3010

ARTIFICIAL INTELLIGENCE

Lecture 5 Online (Real-time) Heuristic Search

Masashi Shimbo

2019-05-21

Today's agenda

- Offline vs. online (real-time) search
- The learning real-time A* (LRTA*) algorithm

The search algorithms introduced in this course thus far

Dijkstra, A*, …

fall into a paradigm called

offline search

Offline search paradigm

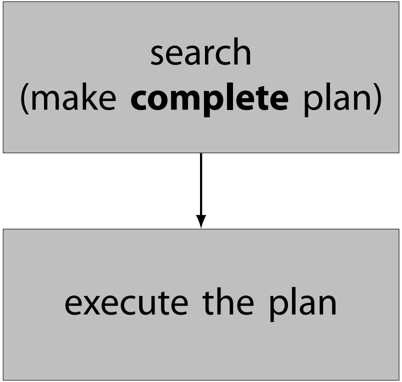
Objective = to find a **complete** plan (action sequence/path) that achieves a goal

- **execution** of the plan is not taken into account

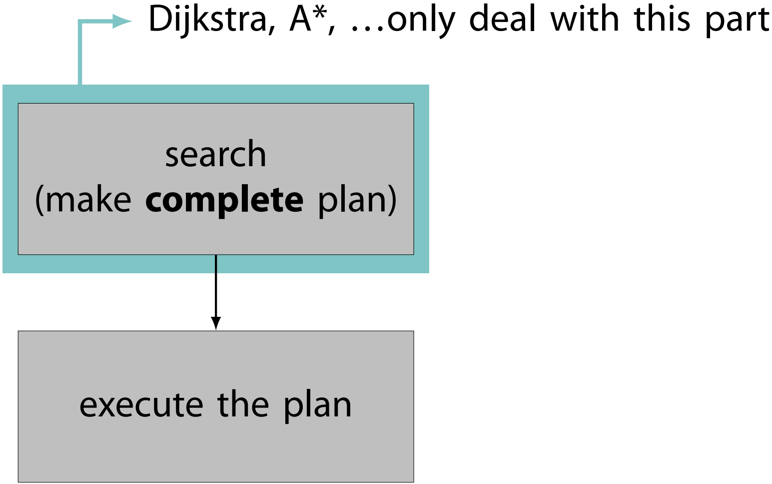
No particular limit on time spent on making a plan

- Finding a complete and optimal plan is the main research concern —even though it might take quite a long time

Offline search paradigm



Offline search paradigm



Offline search paradigm

Dijkstra, A*, …only deal with this part

search
(make **complete** plan)

execute the plan

Execution of the plan is out of their scope

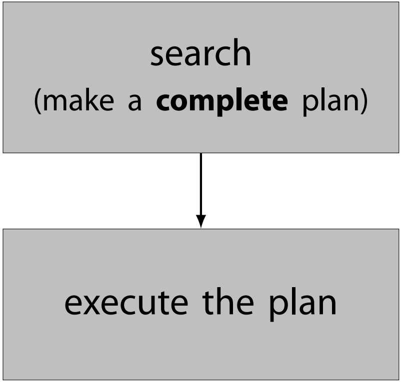
Online search (aka real-time search)

Agent is required to be more “reactive”:

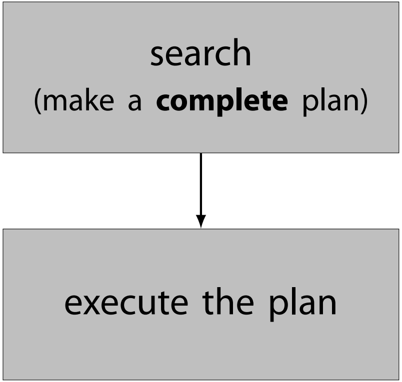
- Not enough time to make a complete plan
- Rather, decide an immediate action to take, on the basis of a **partial** plan, and **execute** the action
- This process is repeated until a goal is reached

➥ **search** and **action execution** are interleaved

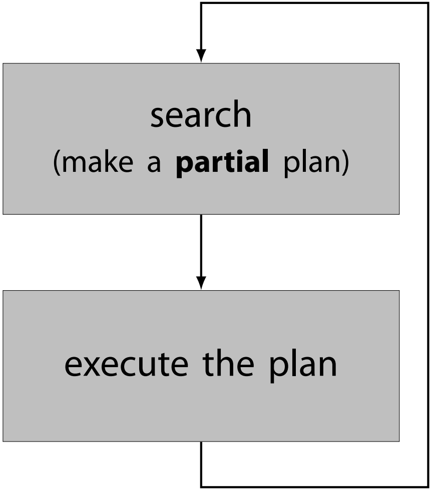
Offline search



Offline search



Online search



Offline vs. online search

Offline search Classic search framework

Dijkstra, A*, …

Make an optimal, complete, plan to achieve a goal

Online search Learning real-time A* (LRTA*), …

Determines an immediate action the agent should take, execute the action, and repeat until a goal is reached.

Online search with one-step lookahead

Let us consider a simple, extreme scenario:

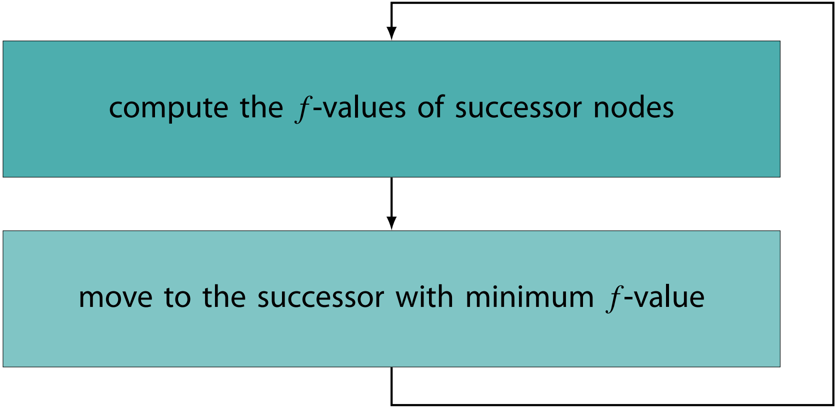
In the planning phase of each iteration, the agent has time just enough for **one-step** lookahead but not more

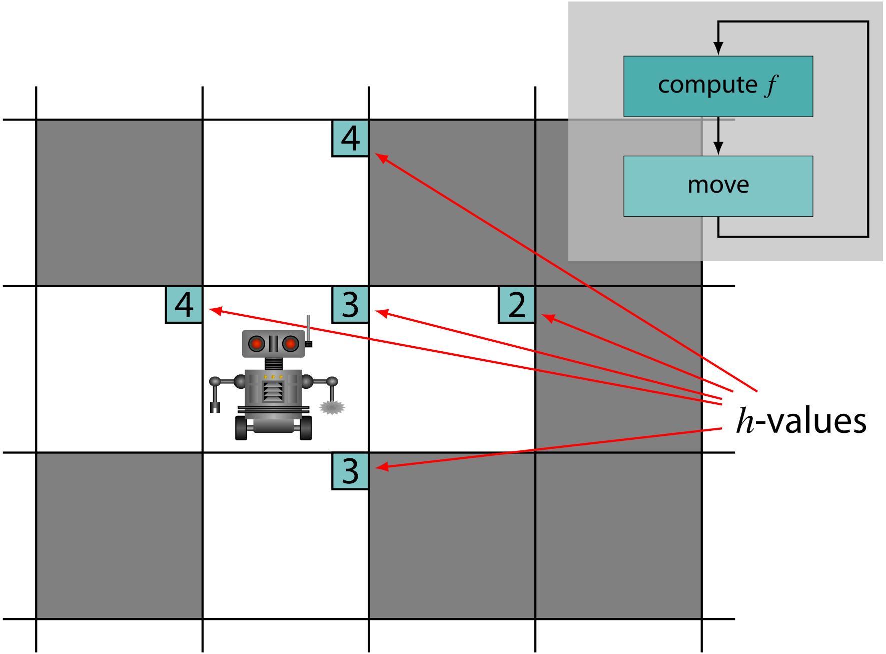
- the agent decides which edge to follow, after examining successor nodes only
- no further search (i.e., beyond successors) is allowed

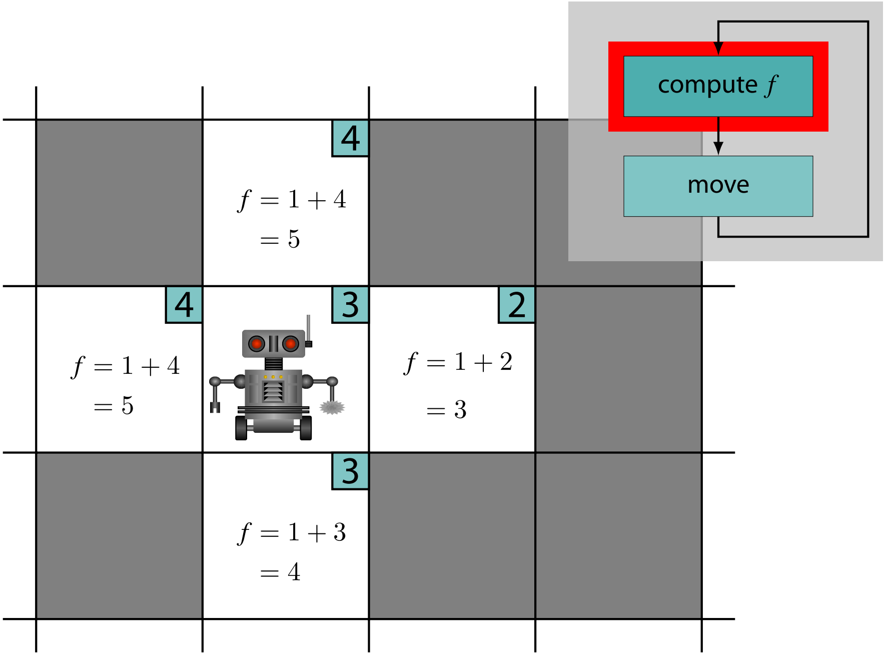
As in A*, heuristic function h is available

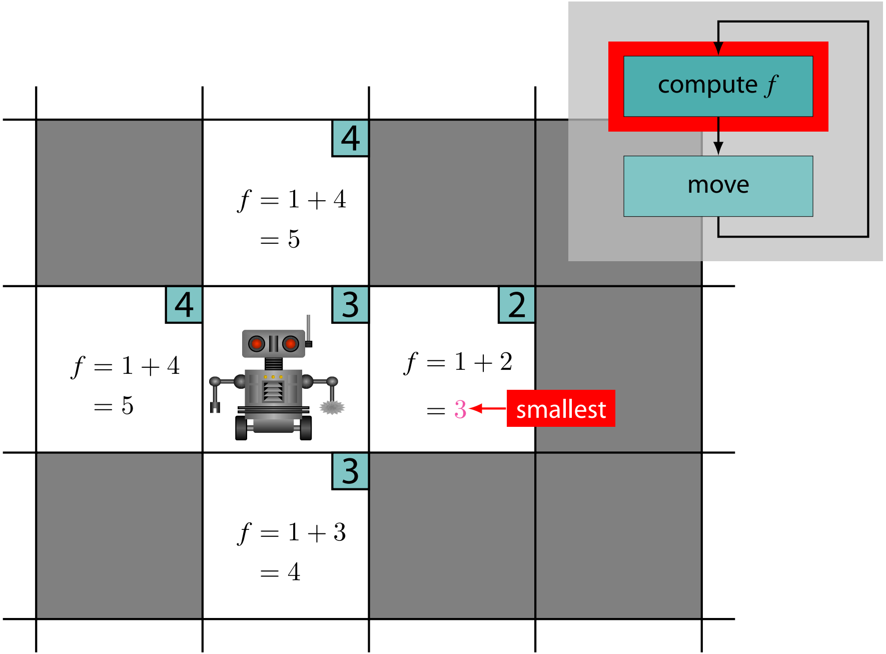
Does the following simple strategy work?

- Consider the current node v of the agent as the initial node, and compute $f(u) = c(v, u) + h(u)$ for every successor node u
- Choose the successor node with the minimum f-value as the destination

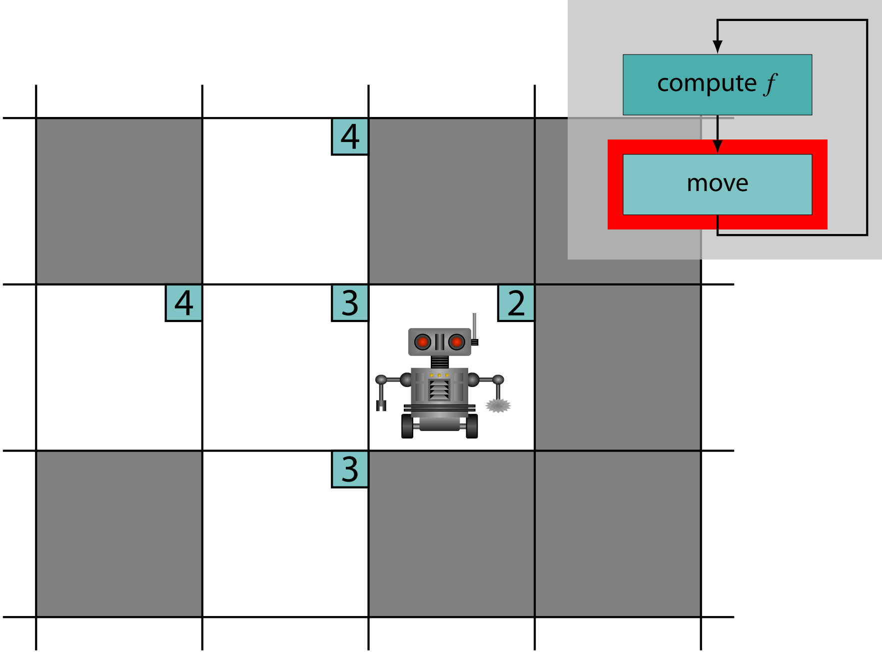


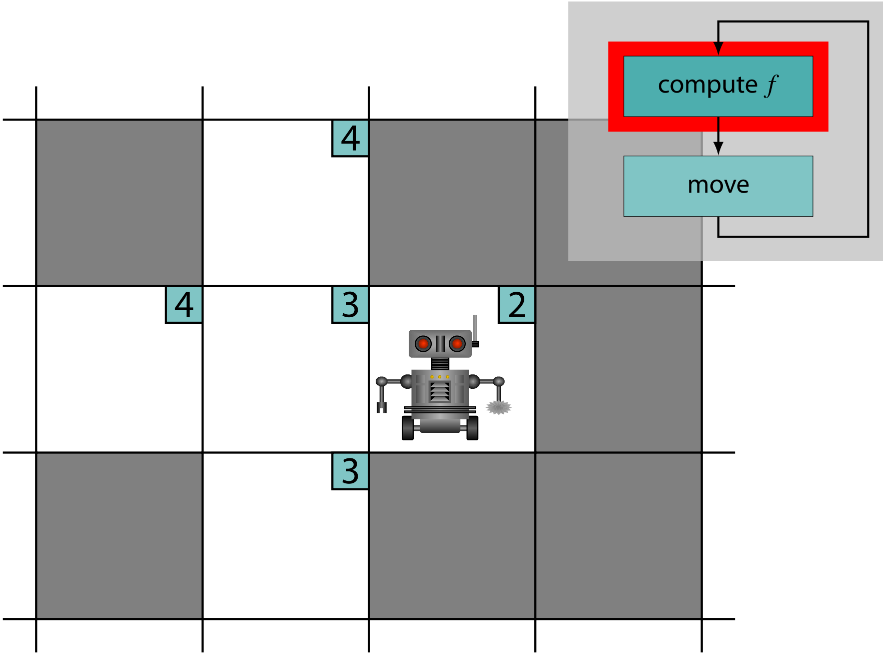

compute f
move
4
4
3
2
3
h-values


compute f
move
4
$f = 1 + 4$
$= 5$
4
3
2
$f = 1 + 4$
$= 5$
$f = 1 + 2$
$= 3$
3
$f = 1 + 3$
$= 4$

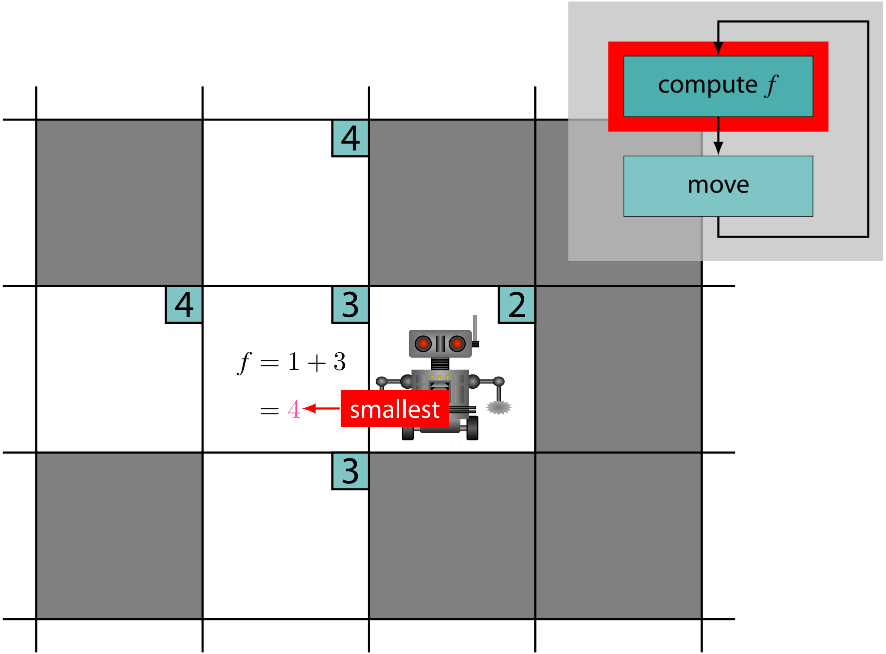


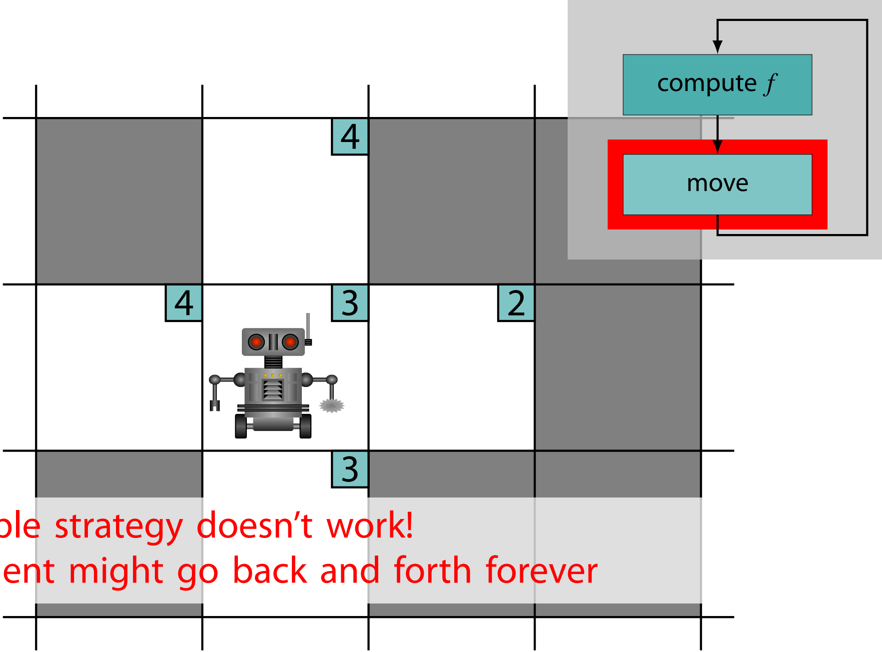
compute f
move
4
$f = 1 + 4$
$= 5$
4
$f = 1 + 4$
$= 5$
3
2
$f = 1 + 2$
$= 3$
smallest
3
$f = 1 + 3$
$= 4$


compute f
move
4
4
3
2
3



compute f
move
4
4
3
2
3


compute f
move
4
4
3
2
3
$f = 1 + 3$
$= 4$
smallest



This simple strategy doesn't work!
…the agent might go back and forth forever

…then, what can be done?

The strategy used by Learning Real-Time A* (LRTA*)

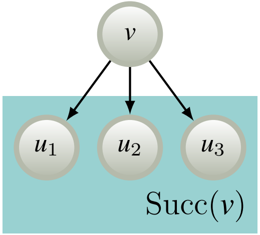
Update h upon leaving the current node!

Let:

$$v = \text{current node}$$

$$v_{\text{next}} = \text{node to move to}$$

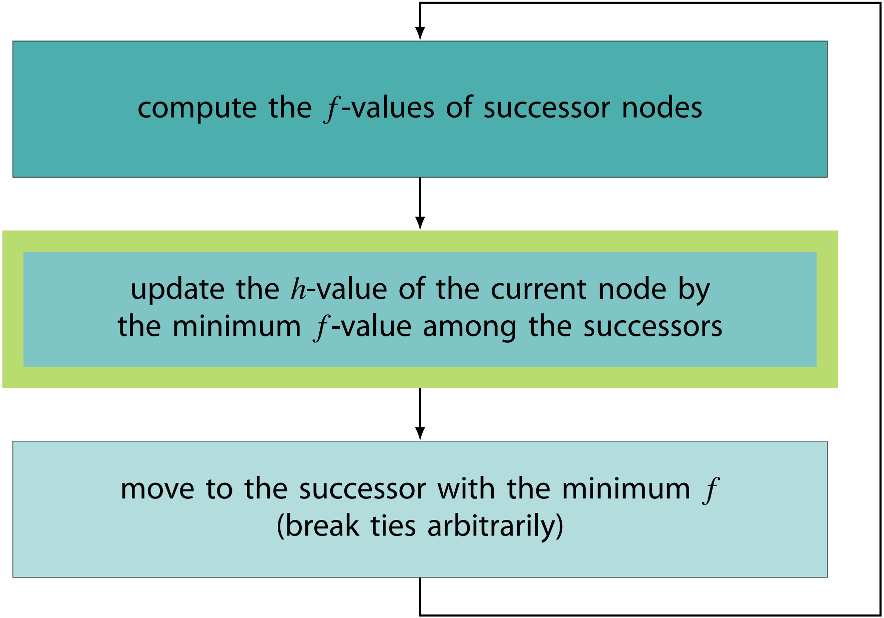
$$v_{\text{next}} = \underset{u \in \mathrm{Succ}(v)}{\text{argmin}} \overbrace{c(v,u) + h[u]}^{f(u)}$$



h-value of the current node v is updated by:

$$h[v] \leftarrow f(v_{\text{next}}) = c(v, v_{\text{next}}) + h[v_{\text{next}}]$$

before moving to the next node v_{next}


compute the f-values of successor nodes
update the h-value of the current node by the minimum f-value among the successors
move to the successor with the minimum f (break ties arbitrarily)

Notice that h-values will change!

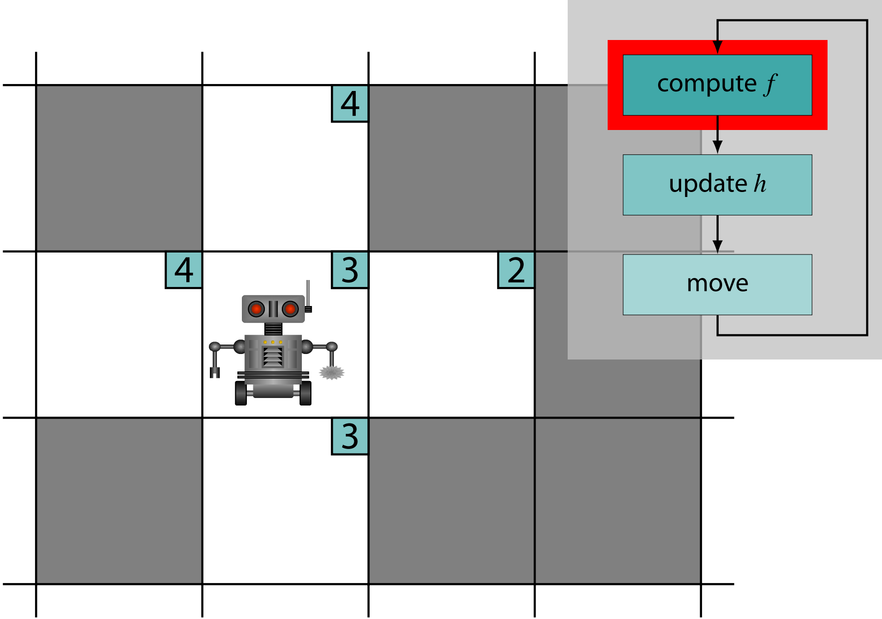
➥ h-values are not static anymore, not as in A*

Initially, their values are given by a static, admissible heuristic evaluation function h (as in A*), but the values may be updated upon the agent's leaving a state

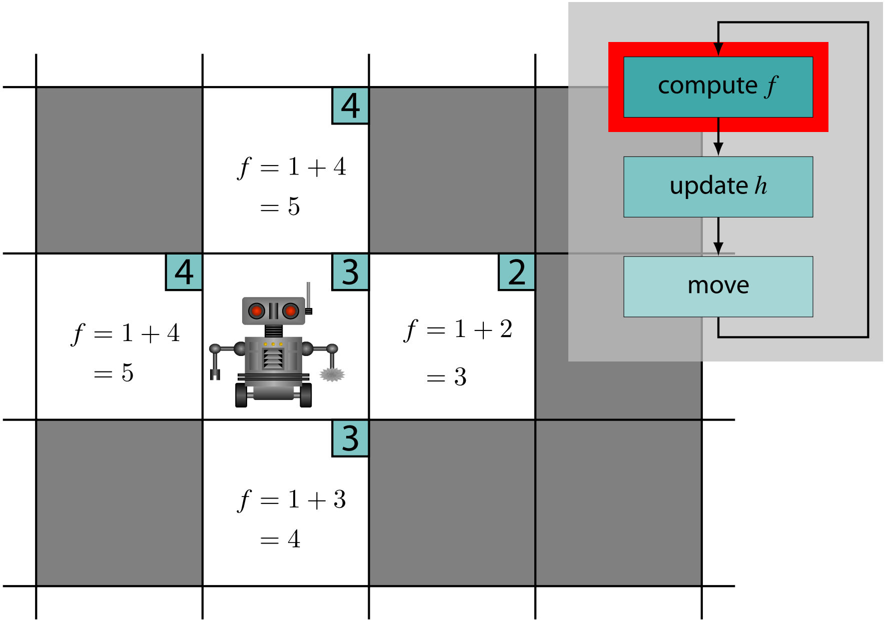
Notation

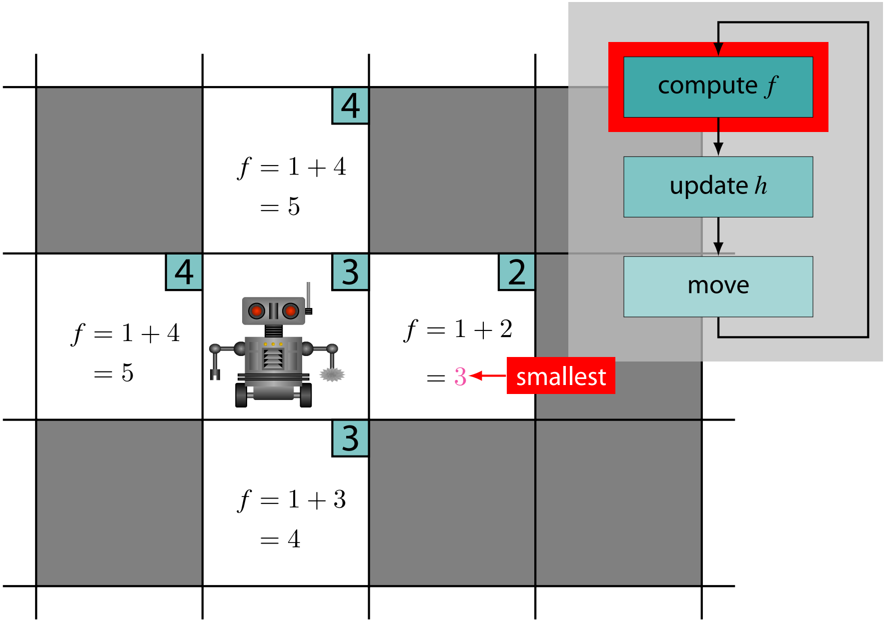
$h(v)$ (with round parentheses) denotes the initial heuristic value for node v

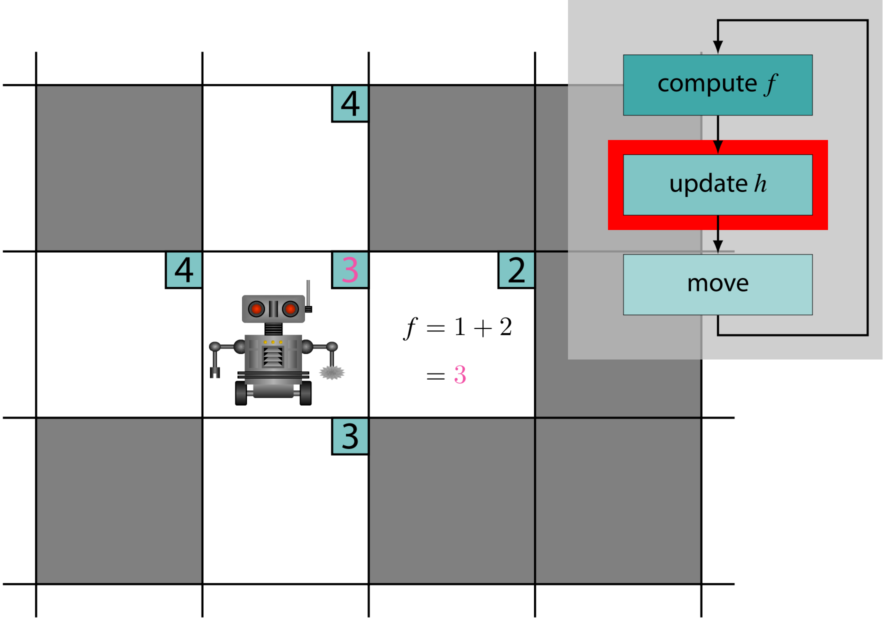
$h[v]$ (with square brackets) denotes the (possibly updated) heuristic value for node v



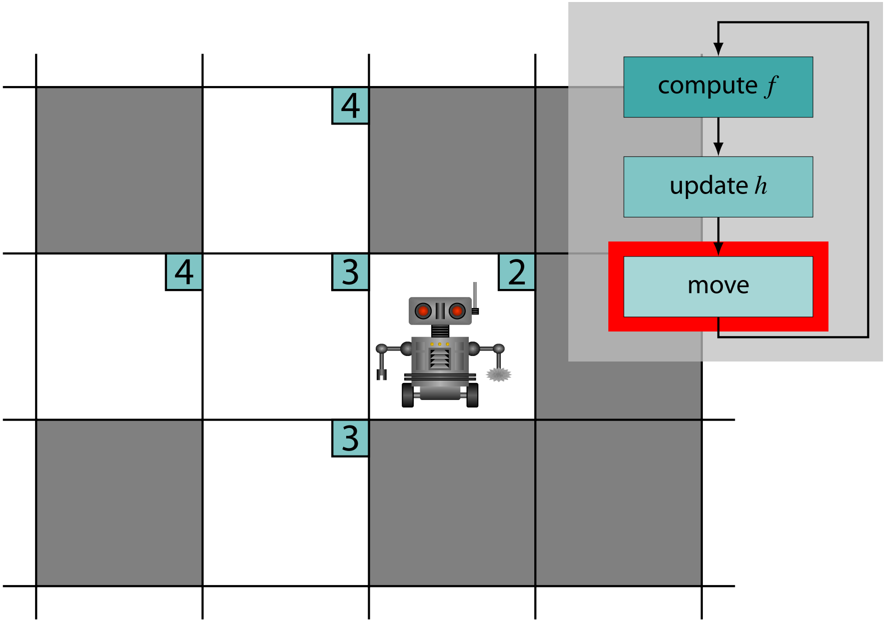
4
4
3
2
3
compute f
update h
move


4
$f = 1 + 4$
$= 5$
4
$f = 1 + 4$
$= 5$
3
2
$f = 1 + 2$
$= 3$
3
$f = 1 + 3$
$= 4$
compute f
update h
move

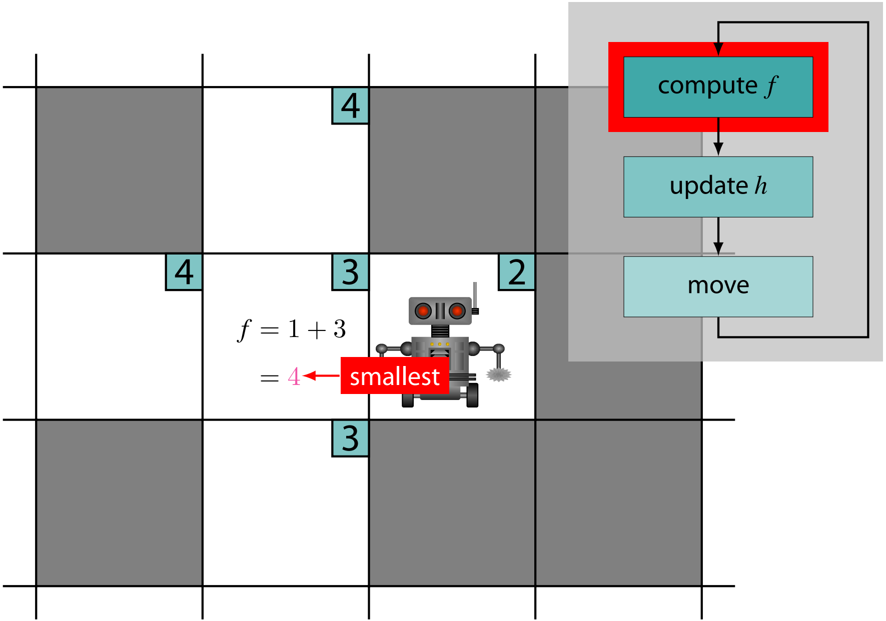

compute f
update h
move
4
$f = 1 + 4$
$= 5$
4
$f = 1 + 4$
$= 5$
3
2
$f = 1 + 2$
$= 3$
smallest
3
$f = 1 + 3$
$= 4$



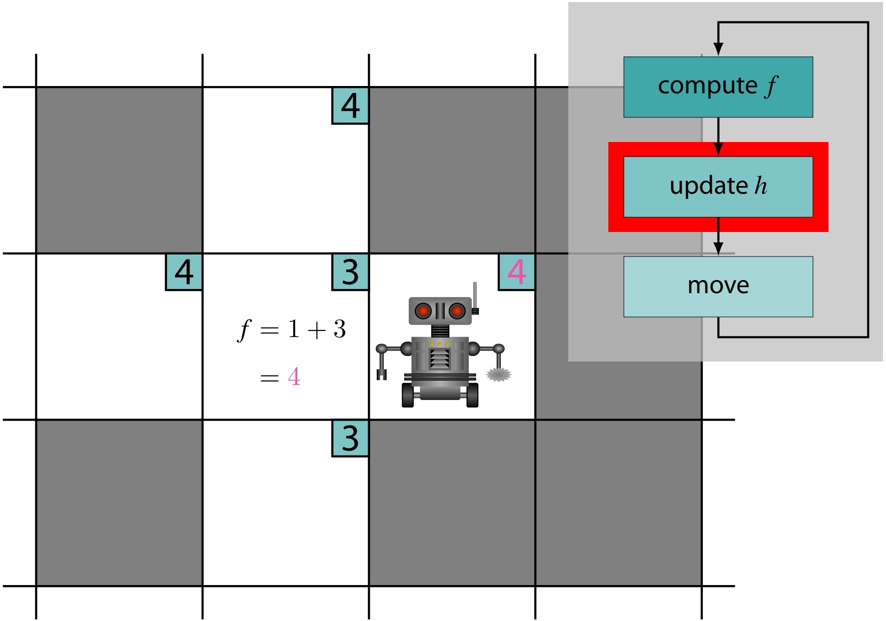
4
4
3
2
3
$f = 1 + 2$
$= 3$
compute f
update h
move

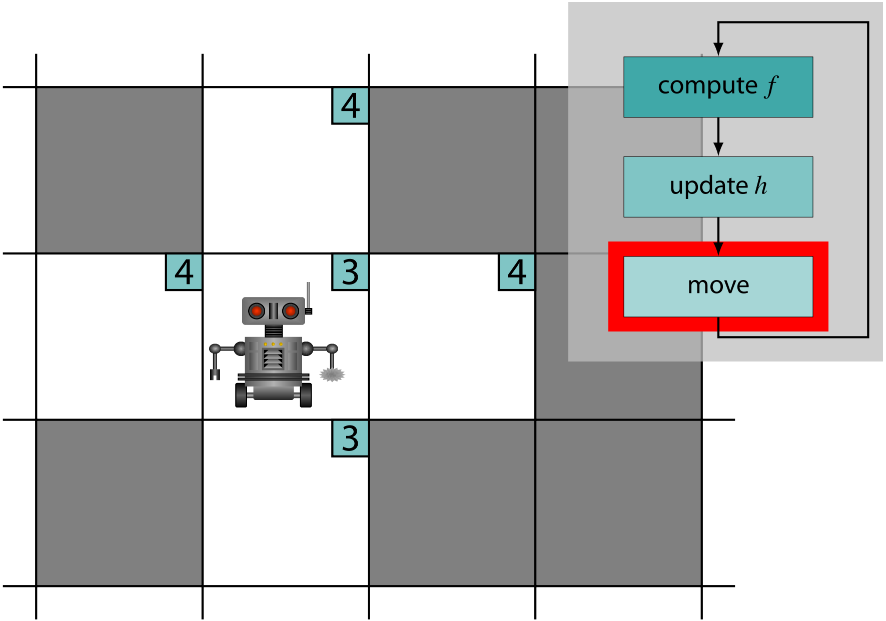


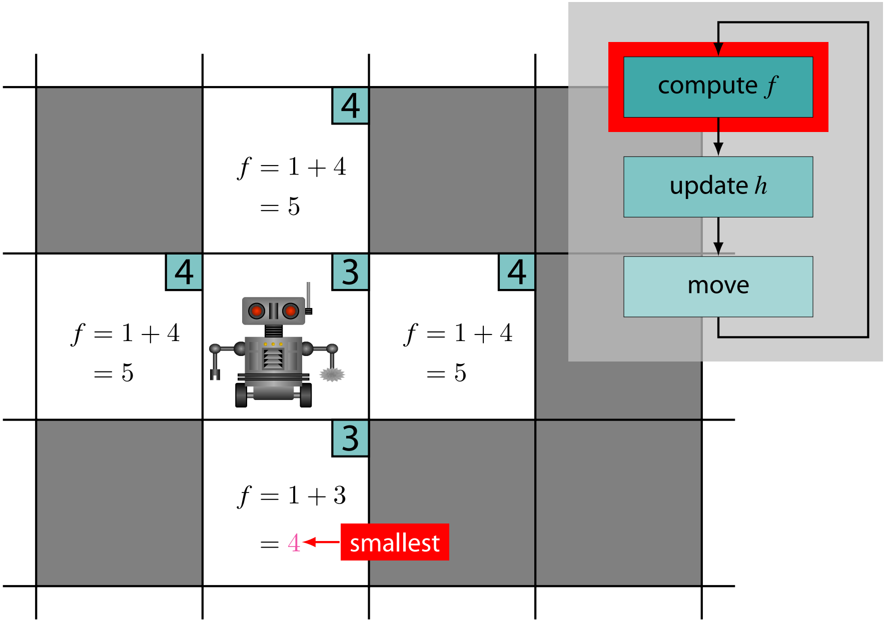
4
4
3
2
3
compute f
update h
move



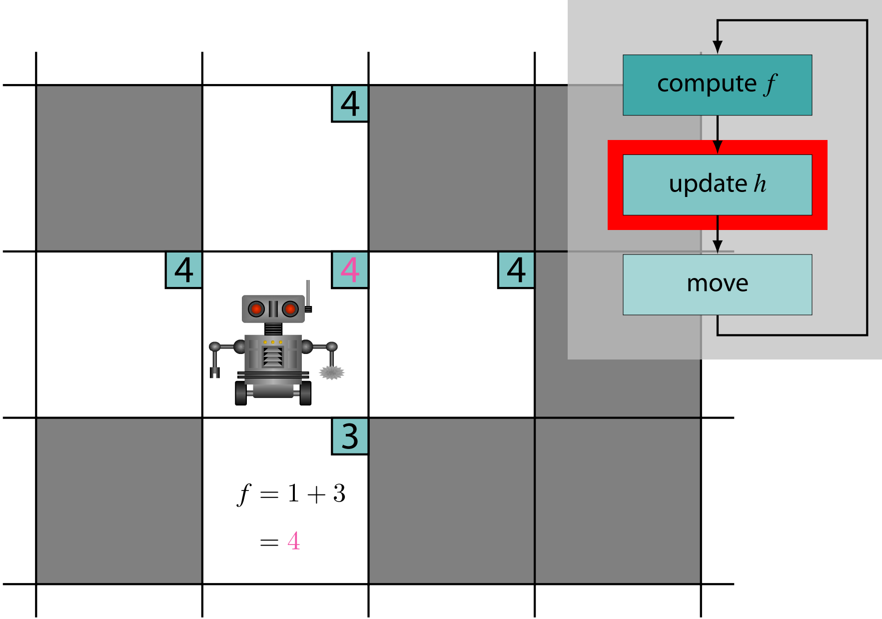
compute f
update h
move
4
4
3
2
3
$f = 1 + 3$
$= 4$
smallest

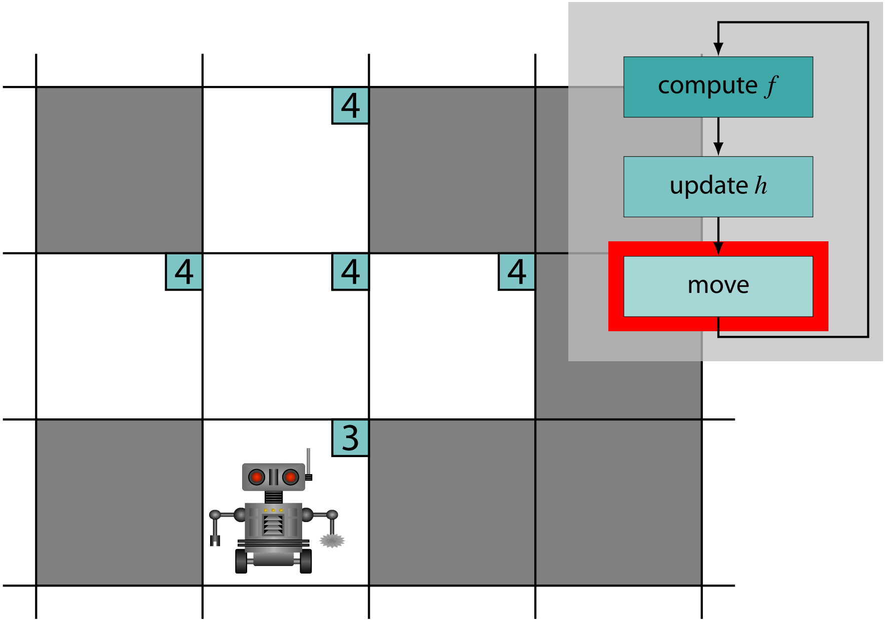

compute f
update h
move
4
4
3
4
3
$f = 1 + 3$
$= 4$

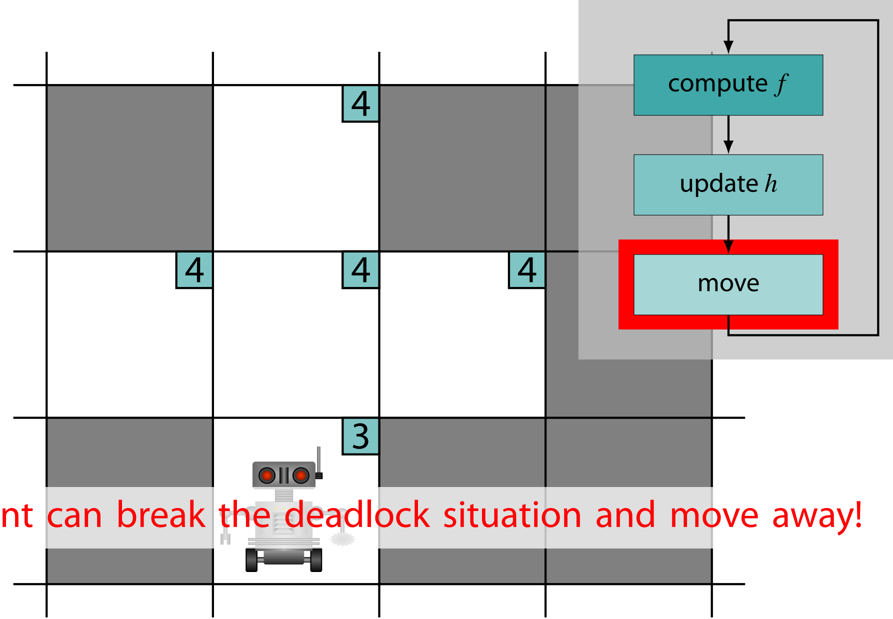

compute f
update h
move
4
4
3
4
3



compute f
update h
move
4
4
3
4
3
$f = 1 + 4$
$= 5$
$f = 1 + 4$
$= 5$
$f = 1 + 4$
$= 5$
$f = 1 + 3$
$= 4$
smallest


compute f
update h
move
4
4
4
4
3
$f = 1 + 3$
$= 4$


compute f
update h
move
4
4
4
4
3



The agent can break the deadlock situation and move away!

Learning real-time A* (LRTA*) [Korf 1990]

procedure $\mathrm{LRTA^*}(s)$

Input : initial state s

$v \leftarrow s$ # v: current state

while not $\mathrm{IsGoal}(v)$ **do**

  if $\mathrm{Succ}(v) = \emptyset$ **then** stop # no successor—search fails

  $f_{\text{best}} \leftarrow +\infty$; $v_{\text{best}} \leftarrow$ nil

  foreach $u \in \mathrm{Succ}(v)$ **do** # find the successor with cheapest $c(v, u) + h(u)$

    if $h[u]$ has not been computed **then** $h[u] \leftarrow h(u)$

    if $c(v, u) + h[u] < f_{\text{best}}$ **then**

      $f_{\text{best}} \leftarrow c(v, u) + h[u]$

      $v_{\text{best}} \leftarrow u$

  $h[v] \leftarrow f_{\text{best}}$ # update; $h[v] \leftarrow \mathrm{argmin}_{u \in \mathrm{Succ}(v)}\, c(v, u) + h[u]$

  $v \leftarrow v_{\text{best}}$ # move; $v \leftarrow \min_{u \in \mathrm{Succ}(v)}\, c(v, u) + h[u]$

LRTA* is complete

If the state space graph is such that

- The numbers of nodes and edges are finite
- No self loops exist (note: this can be easily lifted)
- From any node, at least one path to a goal node exists, which implies
  - $h^*(v)$ is finite for every node v
  - $\mathrm{Succ}(v) \neq \emptyset$ for every non-goal node v

then LRTA* never fails to reach a goal

Proof is given in the following slides…

Terminology

"**time** j" = moment at the beginning of the $(j+1)$st iteration

procedure $\mathrm{LRTA}^*(s)$
$v \leftarrow s$
while not $\mathrm{IsGoal}(v)$ **do**
    if $\mathrm{Succ}(v) = \emptyset$ **then** stop
    $f_{\text{best}} \leftarrow +\infty$; $v_{\text{best}} \leftarrow$ nil
    foreach $u \in \mathrm{Succ}(v)$ **do**
        if $h[u]$ has not been computed **then** $h[u] \leftarrow h(u)$
        if $c(v,u) + h[u] < f_{\text{best}}$ **then**
            $f_{\text{best}} \leftarrow c(v,u) + h[u]$
            $v_{\text{best}} \leftarrow u$
    $h[v] \leftarrow f_{\text{best}}$
    $v \leftarrow v_{\text{best}}$

time $= 0, 1, 2, \ldots$

For $j = 0, 1, 2, \ldots$, let

$$h_j(v) = \begin{cases} (\text{value of } h[v] \text{ at time } j), & \text{if } h[v] \text{ has been computed} \\ (\text{initial } h\text{-value } h(v)), & \text{otherwise} \end{cases}$$

$$v_j = (\text{the state } v \text{ of the agent at time } j)$$

Thus, for example,

$$\begin{aligned} v_0 &= s \\ h_0(v) &= (\text{initial heuristic value for node } v) \\ h_j(v_{j-1}) &= h_{j-1}(v_j) + c(v_{j-1}, v_j) \\ h_j(v) &= h_{j-1}(v) \qquad \text{for all } v \neq v_{j-1} \end{aligned}$$

Lemma: h-values remain admissible if they are initially admissible

If h-values are initially admissible, they remain admissible

➥ h never overestimate the actual cost-to-goal h^*

$$h_j(v) \leq h^*(v) \qquad \text{for every node } v\text{, and time } j$$

Proof is by induction…

Base case (when $j = 0$):

$$h_0(v) \leq h^*(v) \qquad \text{for every node } v$$

because the initial heuristic evaluation function $h_0(v)$ is admissible by assumption

Induction step:

Assume $h_{j-1}(v) \leq h^*(v)$ for every node v, and show $h_j(v) \leq h^*(v)$ for every node v.

In the j-th iteration, the following happens:

- The agent moves from v_{j-1} to v_j
- $h[v_{j-1}]$ is updated from $h_{j-1}(v_{j-1})$ to $h_j(v_{j-1})$

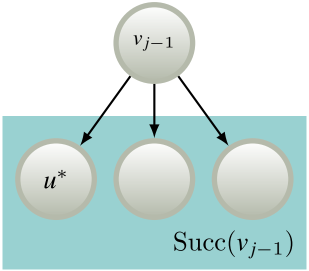
Because v_{j-1} is the only node whose h-value changes between times $j-1$ and j,

$$h_j(v) \leq h^*(v) \qquad \text{for every node } v \text{ other than } v_{j-1}$$

It remains to show $h_j(v_{j-1}) \leq h^*(v_{j-1})$

Let u^* be the successor of v_{j-1} along an optimal path from v_{j-1}:

$$u^* = \operatorname*{argmin}_{u \in \mathrm{Succ}(v_j)} h^*(u) + c(v_{j-1}, u^*)$$



In other words,

$$h^*(v_{j-1}) = h^*(u^*) + c(v_{j-1}, u) \qquad (*)$$

Now,

$$\begin{aligned} h_j(v_{j-1}) &= \min_{u \in \mathrm{Succ}(v_{j-1})} h_{j-1}(u) + c(v_{j-1}, u) && \because \text{update formula} \\ &\leq h_{j-1}(u^*) + c(v_{j-1}, u^*) && \because \min \leq \text{any successor } u \\ &\leq h^*(u^*) + c(v_{j-1}, u^*) && \because \text{inductive assumption} \\ &= h^*(v_{j-1}) && \because \text{from (*) above} \end{aligned}$$

Consider the sum of the h-values over all nodes.

Between times $j-1$ and j, h-values only change at node v_{j-1}. Therefore,

$$\overbrace{\sum_{v\in V} h_j(v)}^{\text{sum after update}} = \overbrace{\sum_{v\in V} h_{j-1}(v)}^{\text{sum before update}} - \overbrace{h_{j-1}(v_{j-1})}^{h(v_{j-1})\text{ before update}} + \overbrace{h_j(v_{j-1})}^{h(v_{j-1})\text{ after update}}$$

Consider the sum of the h-values over all nodes.

Between times $j-1$ and j, h-values only change at node v_{j-1}. Therefore,

$$\overbrace{\sum_{v\in V} h_j(v)}^{\text{sum after update}} = \overbrace{\sum_{v\in V} h_{j-1}(v)}^{\text{sum before update}} - \overbrace{h_{j-1}(v_{j-1})}^{h(v_{j-1})\text{ before update}} + \overbrace{h_j(v_{j-1})}^{h(v_{j-1})\text{ after update}}$$

$$= \sum_{v\in V} h_{j-1}(v) - h_{j-1}(v_{j-1}) + \overbrace{h_{j-1}(v_j) + c(v_{j-1}, v_j)}^{h_j(v_{j-1})}$$

Consider the sum of the h-values over all nodes.

Between times $j-1$ and j, h-values only change at node v_{j-1}. Therefore,

$$\overbrace{\sum_{v\in V} h_j(v)}^{\text{sum after update}} = \overbrace{\sum_{v\in V} h_{j-1}(v)}^{\text{sum before update}} - \overbrace{h_{j-1}(v_{j-1})}^{h(v_{j-1})\text{ before update}} + \overbrace{h_j(v_{j-1})}^{h(v_{j-1})\text{ after update}}$$

$$= \sum_{v\in V} h_{j-1}(v) - h_{j-1}(v_{j-1}) + \overbrace{h_{j-1}(v_j) + c(v_{j-1}, v_j)}^{h_j(v_{j-1})}$$

$$= \sum_{v\in V} h_{j-1}(v) - h_{j-1}(v_{j-1}) + \overbrace{h_j(v_j)}^{h_{j-1}(v_j)} + c(v_{j-1}, v_j)$$

Consider the sum of the h-values over all nodes.

Between times $j-1$ and j, h-values only change at node v_{j-1}. Therefore,

$$\overbrace{\sum_{v\in V} h_j(v)}^{\text{sum after update}} = \overbrace{\sum_{v\in V} h_{j-1}(v)}^{\text{sum before update}} - \overbrace{h_{j-1}(v_{j-1})}^{h(v_{j-1})\text{ before update}} + \overbrace{h_j(v_{j-1})}^{h(v_{j-1})\text{ after update}}$$

$$= \sum_{v\in V} h_{j-1}(v) - h_{j-1}(v_{j-1}) + \overbrace{h_{j-1}(v_j) + c(v_{j-1}, v_j)}^{h_j(v_{j-1})}$$

$$= \sum_{v\in V} h_{j-1}(v) - h_{j-1}(v_{j-1}) + \overbrace{h_j(v_j)}^{h_{j-1}(v_j)} + c(v_{j-1}, v_j)$$

Rearranging terms,

$$\sum_{v\in V} h_j(v) - h_j(v_j) = \sum_{v\in V} h_{j-1}(v) - h_{j-1}(v_{j-1}) + c(v_{j-1}, v_j)$$

Let $S_j = \sum_{v \in V} h_j(v) - h_j(v_j)$. Then,

$$\underbrace{\sum_{v \in V} h_j(v) - h_j(v_j)}_{S_j} = \underbrace{\sum_{v \in V} h_{j-1}(v) - h_{j-1}(v_{j-1})}_{S_{j-1}} + c(v_{j-1}, v_j)$$

$$S_j = S_{j-1} + c(v_{j-1}, v_j)$$

Rearranging,

$$c(v_{j-1}, v_j) = S_j - S_{j-1}$$

Enumerate the equality over $j = 1, 2, \ldots \tau$:

$$
\begin{aligned}
c(v_0, v_1) &= S_1 - S_0 \\
c(v_1, v_2) &= S_2 - S_1 \\
c(v_2, v_3) &= S_3 - S_2 \\
\vdots \quad &\quad \vdots \\
c(v_{\tau-1}, v_\tau) &= S_\tau - S_{\tau-1}
\end{aligned}
$$

Taking sums on both sides of $=$ yields

$$
\sum_{j=1}^{\tau} c(v_{j-1}, v_j) = S_\tau - S_0
$$

Notice that the left-hand side is the cost of the path the agent has traveled up to time τ

$$
\begin{aligned}
\sum_{j=1}^{\tau} c(v_{j-1}, v_j) &= S_\tau - S_0 \\
&\le S_\tau && \because S_0 \ge 0 \\
&= \sum_{v \in V} h_\tau(v) - h_\tau(v_\tau) && \because \text{definition of } S_\tau \\
&\le \sum_{v \in V} h_\tau(v) && \because h_\tau(v_\tau) \ge 0 \\
&\le \sum_{v \in V} h^*(v) = \text{const} && \because h_\tau(v) \le h^*(v)
\end{aligned}
$$

This relation holds for any $\tau = 1, 2, \ldots$ during a run of the algorithm

➥ "The distance (=path cost) that can be traveled is bounded."

➥ The algorithm will terminate.

LRTA* terminates if either (a) it reaches a goal, or (b) there is no successors to v

procedure $\mathrm{LRTA}^*(s)$
$v \leftarrow s$
while not $\mathrm{IsGoal}(v)$ **do**
    if $\mathrm{Succ}(v) = \emptyset$ **then** stop
    $f_{\text{best}} \leftarrow +\infty$; $v_{\text{best}} \leftarrow$ nil
    foreach $u \in \mathrm{Succ}(v)$ **do**
        if $h[u]$ has not been computed **then** $h[u] \leftarrow h(u)$
        if $c(v, u) + h[u] < f_{\text{best}}$ **then**
            $f_{\text{best}} \leftarrow c(v, u) + h[u]$
            $v_{\text{best}} \leftarrow u$
    $h[v] \leftarrow f_{\text{best}}$
    $v \leftarrow v_{\text{best}}$

By assumption, there is always a node to move to

- ➥ The only case the algorithm terminates is when it reaches a goal
- ➥ The algorithm is complete

But the traversed path is in general **not** optimal

— Is there a way to obtain an optimal path?

Repeated application

Call LRTA*(s) repeatedly — after the agent reaches a goal, put it back to the initial state s, and run LRTA* again.

Warning! Do not reset h between runs — reuse the updated h-values at the end of a run as the inital h-values of the next run.

procedure $\mathrm{RepeatedLRTA*}(s)$

Input : initial state s

loop do

  $\mathrm{LRTA*}(s)$

Convergence (Learning)

We can show that after a certain run, the agent only traverses the shortest path

➥ The agent can **learn** the optimal behavior through repeated trials

The following slides give you a proof…

Notation

$$\tau(i) = \text{number of iterations (moves) the agent performed in the } i\text{th run}$$

$$c^{(i)} = \text{cost of the path traversed by the agent in the } i\text{th run}$$

$$K_j^{(i)} = \text{sum of the } h\text{-values over all states at time } j \text{ during the } i\text{th run } (0 \leq j \leq \tau(i))$$

Note: $K_{\tau(i)}^{(i)} = K_0^{(i+1)}$ because h-values are reused for the initial h-values of the next run

$$c^{(i)} = \sum_{j=1}^{\tau(i)} c(v_{j-1}^{(i)}, v_j^{(i)})$$

$$= \overbrace{[K_{\tau(i)}^{(i)} - \underbrace{h^{(i)}(v_{\tau(i)}^{(i)})}_{0 \because v_{\tau(i)} \text{ is a goal state}}]}^{S_{\tau(i)} \text{ in the } i\text{th run}} - \overbrace{[K_0^{(i)} - h^{(i)}(\underbrace{v_0^{(i)}}_{s})]}^{S_0 \text{ in the } i\text{th run}} \qquad \because \text{completeness proof}$$

$$= K_{\tau(i)}^{(i)} - K_0^{(i)} + h^{(i)}(s)$$

$$\leq K_{\tau(i)}^{(i)} - K_0^{(i)} + h^{*}(s) \qquad \because h^{(i)}(s) \leq h^{*}(s)$$

Rearranging $h^{*}(s)$ to the left-hand side:

$$c^{(i)} - h^{*}(s) \leq K_{\tau(i)}^{(i)} - K_0^{(i)}$$

$$= K_0^{(i+1)} - K_0^{(i)} \qquad \because h\text{-values are reused between runs}$$

Enumerate this inequality over runs $i = 1, 2, \ldots, n$:

$$\begin{aligned} c^{(1)} - h^*(s) &\leq K_0^{(2)} - K_0^{(1)} \\ c^{(2)} - h^*(s) &\leq K_0^{(3)} - K_0^{(2)} \\ \vdots \quad &\leq \quad \vdots \\ c^{(n)} - h^*(s) &\leq K_0^{(n+1)} - K_0^{(n)} \end{aligned}$$

Taking the sums on both sides yields:

$$\begin{aligned} \sum_{i=1}^{n} \left(c^{(i)} - h^*(s) \right) &\leq K_0^{(n+1)} - K_0^{(1)} \\ &\leq K_0^{(n+1)} = \sum_{v \in V} h_0^{(n+1)}(v) \\ &\leq \sum_{v \in V} h^*(v) = \text{const} \end{aligned}$$

1. Because $c^{(i)} - h^*(s) \geq 0$, the series on the left-hand side is the sum of non-negative numbers
2. This series is bounded by the constant on the right-hand side, and therefore the sequence $c^{(i)} - h^*(s)$ must converge to 0

➥ After some run, $c^{(i)} = h^*(s)$

➥ The agent evertally traverses the shortest paths only

➥ Agent has **learned** the optimal behavior through trial and error

➥ It can also be proven that, eventually, $h(v) = h^*(v)$ along the shortest paths.

Note: Relation to Q-learning

The updatable h-values can be regarded as an analogue of the "Q-values" in Q-learning (also used by Google's AlphaGo), a form of **reinforcement learning**

For detail, see:

A. G. Barto, S. J. Bradtke, S. P. Singh
Learning to act using real-time dynamic programming *Artificial Intelligence*, 72(1–2): 81–138