SAPIENZA
UNIVERSITÀ DI ROMA

Variants of Learning Real Time A*

Planning and Reasoning

Omar Bayoumi
Matricola 1747042

Table of content

1 Introduction

A* is a great algorithm for finding the **best path** from a start to a goal. However, it is forced to maintain in memory:

- A set of reachable states (**frontier**).
- Heuristics that **never change**.

In particular, the second point is the basis of **LRTA***.
In contrast to A* a hasty algorithm always chooses from a current state the successor state that brings it closer to the goal using very little memory, but this leads to two problems:

- It may find a **suboptimal path**.
- It might get **stuck in a dead end**.

LRTA* is a hasty algorithm that tries to solve these problems and is based on this principle:

> ***"If the heuristic is admissible, h(s) is always less than or equal to the cost to reach the goal from s but a high h(s) is a better estimate of the cost as long as it is admissible, higher is better."***

Assuming there is the perfect heuristic $\boldsymbol{h^*(s)}$, which coincides with the **true cost** from a state to the goal, a heuristic $\boldsymbol{h'(s)}$ is admissible if it remains **less than or equal to** $\boldsymbol{h^*(s)}$ and then increasing the value of $\boldsymbol{h'(s)}$ as long as it remains below makes the heuristic **better but still admissible**.

2 Graphs

The graphs used were generated by me and are **connected and random**. The graph seen in class was also used [Figure 1].
The technique used to allow the graph to be connected is the use of a **recursive function** [Figure 2]:

- **Basic step.** If I have a list of one state, I return that list and no edges, since there is only one state.
- **Recursive step.** I divide the states into the **left half and the right half** and by passing them to the function I am sure to get a **left connected graph and a right connected graph**. At this point I choose one of the two sets randomly and start the edge from a random state of this set to $\boldsymbol{n}$ random states of the other set, where $\boldsymbol{n}$ is a random number $\boldsymbol{n} \in [\boldsymbol{MIN\_EDGES, MAX\_EDGES}]$. Finally, the set of states with the edges is returned.

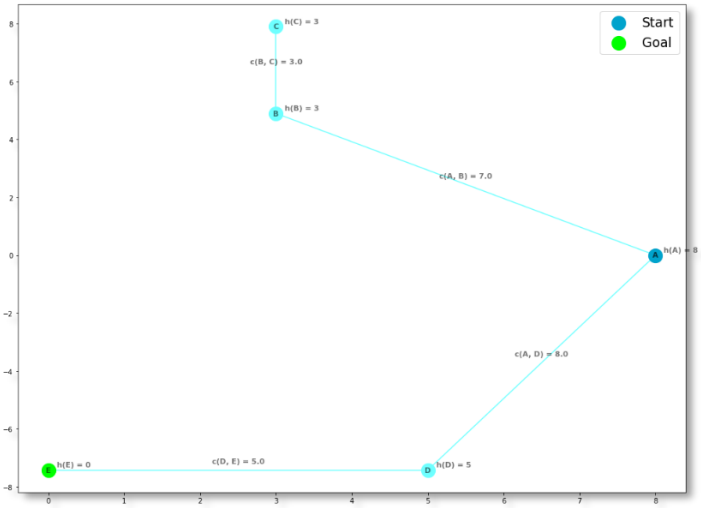

Figure 1. Graph used in lecture

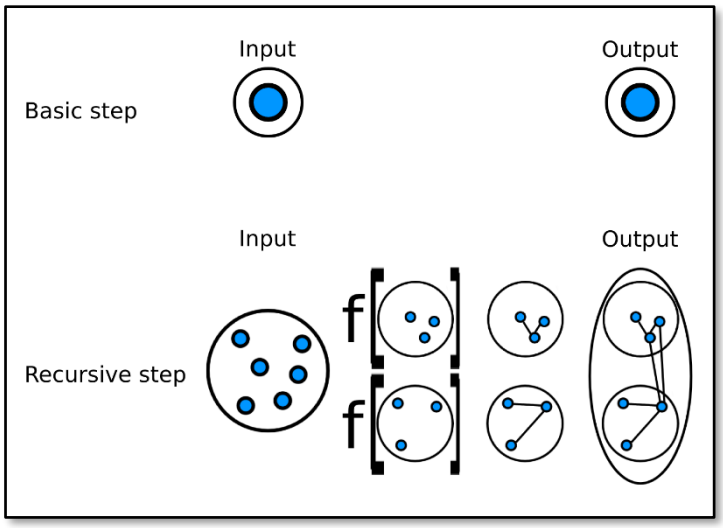

Figure 2. Recursive function for generate connected graphs

3 Algorithms

3.1 LRTA*(k)

The **LRTA*[1] is a hasty algorithm**; in fact, it always looks for the best successor to move. The parameter $\boldsymbol{k}$ indicates how many states ahead one must look before making a move. Specifically, if during this lookahead, at the $\boldsymbol{i\text{-}th}$ **state**, with $\boldsymbol{i < k}$, its heuristics is updated, this update must be **propagated to the successors** of this state until the maximum number $\boldsymbol{k}$ of states to be updated is reached or there are **no more updates available**. Of course, if $\boldsymbol{k = 1}$ an update is done, and we move on.

The **update rule** of the heuristic in **LRTA*(k)** is:

1. Choose the best successor $\boldsymbol{s'}$ of $\boldsymbol{s}$ as $\min_{s' \in succ(s)} \boldsymbol{c(s,s') + h(s')}$.
2. Update the heuristic of the $\boldsymbol{s}$ with the $\boldsymbol{max(h(s), c(s,s') + h(s'))}$.

This works because:

- If $\boldsymbol{h(s) < c(s,s') + h(s')}$ and the **heuristic is admissible**, then the **real cost** of reaching the goal will surely be greater than or equal to the cost of reaching the successor state plus its heuristic.
- Otherwise, it means that $\boldsymbol{h(s)}$ has already been updated before, using the principle just mentioned, and so it is **still admissible but greater than** $\boldsymbol{c(s,s') + h(s')}$ and therefore there is no point in decreasing a heuristic because, as said first... **higher is better**.

After the updates, we proceed to the next best state $\boldsymbol{s'}$ following the $\min_{s' \in succ(s)} \boldsymbol{c(s,s') + h(s')}$.

Animations of the execution of this algorithm are saved in the files in the folder "*./Animations/LRTA_star/*" or shown step by step on the notebook and shows the execution on the graph seen in lecture with different values of $\boldsymbol{k}$ and on a randomly generated graph.

Omar Bayoumi
Matricola 1747042

3.2 LRTA*-LS(k)

The **LRTA*-LS(k)**[2] variant is the same as the regular **LRTA*** but solves one problem. In fact, **if $k = 1$, the reasoning in the algorithm is different but it gets the exact same results as the normal LRTA* with $k = 1$**.

The reasoning behind **LRTA*-LS** is to have a kind of **tree, projected onto the graph**, where the **leaves belong to the frontier list** and the other **inner states belong to the interior list**. The two lists (actually sets) are filled in this way to form the tree:

1. Given a **current state v.**
2. Find y such that:
$$y = \underset{w \in succ(v),\ w \notin Interior}{\mathbf{argmin}} c(v,w) + h(w).$$
3. If $h(v)$ is less than $h(y) + c(v,y)$:
   - **If True**, this is an **interior state.**
   - **If False**, this is a **frontier state.**

After that, updates will be made considering the action $i \rightarrow f$ such that:
$$(i,f) = \underset{v \in Interior,\ w \in Frontier,\ w \in succ(v)}{\mathbf{argmin}} c(v,w) + h(w)$$
The **heuristic of the interior state involved in this action** is updated considering the $max(h(i), c(i,f) + h(f))$ and after that it **becomes part of the frontier** and finally the process is repeated until **no more interior states exist**.

With $k > 1$ the LRTA* problem that has been solved is that **the same states are often inserted multiple times in the queue**, leading to repeated updates in one direction while ignoring the others (see examples of **LRTA* and LRTA*LS with $k = 10$** in which this is shown, and not only... LRTA*LS with $k = 10$ coincides with that with $k = 5$ since **all states have entered in the frontier/interior list**)

Animations of the execution of this algorithm are saved in the files in the folder "*./Animations/LRTA_star_LS/*" or shown step by step on the notebook and shows the execution on the graph seen in lecture with different values of k and on a randomly generated graph.

3.3 TB-LRTA* Falcons

The **TB-LRTA* Falcons**[3] variant tries to improve **LRTA*** through a systematic choice of the minimum. In **LRTA***, the best successor is chosen as:

$$\boldsymbol{f(s) = \underset{s' \in succ(s)}{\operatorname{argmin}}\, c(s, s') + h(s')}.$$

However, this formula could generate a **set of states with the same minimum value**. From this set one is chosen randomly. In my case I have considered the "random" as the first element of this set since the first element depends on the order in which the states are inserted among the successors, which is random.

In **TB-LRTA*** the choice of state in the set of minima is made by giving priority to states obtained **according to a criterion**. From this new set one is chosen systematically. In my case, **always the first one**. This algorithm has a problem: **might have infinite loops**.

The **TB-LRTA* FALCONS solves the infinite loop problem**. It uses the same logic but modifies the **choice of best successors** and the **update rule** due to the use in the calculation of a new function $\boldsymbol{g(s)}$ that represents a **heuristic toward the start state**.
Thus, we have:

1. We find the best successors $\boldsymbol{s'_{set}}$ such as:
$$\boldsymbol{s'_{set} = \underset{s'' \in succ(s)}{\operatorname{argmin}}\, \max\big(g(s'') + h(s''), h(s_{start})\big)}.$$
2. Break ties in $\boldsymbol{s'_{set}}$ in favor of successor $\boldsymbol{s'}$ such that:
$$\boldsymbol{s' = \underset{s'' \in s'_{set}}{\operatorname{argmin}}\, c(s, s'') + h(s)}.$$
3. Break the remaining ties, **always selecting the first one**.
4. Updates (**N.B. My graphs are bidirectional, so successors are the same as predecessors**):
   - Update $\boldsymbol{g(s)}$ if s is not the **start** in this way:
   $$\boldsymbol{max(g(s), s' = \min_{s'' \in pred(s)} g(s'') + c(s'', s)\,,\, \max_{s'' \in succ(s)} g(s'') - c(s, s'')}.$$
   - Update $\boldsymbol{h(s)}$ if s is not the **goal** in this way:
   $$\boldsymbol{max(h(s), s' = \min_{s'' \in pred(s)} h(s'') + c(s'', s)\,,\, \max_{s'' \in succ(s)} h(s'') - c(s, s'')}$$
5. Go to $\boldsymbol{s'}$ and repeat **until the goal is reached**.

Probably because they are directional graphs or because I misinterpreted the paper, but the heuristic must always be a little smaller than what you estimated otherwise you loop forever. For the example graph there were no problems, but in general, for this reason, I used the $\boldsymbol{Euclidean\ distance - 0.01}$ as a heuristic thus avoiding loops. One hypothesis I have is that in both $\boldsymbol{g(s)}$ or $\boldsymbol{h(s)}$ update rule, let's call them $\boldsymbol{gh(s)}$, the formulae $\boldsymbol{\min_{s'' \in pred(s)} gh(s'') + c(s'', s)}$ and $\boldsymbol{\max_{s'' \in succ(s)} gh(s'') - c(s, s'')}$ of one or more states converge to the same value never updating $\boldsymbol{gh(s)}$ and thus going back and forth. This is probably because of the exact calculation of the heuristic following the chosen criterion. Instead, subtracting from that value 0.01 does not converge by a little while avoiding the loop.

Animations of the execution of this algorithm are saved in the files in the folder "*./Animations/TB_LRTA_star_FALCONS/*" or shown step by step on the notebook and shows the execution on the graph seen in lecture and on a randomly generated graph.

4 Results

I performed the following algorithms:

- $\boldsymbol{LRTA^*(1)}$
- $\boldsymbol{LRTA^*(5)}$
- $\boldsymbol{LRTA^*(10)}$
- $\boldsymbol{LRTA^*LS(1)}$
- $\boldsymbol{LRTA^*LS(5)}$
- $\boldsymbol{LRTA^*LS(10)}$
- $\boldsymbol{TB\ LRTA^*\ FALCONS}$

For each of these I ran the algorithm on $\boldsymbol{n}$ (in my case $\boldsymbol{500}$) graphs of $\boldsymbol{m}$ (in my case $\boldsymbol{300}$) states by averaging the following statistics obtained [Table 1]:

- **Iteration**. The number of total repetitions of the algorithm until the heuristics (and the g function in the case of **TB LRTA* FALCONS**) converges.
  This statistic is useful to see **which algorithm converges heuristics the fastest**.
- **Total time**. Total time for the algorithm to arrive at the convergence mentioned above.
  This statistic is useful for measuring **how fast an algorithm achieves convergence. N.B.** this statistic is not 100% reliable since the algorithms, most likely, can be optimized (even the simple minimum calculation done multiple times). However, it still manages to give an idea of how fast an algorithm is, but it is not reliable in a comparison with others.
- **Max time in iterations**. The maximum time recorded for a single iteration.
  This statistic is useful for measuring **how fast an algorithm is at finishing the most complex iteration**, which is not necessarily the first.
  This statistic also carries the problem mentioned above.
- **Total number of step forwards**. Number of steps taken to achieve goal and convergence (forward search steps do not count).
  This statistic is useful for measuring **how many steps are needed to reach convergence for an algorithm**. It is also useful for comparison with other algorithms.
- **Max number of step forwards in iterations**. Maximum number of steps taken in a single iteration (forward search steps do not count).
  This statistic is useful for measuring **how many steps are needed to reach the goal in the most expensive iteration**. It is also useful for comparison with other algorithms.
- **Max number of states in memory**. Maximum number of states in memory throughout the algorithm.
  This statistic is useful to show **how much memory is being occupied**. In general, it shows how much the trade-off of adding a few states in memory reduces the number of: tries before convergence is reached and the number of steps taken.
  Whenever a state is assigned to a variable or inserted into a list $\boldsymbol{1}$ is added to this statistic, otherwise $\boldsymbol{1}$ is taken away. However, throughout the process the maximum value of states is remembered to simulate the computational $\boldsymbol{O()}$ of the space.

Taking **LRTA*(k)** with $\boldsymbol{k = 1}$ as a reference we can make some comments on the variants [Table 1]:

- The basic versions with $\boldsymbol{k > 1}$ greatly **improve convergence**, with fewer steps paying little memory and very little extra time. **N.B.** in this case, since it is the **same algorithm**, the times can be compared.
  Probably this occurs because looking $\boldsymbol{k}$ **states ahead** explore solutions that would have been reached anyway by the basic algorithm but starting over and thus **repeating "unnecessary" steps**.

- The **LRTA*LS(k)** variant, you can see [Table 1], as per theory, that it corresponds to **LRTA*(k)** if $k = 1$ because essentially no queue is initialized and thus the resulting tree coincides with the successors of the current state, and thus **the choice is the same**. The only statistics that change are the temporal and spatial ones:
  - **The temporal one**, as already mentioned, cannot be compared between different algorithms for optimization reasons.
  - **The spatial one**, on the other hand, unlike **LRTA***, a tree with successors is built and kept in memory and that is why it takes up **more memory**.

  Exactly like **LRTA***, **LRTA*LS** with $k > 1$ also has the same advantages and disadvantages. In conclusion, as can be seen from the table [Table 1], **LRTA*LS** was the **best algorithm in terms of efficiency but the worst in terms of space**.

- The **TB LRTA* Falcons** variant aims to **speed up the convergence of LRTA*** and indeed it succeeds. This variant can be thought of as a **trade-off between LRTA* and LRTA*LS** in that it is not the best in convergence and time (**N.B.** here it is probably an optimization problem) but carries with it **few states in memory**. Another statistic that might be useful to see is the number of steps taken. Because of this large number of steps, this algorithm probably **improves the heuristics better than the others**.

	Iteration	Total_time	Max_time_in_iterations	Total_number_of_step_forward	Max_number_of_step_forward_in_iterations	Max_number_of_states_in_memory
LRTA*(k) with k = 1	19.728000	0.791000	0.231000	3301.524000	948.562000	3.000000
LRTA*(k) with k = 5	11.842000	0.843000	0.324000	1676.614000	604.848000	5.974000
LRTA*(k) with k = 10	9.108000	0.850000	0.371000	1130.018000	456.810000	10.232000
LRTA-LS*(k) with k = 1	19.728000	1.228000	0.367000	3301.524000	948.562000	12.256000
LRTA-LS*(k) with k = 5	9.130000	0.816000	0.346000	1065.116000	418.206000	23.068000
LRTA-LS*(k) with k = 10	7.038000	0.891000	0.425000	803.254000	353.710000	31.698000
TB LRTA* FALCONS	13.128000	3.526000	0.805000	5499.972000	1240.168000	9.818000

Table 1. Table with all the average statistics collected on the various algorithms. The worst results are highlighted in red and the best in green

5 References

[1] LRTA*(k) - Carlos Hernández and Pedro Mesuguer
[2] Improving LRTA*(k) - Carlos Hernández and Pedro Mesuguer
[3] Speeding up the Convergence of Real-Time Search - David Furcy and Sven Koenig