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

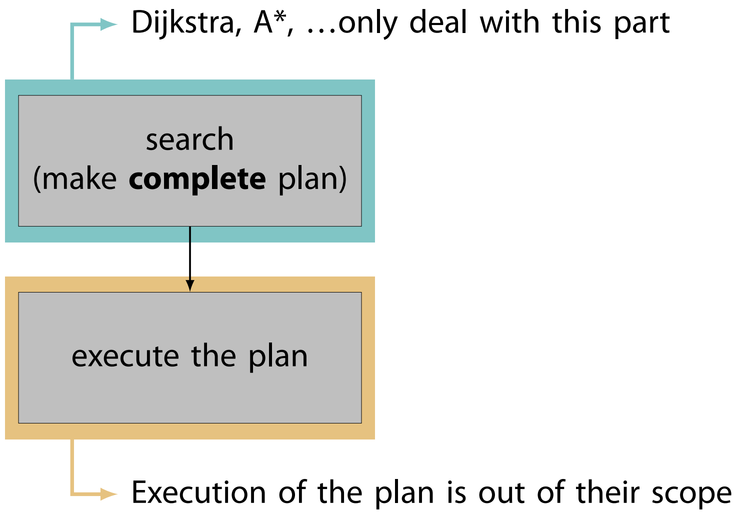
Objective = to find a **complete** plan (action sequence/path) that achieves a goal

➥ **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

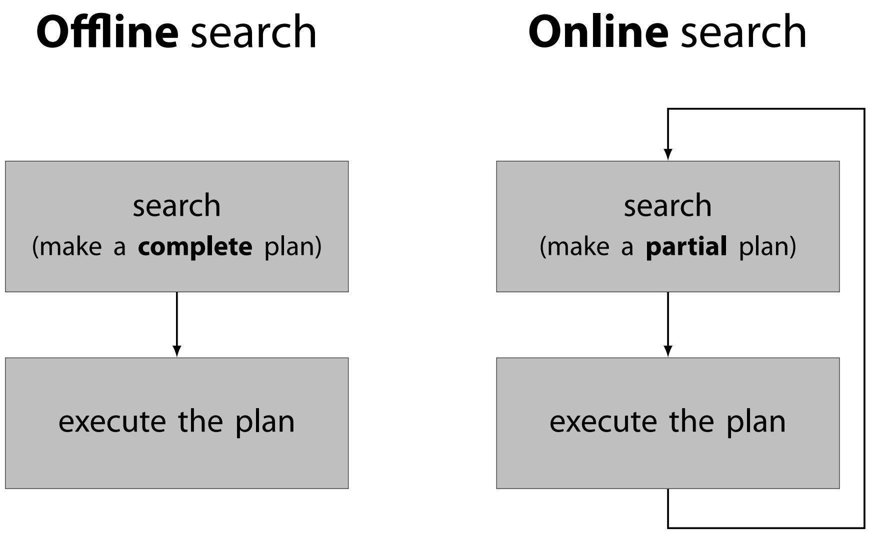


Online search (aka real-time search)

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 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:

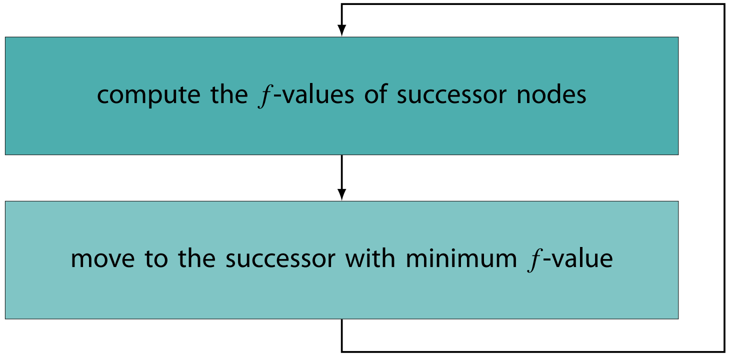
In the planning phase of each iteration, the agent has time just enough for **one-step** lookahead but not more

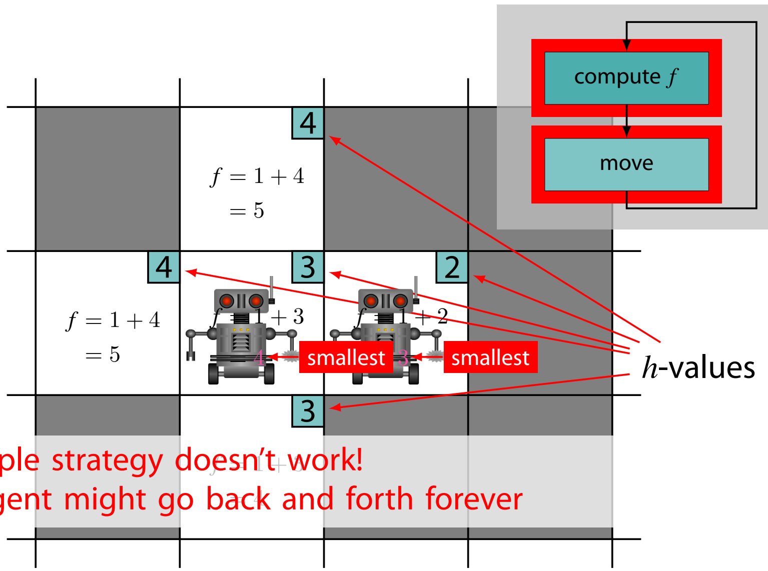
- the agent decides which edge to follow, after examining successor nodes only
- no further search (i.e., beyond successors) is allowed

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





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

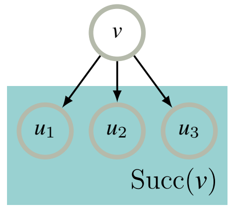
…then, what can be done?

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

Update h upon leaving the current node!

Let:

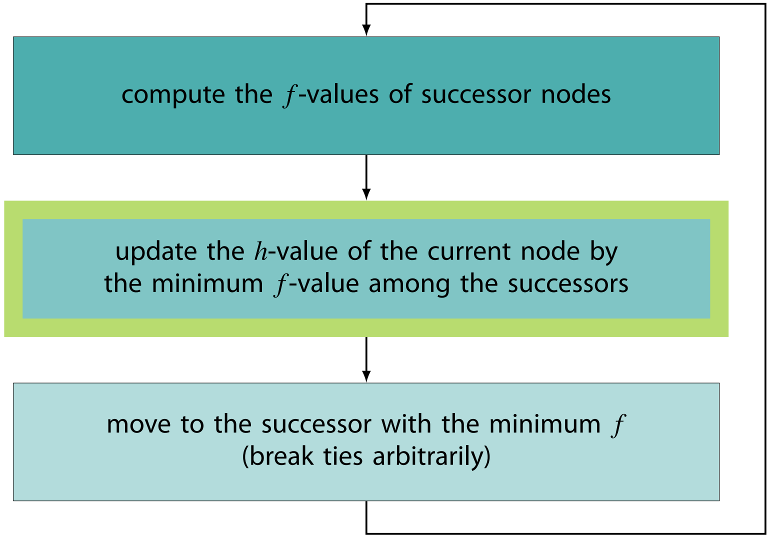
$$v = \text{current node}$$
$$v_{\text{next}} = \text{node to move to}$$
$$v_{\text{next}} = \operatorname*{argmin}_{u \in \mathrm{Succ}(v)} \overbrace{c(v, u) + h[u]}^{f(u)}$$



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}



13

Notice that h-values will change!

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

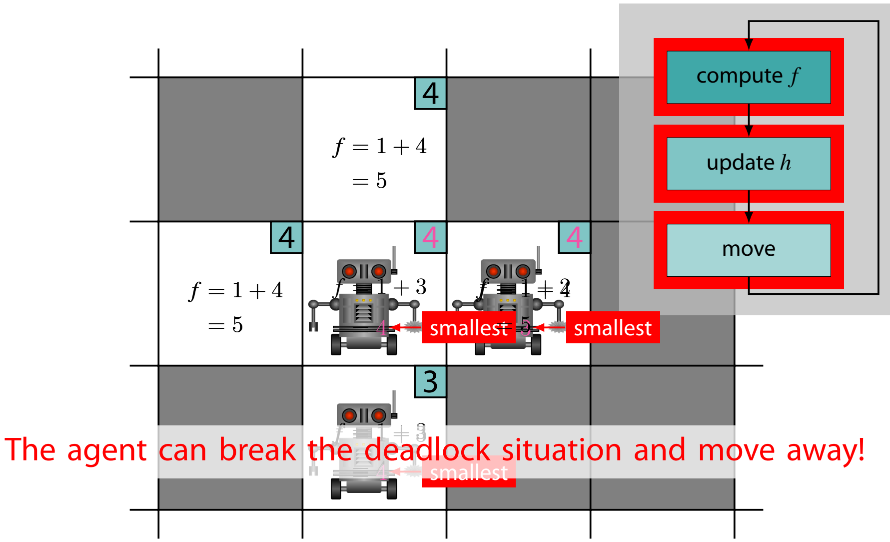
Initially, their values are given by a static, admissible heuristic evaluation function h (as in A*), but the values may be updated upon the agent's leaving a state

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

14



15

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

```
procedure LRTA*(s)
Input    : initial state s
v ← s                                                   # v: current state
while not IsGoal(v) do
    if Succ(v) = ∅ then stop                            # no successor—search fails
    f_best ← +∞; v_best ← nil
    foreach u ∈ Succ(v) do                              # find the successor with cheapest c(v,u) + h(u)
        if h[u] has not been computed then h[u] ← h(u)
        if c(v,u) + h[u] < f_best then
            f_best ← c(v,u) + h[u]
            v_best ← u
    h[v] ← f_best                                       # update; h[v] ← argmin_{u∈Succ(v)} c(v,u) + h[u]
    v ← v_best                                          # move; v ← min_{u∈Succ(v)} c(v,u) + h[u]
```

16

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 LRTA*(s)
v ← s
while not IsGoal(v) do          ← time = 0, 1, 2, ...
    if Succ(v) = ∅ then stop
    f_best ← +∞; v_best ← nil
    foreach u ∈ Succ(v) do
        if h[u] has not been computed then h[u] ← h(u)
        if c(v, u) + h[u] < f_best then
            f_best ← c(v, u) + h[u]
            v_best ← u
    h[v] ← f_best
    v ← v_best
```


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 $j = 0$):

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

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

Induction step:

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

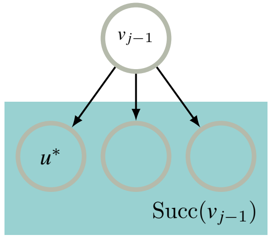
Because v_{j-1} is the only node whose h-value changes between times $j-1$ and j,

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

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

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

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



In other words,

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

Now,

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

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,

$$\begin{aligned} \overbrace{\sum_{v \in V} h_j(v)}^{\text{sum after update}} &= \overbrace{\sum_{v \in V} h_{j-1}(v)}^{\text{sum before update}} - \overbrace{h_{j-1}(v_{j-1})}^{h(v_{j-1}) \text{ before update}} + \overbrace{h_j(v_{j-1})}^{h(v_{j-1}) \text{ after update}} \\ &= \sum_{v \in V} h_{j-1}(v) - h_{j-1}(v_{j-1}) + \overbrace{h_{j-1}(v_j) + c(v_{j-1}, v_j)}^{h_j(v_{j-1})} \\ &= \sum_{v \in V} h_{j-1}(v) - h_{j-1}(v_{j-1}) + \overbrace{h_j(v_j)}^{h_{j-1}(v_j)} + c(v_{j-1}, v_j) \end{aligned}$$

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 $j = 1, 2, \dots \tau$:

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

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

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

This relation holds for any $\tau = 1, 2, \dots$ 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 LRTA*(s)
v ← s
while not IsGoal(v) do
    if Succ(v) = ∅ then stop
    f_best ← +∞; v_best ← nil
    foreach u ∈ Succ(v) do
        if h[u] has not been computed then h[u] ← h(u)
        if c(v, u) + h[u] < f_best then
            f_best ← c(v, u) + h[u]
            v_best ← u
    h[v] ← f_best
    v ← v_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

$\tau(i)$ = number of iterations (moves) the agent performed in the ith run

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

$K_j^{(i)}$ = sum of the h-values over all states at time j during the ith 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

33

$$
\begin{aligned}
c^{(i)} &= \sum_{j=1}^{\tau(i)} c(v_{j-1}^{(i)}, v_j^{(i)}) \\
&= \overbrace{[K_{\tau(i)}^{(i)} - \underbrace{h^{(i)}(v_{\tau(i)}^{(i)})}_{0 \because v_{\tau(i)} \text{ is a goal state}}]}^{S_{\tau(i)} \text{ in the } i\text{th run}} - \overbrace{[K_0^{(i)} - h^{(i)}(\underbrace{v_0^{(i)}}_{s})]}^{S_0 \text{ in the } i\text{th run}} && \because \text{completeness proof} \\
&= K_{\tau(i)}^{(i)} - K_0^{(i)} + h^{(i)}(s) \\
&\leq K_{\tau(i)}^{(i)} - K_0^{(i)} + h^*(s) && \because h^{(i)}(s) \leq h^*(s)
\end{aligned}
$$

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

$$
\begin{aligned}
c^{(i)} - h^*(s) &\leq K_{\tau(i)}^{(i)} - K_0^{(i)} \\
&= K_0^{(i+1)} - K_0^{(i)} && \because h\text{-values are reused between runs}
\end{aligned}
$$

34

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:

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

35

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

36

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