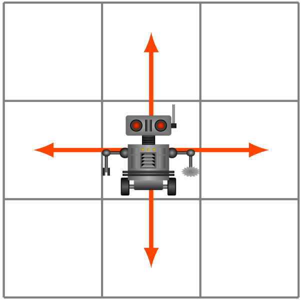
"Relaxed problems" approach:

1. Make an easier problem RP by removing constraints in the original problem OP.
2. Find the optimal solution for RP
3. Use the cost of the solution as heuristic h for OP.

The optimal solution (with cost h^*) for OP is also a solution of RP (but not necessarily optimal in RP; better solutions may exist)

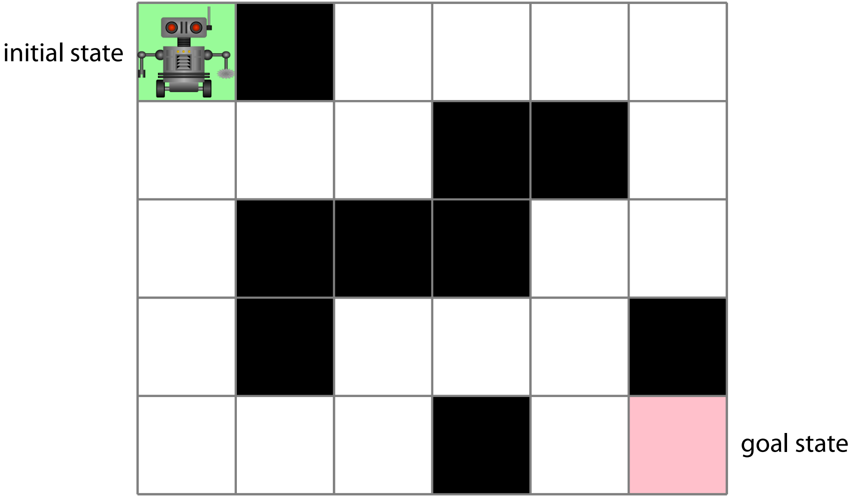
➥ $h \leq h^*$

Gridworld



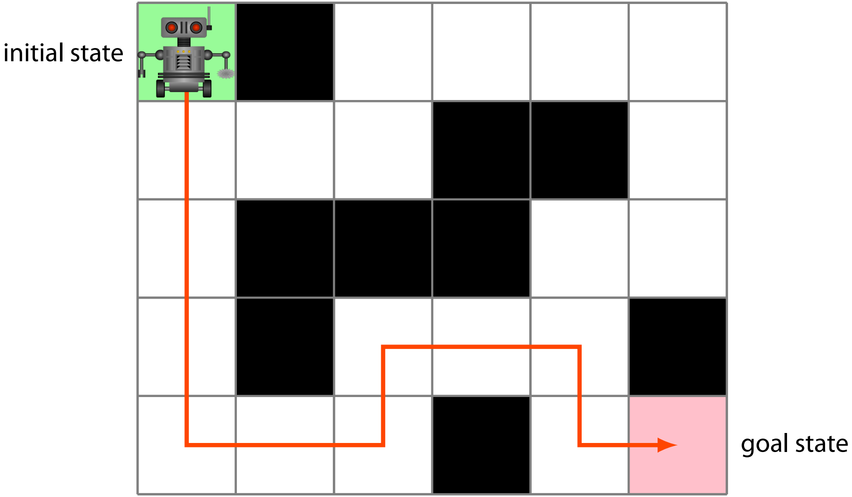
Gridworld

Many obstacles exist 障害物が多数存在



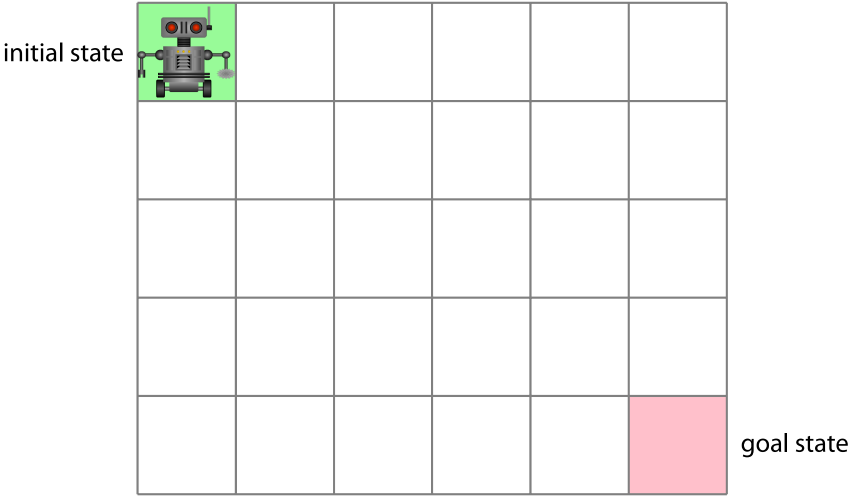
Gridworld

Many obstacles exist 障害物が多数存在



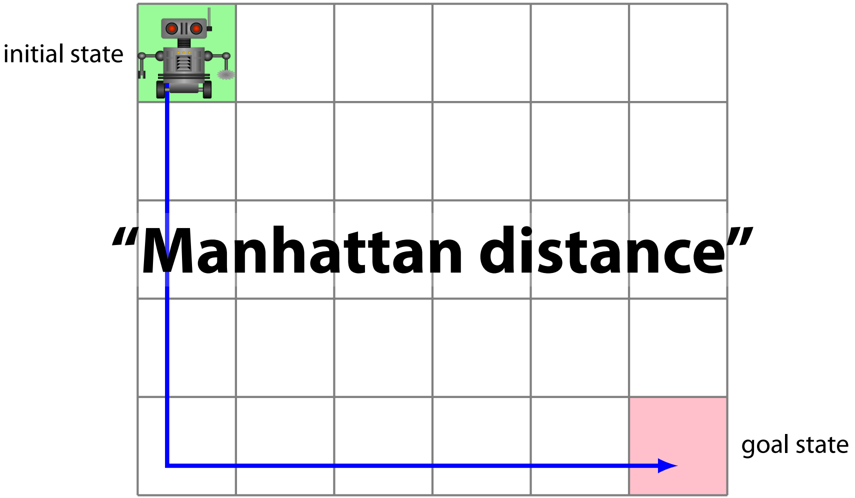
Gridworld: relaxed problem 緩和問題

Assume obstacles do not exist 障害物が一切存在しないと仮定

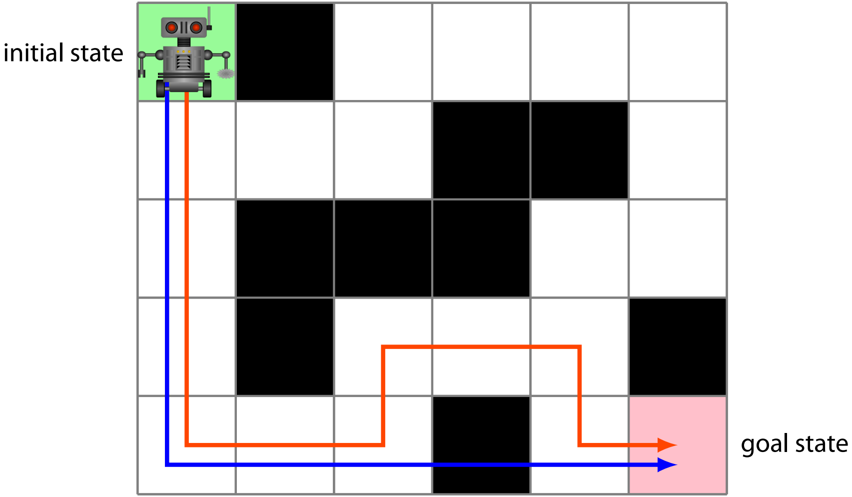


Gridworld: relaxed problem 緩和問題

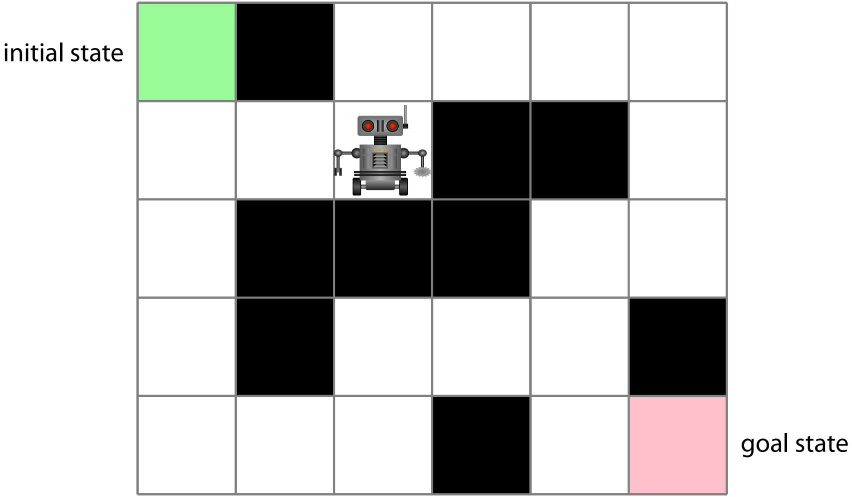
Assume obstacles do not exist 障害物が一切存在しないと仮定



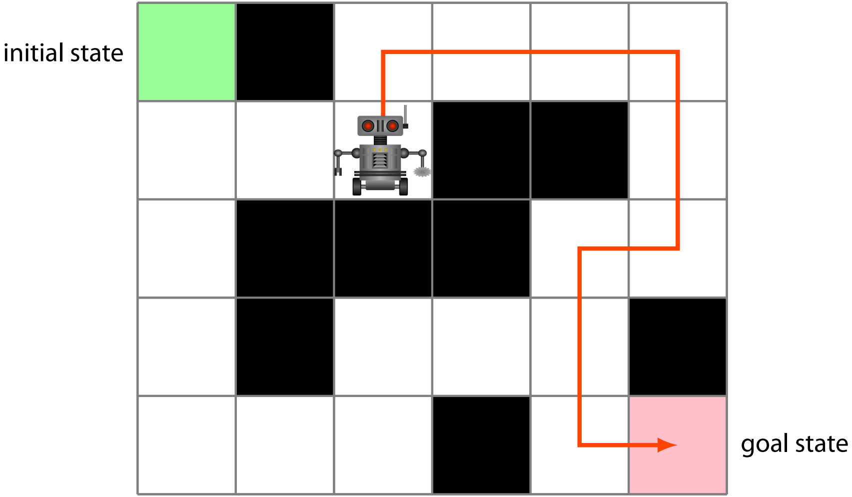
Gridworld: Manhattan-distance heuristic



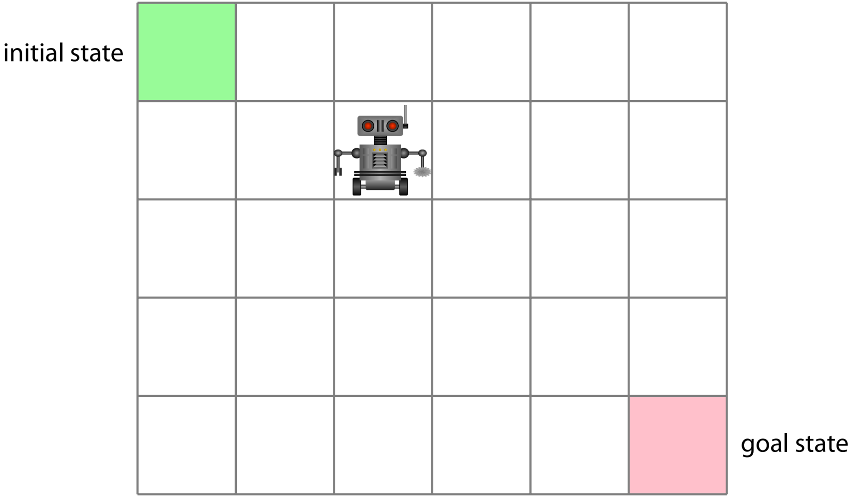
Gridworld: Manhattan-distance heuristic



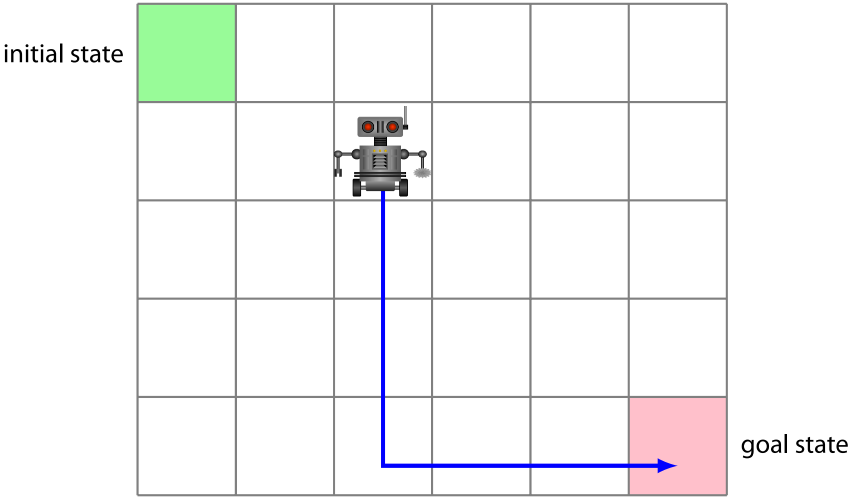
Gridworld: Manhattan-distance heuristic



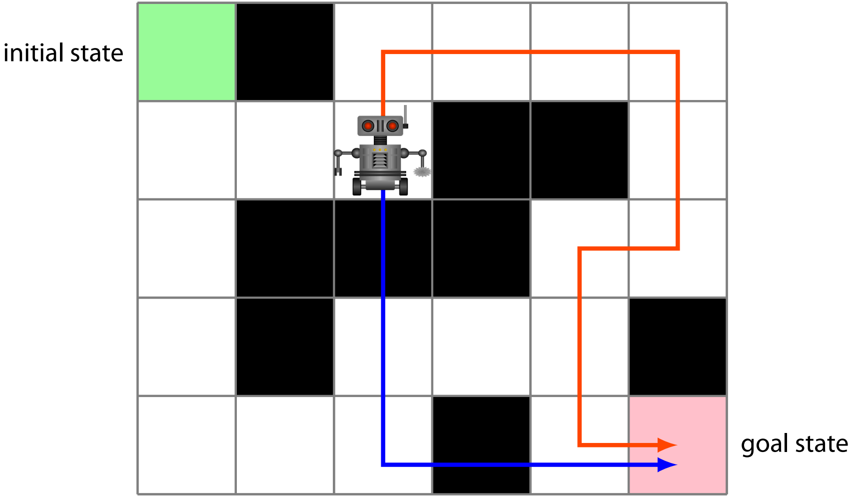
Gridworld: Manhattan-distance heuristic



Gridworld: Manhattan-distance heuristic



Gridworld: Manhattan-distance heuristic



Gridworld: Manhattan-distance heuristic

Computation is easy

v: arbitrary state

Let

(x, y)	coordinates of state v
(x_t, y_t)	coordinates of the goal state

Then,

$$h(v) = |x - x_t| + |y - y_t|$$

Admissible heuristic via relaxed problems: Another example

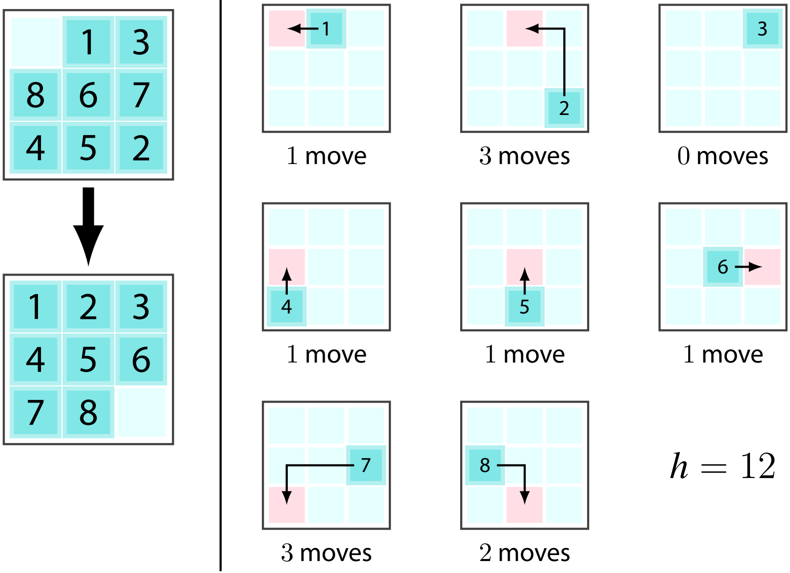
Manhattan distance heuristic for $(n^2 - 1)$-puzzles:

Sum of the Manhattan distance from each tile to its goal position.

$\updownarrow$

Relaxed problem: tiles can overlap with each other.

Eight-puzzle: Manhattan distance heuristic



$$h = 12$$

Requirement for heuristic functions: They must be efficiently computable

ヒューリスティック関数には「簡単に計算できること」が求められる

No use if computing a heuristic function takes equal or more time and space than actually searching the state space, no matter how accurate its estimates are.

どんな正確なヒューリスティック関数でも, 実際の探索を行う以上の時間やメモリが計算に必要なら, そもそも使う意味がない

Heuristic evaluation function h: Summary

- h associates a non-negative real number $h(v)$ to each state v
- $h(v)$ is an **estimate** of the actual cheapest cost $h^*(v)$ necessary to reach a goal state from state v
- h must be efficiently computable
- h is said to be **admissible** if $h(v) \leq h^*(v)$ for every state v
- One way to construct an admissible h is to consider relaxed problems

The A* algorithm A* アルゴリズム

The A* algorithm

[Hart, Nilsson & Raphael 1968]

Idea

Use the sum of

- the path cost from the initial state to state v
- the estimated cost from v to a goal state

to evaluate how “promising” it is to expand state v.

The A* algorithm

[Hart, Nilsson & Raphael 1968]

Idea

Use the sum of

- $g[v]$
- $h(v)$

to evaluate how "promising" it is to expand state v.

Evaluation function $f[v]$ of A*

$$f[v] = g[v] + h(v)$$

where

$g[v]$ tentative minimum cost from the initial state to state v s から v への, これまで見つかった経路のなかで最小のコスト

$h(v)$ estimated cost from state v to the nearest goal state v から最も近い目標節点への経路コストの見積もり

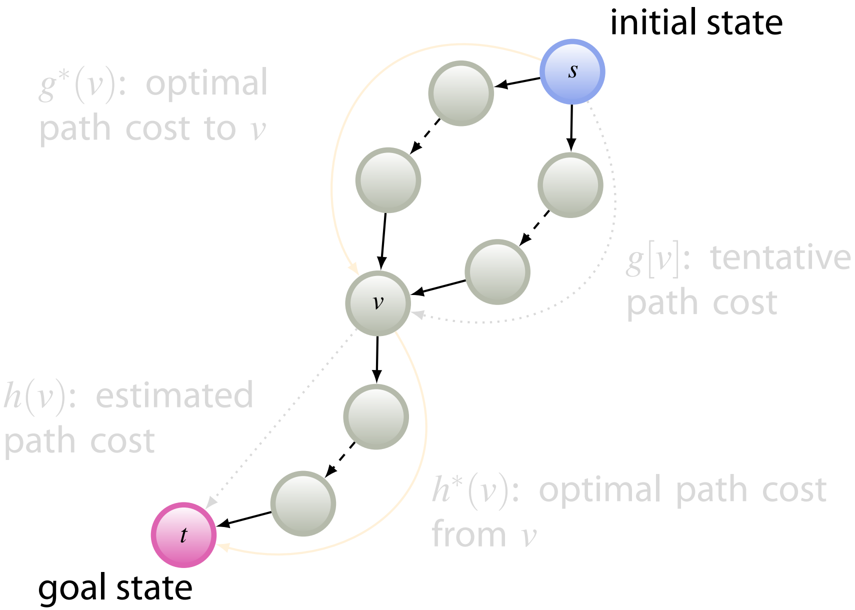
Idea:

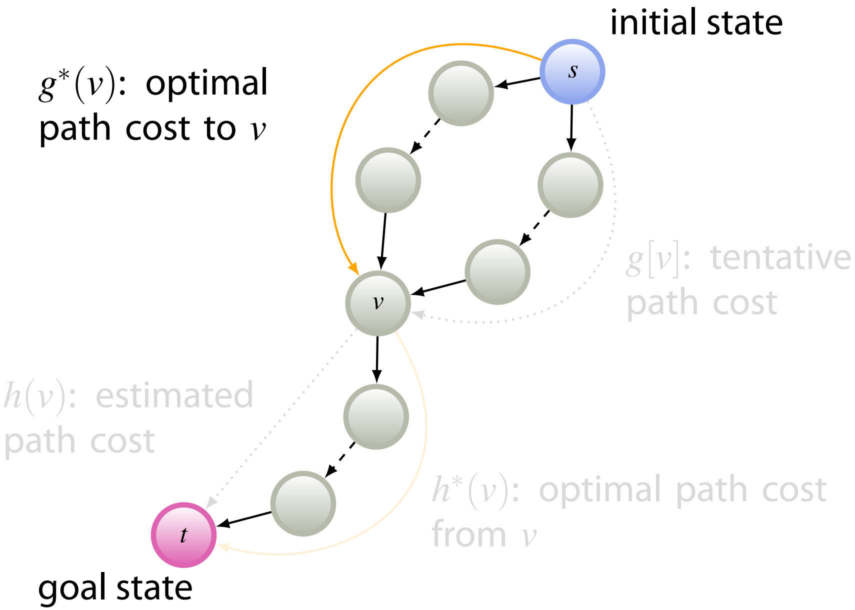
$f[v]$ smaller $\leftrightarrow$ v more promising

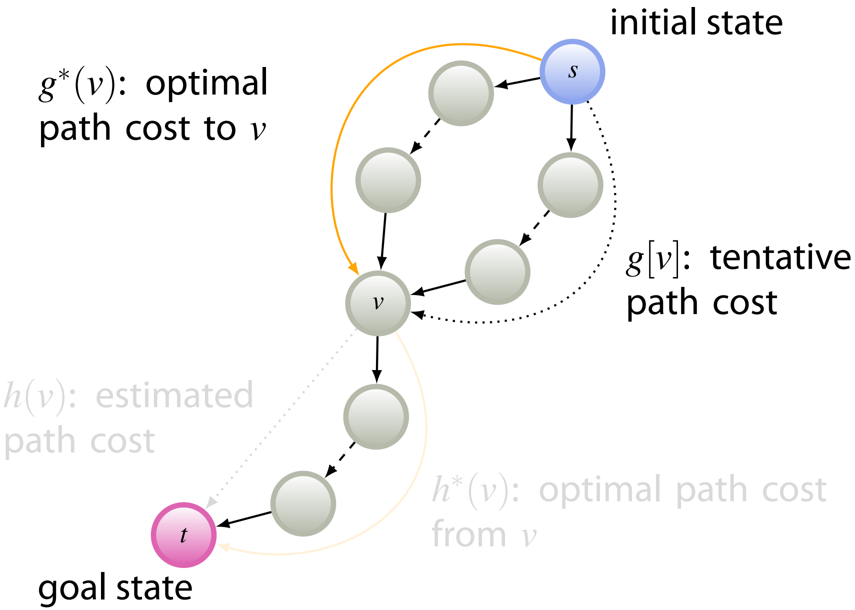
f が小さい節点ほど, より有望だ, とみなす

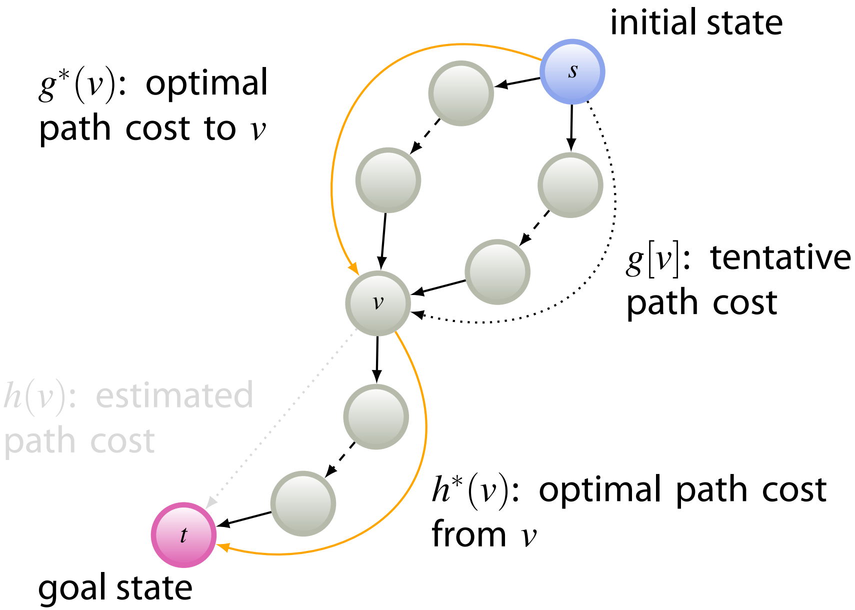
- $g[v]$: value may get updated if a better path from s to v is found later
- $h(v)$: once computed, the value will not change

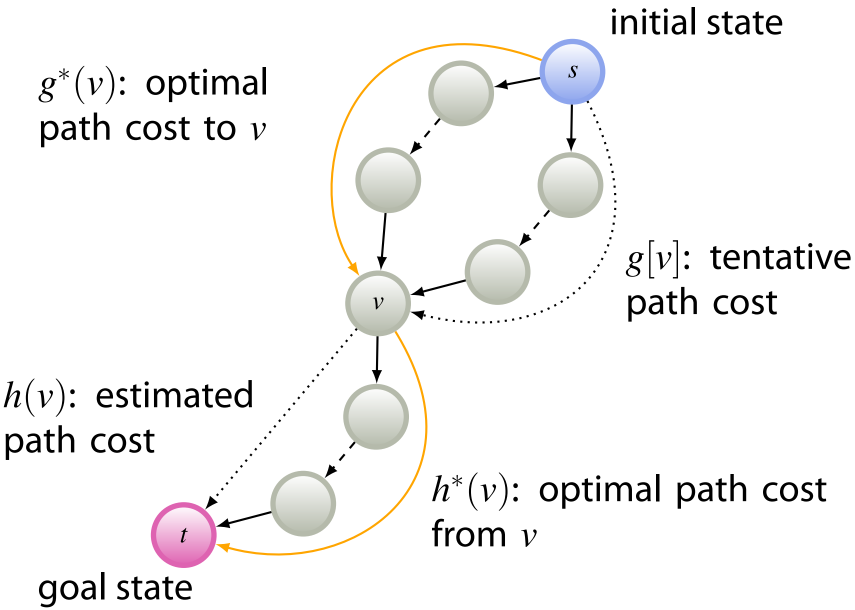
- $g[v]$: upper-bound of the optimal cost $g^*(v)$ (from s to v)
- $h(v)$: lower-bound of the optimal cost $h^*(v)$ (from v to goal t) provided that $h(v)$ is admissible.


initial state
$g^*(v)$: optimal path cost to v
s
$g[v]$: tentative path cost
v
$h(v)$: estimated path cost
$h^*(v)$: optimal path cost from v
t
goal state


initial state
s
$g^*(v)$: optimal path cost to v
$g[v]$: tentative path cost
v
$h(v)$: estimated path cost
$h^*(v)$: optimal path cost from v
t
goal state


initial state
$g^*(v)$: optimal path cost to v
s
$g[v]$: tentative path cost
v
$h(v)$: estimated path cost
$h^*(v)$: optimal path cost from v
t
goal state


initial state
$g^*(v)$: optimal path cost to v
s
$g[v]$: tentative path cost
v
$h(v)$: estimated path cost
$h^*(v)$: optimal path cost from v
t
goal state


initial state
s
$g^*(v)$: optimal path cost to v
$g[v]$: tentative path cost
v
$h(v)$: estimated path cost
$h^*(v)$: optimal path cost from v
t
goal state

A* subsumes Dijkstra's shortest path algorithm as a special case

A* reduces to Dijkstra's algorithm if $h(v) = 0$ for every node v.

The A* algorithm

The algorithm is identical to Dijkstra's, except

- OPEN is a priority queue with priority function $f[v] = g[v] + h(v)$ (not $g[v]$).
- For each generated node v, $f[v]$ is recorded along with $g[v]$.
- Nodes in CLOSED can be **re-opened**.

The A* algorithm

The algorithm is identical to Dijkstra's, except

- OPEN is a priority queue with priority function $f[v] = g[v] + h(v)$ (not $g[v]$).
- For each generated node v, $f[v]$ is recorded along with $g[v]$.
- Nodes in CLOSED can be **re-opened**.

The A* algorithm

The algorithm is identical to Dijkstra's, except

- OPEN is a priority queue with priority function $f[v] = g[v] + h(v)$ (not $g[v]$).
- For each generated node v, $f[v]$ is recorded along with $g[v]$.
- Nodes in CLOSED can be **re-opened**.

Priority queue

Two functions for manipulating priority queue P_f:

$\mathrm{Insert}_f(P_f, v)$

Put item v in P_f

$\mathrm{DeleteMin}_f(P_f)$

Remove and return an item with the minimum f-value from P_f.
Thus, the returned item v is the one with

$$v = \operatorname*{argmin}_{u \in P_f} f[u]$$

before removal

How a node changes its status in A*

Closed nodes can be re-opened

Status	Description
Unexplored	
$\downarrow$	(when a parent node is expanded)
OPEN	the node is generated but not expanded
$\downarrow\uparrow$	(when the node itself is expanded)
CLOSED	the node is generated and expanded

Dijkstra's shortest path algorithm

Main routine

OPEN $\leftarrow$ **new** PriorityQueue_g
$g[s] \leftarrow 0$
$\text{Insert}_g(\text{OPEN}, s)$ # OPEN: set of states generated but not expanded
CLOSED $\leftarrow \emptyset$ # CLOSED: set of expanded states
loop do
    if IsEmpty(OPEN) **then return** "failure"
    $v \leftarrow \text{DeleteMin}_g(\text{OPEN})$ # choose a node with the smallest g
    CLOSED $\leftarrow$ CLOSED $\cup \{v\}$ # put v in CLOSED
    if IsGoal(v) **then return** Solution(v, s)
    Expand(v)

The A* algorithm

Main routine

$\text{OPEN} \leftarrow$ **new** PriorityQueue_f # priority is based on $f = g + h$

$g[s] \leftarrow 0;\ f[s] \leftarrow h(s)$ # $f[s] = g[s] + h(s) = 0 + h(s) = h(s)$

$\text{Insert}_f(\text{OPEN}, s)$ # OPEN: set of states generated but not expanded

$\text{CLOSED} \leftarrow \emptyset$ # CLOSED: set of expanded states

loop do

  if $\text{IsEmpty}(\text{OPEN})$ **then return** "failure"

  $v \leftarrow \text{DeleteMin}_f(\text{OPEN})$ # choose a node with the smallest f

  $\text{CLOSED} \leftarrow \text{CLOSED} \cup \{v\}$

  if $\text{IsGoal}(v)$ **then return** $\text{Solution}(v, s)$

  $\text{Expand}(v)$

procedure $\text{Expand}(v)$ **for Dijkstra's algorithm**

foreach $u \in \text{Succ}(v)$ **do**
  if $u \notin \text{OPEN} \cup \text{CLOSED}$ **then** # if u is a new state
    Reserve memory for $g[u]$, $\text{Parent}[u]$
    $g[u] \leftarrow g[v] + c(v, u)$ # memorize $g[u]$
    $\text{Parent}[u] \leftarrow v$ # also memorize Parent
    $\text{Insert}_g(\text{OPEN}, u)$
  else if $u \in \text{OPEN}$ **then** # "relax" edge (v, u) if $u \in \text{OPEN}$
    if $g[v] + c(v, u) < g[u]$ **then** # if it is better than the current path
      $g[u] \leftarrow g[v] + c(v, u)$ # update g if path through (v, u) is shorter
      $\text{Parent}[u] \leftarrow v$ # update Parent, too

procedure $\mathrm{Expand}(v)$ for A* algorithm

foreach $u \in \mathrm{Succ}(v)$ **do**

  if $u \notin \mathrm{OPEN} \cup \mathrm{CLOSED}$ **then** # if u is a new state

    Reserve memory for $g[u]$, $f[u]$, and $\mathrm{Parent}[u]$

    $g[u] \leftarrow g[v] + c(v,u)$; $f[u] \leftarrow g[u] + h(u)$ # memorize $f[u]$ as well as $g[u]$

    $\mathrm{Parent}[u] \leftarrow v$ # also memorize Parent

    $\mathrm{Insert}_f(\mathrm{OPEN}, u)$

  else if $u \in \mathrm{OPEN}$ **then** # "relax" edge (v,u) if $u \in \mathrm{OPEN}$

    if $g[v] + c(v,u) < g[u]$ **then** # if path through (v,u) is shorter

      $g[u] \leftarrow g[v] + c(v,u)$; $f[u] \leftarrow g[u] + h(u)$ # update g and f

      $\mathrm{Parent}[u] \leftarrow v$ # update Parent, too

  else # if $u \in \mathrm{CLOSED}$, "relax" edge (v,u) and re-open u if necessary

    if $g[v] + c(v,u) < g[u]$ **then** # if a cheaper path is found

      $g[u] \leftarrow g[v] + c(v,u)$; $f[u] \leftarrow g[u] + h(u)$ # update g and f

      $\mathrm{Parent}[u] \leftarrow v$ # update Parent, too

      $\mathrm{CLOSED} \leftarrow \mathrm{CLOSED} \setminus \{u\}$ # then take u out of CLOSED

      $\mathrm{Insert}_f(\mathrm{OPEN}, u)$ # and put it back into OPEN

A* and Dijkstra's algorithm: Difference in $\mathrm{Expand}(v)$

Case	Dijkstra	A*
$u \notin \mathrm{OPEN}$ nor $u \notin \mathrm{CLOSED}$	$g[u] \leftarrow g[v] + c(v,u)$ $\mathrm{Insert}_g(\mathrm{OPEN}, u)$ $\mathrm{Parent}[u] \leftarrow v$	$g[u] \leftarrow g[v] + c(v,u)$ $f[u] \leftarrow g[u] + h(u)$ $\mathrm{Insert}_f(\mathrm{OPEN}, u)$ $\mathrm{Parent}[u] \leftarrow v$
$u \in \mathrm{OPEN}$	**if** $g[v] + c(v,u) < g[u]$ **then** $\quad g[u] \leftarrow g[v] + c(v,u)$ $\quad \mathrm{Parent}[u] \leftarrow v$	**if** $g[v] + c(v,u) < g[u]$ **then** $\quad g[u] \leftarrow g[v] + c(v,u)$ $\quad f[u] \leftarrow g[u] + h(u)$ $\quad \mathrm{Parent}[u] \leftarrow v$
$u \in \mathrm{CLOSED}$	Do nothing (always $g[u] \leq g[v] + c(v,u)$)	**if** $g[v] + c(v,u) < g[u]$ **then** $\quad g[u] \leftarrow g[v] + c(v,u)$ $\quad f[u] \leftarrow g[u] + h(u)$ $\quad \mathrm{CLOSED} \leftarrow \mathrm{CLOSED} \setminus \{u\}$ $\quad \mathrm{Insert}_f(\mathrm{OPEN}, u)$ $\quad \mathrm{Parent}[u] \leftarrow v$

v = node just expanded / u = a successor of v

Properties of A*

Assumptions

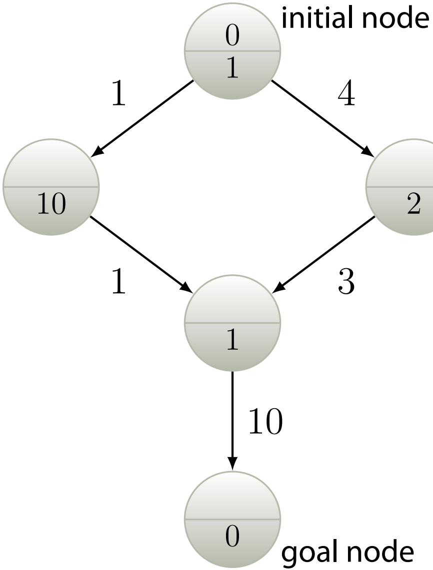
- At least one solution (path from the initial state to a goal state) exists
- function $h(\cdot)$ is admissible

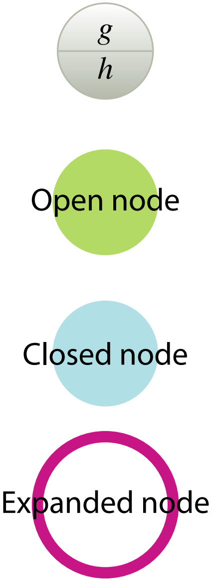
Completeness

A* never fails to find a solution

Admissibility

The solution found by A* is optimal


0
1
initial node
1
4
10
2
1
3
1
10
0
goal node


g
h
Open node
Closed node
Expanded node

Exercise

Trace the execution of the A* algorithm on this graph. In particular,

1. Trace which nodes are on OPEN and which are on CLOSED
2. Compute the g-value of each node at each stage
3. In what order are nodes expanded?
4. How many iterations are necessary before termination?

If you still have time left, trace the behavior of Dijkstra's algorithm (i.e., by setting $h = 0$ for all nodes) on the same graph

A* may reopen closed nodes

Cf. Dijkstra's algorithm never reopens a node.

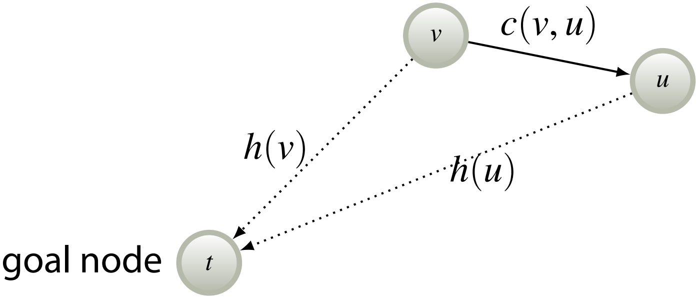
Is there a class of heuristic functions h such that A* does not open a node more than once?

➥ **Monotone heuristic functions**

Monotone heuristic function

Heuristic function h is said to be **monotone** if

- $h(v) \leq c(v, u) + h(u)$ holds for **every** edge (v, u)
- $h(t) = 0$ for every goal node t



Monotonicity and admissibility

Monotonicity implies admissibility

h is monotone $\to$ h is admissible

A* guided by monotone heuristic function h **never** re-opens a node

A* algorithm

Main routine

$\text{OPEN} \leftarrow$ **new** PriorityQueue_f # priority is based on f
$g[s] \leftarrow 0$
$f[s] \leftarrow h(s)$ # $f[s] = g[s] + h(s) = h(s)$
$\text{Insert}_f(\text{OPEN}, s)$ # OPEN: set of states generated but not expanded
$\text{CLOSED} \leftarrow \emptyset$ # CLOSED: set of expanded states
loop do
  if $\text{IsEmpty}(\text{OPEN})$ **then return** "failure"
  $v \leftarrow \text{DeleteMin}_f(\text{OPEN})$ # choose a node with the smallest f
  $\text{CLOSED} \leftarrow \text{CLOSED} \cup \{v\}$
  if $\text{IsGoal}(v)$ **then return** $\text{Solution}(v, s)$
  $\text{Expand}(v)$

A* algorithm with monotone h

Main routine — No change from the original A*

$\text{OPEN} \leftarrow$ **new** PriorityQueue_f # priority is based on f

$g[s] \leftarrow 0$

$f[s] \leftarrow h(s)$ # $f[s] = g[s] + h(s) = h(s)$

$\text{Insert}_f(\text{OPEN}, s)$ # OPEN: set of states generated but not expanded

$\text{CLOSED} \leftarrow \emptyset$ # CLOSED: set of expanded states

loop do

  if $\text{IsEmpty}(\text{OPEN})$ **then return** "failure"

  $v \leftarrow \text{DeleteMin}_f(\text{OPEN})$ # choose a node with the smallest f

  $\text{CLOSED} \leftarrow \text{CLOSED} \cup \{v\}$

  if $\text{IsGoal}(v)$ **then return** $\text{Solution}(v, s)$

  $\text{Expand}(v)$

Procedure $\text{Expand}(v)$

for A* algorithm

foreach $u \in \text{Succ}(v)$ **do**
  if $u \notin \text{OPEN} \cup \text{CLOSED}$ **then** — # if u is a new state
    Reserve memory for $g[u]$, $f[u]$, and $\text{Parent}[u]$
    $g[u] \leftarrow g[v] + c(v,u);\; f[u] \leftarrow g[u] + h(u)$ — # memorize $f[u]$ as well as $g[u]$
    $\text{Parent}[u] \leftarrow v$ — # also memorize Parent
    $\text{Insert}_f(\text{OPEN}, u)$
  else if $u \in \text{OPEN}$ **then** — # “relax” edge (v,u) if $u \in \text{OPEN}$
    if $g[v] + c(v,u) < g[u]$ **then** — # if it gives a better path than the current one
      $g[u] \leftarrow g[v] + c(v,u);\; f[u] \leftarrow g[u] + h(u)$ — # update g and f
      $\text{Parent}[u] \leftarrow v$ — # update Parent, too
  else — # if $u \in \text{CLOSED}$, “relax” edge (v,u) and re-open u if necessary
    if $g[v] + c(v,u) < g[u]$ **then** — # if a cheaper path is found
      $g[u] \leftarrow g[v] + c(v,u);\; f[u] \leftarrow g[u] + h(u)$ — # update g and f
      $\text{Parent}[u] \leftarrow v$ — # update Parent
      $\text{CLOSED} \leftarrow \text{CLOSED} \setminus \{u\}$ — # take u out of CLOSED
      $\text{Insert}_f(\text{OPEN}, u)$ — # and put it back in OPEN

Procedure $\mathrm{Expand}(v)$

for A* algorithm when h is monotone

foreach $u \in \mathrm{Succ}(v)$ **do**
    if $u \notin \mathrm{OPEN} \cup \mathrm{CLOSED}$ **then** # if u is a new state
        Reserve memory for $g[u]$, $f[u]$, and $\mathrm{Parent}[u]$
        $g[u] \leftarrow g[v] + c(v,u);\ f[u] \leftarrow g[u] + h(u)$ # memorize $f[u]$ as well as $g[u]$
        $\mathrm{Parent}[u] \leftarrow v$ # also memorize Parent
        $\mathrm{Insert}_f(\mathrm{OPEN}, u)$
    else if $u \in \mathrm{OPEN}$ **then** # "relax" edge (v,u) if $u \in \mathrm{OPEN}$
        if $g[v] + c(v,u) < g[u]$ **then** # if it gives a better path than the current one
            $g[u] \leftarrow g[v] + c(v,u);\ f[u] \leftarrow g[u] + h(u)$ # update g and f
            $\mathrm{Parent}[u] \leftarrow v$ # update Parent, too
    else # if $u \in \mathrm{CLOSED}$, "relax" edge (v,u) and re-open u if necessary
        if $g[v] + c(v,u) < g[u]$ **then** # if a cheaper path is found
            $g[u] \leftarrow g[v] + c(v,u);\ f[u] \leftarrow g[u] + h(u)$ # update g and f
            $\mathrm{Parent}[u] \leftarrow v$ # update Parent
            $\mathrm{CLOSED} \leftarrow \mathrm{CLOSED} \setminus \{u\}$ # take u out of CLOSED
            $\mathrm{Insert}_f(\mathrm{OPEN}, u)$ # and put it back in OPEN

This test never succeeds if h is monotone

Procedure $\mathrm{Expand}(v)$

for A* algorithm when h is monotone

foreach $u \in \mathrm{Succ}(v)$ **do**
- **if** $u \notin \mathrm{OPEN} \cup \mathrm{CLOSED}$ **then** # if u is a new state
  - Reserve memory for $g[u]$, $f[u]$, and $\mathrm{Parent}[u]$
  - $g[u] \leftarrow g[v] + c(v,u);\ f[u] \leftarrow g[u] + h(u)$ # memorize $f[u]$ as well as $g[u]$
  - $\mathrm{Parent}[u] \leftarrow v$ # also memorize Parent
  - $\mathrm{Insert}_f(\mathrm{OPEN}, u)$
- **else if** $u \in \mathrm{OPEN}$ **then** # "relax" edge (v,u) if $u \in \mathrm{OPEN}$
  - **if** $g[v] + c(v,u) < g[u]$ **then** # if it gives a better path than the current one
    - $g[u] \leftarrow g[v] + c(v,u);\ f[u] \leftarrow g[u] + h(u)$ # update g and f
    - $\mathrm{Parent}[u] \leftarrow v$ # update Parent, too

Thus, this part can be safely removed if we know h is monotone for sure

Dijkstra and A* : Difference in $\mathrm{Expand}(v)$

Case	Dijkstra	A*
$u \notin \mathrm{OPEN}$ nor $u \notin \mathrm{CLOSED}$	$g[u] \leftarrow g[v] + c(v,u)$ $\mathrm{Insert}_g(\mathrm{OPEN}, u)$ $\mathrm{Parent}[u] \leftarrow v$	$g[u] \leftarrow g[v] + c(v,u)$ $f[u] \leftarrow g[u] + h(u)$ $\mathrm{Insert}_f(\mathrm{OPEN}, u)$ $\mathrm{Parent}[u] \leftarrow v$
$u \in \mathrm{OPEN}$	**if** $g[v] + c(v,u) < g[u]$ **then** $g[u] \leftarrow g[v] + c(v,u)$ $\mathrm{Parent}[u] \leftarrow v$	**if** $g[v] + c(v,u) < g[u]$ **then** $g[u] \leftarrow g[v] + c(v,u)$ $f[u] \leftarrow g[u] + h(u)$ $\mathrm{Parent}[u] \leftarrow v$
$u \in \mathrm{CLOSED}$	Do nothing (always $g[u] \leq g[v] + c(v,u)$)	**if** $g[v] + c(v,u) < g[u]$ **then** $g[u] \leftarrow g[v] + c(v,u)$ $f[u] \leftarrow g[u] + h(u)$ $\mathrm{Parent}[u] \leftarrow v$ $\mathrm{CLOSED} \leftarrow \mathrm{CLOSED} \setminus \{u\}$ $\mathrm{Insert}_f(\mathrm{OPEN}, u)$

$v =$ node just expanded / $u =$ a successor of v

Dijkstra and A* with monotone h: Difference in $\mathrm{Expand}(v)$

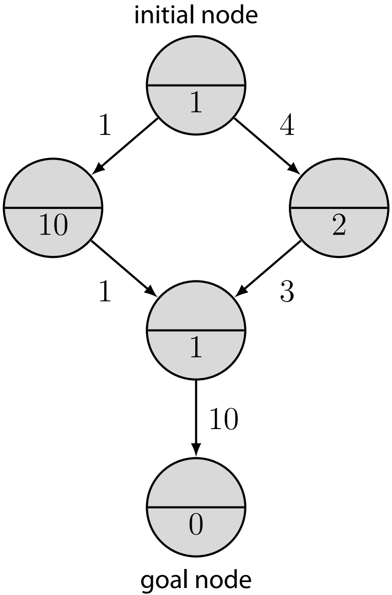
Case	Dijkstra	A*
$u \notin \mathrm{OPEN}$ nor $u \notin \mathrm{CLOSED}$	$g[u] \leftarrow g[v] + c(v,u)$ $\mathrm{Insert}_g(\mathrm{OPEN}, u)$ $\mathrm{Parent}[u] \leftarrow v$	$g[u] \leftarrow g[v] + c(v,u)$ $f[u] \leftarrow g[u] + h(u)$ $\mathrm{Insert}_f(\mathrm{OPEN}, u)$ $\mathrm{Parent}[u] \leftarrow v$
$u \in \mathrm{OPEN}$	**if** $g[v] + c(v,u) < g[u]$ **then** $g[u] \leftarrow g[v] + c(v,u)$ $\mathrm{Parent}[u] \leftarrow v$	**if** $g[v] + c(v,u) < g[u]$ **then** $g[u] \leftarrow g[v] + c(v,u)$ $f[u] \leftarrow g[u] + h(u)$ $\mathrm{Parent}[u] \leftarrow v$
$u \in \mathrm{CLOSED}$	Do nothing (always $g[u] \le g[v] + c(v,u)$)	Do nothing (always $g[u] \le g[v] + c(v,u)$)

v = node just expanded / u = a successor of v

How difficult is it to design a monotone heuristic function?

Good News!

Almost all well-known “natural” heuristics (e.g., those computed from relaxed problems) are monotone



Note: the heuristic used for the exercise was artificially constructed

It was

- admissible
- but **not** monotone