Report Planning & Reasoning Project
A Novel approach for Learning Real-Time A*

Fabrizio Casadei - Matricola: 1952529

SAPIENZA
UNIVERSITÀ DI ROMA

January 23, 2022

Chapter 1

Overview

This report talks about the *Learning Real-time A** and a set of new possible variants proposed.
First, we will start with describing of the classical algorithm and how it works, then we will pass on the introduction of the problems used for testing, at the end we will expose and explain the new variants with a special highlight on the outcomes achieved.
The whole report is based on the relative Python code implementation developed for this purpose.
The *LRTA**, and its variants born like solver for search problems, to be more precise for informed search problems, in which an estimation of how far or how close a certain state respects the goal, can be determined. These algorithms, like *Enforced Hill Climbing* or *Best First Search*, are used in many planners, one example is the *Fast Forward* (FF).
These search Algorithms suffer of different problems, both in terms of performance and final results. Referring to the performance expected, we can have big time duration to achieve a result or we can have a memory usage too large to search in the state space. While for the results, search can lead to the goal but not in the optimal manner (local maximum)
Keeping in mind this, we are now going to describe The LRTA*.

Chapter 2

The LRTA*

How described in the introduction, the *Learning Real-time A** is an informed search algorithm that extends *A**, hence, utilizes the concept of *Heuristic* to determine how distant is a certain state from the Goal.
The *Heuristic* is then combined with the affective cost to reach a new State from the current State to establish the *Evaluation function* with this formula: $f(s') = c(s, s') + h(s')$ with s the actual state and s' the next state.
The *heuristic* $h(s)$, must be admissible so: $h(s) <= h(s)^*$ with $h(s)^*$ the real cost to reach the goal. In other terms $h(s)$ must be an underestimation or equal to the real cost.
The *LRTA** algorithm is optimal, so it always outcomes the best solution, and whether the goal can be reached from every state (explorable domains) it's complete, so always find a solution. The bigger drawback is the time needed to reach the optimal plan.
Real-time search algorithms like *LRTA**, are designed for agent tasks in which the environment is initially unknown. The actions should be chosen in a limited time while having information only in a restricted part of the problem, the actual state.
The *heuristic* is used to have a limited lookahead for better choices.
The key concept around the *LRTA** is reducing this gap between $h(s)$ and $h(s)*$ through updates, in order to have a higher $h(n)$ which is still admissible (*heuristic refinement*).
For doing this we divide the algorithm into two main parts: the **simulations**, and the **execution**.
These two parts can be also implemented to run in a simultaneous manner, but for this treatment, we assume a sequentiality between them.
In the section of simulations, we update the *heuristic* of states crossed during the execution. This is done, taking the successors from a state s, and estimating for each one the cost function $f(s')$ and taking the lowest one. If the minimal $f(s')$ is greater than the actual $h(s)$, the heuristic of the state s is updated with this *cost function*. This is continuously performed until the finish of the simulation.
Each simulation ends whether the goal is reached or a certain chosen depth limit for the search has been reached.
The number of simulations can be fixed, in this case, the optimality is not guaranteed, or illimited.
In the latter case, the simulations end when a *Convergence* is found, which means a trial with no updates and the goal reached. This is the critical part in terms of time spent for an optimal solution.
In the second part, we use the *heuristic* learned, or better, updated during the simulation,

to execute and have an optimal plan (fewer number of states visited) which corresponds with the last simulation.
To sum up everything, it's shown the *pseudocode* of the algorithm:

LRTA*

```
1  BEGIN
2  initialize h = h_0;
3  REPEAT
4        LRTA*-simulation();
5  UNTIL(Convergence);
6  END
```

LRTA*-SIMULATION

```
 1  BEGIN
 2  set current state s = s_0;
 3  REPEAT
 4        expand the state s;
 5        find state s' with lowest f(s') = c(s, s') + h(s')
 6        IF f(s') > h(s)
 7              update h(s) to f(s')
 8        move to s'
 9  UNTIL s ∈ S_goal || depth ≥ max_depth
10  END
```

Chapter 3

The Problems

In this chapter, we are going to describe problems used to test both the classical *LRTA** and the new versions introduced. The *search-space* can be defined with the following tuple: $(S, A, c, S_{init}, S_{goal}, h)$, in which:

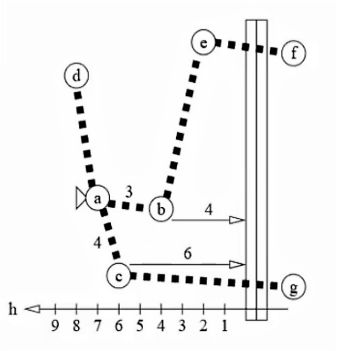
- S, is the set of finite states for the problem;
- A, is the set of finite actions for each state s;
- c, is the cost function from s to s', with $s' = a(s)$ and $a \in A$. Can be also expressed in this form $c(s, s')$;
- s_{init}, is the initial state of the problem
- S_{goal}, is the final goal state/states for the problem
- h, is the heuristic that guides the solving of the problem. At the beginning, $h = h_0$ with h_0 the initial heuristic defined freely.

Two important assumptions are been adopted: (i) in every action and in every state exists a reverse action to go back to the previous state, (ii) from the s_{init} at least one goal can be reached.
The *state-space* can be always represented using the graph data structure, in which states are nodes and actions are represented by the edges. From the assumption on the reverse action we know that graph is oriented and not directed. During this treatment, the concepts of states and actions can be exchanged with nodes and edges. The first two problems are used to test two important properties of the *LRTA**, which is the *optimality* and the *completeness* (no dead ends). These problems are been taken directly from the course held by *Prof. Paolo Liberatore* and can be found at the following website: `http://www.diag.uniroma1.it/liberato/planning/`.
The other two problems are represented by the famous *8-puzzle* brain teaser (useful to test speed on convergence given the big state space), and a Problem called *Escape*.

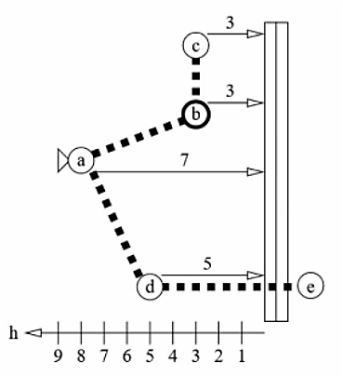
3.1 Optimality and Completeness test problem

These two problems are of the same typology, we start from an initial node, in both cases a, and we want to find a state at left of the barrier. In the first problem, the goal states

S_{goal} are f and g, while in the second, only e.
The cost from one state to the other is represented by the squares between them, i.e. cost(b,e) in the first problem is 10. The initial *heuristic* h_0 has been evaluated relaxing the problem, and using the horizontal distance from a state to the barrier, so from the barrier to the right side we have $h_0 = 0$, while on the left side, values greater than 0. for example in the first problem $h_0(c) = 6$.
The first problem shows how even if guided from a *heuristic* not proper to achieve directly the optimal result, in fact, it prefers the path $a \to b \to e \to f$ rather than $a \to c \to g$ (optimal), the algorithm updating h, find the *optimality*.
The second problem for the same over relaxed *heuristic* reason faces the algorithms with a *dead-end* node c, and prove that the updates solve this issue, increasing the value of the heuristics in order to choose different paths.



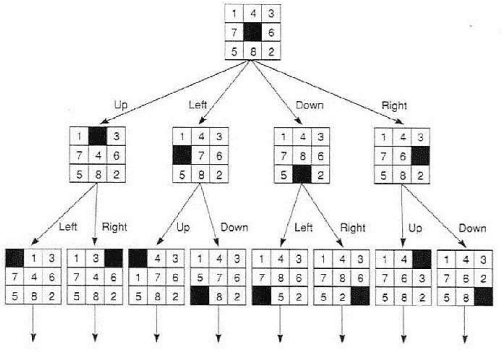
Optimality test problem



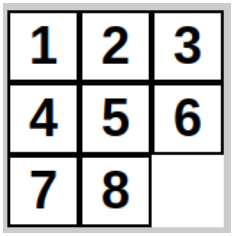
Dead-End test problem

3.2 8-puzzle

The*8-puzzle* game is a very popular brain teaser obtained from a simplification on the more famous *15-puzzle* born around 1900. The aim of this game, is from an initial configuration like in the image above, to arrive at a final configuration of numbers as shown below:



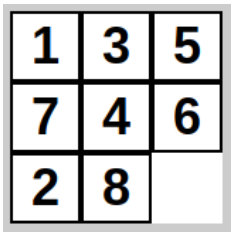
The 8-puzzle problem



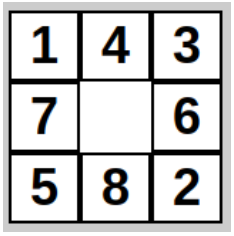
Final configuration

Each move can be done by exchanging the free tile with a tile that is horizontally or vertically adjacent to it. How is shown in the image above.
Each move has a cost equal to 1. The initial heuristic has been obtained relaxing the

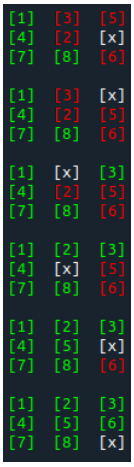
problem and is equal to 8 minus the number of misplaced tiles (excluding the free one). For example, the $h(s_{init})$ in the above image is 4.
This problem is useful to test the speed of the algorithm, given the bigger state-space respect the other problems. Considering only the configurations that are solvable, so with an even number of inversions, we obtain $9!/2 = 181440$ possible states.
As problems are been used two configurations: an easy one (at least 10 moves necessary), and a medium one (at least 14 moves necessary).



Easy difficulty config.

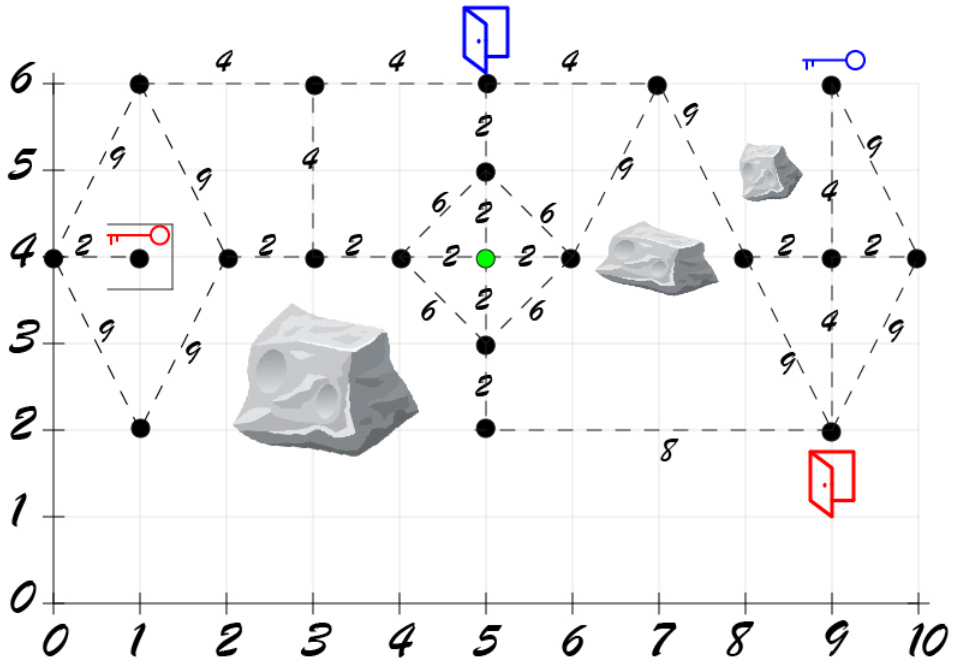


Medium difficulty config.



Example of an optimal plan for a straight forward case.

3.3 Escape



The graphical representation of the Escape problem

Escape represents a problem that has as aim to get out of the environment. You can leave through two different doors, red or blue, but these doors are locked. You can unlock them using the key of the respective colour.

The states in this problem are formed by a tuple: (p_x, p_y, rk, bl) which are:

- p_x, the horizontal position on the Cartesian plane;
- p_y, the vertical position on the Cartesian plane;
- rk, a boolean value which indicates the holding of the red key
- bk, a boolean value which indicates the holding of the blue key

With the actions, you can move from one position (which doesn't correspond with the state in this problem) to another with a certain cost $c(s, s')$.If the move is horizontal or vertical the cost is around two times the *Euclidean distance*, otherwise three times (diagonal).
The initial state is represented in position from the green dot. Therefore is the state $s_{init} = (4, 5, False, False)$. The goals S_{goal} as explained are represented by the following four states:

1. $(6, 5, False, True)$
2. $(6, 5, True, True)$

3. $(1, 9, True, False)$
4. $(1, 9, True, True)$

Obviously, it's easy to understand that goals number 2 and 4 are for sure not optimal. The initial heuristic h_0 is obtained combing different times the *Manhattan distance*, this is the pseudocode:

ESCAPE h_0

```
1  BEGIN
2  s = the actual state;
3  pos = position in s
4  dist = the Manhattan distance function
5  red distance = dist(pos, red key position) + dist(red key position, red door position)
6  blue distance = dist(pos, red blue position) + dist(blue key position, blue door position)
7  IF s has red key
8        red distance = dist(pos, red door position)
9  IF s has blue key
10       blue distance = dist(pos, blue door position)
11 h_0 = min(red distance, blue distance)
12 END
```

Chapter 4

The novel LRTA*

Having treated the classical *LRTA** and the problems used to test the algorithms, we can finally pass to the description of the Variants introduced for the *LRTA**. In this chapter we analyze and discuss the proposals, then in the final chapter, we expose the results.
Several types of *LRTA** variants have been theorized in the past years, improving performance in terms of memory and time. With this work, we focus on optimizing time.
We propose different modifications that can be used as modules to inject directly into the *LRTA**. These modules are independents and can be combined at will.

4.1 f(s) disambiguation

This variant, introduce in the *LRTA** the possibility of an additional insight when we look for the next state. In fact, we know from the basic algorithm that we always want to pass at the next node with the minimum evaluation function $f(s)$. But what do when the minimum $f(s)$ are equal between two or more states?
In the classical approach, each one can be selected, is not important. In theory, this choice can lead to a very important contribution in many problems, leading to a shallower search or fewer simulation numbers for the convergence.
For doing this has been proposed a one-step look ahead in the states that have equal $f(s')$. which is the minimal one. Hence we memorize these states: $s1', s2', \ldots, sn'$ with equal $f(s')$, for each one we expand and store information about the evaluation function of the children, looking for the minimum one.
If there is a fewer value of $f(s'')$ respect the others we chose the parent node as the next node for the search. Otherwise can be selected randomly as before.
Doing this we have a trade-off from which not all the problems can benefit: the number of states analyzed increases, but this can lead to the temporal avoidance of visits for states that are of less worth in that specific step.
For sure using this technique we expect to perform a less number of simulations, which doesn't mean automatically less time, another key aspect is the depth of the search.
In the appendix section is also presented an alternative version of this technique.

4.2 Dynamic depth limit

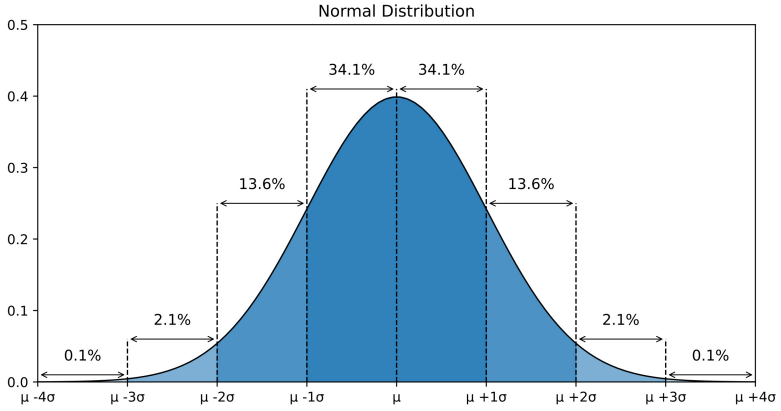
This and the following modules affect the depth of the search, but they do this in a totally different manner.
Before explaining the concepts behind it, it's important to highlight the fact that here we are assuming to have a certain max depth value for the search, while an unlimited number of simulation numbers are available to find convergence.
This variant exploits the notion of the depth used to reach the goal in a certain simulation n, to limit the depth in the simulation $n+1$. This limitation is equal to the depth reached in the simulation n divided by a *scale factor* which can be chosen freely. In the execution performed has been chosen a *scale factor* equal to two. Hence we limit the next iteration to half of the actual depth.
If a simulation doesn't find the goal within the actual max depth, the depth limit is reset to the default one.
This limitation can lead to a faster updating for the heuristics that are really important in the search. Considering that an optimal plan can have a number of actions, and Consequently a number of states visited, very smaller respect the maximum depth used (different orders of magnitude greater), the updates of a limited part closer to the initial state, speed-up the convergence in terms of time.
This concept of updating the states closer to the start should not be used to reduce too much the depth, increasing the scale factor, because even if it's slow to update using deep states, can have a key role in the learning.
Using this variant is expected a greater number of iterations that are smaller in depth.
Below is shown the pseudocode of this variant, which modifies the simulation loop.

LRTA* Dynamic Depth-simulation

```
1  BEGIN
2  max_depth = depth limit
3  set scale factor s_f
4  set current state s = s_0;
5  REPEAT
6       expand the state s;
7       find state s' with lowest f(s') = c(s, s') + h(s')
8       IF f(s') > h(s)
9            update h(s) to f(s')
10      move to s'
11      IF s ∈ S_goal
12           max_depth = depth / s_f
13      ELSE-IF depth = max_depth
14           max_depth = depth limit
15 UNTIL s ∈ S_goal || depth ≥ max_depth
16 END
```

4.3 Gaussian restart

This technique and the next one uses the concept of restarting the simulations even if the depth limit has not been attained. A simulation can restart following certain criteria, rebooting the execution from the initial state with the heuristics of the intermediate states updated.
The main idea is the introduction of a *random uniform distributed variable* between 0 and 1. This value will be the random criterion to restart the simulation. The discriminant which decides whether the random value is sufficient or not is a *Gaussian distribution*. More in detail, we define a *Normal distribution* in which the x-axis represents the iteration number in the simulation while on the y-axis we have the probability value for the restart. Therefore each iteration has not the same probability to be the last one.



The Normal probability distribution

The *mean* of the normal distribution is given by half of the max depth for the search, scaled to be a value between 0 and 10, while the *variance* can be chosen at will, in order to have more or less restarts.
Certainly, also the number of iteration must be scaled to evaluate the correct probability, and it's scaled like the mean, dividing with 10 to the *order of magnitude* of the iteration number.
This allows to have a *stochastic* mechanism for managing the restarting. In the execution has been used a variance (sigma) = 1, which is expected to bring a big increase of simulations.

4.4 Increment restart

This final Variant on the restart of the simulation, make a simple reasoning.
When in a certain step of the simulation there is an update of the heuristic which is very consistent (how much is consistent it's up to you to decide), can be a good idea to restart without losing time and memory continuing the search of states, that you could not visit in future given the important update.
So when you update, before passing to the next node s', you compute the difference between the old value and the new value of the heuristic for the actual state s:
$\Delta = h_{new}(s) - h_{old}(s)$, then it's computed the delta percentage respect the old value of the heuristic: $\Delta\% = \frac{\Delta}{h_{old}(s)} \cdot 100$.
After this you use a pre-fixed threshold which is an important discriminant for the update, to decide whether to restart or continue the current simulation.
Even there is expected to reach a greater number of simulations, that are shorter.

Chapter 5

Conclusion

In this last section, we are going to talk about the results achieved. Even if we have tested different types of algorithms with all the problems, we suggest focusing the attention on numbers coming from 8-puzzle.
Escape problem it's an interesting example of performance on small state space, and we will see how these variants behave in this other case. To be more complete are also shown time performance for Optimality and Dead-end test.
Starting from these lasts, first the Optimality test problem:

	time [ms]	**n. of simulations**
Classic LRTA*	3.390	4
LRTA*: f(s) disambiguation	0.684	4
LRTA*: Dynamic depth	0.789	4
LRTA*: Gaussian restart	10.696	5
LRTA*: Increment restart	0.751	5
LRTA*: all variants included	10.286	6
LRTA*: all except f(s) disambiguation	8.571	5

The dead-end test problem:

	time [ms]	**n. of simulations**
Classic LRTA*	3.512	3
LRTA*: f(s) disambiguation	0.618	3
LRTA*: Dynamic depth	1.213	4
LRTA*: Gaussian restart	8.814	5
LRTA*: Increment restart	1.074	6
LRTA*: all variants included	11.254	7
LRTA*: all except f(s) disambiguation	10.861	7

From these data, we can infer that there is not an advantage from these techniques, but let's see how these numbers change, increasing the problem complexity and the consequent state space. We pass now the Escape problem.

	time [ms]	n. of simulations
Classic LRTA*	82.732	20
LRTA*: f(s) disambiguation	36.386	20
LRTA*: Dynamic depth	31.695	21
LRTA*: Gaussian restart	243.307	21
LRTA*: Increment restart	45.945	22
LRTA*: all variants included	220.712	25
LRTA*: all except f(s) disambiguation	234.589	24

In the end, we present the results for two configurations of the 8-puzzle.

	Easy configuration		Medium configuration	
	time [s]	n. sim.	time [s]	n. sim.
Classic LRTA*	32.957	112	9390.83	2335
LRTA*: f(s) disambiguation	44.499	110	26935.813	2204
LRTA*: Dynamic depth	15.110	133	6721.887	2358
LRTA*: Gaussian restart	10.537	112	2887.526	2533
LRTA*: Increment restart	16.239	112	7992.108	2490
LRTA*: all variants included	6.480	127	6356.298	2595
LRTA*: all except f(s) disambiguation	6.295	133	1583.447	2788

One first observation from the results that we can make, how suggested in the previous chapters, is the correlation between time and the number of simulations, which change from problem to problem.
As it's possible to see in the 8-puzzle case, all the techniques related to the depth are been behaved very performative. The one step ahead technique for disambiguation has not led to good results in this case, but we can intuitively imagine a real dependence on the problem type, How many values are similar? In the 8-puzzle, having a unitary cost for the actions, checks are made very often, even for states that cannot already benefit from this control.
One possible clue is to enable to *f(s) disambiguation* after a certain number of simulations, or more in deep in the search. In fact, we can see how the "Escape" problem reduces the execution time for the convergence, having from the start very different values of $f(s)$.
Dynamic depth limit & Increment restart are very easy to introduce in each kind of problem and produce a good improvement to the time necessary, the first depends only on the depth limit, while the second only on the threshold value used.
The Gaussian restart it's a bit harder to configure, really depending on the problem. More in detail, it's not obvious how to fit the normal distribution over the iterations to achieve the best result, but we can see how much improve the execution.
Combining these three last methods, it's preferred to choose a greater depth limit with respect to the one used in the classic LRTA*.

Appendix

An alternative f(s) disambiguation method

In this section, we propose a softly different approach for the f(s) disambiguation variant. In the one present previously, we perform the one-step look ahead for states that have the same minimum $f(s)$. Here we want to consider states not only with the minimal value of $f(s)$, but also the ones that have $f(s)$ enough close.
For doing this we have to explicit this concept of "enough close", introducing a parameter which has been named *Tolerance*. How the name suggests, we accept states for the look ahead that no exceed the value of this parameter, which should be selected between 0 (no tolerance, hence classical $f(s)$ disambiguation) and 1(accept $f(s)$ that are the double respect minimum).
So for each state we compute this value: $\frac{f(s)}{f(s)_{min}} - 1$, and we look if it's fewer respect the *tolerance*, if pass this check, we enlist it for the disambiguation. Then follow the same operations performed in the previous version of the f(s) disambiguation.
The result obtained, compared with the ones coming from the earlier variant, are shown below.

Problems	**LRTA*: f(s) disambiguation with tolerance**		***LRTA*: f(s) disambiguation***	
	time [s]	**n. sim.**	**time [s]**	**n. sim.**
Optimality	$0.649 \cdot 10^{-3}$	3	$0.684 \cdot 10^{-3}$	4
Completeness	$0.905 \cdot 10^{-3}$	4	$0.618 \cdot 10^{-3}$	3
Escape	$50.745 \cdot 10^{-3}$	20	$36.386 \cdot 10^{-3}$	20
8-puzzle: easy config.	3.726	27	44.499	110
8-puzzle: mid config.	$11.663 \cdot 10^{3}$	617	$26.936 \cdot 10^{3}$	2204

How it's possible to appreciate from these data, with this modification, the results are really satisfactory. If we look at the times and iterations number for the 8-puzzle, which is the best kind of benchmark here, we have more than 10 times fewer timings for the easy config. and less than half for the medium configuration, iterations number are reduced significantly as well.
Of course, the outcomes depend on the complexity of the problem, but also on the initial heuristics. In fact, making a lot of one-step look ahead when the heuristic is not properly defined, can lead to a worsening of the performance for problems that are quite straightforward, like in the *escape* case.