A* in Dynamic Environments

by Łukasz Stafiniak

Instytut Informatyki Uniwersytetu Wrocławskiego

Real-Time Adaptive A^*

A^* maintains ("A New Principle for Incremental Heuristic Search: Theoretical Results" Maxim Likhachev Sven Koenig):

- estimated minimal cost $g(s)$ from current state s_{curr} to every state s (initially $+\infty$)
- a heuristic $h(s)$ estimating the goal distance
- a priority queue (open list=fringe nodes $\mathcal{F}$) weighted by $f(s) = g(s) + h(s)$ initially with s_{curr} only
- pop a state s, update $g(s')$ for each successor of s, push onto the queue each successor for which $g(s')$ dropped, put s into closed list=internal nodes $\mathcal{I}$.

Take $s \in \mathrm{closed}$ (an expanded state=internal node).
After a complete A^* search, $g(s)$ is the cost-minimal path from s_{start} to s.

Let $\mathrm{gd}(s)$ – the goal distance of s, $f^* = \mathrm{gd}(s_{\mathrm{start}})$ – the optimal cost. Then:

$$f^* - g(s) \leqslant \mathrm{gd}(s)$$

so the updated heuristic $f^* - g(s)$ is admissible. It is a better heuristic:

$$h(s) \leqslant f^* - g(s)$$

Adaptive A^* repeatedly finds cost-minimal paths for problems with the **same goal vertices** and **non-decreasing edge costs**.
The updated heuristic is consistent:

$$h'(s) \leqslant h'(\mathrm{succ}(s,a)) + c(s,a) \leqslant h'(\mathrm{succ}(s,a)) + c'(s,a)$$

[Compare admissibility and increasing costs requirement to exploration.]

("Real-Time Adaptive A*" Sven Koenig Maxim Likhachev)
s_{term}–a state that bounded-lookahead A* is about to expand when it terminates.

`procedure` realtime_adaptive_astar():

  `while` ($s_{\mathrm{curr}} \notin \mathrm{GOAL}$) `do`

    lookahead := any desired integer greater than zero;

```
astar();
if s = FAILURE then
        return FAILURE;
for all s ∈ CLOSED do
        h(s) := g(s_term) + h(s_term) − g(s);
movements := any desired integer greater than zero;
while (s_curr ≠ s_term ∧ movements > 0) do
        a := the action in A(s_curr) on the cost-minimal trajectory
        from s_curr to s_term;
        s_curr := succ(s_curr, a);
        movements := movements − 1;
        for any desired number of times (including zero) do
                increase any desired c(s, a) where s ∈ S and a ∈ A(s);
        if any increased c(s, a) is on the cost-minimal trajectory
        from s_curr to s_term then
                break;
        return SUCCESS;
```

Model learning is easy: start with the most optimistic $c(s,a) = \varepsilon$ (e.g., $\varepsilon = 1$), update costs from experience (e.g. $c(s,a) := \infty$ when $a \notin A(s)$).

Because of simplicity, RTAA* is the best method for heavily time-constrained domains.

Prioritorized Learning Real-Time A*

("Real-Time Heuristic Search with a Priority Queue" D. Chris Rayner, Katherine Davison, Vadim Bulitko, Kenneth Anderson, Jieshan Lu)

function PLRTA$^*(s)$:

  while $s \neq s_{\text{goal}}$ **do**

    StateUpdate(s)
    repeat

      $p = $ queue.Pop()
      if $p \neq s_{\text{goal}}$ **then** StateUpdate(p)

    until N states are updated or queue $= \varnothing$
    $s \leftarrow$ neighbor s' with lowest $f(s,s') = c(s,s') + h(s')$

function StateUpdate(s)

  find neighbor s' with lowest $f(s,s') = c(s,s') + h(s')$
  $\Delta \leftarrow f(s,s') - h(s)$
  if $\Delta > 0$ **then**

    $h(s) \leftarrow f(s,s')$
    for all neighbors n of s **do**

      AddToQueue(n, Δ)

function AddToQueue(s, Δ_s)

  if $s \notin$ queue **then**

    if queue.Full() **then**

      $(r, \Delta_r) \leftarrow$ queue.Pop()
      if $\Delta_r < \Delta_s$ **then** queue.Push(s, Δ_s)
      else queue.Push(r, Δ_r)

    else queue.Push(s, Δ_s)

Moving Target Adaptive A*

- instead of the "optimal" minimax (escaping opponent), just generalize A*
- based on Adaptive A* (see the previous notes file)
- we want linear space complexity (don't store distances between arbitrary states visited)
- correct the h values when the goal state changes; let H be the original heuristic

$$h(s) := \max\left(H(s, s'_{\text{target}}), h(s) - h(s'_{\text{target}})\right)$$

- the lazy version accumulates updates and applies them when a state is visited (remembering that the previous update was n times ago)

D* Lite

("D* Lite" Sven Koenig, Maxim Likhachev)

D* Lite is based on **Lifelong Planning A*** (LPA*) which performs A* and accommodates cost changes by replanning backwards from the change points only backing-up distance from start for relevant states.

- no assumptions about how the costs change (up or down, close or far from the current/start state)
- the priority queue always contains the locally inconsistent states $g(s) \neq \mathrm{rhs}(s)$
- Initialize() should perform lazily not to tabulate all states

D* Lite lets the current state move by planning backwards from the goal state, replacing as g the distance from current state by the more stable distance from goal state. Heuristic h now measures distances from the current state.

- h admissible $h(s, s') \leqslant c^*(s, s')$ and obeys the triangle inequality
- not to reorder the queue, accumulate changes $h(s_{\mathrm{prev}}, s_{\mathrm{curr}})$

Minimax LPA* algorithm results by replacing $\min_{s' \in \mathrm{Succ}(u)} (c(u, s') + g(s'))$ with $\min_{a \in \mathcal{A}(u)} \max_{s' \in \mathrm{Succ}(u)} (c(u, s') + g(s'))$ (and performing $\arg\min_{a \in \mathcal{A}(u)}$).

Lifelong Planning A* algorithm

procedure CalculateKey(s):

$$\textbf{return}\ (\min(g(s), \text{rhs}(s)) + h(s, s_{\text{goal}}), \min(g(s), \text{rhs}(s)))$$

procedure Initialize():

$U \leftarrow \varnothing$;
for all $s \in \mathcal{S}$ **do** $\text{rhs}(s) \leftarrow g(s) \leftarrow \infty$
$\text{rhs}(s_{\text{start}}) \leftarrow 0$
$U.\text{Insert}(s_{\text{start}}, \text{CalculateKey}(s_{\text{start}}))$

procedure UpdateVertex(u):

if $u \neq s_{\text{start}}$ **then** $\text{rhs}(u) \leftarrow \min_{s' \in \text{Pred}(u)} (g(s') + c(s', u))$
if $u \in U$ **then** $U.\text{Remove}(u)$
if $g(u) \neq \text{rhs}(u)$ $\{ \wedge \text{NotYet}(u) \}$ **then** $U.\text{Insert}(u, \text{CalculateKey}(u))$

```
procedure ComputeShortestPath():
    while U.TopKey() <· CalculateKey(s_goal) ∨ rhs(s_goal) ≠ g(s_goal)
        u ← U.Pop()
        if g(u) > rhs(u)
            g(u) ← rhs(u)
            for all s ∈ Succ(u) do UpdateVertex(s)
        else
            g(u) ← ∞
            for all s ∈ Succ(u) ∪ {u} do UpdateVertex(s)
procedure Main():
    Initialize()
    forever
        ComputeShortestPath(); {insert all inconsist. states to U}
        Wait for changes in edge costs
        for all directed edges (u, v) with changed costs
            update c(u, v)
            UpdateVertex(v)
```

D* Lite algorithm

procedure CalculateKey(s):

return $(\min(g(s), \mathrm{rhs}(s)) + h(s_{\mathrm{start}}, s) + k_m, \min(g(s), \mathrm{rhs}(s)))$

procedure Initialize():

$U \leftarrow \varnothing$
$k_m \leftarrow 0$
for all $s \in \mathcal{S}$ **do** $\mathrm{rhs}(s) \leftarrow g(s) \leftarrow \infty$
$\mathrm{rhs}(s_{\mathrm{goal}}) \leftarrow 0$
$U.\mathrm{Insert}(s_{\mathrm{goal}}, \mathrm{CalculateKey}(s_{\mathrm{goal}}))$

procedure UpdateVertex(u):

if $u \neq s_{\mathrm{goal}}$ **then** $\mathrm{rhs}(u) \leftarrow \min_{s' \in \mathrm{Succ}(u)} (c(u, s') + g(s'))$
if $u \in U$ **then** $U.\mathrm{Remove}(u)$
if $g(u) \neq \mathrm{rhs}(u)$ $\{\wedge \mathrm{NotYet}(u)\}$ **then** $U.\mathrm{Insert}(u, \mathrm{CalculateKey}(u))$

```
procedure ComputeShortestPath():
    while U.TopKey() <· CalculateKey(s_start) ∨ rhs(s_start) ≠ g(s_start)
        k_old ← U.TopKey()
        u ← U.Pop()
        if k_old <· CalculateKey(u)
            U.Insert(u, CalculateKey(u))
        else if g(u) > rhs(u)
            g(u) ← rhs(u)
            for all s ∈ Pred(u) do UpdateVertex(s)
        else
            g(u) ← ∞
            for all s ∈ Pred(u) ∪ {u} do UpdateVertex(s)
    {insert remaining locally inconsistent states to U}
```

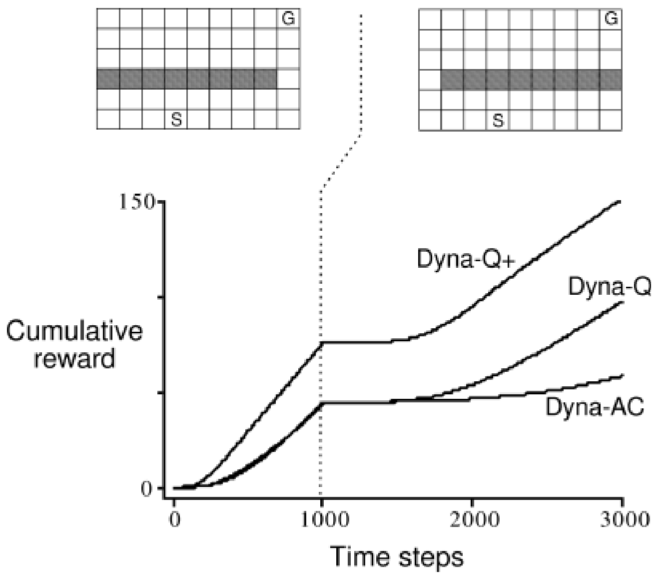
```
procedure Main():
    s_last ← s_start
    Initialize()
    ComputeShortestPath()
    while s_start ≠ s_goal
        s_start ← arg min_{s' ∈ Succ(s_start)} (c(s_start, s') + g(s'))
        Move to s_start
        Scan graph for any changed edge costs
        if any edge cost changed
            k_m ← k_m + h(s_last, s_start)
            s_last ← s_start
            for all directed edges (u, v) with changed costs
                Update c(u, v)
                UpdateVertex(u)
            ComputeShortestPath()
```

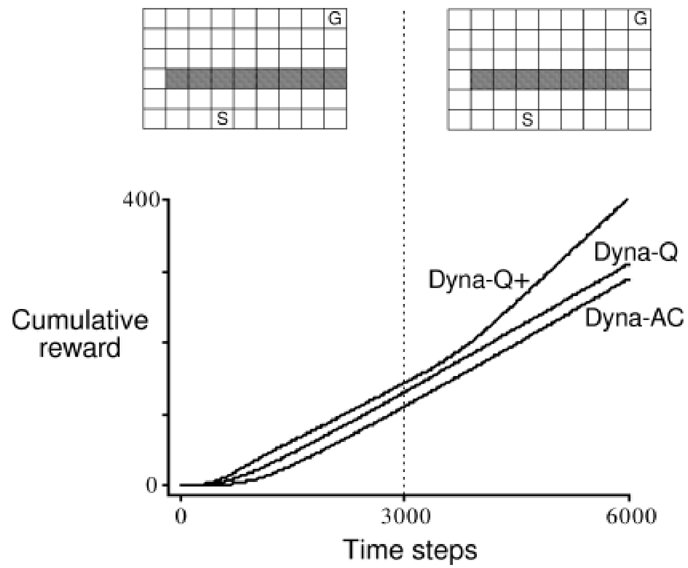
A Generalized Framework for LPA* with inconsistent heuristics

("A Generalized Framework for Lifelong Planning A* Search" Sven Koenig, Maxim Likhachev)

- Heuristic Search-based Planning (HSP) used in modern symbolic planners
- planners that find minimum-cost plans do not scale to large domains: the need to use more informed but inconsistent heuristics
  - the bigger the cost heuristic, the less states expanded
- experimentally A* works better with inconsistent heuristics, when for f-value tie bigger g is prefered (not smaller)
  - LPA* is not even correct: it can fail to find a finite cost solution
  - no matter whether smaller or bigger g is prefered, LPA* is worse than A* (replanning from scratch) on inconsistent heuristics
- solution: the priority queue of GLPA* does not contain all locally inconsistent states but only those not yet expanded as overconsistent; however, GLPA* updates the priority queue to contain all locally inconsistent states between calls to ComputePlan(); (NotYet in the algorithms)
  - now one can use inconsistent heuristics and reversed g tie breaks

When the Model is Wrong – Dyna+





Dyna-$Q+$ uses a heuristic to enforce exploration (especially useful in non-stationary environments): if a transition has not been tried in n time steps, then planning backups are done as if that transition produced a reward of $r + \kappa\sqrt{n}$, for some small κ. Perhaps a similar idea can be incorporated into above algorithms.

Appendix

D* Lite optimized algorithm

`procedure` CalculateKey(s):

$$\texttt{return}\ (\min(g(s), \mathrm{rhs}(s)) + h(s_{\mathrm{start}}, s) + k_m, \min(g(s), \mathrm{rhs}(s)))$$

`procedure` Initialize():

$U \leftarrow \varnothing$
$k_m \leftarrow 0$
`for all` $s \in \mathcal{S}$ `do` $\mathrm{rhs}(s) \leftarrow g(s) \leftarrow \infty$
$\mathrm{rhs}(s_{\mathrm{goal}}) \leftarrow 0$
$U.\mathrm{Insert}(s_{\mathrm{goal}}, (h(s_{\mathrm{start}}, s_{\mathrm{goal}}), 0))$

`procedure` UpdateVertex(u):

`if` $u \neq s_{\mathrm{goal}} \wedge u \in U$ `then` $\mathrm{rhs}(u) \leftarrow \min_{s' \in \mathrm{Succ}(u)} (c(u, s') + g(s'))$
`if` $u \in U \wedge g(u) = \mathrm{rhs}(u)$ `then` $U.\mathrm{Remove}(u)$
`if` $g(u) \neq \mathrm{rhs}(u) \wedge u \in U$ $\{\wedge \mathrm{NotYet}(u)\}$

$U.\mathrm{Insert}(u, \mathrm{CalculateKey}(u))$

```
procedure ComputeShortestPath():
    while U.TopKey() ≺ CalculateKey(s_start) ∨ rhs(s_start) ≠ g(s_start)
        k_old ← U.TopKey()
        u ← U.Pop()
        k_new ← CalculateKey(u)
        if k_old ≺ k_new
            U.Insert(u, k_new)
        else if g(u) > rhs(u)
            g(u) ← rhs(u)
            for all s ∈ Pred(u)
                if s ≠ s_goal
                    rhs(s) ← min (rhs(s), c(s, u) + g(u))
                UpdateVertex(s)
        else
            g_old ← g(u)
            g(u) ← ∞
```

$\texttt{for all } s \in \text{Pred}(u) \cup \{u\}$

$\texttt{if } \text{rhs}(s) = c(s, u) + g_{\text{old}} \wedge s \neq s_{\text{goal}}$

$\text{rhs}(s) \leftarrow \min\left(\text{rhs}(s), c(s, u) + g(u)\right)$

$\text{UpdateVertex}(s)$

{insert remaining locally inconsistent states to U}

```
procedure Main():
    s_last ← s_start
    Initialize()
    ComputeShortestPath()
    while s_start ≠ s_goal
        s_start ← arg min_{s'∈Succ(s_start)} (c(s_start, s') + g(s'))
        Move to s_start
        Scan graph for any changed edge costs
        if any edge cost changed
            k_m ← k_m + h(s_last, s_start)
            s_last ← s_start
            for all directed edges (u, v) with changed costs
                c_old ← c(u, v)
                Update c(u, v)
                if c_old > c(u, v)
                    if u ≠ s_goal
                        rhs(u) ← min (rhs(u), c(u, v) + g(v))
```

`else if` $\text{rhs}(u) = c_{\text{old}} + g(v)$

`if` $u \neq s_{\text{goal}}$

$\text{rhs}(u) \leftarrow \min_{s' \in \text{Succ}(u)} (c(u,\, s') + g(s'))$

$\text{UpdateVertex}(u)$

$\text{ComputeShortestPath}()$