Int. J. Appl. Math. Comput. Sci., 2017, Vol. 27, No. 2, 273–290
DOI: 10.1515/amcs-2017-0020

D* EXTRA LITE: A DYNAMIC A* WITH SEARCH–TREE CUTTING AND FRONTIER–GAP REPAIRING

MACIEJ PRZYBYLSKI [a,*], BARBARA PUTZ [a]

[a]Institute of Automatic Control and Robotics
Warsaw University of Technology, ul. św. A. Boboli 8, 02-525 Warsaw, Poland
e-mail: {maciej.przybylski,bputz}@mchtr.pw.edu.pl

Searching for the shortest-path in an unknown or changeable environment is a common problem in robotics and video games, in which agents need to update maps and to perform re-planning in order to complete their missions. D* Lite is a popular incremental heuristic search algorithm (i.e., it utilizes knowledge from previous searches). Its efficiency lies in the fact that it re-expands only those parts of the search-space that are relevant to registered changes and the current state of the agent. In this paper, we propose a new D* Extra Lite algorithm that is close to a regular A*, with reinitialization of the affected search-space achieved by search-tree branch cutting. The provided worst-case complexity analysis strongly suggests that D* Extra Lite's method of reinitialization is faster than the focused approach to reinitialization used in D* Lite. In comprehensive tests on a large number of typical two-dimensional path-planning problems, D* Extra Lite was 1.08 to 1.94 times faster than the optimized version of D* Lite. Moreover, while demonstrating that it can be particularly suitable for difficult, dynamic problems, as the problem-complexity increased, D* Extra Lite's performance further surpassed that of D*Lite. The source code of the algorithm is available on the open-source basis.

Keywords: shortest-path planning, incremental heuristic search, mobile robot navigation, video games.

1. Introduction

Goal-directed navigation without accurate knowledge of the environment is a common problem in robotics and video games. As an agent follows a path to a stationary goal, it can discover changes within a certain range of sensors, which will require re-planning. Incremental heuristic algorithms are beneficial in this context; able to reuse knowledge from previous searches, substantially less computation time is needed for re-planning. In this paper, we focus on optimal algorithms, with limited discussion of suboptimal variants.

Incremental shortest-path searching algorithms are typically used in a sense-plan-act scheme. During the planning phase a stationary snapshot of the environment is used. Discrepancies between the known map and accurate map pertain to the appearance and disappearance of obstacles. Although in both the cases (appearance and disappearance) a previous search can be reused, some techniques can be optimal in some cases and not in others. Beyond the field of robotics and video games, these kinds of algorithms are applied to vehicle navigation, in which quick re-computation of the optimal path is crucial. Furthermore, as the majority of these algorithms are general enough to be applied to shortest-path searching in graphs, they are used across a variety of applications. In this paper, we describe D* Extra Lite that is a novel, general purpose, incremental shortest-path searching algorithm.

In the present study, we analyzed several algorithms that are suitable to general use (i.e., both appearance and disappearance of obstacles) (see, e.g., Trovato, 1990; Stentz, 1995; Podsędkowski, 1998; Podsędkowski *et al.*, 2001; Koenig *et al.*, 2004; Koenig and Likhachev, 2005b; Sun and Koenig, 2007; Sun *et al.*, 2008; Hernández *et al.*, 2015). In a test framework, we then implemented two of them; the popular, state-of-the-art D* Lite (optimized version) (Koenig and Likhachev, 2005b) and the recently-developed MPGAA* algorithm, which has been shown to outperform D* Lite (Hernández *et al.*, 2015) in some problems. Following the results of our analysis, we propose a new algorithm that appears to outperform both D* Lite (optimized version) and

[*]Corresponding author

MPGAA* in two-dimensional path-planning problems. Accordingly, we called the new algorithm ‘D* Extra Lite’. The main difference between D* Lite and D* Extra Lite is in the way in which affected portions of the map are reinitialized. The superior reinitialization process used by D* Extra Lite allows the main searching function to remain as simple as the original A* (Hart *et al.*, 1968).

In addition, we analyze algorithms that are only applicable to planning with the freespace assumption (e.g., Koenig and Likhachev, 2005; Hernández *et al.*, 2009; 2011; 2014) due to techniques that can be also used in general-purpose algorithms. The freespace assumption is a special case of incremental path-planning, in which an agent does not know the environment, and it is assumed that this unknown space is free; thus only obstacles can appear.

The paper is organized as follows. In Section 2 referring to problems in terms of graph searching, we present the aim of this paper. In a review of related work (Section 3), we discuss the present state of the art in this area. At the beginning of Section 4, we outline the main idea behind the D* Extra Lite algorithm. Next, we compare D* Extra Lite with D* Lite (its optimized version). Section 4, beyond the algorithm itself, includes a theoretical discussion of the properties of D* Extra Lite, as well as an extensive list of examples that illustrate the behavior of this algorithm in typical cases of environmental changes. Finally, in Section 5, we present and discuss the results of some tests.

2. Problem formulation

Similarly to other texts concerned with path-planning (e.g., Sturtevant, 2012; Stentz, 1995; Koenig and Likhachev, 2005b; Hernández *et al.*, 2015), in this analysis, we utilized a two-dimensional path-planning on an occupancy grid as our primary example. Within such a domain, usually an agent’s movements can be made to each of the four neighboring cells (used in the examples throughout the paper) or to each of the eight neighboring cells (used in the final experiments). If the neighboring map-cell is occupied, we assume that the transition remains possible, but the cost is infinite.

All feasible positions s of an agent on a map form a state-space S. A transition between two states is possible by execution of an action $a_{s,s'} \in A$, where A is the set of all feasible actions. For each action a cost function is defined: $cost(a_{s,s'}) \equiv cost(s,s') : A \to \mathbb{R}^+$. Although within a single search episode, the action cost is constant, it may change as observations are made between search episodes.

A transition function $\gamma(s, a_{s,s'}) : S \times A \to S$ returns the state s' achieved by execution of action $a_{s,s'}$. An inverse transition function $\gamma^{-1}(s, a_{s',s}) : S \times A \to S$ returns the state s' from which state s can be achieved. Herein it is assumed that for each pair of states, only one action joins them.

With the transition function we can define a set of successors $Succ(s) = \{s' \in S | s' = \gamma(s, a_{s,s'})\}$ and a set of predecessors $Pred(s) = \{s' \in S | s' = \gamma^{-1}(s, a_{s',s})\}$.

A path from s_1 to s_n is a sequence of such states, and for each pair of consecutive states $\langle s_i, s_{i+1} \rangle$ an action $a_{s_i,s_{i+1}}$ in A must exist.

The domain used in this paper can be represented as a graph $G = (V, E)$, assuming that there exists a direct state-to-node mapping $s \in S \to v \in V$ and a direct action-to-edge mapping $a \in A \to e \in E$. Therefore, in this paper, we use the terms *state* and *node*, and *action* and *edge*, interchangeably. However, it should be noted that with the implementation of graph-searching algorithms, nodes hold additional information, such as

- $parent(s)$, a neighboring node that leads to the start-state (or a goal state in the case of a backward search),
- $g(s)$, a value that represents the cost from the starting node to s (or the cost from the goal node to s in the case of a backward-search), calculated as follows: $g(s) = g(parent(s)) + cost(parent(s), s)$. In the case the state s can be omitted, this value is referred to as the g value.

The shortest-path search can be conducted from the starting node (forward-search), and from the goal node (backward-search). If the shortest path is found, an agent follows it until a change in the environment that may affect the path is observed. Such an observation is reflected by a cost change at the edges of the graph (costs of actions), and path re-planning is necessary.

3. Related work

The work of Stentz, who developed algorithms D* (Stentz, 1994) and Focussed D* (Stentz, 1995), and that of Koenig and Likhachev (2005b), who developed D* Lite, are the most recognizable contributions to date to address the problem of incremental shortest-path planning. All three of the above-mentioned algorithms run backwards, making them especially useful, as only the start-state changes between search episodes, and the goal-state remains unchanged; therefore, a significant part of the explored search-space remains relevant. The general aim behind these algorithms is to repair only the nodes (map cells) in the affected part of the map, that is, unless the current state of the agent is achieved by the searching algorithm, in which case, preference is to nodes that will lead to the current state of the agent.

Due to changes in the environment, a part of the search-space may become inconsistent. While one part may include under-consistent nodes with underestimated

g values (when new obstacles appear), the other part may feature over-consistent nodes with overestimated g values (when obstacles disappear). In the case of underestimated g values, the nodes need to be reinitialized. This allows the algorithm to assign new values, which will most likely be higher. In D*, Focussed D* and D* Lite, reinitialization occurs during the search. In order to detect an inconsistent node, each node has an additional value, denoted by the k value in Focussed D*, and the rhs value in D* Lite. Moreover, both algorithms utilize a heuristic cost to the agent's current state in order to guide searching. In the searching phase, both the algorithms perform the following two operations for each underestimated node. Before the new g value is set, each underestimated node is reinitialized. To assure that reinitialization precedes setting the g value, the list of open nodes must be sorted using a complex key value (i.e., $\min(rhs(s), g(s))$).

A different approach is to reinitialize the affected portion of the map and then to conduct a new search of only that part. As argued by Stentz (1995), such an approach is "inefficient when the robot is near the goal and the affected portions of the map have long 'shadows'." This approach, i.e., reinitialization of the entire affected section of the search-space, can be found in the work of Podsedkowski (1998; 2001), as well as in the Differential A* algorithm proposed by Trovato (1990), revisited and extended by Trovato and Dorst (2002). Differential A* may be the most similar algorithm to the D* Extra Lite presented in this paper. Unfortunately, an experimental comparison of Differential A* with Focussed D* or D* Lite has been neglected so far. However, with the understanding gained from the work on D* Extra Lite, also stated by Koenig and Likhachev (2005b), we can propose that the reinitialization of the entire open-list before each search episode inhibits efficient functionality of the Differential A* algorithm ($recompute\_OPEN()$ procedure in the pseudocode presented by Trovato and Dorst (2002)).

In order to avoid re-computation of the entire open-list, Focussed D*, D* Lite and D* Extra Lite use a biased key value. A biased key value, in addition to the calculated cost g and the heuristic cost h, includes the k_m value, which grows proportionally to the cost of the agent's transition between each search episode. Accordingly, it is ensured that nodes that were pushed to the open-list in the previous and subsequent episodes, will be popped in an order that will satisfy optimality requirements without reordering the entire open-list.

D*, Focussed D* and D* Lite each run backwards, from the goal to the current state of the agent, which allows for the unaffected search tree to be easily reused. However, it should be noted that it is also possible for forward-search algorithms to reuse a previously explored the search tree, e.g., LPA* (Koenig *et al.*, 2004) or Fringe-Saving A* (Sun and Koenig, 2007). Moreover, adaptive algorithms, which do not reuse search-tree, are continuously running from scratch in a forward direction. These algorithms improve their h values (they learn heuristics from previous searches), which substantially accelerates subsequent search episodes. The basic algorithm for this type is Adaptive A* (Koenig and Likhachev, 2005a). Findings strongly suggest that AA* is quicker than repeated A*, and can be faster then D* Lite, however only with the use of buckets in place of the heap as a priority queue for open-list managing.

We have discussed search-tree reuse and adaptive heuristic learning techniques; however, there are other algorithms based on AA* that make use of previously found paths, i.e., Path Adaptive A* (Hernández *et al.*, 2009) and Multi-Path Adaptive A* (MPAA*) (Hernández *et al.*, 2014). While these algorithms also run forward from the starting state, they can terminate before achieving the goal node. As these algorithms record previously constructed path(s), it is sufficient to construct a path that remains connected with the goal node (it has not been disconnected through changes in the environment). The most complex adaptive algorithm may be Tree Adaptive A* (Hernández *et al.*, 2011), which combines a reusable tree (like D* or LPA* algorithms) with reusable paths, and accordingly, heuristic improving.

All the adaptive algorithms mentioned above are limited to freespace assumption. Consequently, although they can manage new obstacles, where there is a shortcut available, their solutions fail to remain optimal. This problem has been solved with Generalized Adaptive A* (GAA*) (Sun *et al.*, 2008), which, in the case of a decrease in the edge cost, reestablishes the consistency of h values by performing an uninformed backward search throughout the explored search space. The recent Multi-Path Generalized Adaptive A* (MPGAA*) (Hernández *et al.*, 2015) algorithm demonstrates the benefits to be gained from the path reuse technique, by combining it with the GAA* algorithm.

As MPGAA* can deal with increasing, as well as decreasing edge costs, it can be compared with the D* Lite. In the work of Hernández *et al.* (2015), MPGAA* outperformed D* Lite in most cases. Therefore, in this paper, we compare D* Extra Lite with MPGAA* and D* Lite (an optimized version).

It is worth noting that GAA* (and MPGAA*) is similar to LSS-LRTA* (Koenig and Sun, 2009). LSS-LRTA* is a learning real-time algorithm that searches forwards and can be stopped before it finds the global solution. Reaching the goal can be assured, as after each searching phase there is a learning phase in which the algorithm updates the h values of each visited node. The learning phase is a function equal to the consistency reestablishing performed by GAA*. Without a computation time limit, LSS-LRTA* undertakes a global search, which makes it comparable to incremental planning algorithms. LSS-LRTA* has been shown to outperform D* Lite in some settings; for a discussion, refer to Koenig and Sun (2009).

Another novel and interesting approach is found in the Dynamically Pruned A* (DPA*) algorithm (van Toll and Geraerts, 2015). During its search, DPA* reuses previously found paths for states pruning. For nodes on a previously found path, DPA* does not expand the entire neighborhood, it rather expands only subsequent nodes from this path. In contrast, although DPA* reuses previously found paths to the aforementioned adaptive algorithms, it does not learn heuristics. DPA* was designed for crowd simulation on navigation meshes (sparse graphs), in which low memory requirements were more important than short running times. Although DPA* has been shown to be faster than repeated A* by 62%, D* Lite is faster than repeated A* by at least one order of magnitude (Koenig and Likhachev, 2005b). Therefore DPA* was not selected for our comparison.

4. D* Extra Lite

4.1. Intuition. In most of incremental heuristic search algorithms, the first search episode is equivalent to the regular A* algorithm, which expands consecutive search-space nodes until it reaches the goal node. This can be in the case of a forward search (e.g., LPA*, MPGAA*) or a backward search (e.g., D*, D* Lite). If the algorithm sets parent pointers (for example, D* Lite does not), these pointers form a tree with a root in the node from which the search originated. We refer to this tree as the search tree.

If any change is observed to affect the explored search space, particularly, an edge cost $e(s_1, s_2)$ has changed, a part of the visited search space (a branch of the search tree) has become inconsistent and must be re-explored. The inconsistent part of a search tree can be defined as a branch of a search tree that contains nodes supported by an edge $e(s_1, s_2)$. A node s_2 is supported by an edge $e(s_1, s_2)$ if the node s_1 is a parent of node s_2; furthermore, if a node s_2 is a parent of node s_3 and s_2 is supported by $e(s_1, s_2)$, then s_3 must also be supported by $e(s_1, s_2)$.

The g values of nodes that belong to an inconsistent search-tree branch, are either too high or too low. At that point, all incremental algorithms (such as D*, D* Lite, MPGAA*) distinguish between two situations—when the cost of an edge has increased and when it has decreased.

If the root of the search-tree has not changed, such is the case of incremental search algorithms running backwards (e.g., D* and D* Lite), the following observations can be made.

In the situation in which the cost of an edge $e(s_1, s_2)$ decreases, it is sufficient to reopen the s_1 node and continue the search. This is owing to the fact that the g values in the inconsistent part of the search-tree

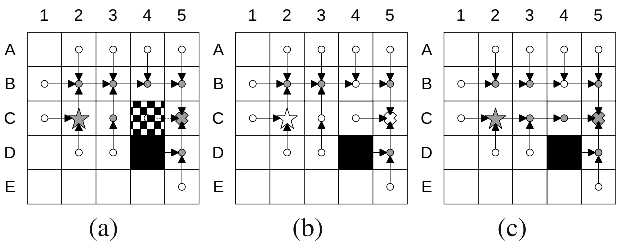

Fig. 1. Agent (star), following the move from $C1$ to $C2$, observes cell $C4$ to become free (chessboard) (a). Therefore, nodes s_{C3}, s_{B4} and s_{C5} have to be re-opened (b). Figure (c) illustrates search-space following re-planning. White inner shape: open nodes, gray inner shape: closed nodes, arrows: parent node pointers, cross: goal node, black squares: obstacles, dashed line: affected edges.

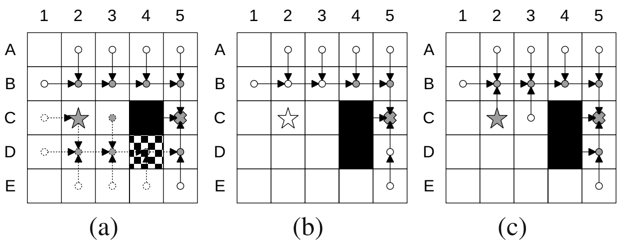

Fig. 2. Agent (star), following the move from $C1$ to $C2$, observes cell $D4$ to be occupied (chessboard) (a). The entire branch supported by the edge $e(s_{C4}, s_{C5})$ must then be cut (b). As the space left by the cut branch may be filled by another branch, the nodes that neighbor the cut branch are re-opened. A new optimal solution emerges that is on a different branch from the initial branch (c).

are higher than should be (nodes are over-consistent). Optimal path-searching algorithms change the g value only if $g(s_2) > g(s_1) + cost(s_1, s_2)$, which also functions to prevent the algorithm from re-exploration of consistent nodes. Moreover, in this situation, the affected branch of the search tree cannot shrink (it can grow or stay unchanged). In Fig. 1, an example of edge-cost decrease is depicted. As cell $C4$ became free, the cost of the corresponding edges $e(s_{C3}, s_{C4})$, $e(s_{B4}, s_{C4})$ and $e(s_{C5}, s_{C4})$ decreased from infinity to one (Fig. 1(a)). As explained, there is no need to cut any branch; however, nodes s_{C3}, s_{B4} and s_{C5} must be re-opened (Fig. 1(b)).[1]

If the cost of an edge $e(s_1, s_2)$ increases, all nodes in the branch of the search-tree supported by this edge become under-consistent, which means that its g values are lower than they should be. As the condition $g(s_2) > g(s_1) + cost(s_1, s_2)$ is not fulfilled, the simple reopening of s_1 will not lead the algorithm to re-establish consistency. Therefore, before the algorithm begins new search, it must make such nodes over-consistent. This

[1]In the example, in addition to nodes s_{C3}, s_{B4} and s_{C5}, a start node s_{C2} has been opened to properly handle the termination condition, which is a property of the D* Extra Lite algorithm explained later.

is achieved by setting its g values to infinity or by marking them as unvisited. If the cost of the $e(s_1, s_2)$ edge increases, the affected search-tree branch may shrink or even—covered by other unaffected branches—it may disappear. Thus, parent nodes of nodes that belong to the affected area may change radically, as shown in Fig. 2.

The main issue is to decide which nodes should be made over-consistent and which nodes should be re-explored. The idea behind the D* Lite algorithm is to make over-consistent and to re-explore only those nodes that lead towards the starting-node (i.e., the current state of the agent). For both the operations (making node over-consistent and node re-exploration) the same open-list is used. This is made possible by introduction of the rhs value for each node, which ensures that nodes will be made over-consistent and re-explored in the correct order. An advantage of this approach is that only necessary nodes are reinitialized and re-explored. A disadvantage of D* Lite is that making some nodes inconsistent and reopening their neighbors is a part of a search loop that involves operations on the open list, and this may hinder the efficiency of the algorithm.

As already discussed, following an increase in the cost of the edge $e(s_1, s_2)$, the entire affected searchtree branch becomes inconsistent. The main idea behind the D* Extra Lite algorithm is to cut the whole branch at once. This is unlike D* Lite, which while searching, reinitializes single nodes. Branch cutting is a simple recursive operation that makes nodes unvisited without employing the open list. After the branch cut there will be a gap in the frontier fringe to be repaired. Therefore, apart from reopening the s_1 node, all nodes belonging to neighboring branches will also need to be reopened. Only then is the search-space reinitialized and ready for a new search episode.

4.2. D* Extra Lite algorithm.

D* Extra Lite, like other incremental algorithms, operates on procedures that utilize a sense-plan-act scheme. In our implementation, D* Extra Lite shares such basic procedures with D* Lite (Algorithm 1). The algorithms start with an initial map update and a search-space initialization (lines 3–4 in Algorithm 1). The main loop (lines 5–10 in Algorithm 1) iteratively runs searching, action selection and execution, map update and reinitialization. The SEARCH() procedure consists of another loop that repeatedly performs SEARCHSTEP() while a goal condition has not been met, and the open list is not empty. The ACTIONSELECTION() procedure chooses the action (leading to successive state) that will achieve the goal with the least cost. The REINITIALIZE() procedure will instantly terminate if no change is observed.

Procedures that distinguish D* Lite from D* Extra Lite are shown side by side in listings Algorithms 2[2] and 3, respectively, but even within these procedures, several elements are similar.

Algorithm 1. Procedures common for the D* Lite and D* Extra Lite algorithms.

```
1: function MAIN()
2:     $s_{last} = s_{start}$
3:     MAPUPDATE()
4:     INITIALIZE()
5:     while $s_{start} \neq s_{goal}$ do
6:         if NOT SEARCH() then
7:             return goal is not reachable
8:         $s_{start}$ =ACTIONSELECTION($s_{start}$)
9:         MAPUPDATE()
10:        REINITIALIZE()
11: function SEARCH()
12:     while open-list is not empty do
13:         if SOLUTIONFOUND() then
14:             return true
15:         SEARCHSTEP()
16:     return false
17: function ACTIONSELECTION($s_{start}$)
18:     return $\arg\min_{s' \in Succ(s_{start})}(cost(s_{start}, s') + g(s'))$
```

Both the algorithms must operate while the agent's start state changes between searching episodes. In heuristic search algorithms, the key value to prioritizing an open list is usually calculated as a sum of an heuristic value $h(s_{start}, s)$, which in the case of a backward-search is the cost-to-start, and the $g(s)$ value, is the cost-from-goal. Owing to an agent's transitions toward decreasing g values, h values should be recalculated. Recalculation of the key for each open node, and reordering of an open list, would hinder the efficiency of the search algorithm. Therefore, another solution is used.

If the agent changes its state from the previous start state s_{t0} to the new start state s_{t1}, than for some nodes previously calculated h values are underestimated, while other are overestimated. If the h value is underestimated, i.e., $h(s_{t0}, s) < h(s_{t1}, s)$, the node will be removed from the top of the open list too early; thus its key has to be recalculated and the node has to be re-pushed to the open list (lines 14–17 in Algorithms 2 and 3). A more serious situation is when the h value of a node is overestimated, i.e., $h(s_{t0}, s) > h(s_{t1}, s)$. In this case, the node might be removed from the top of the open list too late and the algorithm will not find the optimal solution. In the worst case, the h value will be overestimated by $h(s_{t0}, s_{t1})$ (Fig. 3). To avoid overestimated h values, a bias value k_m can be added to the heuristic calculated for nodes added after agent's transition (line 2 in Algorithms 2 and 3). If the k_m value is increased by $h(s_{t0}, s_{t1})$, all nodes pushed to the open list before the agent's transition will have been underestimated (or exactly) h values, i.e.,

[2]The pseudocode of D* Lite (optimized version) presented here is not an exact copy of the pseudocode by Koenig and Likhachev (2005b); however, it is the same algorithm without any modifications.

Algorithm 2. D* Lite (optimized version) procedures.

```
 1: function CalculateKey($s$)
 2:     return
    $[\min(g(s), rhs(s)) + h(s_{start}, s) + k_m; \min(g(s), rhs(s))]$
 3: function SolutionFound()
 4:     return $key$(TopOpen()) $>=$ CalculateKey($s_{start}$)
    AND $rhs(s_{start}) <= g(s_{start})$
 5: function Initialize()
 6:     $k_m = 0$
 7:     for all $s \in S$ do
 8:         $g(s) = rhs(s) = \infty$
 9:     $rhs(s_{goal}) = 0$
10:     PushOpen($s_{goal}$, CalculateKey($s_{goal}$))
11: function SearchStep()
12:     $s =$TopOpen()
13:     PopOpen()
14:     $k_{old} = key(s)$
15:     $k_{new} =$ CalculateKey($s$)
16:     if $k_{old} < k_{new}$ then
17:         PushOpen($s$, CalculateKey($s$))
18:     else if $g(s) > rhs(s)$ then
19:         $g(s) = rhs(s)$
20:         RemoveOpen($s$)
21:         for all $s' \in Pred(s)$ do
22:             if $s' \neq s_{goal}$ then
23:                 $rhs(s') = \min(rhs(s'), cost(s', s) + g(s))$
24:             UpdateVertex($s'$)
25:     else
26:         $g_{old} = g(s)$
27:         $g(s) = \infty$
28:         for all $s' \in Pred(s) \cup s$ do
29:             if $rhs(s') = cost(s', s) + g_{old}$ AND $s' \neq s_{goal}$ then
30:                 EvaluateRhs($s'$)
31:             UpdateVertex($s'$)
32: function Reinitialize()
33:     if any edge cost changed then
34:         $k_m = k_m + h(s_{last}, s_{start})$
35:         $s_{last} = s_{start}$
36:         for all directed edges $(u, v)$ with changed cost do
37:             $c_{old} = cost(u, v)$
38:             update edge cost $cost(u, v)$
39:             if $c_{old} > cost(u, v)$ then
40:                 if $u \neq s_{goal}$ then
41:                     $rhs(u) = \min(rhs(u), cost(u, v) + g(v))$
42:             else if $rhs(u) = c_{old} + g(v)$ then
43:                 if $u \neq s_{goal}$ then
44:                     EvaluateRhs($u$)
45:             UpdateVertex($u$)
46: function UpdateVertex($s$)
47:     if $g(s) \neq rhs(s)$ then
48:         PushOpen($s$, CalculateKey($s$))
49:     else if $g(s) = rhs(s)$ AND $open(s)$ then
50:         RemoveOpen($s$)
51: function EvaluateRhs($s$)
52:     $rhs(s) = \infty$
53:     for all $s' \in Succ(s)$ do
54:         $rhs(s) = \min(rhs(s), cost(s, s') + g(s'))$
```

In the pseudocode, the following (though not explained elsewhere) functions are also used: TopOpen()—returns the node with the lowest key in the open list; PopOpen()—removes the node with the lowest key in the open list; PushOpen(s, k)—if node s is not open, it inserts s to the open list with key k, if node s is open, it updates the priority (if necessary); RemoveOpen(s)—removes node s from the open list; $open(s)$—indicates if node s is in the open list.

Algorithm 3. D* Extra Lite procedures.

```
 1: function CalculateKey($s$)
 2:     return
    $[g(s) + h(s_{start}, s) + k_m; g(s)]$
 3: function SolutionFound()
 4:     return TopOpen() $= s_{start}$
    OR ($visited(s_{start})$ AND NOT $open(s_{start})$)
 5: function Initialize()
 6:     $k_m = 0$
 7:     $visited(s_{goal}) = true$
 8:     $parent(s_{goal}) = NULL$
 9:     $g(s_{goal}) = 0$
10:     PushOpen($s_{goal}$, CalculateKey($s_{goal}$))
11: function SearchStep()
12:     $s =$TopOpen()
13:     PopOpen()
14:     $k_{old} = key(s)$
15:     $k_{new} =$ CalculateKey($s$)
16:     if $k_{old} < k_{new}$ then
17:         PushOpen($s$, CalculateKey($s$))
18:     else
19:         for all $s' \in Pred(s)$ do
20:             if NOT $visited(s')$ OR $g(s') > cost(s', s) + g(s)$ then
21:                 $parent(s') = s$
22:                 $g(s') = cost(s', s) + g(s)$
23:                 if NOT $visited(s')$ then
24:                     $visited(s') = true$
25:                 PushOpen($s'$, CalculateKey($s'$))
26: function Reinitialize()
27:     if any edge cost changed then
28:         CutBranches()
29:         if $seeds \neq \emptyset$ then
30:             $k_m = k_m + h(s_{last}, s_{start})$
31:             $s_{last} = s_{start}$
32:             for all $s \in seeds$ do
33:                 if $visited(s)$ AND NOT $open(s)$ then
34:                     PushOpen($s$, CalculateKey($s$))
35:             $seeds = \emptyset$
36: function CutBranches()
37:     $reopen\_start = false$
38:     for all directed edges $(u, v)$ with changed cost do
39:         if $visited(u)$ AND $visited(v)$ then
40:             $c_{old} = cost(u, v)$
41:             update edge cost $cost(u, v)$
42:             if $c_{old} > cost(u, v)$ then
43:                 if $g(s_{start}) > g(v) + cost(u, v) + h(s_{start}, u)$ then
44:                     $reopen\_start = true$
45:                 $seeds = seeds \cup v$
46:             else if $c_{old} < cost(u, v)$ then
47:                 if $parent(u) = v$ then
48:                     CutBranch($u$)
49:     if $reopen\_start = true$ AND $visited(s_{start})$ then
50:         $seeds = seeds \cup s_{start}$
51: function CutBranch($s$)
52:     $visited(s) = false$
53:     $parent(s) = NULL$
54:     RemoveOpen($s$)
55:     for all $s' \in Succ(s)$ do
56:         if $visited(s')$ AND NOT $parent(s') = s$ then
57:             $seeds = seeds \cup s'$
58:     for all $s' \in Pred(s)$ do
59:         if $visited(s')$ AND $parent(s') = s$ then
60:             CutBranch($s'$)
```

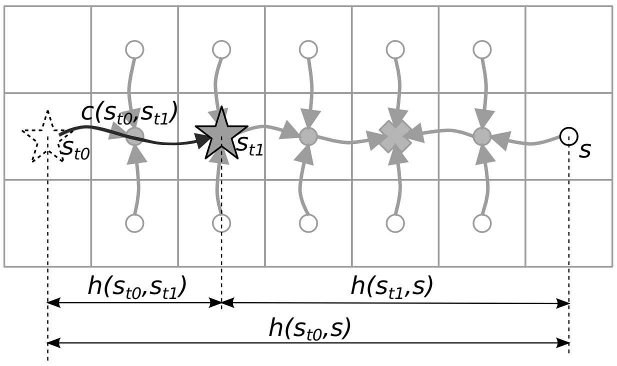

Fig. 3. If the agent (star) moves from s_{t0} to s_{t1}, then the h value of node s will change. For the s node, the previous h value was $h(s_{t0}, s)$; however, following the agent's transition, a new h value should be $h(s_{t1}, s)$. Assuming that for the heuristic function the triangle inequality $h(s_{t0}, s_{t1}) + h(s_{t1}, s) \geq h(s_{t0}, s)$ holds, the worst case of overestimation of the h value for node s, i.e., $h(s_{t0}, s) - h(s_{t1}, s)$, will be equal to $h(s_{t0}, s_{t1})$, though, in general, the traveled path cost $c(s_{t0}, s_{t1}) \geq h(s_{t0}, s_{t1})$. If, in the next search episode, the old h value is used for open-list prioritizing, the s node may be removed from the top of the open list too late. White inner shape: open nodes, gray inner shape: closed nodes.

$h(s_{t0}, s) \leq h(s_{t1}, s) + k_m$. The k_m value is increased at each reinitialization step (line 34 in Algorithm 2, line 30 in Algorithm 3). Koenig and Likhachev (2001) provide a proof that the use of a biased key value does not affect the optimality of the D* Lite algorithm. As D* Extra Lite differs from D* Lite only in the reinitialization, the proof presented by Koenig and Likhachev (2001), to some extent, can also be adapted to D* Extra Lite.

Another common element of D* Lite and D* Extra Lite is that a key used for open-list sorting is a pair of values: $key(s) = [key_1(s), key_2(s)]$ (line 2 in Algorithms 2 and 3). As both algorithms allow for the reopening of previously closed nodes, in the case of multiple nodes with equal $key_1(s)$ values, the tie-breaking rule should be to favor nodes with lower $key_2(s)$, which is $g(s)$ in D* Extra Lite and $\min(rhs(s), g(s))$ in D* Lite. This measure preserves optimality.

Referring to the previous explanations, changes in the environment affect branches of the search tree. The affected branch must be re-explored with special attention given to the case of edge cost increase. To re-explore such a branch, it must be reinitialized. D* Lite recognizes and reinitializes affected nodes while searching. Such an approach requires the use of the $rhs(s)$ value, which can be understood as a pre-$g(s)$ value ($g(s)$ takes the $rhs(s)$ value when s is over-consistent; line 19 in Algorithm 2). Comparing $rhs(s)$ and $g(s)$ values is a basic method for recognizing affected nodes (specifically under-consistent).

In contrast to D* Lite, the D* Extra Lite algorithm always reinitializes the entire affected under-consistent branch of a search tree. In consequence, at the beginning of searching, the search space has no under-consistent nodes, but over-consistent, consistent and unvisited nodes only. Therefore, it is possible to keep the SEARCHSTEP() (Algorithm 3) almost as simple as the A* algorithm.

The REINITIALIZE() procedure (Algorithm 3) is executed if the cost of any visited edge has changed. Tree-cutting is the first step of reinitialization (CUTBRANCHES() in Algorithm 3). For each edge with changed cost, the CUTBRANCHES() procedure does one of two possible operations. If the cost of the $e(u, v)$ edge has decreased, the v node is added to the list of seeds to be reopened later (lines 42, 45 in Algorithm 3). If the cost of the $e(u, v)$ edge has increased and node v is the parent of node u, the branch is cut starting from u (lines 46–48 in Algorithm 3). The cutting operation is simple, marking nodes unvisited. The CUTBRANCH() procedure is the recursive procedure which traverses throughout the branch, i.e., a next node to cut s' has to be such a predecessor of a current node s, that s is the parent of s' (lines 58–60 in Algorithm 3).

Each successor node s', such that $s \neq parent(s')$ is placed in the list of seeds (lines 55–57 in Algorithm 3). Although seeds are simply nodes to reopen, as they might be cut later, they cannot be merely pushed to the open list. Following the CUTBRANCHES() procedure, the REINITIALIZE() procedure pushes to the open list only these nodes from the $seeds$ list that remain visited and are not already open (lines 32–34 in Algorithm 3). This operation repairs the frontier-gap made by branch cutting.

D* Extra Lite succeeds if the start node is on the top of the open list, such as in the A* algorithm, or, if the start node has been closed in some previous searching episode and has not been cut in the reinitialization.

In the case of an edge-cost decrease, there may be a shorter path. Therefore, to preserve optimality, the start node should be reopened. However, not in every case of the edge-cost decrease does the start node need to be reopened. Herein, another optimization of the algorithm is possible. Assuming that $h(s_{start}, u)$ is admissible, for a decreased $e(u, v)$ edge cost, the start node s_{start} requires reopening only if $g(s_{start}) > g(v) + cost(u, v) + h(s_{start}, u)$. Otherwise, it is impossible for a path containing an $e(u, v)$ edge to be shorter. This condition is checked and applied in lines 42–44 and 49–50 of Algorithm 3. Such a situation is depicted in Fig. 4 in Episode 5.

4.3. Discussion of the algorithm.

D* Extra Lite is very similar to D* Lite, thus these algorithms have similar both time and space complexities. For example, the implementation using a binary heap has $O(n \log n)$ time complexity, where n is the number of expanded nodes. If there are only over-consistent or uninitialized nodes, both the algorithms are almost equivalent. In this case, for D* Lite, only lines 12–24 (Algorithm 2) of the SEARCHSTEP() function are used, while the SEARCHSTEP() function of the D* Extra Lite algorithm is designed for such a case exclusively. If edge costs decrease, reinitialization is also similar for both the algorithms—both reopen nodes that support a changed edge, (lines 39–41 in Algorithm 2 and lines 42–45 in Algorithm 3).

The main difference between D* Lite and D* Extra Lite is in the edge-cost increase. D* Lite reinitializes and re-expands only those nodes that lead toward the agent's current state, while D* Extra Lite always reinitializes the entire under-consistent branch of the search-tree. Therefore, in this study we investigate the complexity of reinitialization, only. We do this for the n-th searching episode. Let us consider a sufficiently large graph and an agent with a finite observation range. Since the observation range is finite, the number of changed edges is negligibly small. Therefore, the cost of operations in lines 32–45 (Algorithm 2) of D* Lite and lines 36–46 (Algorithm 3) of D* Extra Lite can also be neglected. The crucial operations are in lines 26–31 (Algorithm 2, which are a part of the SEARCHSTEP() function) and 32–34, 48 and 51–60 (Algorithm 3, which are mainly the CUTBRANCH() function).

Now, let us introduce the following numbers relevant to D* Extra Lite: n_{to_cut}—number of under-consistent nodes to cut; n_{to_open}—number of nodes to open in order to repair the frontier gap; $n_{open,DEL}$—number of nodes in the open-list; n_{tree}—number of nodes in the search tree after the previous search episode, such that $n_{to\_cut} + n_{to\_open} \leq n_{tree}$ and $n_{open,DEL} \leq n_{tree}$.

For D* Lite, we can define numbers as follows: n_{to_reinit}—number of nodes to be reinitialized for which $key(s) \leq key(s_{start})$; $n_{open,DL}$—number of nodes in the open-list; $n_{ever\_visited}$—number of nodes visited, such that $n_{to\_reinit} \leq n_{ever\_visited} - 1$ and $n_{open,DL} \leq n_{ever\_visited}$. In some cases, all visited nodes must be reinitialized. For example, this could happen when, after traversing a long corridor, right before reaching the goal, an agent encounters a dead end and must take a new path through a corridor that was not initially chosen.

Furthermore, at each n-th step of the agent, the relation $n_{tree} \leq n_{ever\_visited}$ holds. As D* Extra Lite instantly cuts all under-consistent branches, it is likely that $n_{tree} < n_{ever\_visited}$.

The computation times of reinitialization of under-consistent nodes for D* Lite and D* Extra Lite are

$$\begin{aligned} &T_{reinit,DL} \\ &\quad\approx n_{to\_reinit} \cdot (c_{pop;13}(n_{open,DL}) \\ &\qquad + c_{push;31,48}(n_{open,DL}) \\ &\qquad + c_{h;31,48} + c_{preds;28} + b \cdot c_{cost;29} \\ &\qquad + \log b \cdot (c_{succs;30,53} + b \cdot c_{cost;30,53}) \\ &\qquad + \log b \cdot (c_{push|del;31,48|50}(n_{open,DL}) \\ &\qquad + c_{h;31,48})), \end{aligned} \tag{1}$$

$$\begin{aligned} &T_{reinit,DEL} \\ &\quad\approx n_{to\_cut} \cdot (c_{del;54}(n_{open,DEL}) + c_{preds;58} \\ &\qquad + c_{succs;55}) + n_{to\_open} \cdot (c_{push;34}(n_{open,DEL}) \\ &\qquad + c_{h;34}). \end{aligned} \tag{2}$$

As expected, iterations over changed edges have been omitted. Computation times of particular functions are marked as $c_{<\text{function name}>;<\text{code lines}>}$, where calculation times of the open-list operations, such as push, pop and delete, may depend on the number of open elements. b is a domain-specific branching factor (number of neighbors).

Now, let us consider the worst-case scenario, in which all nodes that have ever been visited must be reinitialized. In such a case, D* Lite $n_{to\_reinit} = n_{ever\_visited} - 1$, and D* Extra Lite $n_{to\_cut} = n_{ever\_visited} - 1$. Additionally, for D* Extra Lite, $n_{to\_open} = 1$, i.e., only the root (the goal node) will be reopened. The worst-case computation times are

$$\begin{aligned} &T_{reinit,DL} \\ &\approx n_{ever\_visited} \cdot (\\ &\quad c_{pop;13}(n_{open,DL}) + c_{push;31,48}(n_{open,DL}) \\ &\quad + c_{h;32,48} + c_{preds;28} + b \cdot c_{cost;29} \\ &\quad + \log b \cdot (c_{succs;30,53} + b \cdot c_{cost;30,53}) \\ &\quad + \log b \cdot (c_{push|del;31,48|50}(n_{open,DL}) + c_{h;31,48})), \end{aligned} \tag{3}$$

$$\begin{aligned} &T_{reinit,DEL} \\ &\quad\approx n_{ever\_visited} \cdot (c_{del;54}(n_{open,DEL}) \\ &\qquad + c_{preds;58} + c_{succs;55}). \end{aligned} \tag{4}$$

Although there exists no general relationship between $n_{open,DL}$ and $n_{open,DEL}$, for both algorithms, those numbers may be large; for example, while D* Lite removes only consistent nodes, it may keep many inconsistent nodes from any of previous searches, D* Extra Lite, due to branch cutting, may maintain a rugged frontier. However, for a sufficiently large graph, it can be assumed that $n_{open,DL} \approx n_{open,DEL} \approx n_{open}$. Furthermore, assuming that an open-list is implemented using a binary heap, the time of pushing, popping and deletion operations is the same, namely $c_{heap}(n_{open}) \approx$

$\log n_{open}$. Equations (3) and (4) can be transformed as follows:

$$\begin{aligned} &T_{reinit,DL} \\ &\quad\approx n_{ever\_visited} \cdot ((2+\log b) \\ &\quad\quad \cdot c_{heap}(n_{open}) + (1+\log b)\cdot c_h + c_{preds} \\ &\quad\quad + \log b \cdot c_{succs} + b\cdot(1+\log b)\cdot c_{cost}), \end{aligned} \tag{5}$$

$$\begin{aligned} &T_{reinit,DEL} \\ &\quad\approx n_{ever\_visited} \cdot (c_{heap}(n_{open}) \\ &\quad\quad + c_{preds} + c_{succs}). \end{aligned} \tag{6}$$

From (5) and (6) it is clear that in the worst-case scenario, the following relation holds: $T_{reinit,DEL} < T_{reinit,DL}$. Moreover, as the reinitialization time (1) depends on branching factor b, D* Lite is more domain sensitive than D* Extra Lite.

In contrast, there exists a scenario that D* Lite could solve easily while D* Extra Lite would struggle. Let us assume that there is only one under-consistent node for which $key(s) < key(s_{start})$. In such a case $n_{to\_reinit} = 1$. D* Extra Lite would need to cut an entire branch and reopen all nodes that are neighboring to this branch. For a sufficiently large graph, it is likely that $n_{to\_cut} + n_{to\_open} > n_{to\_reinit}$ thus, from (1) and (2), $T_{reinit,DEL} > T_{reinit,DL}$. Indeed, such a scenario is observable on random maps with low fill-ratio. However, these random maps are artificial, and therefore specific with their salt-and-pepper-like changes. For typical maps from video games as well as better structured maps of rooms, D* Extra Lite remains quicker than both D* Lite and MPGAA*.

A further improvement of D* Extra Lite is possible for undirected graphs. For domains such as presented here path-planning on a grid-map, in which $Succ(s) \equiv Pred(s)$, the CUTBRANCH() function can be simplified. This is demonstrated in Algorithm 4.

Algorithm 4. CUTBRANCH() procedure of the D* Extra Lite algorithm for domains in which $Succ(s) \equiv Pred(s)$.

```
1: function CUTBRANCH($s$)
2:     $visited(s) = false$
3:     $parent(s) = NULL$
4:     REMOVEOPEN($s$)
5:     for all $s' \in Pred(s)$ do
6:         if $visited(s')$ AND $parent(s') = s$ then
7:             CUTBRANCH($s'$)
8:         else
9:             $seeds = seeds \cup s'$
```

In our opinion, in addition to a superior reinitialization-time, D* Extra Lite is easier to implement and more reliable than D* Lite. For example, while to ascertain if a node is a parent of another node, D* Extra Lite relies on topological relations only (i.e., parent(s); lines 47, 56 and 59 in Algorithm 3), D* Lite uses comparison $rhs(s) = cost(s, s') + g(s')$ (lines 29, 42 in Algorithm 2). A comparison in the case of a cost expressed with real numbers, is an error-prone operation for computers. If, due to numerical issues, the admissibility of a heuristic is broken, D* Lite may produce local minima. This would make it impossible to reconstruct the path. MPGAA*, another algorithm implemented and used in our benchmark, is also vulnerable to numerical errors; numerical issues may affect the output of the GOALCONDITION() function, which would lead the algorithm to run unnecessary search steps.

4.4. Example.

In this example, we refer to a grid-world domain. An agent can move in cardinal directions only. The cost of the motion between two unoccupied neighboring cells is one. An occupied cell is treated as a regular cell; however, the cost of entering or leaving such a cell is infinite. The heuristic function uses the Manhattan distance. For example, while to ascertain if a node is a parent of another node, D* Extra Lite relies on topological relations only (i.e., parent(s); lines 47, 56 and 59 in Alg. 3), D* Extra Lite in action is presented in Fig. 4. Each sub-figure depicts the complete state of a search space. Consecutive episodes are organized in rows. Each episode begins with the initialization/reinitialization of the search space, after which searching commences. When a solution is found, an agent follows decreasing g values leading towards the goal. After each step, an observation is performed. Any change observed in the explored search space ends the current episode and starts a new episode from reinitialization.

In Fig. 4, each visited grid cell (i.e., visited node) has been assigned four values, which are the h value in the bottom left, the g value in the top left, the f value in the top right, and the k_m value in the bottom right. The value $f = h + g + k_m$ is the first part of a key value calculated in line 2 in Algorithm 3. Closed nodes have gray-filled inner shape. Nodes with a white-filled inner shape remain on the open-list. An arrow between nodes always points to the parent node.

Episode 1 commences with initialization. In Fig. 4 the goal cell is marked with a cross sign. Initially, the goal node has $h = 3$, $g = 0$, $f = 3$. The k_m value is set to 0. The current state of an agent is depicted with a star. The second sub-figure in the row in Fig. 4 depicts the state of the search space after searching. If the start node is visited and it is on top of the open list or has already been closed in a previous search episode, the searching algorithm succeeds (line 4 in Algorithm 3). After searching, the agent follows decreasing g values, until it notices any change in the environment. In Episode 1, after two steps of the agent, the cell $C3$ changed its state to ‘occupied’. The affected search-tree branch is indicated with a dashed line (the

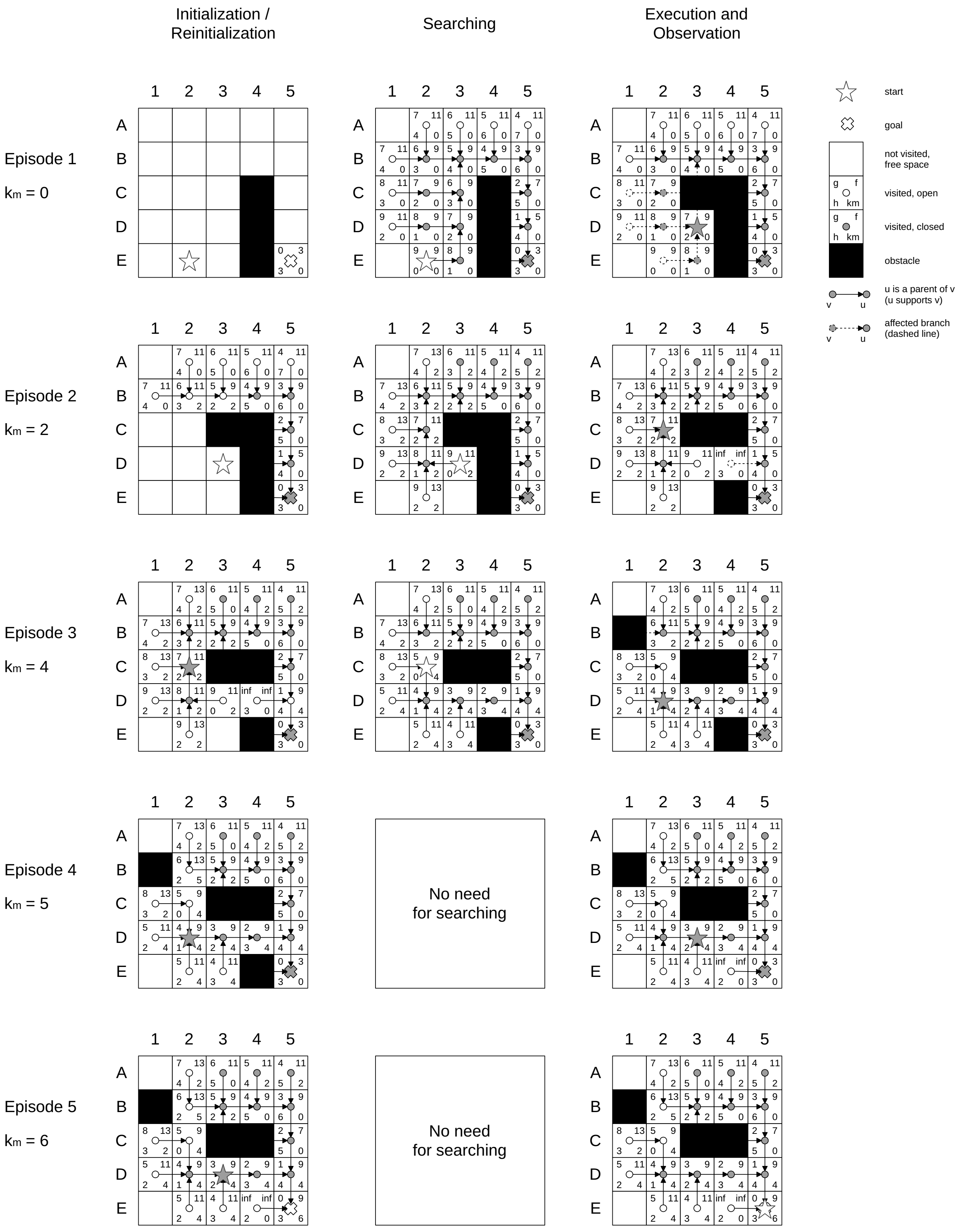

Fig. 4. D* Extra Lite example in action (white inner shape: open nodes, gray inner shape: closed nodes).

third sub-figure of Episode 1 in Fig. 4). Observation of this change ends Episode 1.

Episode 2 commences with reinitialization. Due to a change in the $C3$ cell, the costs of edges $e(s_{C3}, s_{B3})$, $e(s_{C2}, s_{C3})$ and $e(s_{D3}, s_{C3})$ have increased. Consequently, nodes supported by these edges became under-consistent (theirs g values are lower than they should be). In this case, the reinitialization procedure will cut the entire affected branch (lines 46–48 in Algorithm 3). As no particular order is required, cutting may start from any of the $C2$, $D3$ or $C3$ nodes. If the cutting procedure starts from node $C2$, the branches supported by $D3$ or $C3$ will be cut later. During branch cutting, neighboring nodes are pushed to the list of seeds (line 57 in Algorithm 3, or line 9 in Algorithm 4 for the undirected graph). According to the CUTBRANCHES() procedure, $seeds$ contains a number of nodes of which only a select few remain visited. The nodes that are not yet on the open list are reopened (lines 32–34 in Algorithm 3). Following the reinitialization that began Episode 2, nodes $B2$ and $B3$ have been reopened. During the reinitialization, a k_m has been increased by $h(s_{E2}, s_{D3}) = 2$ (line 30 in Algorithm 3). Episode 2 ends after the change observed in cell $D4$.

In the reinitialization at the beginning of Episode 3, the value of k_m is increased by $h(s_{D3}, s_{C2}) = 2$. The node corresponding to cell $D5$ is reopened (lines 45, 32–34 in Algorithm 3). The values of the $D2$ node has been set to $h = 4$, $g = 1$, $k_m = 3$ which results in $f = 8$. As $g(s_{start}) > g(s_{D5}) + cost(s_{D4}, s_{D5}) + h(s_{start}, s_{D4})$, the start node must be reopened (lines 43–44, 49–50 in Algorithm 3).

Episode 3 ends with the observation of cell $B1$ becoming occupied. At the beginning of Episode 4, the affected branch is cut. However, the node corresponding to the agent's state is closed and unaffected; therefore, the success condition is realized (line 4 in Algorithm 3) and no further searching is required.

Following the transition from $D2$ to $D3$, the agent observes that the cost of the $e(s_{E4}, s_{E5})$ edge has decreased, which ends Episode 4. However, there is no need for further searching. This is because $g(s_{start}) = g(s_{E5}) + cost(s_{E4}, s_{E5}) + h(s_{start}, s_{E4})$, and no shorter path can exist (no need to reopen the start node, which is checked in line 43 in Algorithm 3).

5. Experimental results

In the experiments we compared the following three algorithms: D* Lite (optimized version) (Koenig and Likhachev, 2005b), MPGAA* (Hernández *et al.*, 2015) and D* Extra Lite. These experiments were run on an Intel(R) Core(TM) i7-3520M CPU @ 2.90 GHz machine, with 8 GB of RAM, running 64-Bit Linux.

All the three algorithms have been implemented in C++ within the same programming framework[3] and compiled using the gcc (4.8.4) compiler with o3 level of optimization.

This framework, in addition to the algorithms, provides a heap implementation. Heap implementation realizes a lazy node removal and update, i.e., the REMOVEOPEN() function (line 50 in Algorithm 2, and line 54 in Algorithm 3) marks only the node(s) to be removed. An actual node removal takes place when the marked node is on top of the open list. The following domain-specific functions for 2D grid-based path planning are also provided: benchmark maps and problem-loading, $cost$ and $heuristic$ functions (both use the Euclidean distance represented with integer numbers multiplied by a factor of 1000), a neighborhood selection function (an eight-neighbor grid) and the MAPUPDATE() function, which simulates a 360° rangefinder working with a resolution of 1° at a specific observation range. For that reason, simple ray tracing is used. (If a laser beam encounters an obstacle, ray tracing for that beam is stopped.) For each algorithm tested within the framework, the graph representing the entire search-space (equal to the size of the map) is allocated at the beginning.

Every benchmark problem has been solved in accordance with the main function presented in Algorithm 1, that is, the map is updated after each step of an agent. If any change observed in the map affects the shortest path, reinitialization and a consecutive search are performed, though such necessity is checked by each algorithm in a different way. While the main function is running, a number of parameters are logged; these are: search function running time, reinitialization running time, search steps count, open list operations count, predecessor list query count, successor list query count, and traveled path cost. The total running time is simply the sum of the search and reinitialization function running time; thus, it does not include the map-update time.

These algorithms have been tested within the following two settings: planning with the freespace assumption (Setting 1), in which obstacles are only added, and planning on maps with shortcuts and barriers (Setting 2), in which obstacles may appear or disappear.

In the experiments, maps and problems from the benchmark prepared by Sturtevant (2012) have been used. This benchmark provides a number of maps and randomly generated problems for 2D grid-based path planning, of which the following map sets were used: *random_10*, artificially generated maps with a fill-ratio of 10% (Fig. 5(a)); *rooms*, artificially generated maps consisting of square rooms of different size (8–64 pixels)

[3] The source code is available at `https://bitbucket.org/maciej_przybylski/heuristic_search`.

with narrow—of a single pixel size—passages in walls (Fig. 5(b)); *wc3*, maps from the World of Warcraft 3 video game (Fig. 5(c)); *sc*, maps from the Starcraft video game (Fig. 5(d)); *mazes*, artificially generated maps with passages of varying widths (1–32 pixels) (Fig. 5(e)).

As Sturtevant (2012) argues, the *wc3* and *sc* maps are a good approximation of outdoor environments, and while the *rooms* maps simulate indoor environments very well, *random* map problems are typically used for the benchmarking of incremental path planning algorithms (Stentz, 1995; Koenig and Likhachev, 2005b; Hernández *et al.*, 2015). Finally, this *mazes* map set benefits from many difficult problems with dead-ends.

Each data-set features distinct characteristics. In the *mazes* map set, it is possible for even a minor change to cause an extensive modification to the shortest path. Additionally, this *random* dataset is characterized by many small changes that do not significantly affect the path. Such a property of maps is related to the dimension parameter described and calculated for each map set by Sturtevant (2012). (Refer to Table 1, for a dimension value for each map set used in the experiments.) The dimension of a map set describes the increase in the number of nodes at each depth of searching. As explained by Sturtevant (2012), this dimension is an estimate of the branching factor of the search tree generated while searching (not to be confused with the number of applicable actions, which is domain specific).

In each setting, we used maps sized 512×512 (except for the *sc* map set in which few maps are larger) with an observation range of 10 map cells. For each original map, a map with modifications was prepared. According to the suggestions of Sturtevant (2012), our experimental results were ordered by the problem length (plots in Figs. 6 and 7). In Setting 1, in Fig. 6, the x-coordinate indicates the true shortest path calculated by the A* algorithm running on a modified map, which is initially unknown for the agent. In Setting 2, in Fig. 7, the x-coordinate indicates the overhead of the initially known shortest path over the true shortest path, that is, $true\_shortest\_path\_cost - initial\_shortest\_path\_cost$.

The plots in Figs. 6 and 7 were created by grouping problems into buckets. This was according to Sturtevant's (2012) proposition, except that the bucket size may vary between map sets. Additionally, in the background of each plot, a histogram illustrates a coverage by problems. Plot values were calculated for buckets that contained a minimum of seven successfully solved problems. For each bucket, a mean was calculated to be presented in the plot. This value excluded the two most extreme values in that bucket.

5.1. Planning with the freespace assumption.

Planning with the freespace assumption is a scenario in which the first planning episode is performed on an empty map. Obstacles are added to the map only if they are observed within the observation range, which means that action costs can only increase.[4] As there is no need to reestablish values consistency for h, in this case MPGAA* (Hernández *et al.*, 2015) behaves similarly to MPAA* (Hernández *et al.*, 2014). Although this is a preferable situation for MPGAA*, as it requires reinitialization of underestimated nodes, it is the most demanding scenario for D* Lite and D* Extra Lite.

In Table 1, average parameters values logged for planning with the freespace assumption are shown. For each map set, 10,000 randomly selected problems have been solved. In the majority of cases, D* Extra Lite is the quickest (highlighted T_t values in Table 1), only for random maps with 10% fill-ratio does D* Lite outperform the other two algorithms. Given that artificial maps have the highest dimension, generally, they are less cluttered. As discussed in the complexity analysis, this is the most preferable scenario for D* Lite.

Other parameters listed in Table 1 offer further support to results of the complexity analysis. Since D* Lite performs reinitialization while searching, the number of search steps for D* Extra Lite is lower than that for D* Lite, as well as the number of operations on the heap. In turn, due to search-tree branch cutting, D* Extra Lite performs many more iterations over the list predecessors. The number of iterations over the list of successors for the *mazes* map set is also interesting, since it is higher for D* Lite than for D* Extra Lite. These results are consistent with the worst-case time complexity of D* Lite (Eqn. (3)) and D* Extra Lite (Eqn. (4)), such that D* Lite may perform the iteration over successors up to $\log b$ times more often than D* Extra Lite.

Finally, in Table 1, the traveled-path cost is presented. D* Lite and D* Extra Lite are equivalent; however, the traveled path cost varies for MPGAA*. This is due to the fact that, while many paths of the same length exist in the 2D grid domain, different algorithms break ties in different ways. (For example, D* Lite and D* Extra Lite break ties as shown in line 18 of Algorithm 1.) Although all three algorithms are optimal, the consequences of the selected next step are unpredictable, and a chosen path could be a dead end.

In Figs. 6(a)–(f) the total time is shown as a function of problem length. For problems of a short length, the algorithms finished missions in a comparable time, although with an increasing path length, differences become more pronounced. For the *random_10* maps (Fig. 6(a)), D* Lite is noticeably the quickest. However, with an increasing problem complexity, differences between the algorithms increase in favor of D* Extra

[4] A video demonstrating the three algorithms tested in this study in planning with freespace assumption is available at `https://youtu.be/al2L_TJXnoY`.

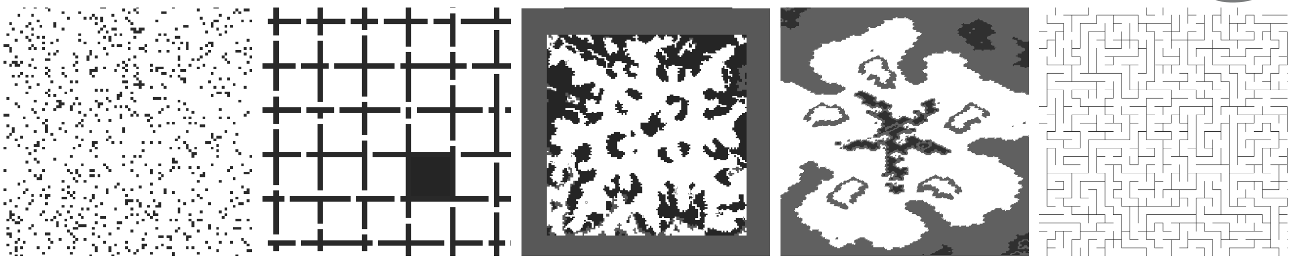

Fig. 5. Sample maps from the Sturtevant (2012) benchmark: portion of a random map (a), portion of a rooms map (2), wc3 (World of Warcraft 3) (c), sc (Starcraft) (d), maze (e).

Table 1. Average experimental results for planning with freespace assumption. Dim.: dimension (Sturtevant, 2012), T_r: reinitialization time [ms], T_s: search time [ms], T_t: total time [ms], R_t: total time ratio, #S.Steps: number of search steps, #Heap: number of heap operations, #Preds: number of iterations over predecessors, #Succs: number of iterations over successors, P. Cost: traveled path cost [map cells].

Map set	Dim.	T_r	T_s	T_t	R_t	#S.Steps	#Heap	#Preds	#Succs	P. Cost	Algorithm
random 10%	1.13	3.53	21.51	25.04	1.00	26151	86745	41032	17773	378.844	D* Extra Lite
		0.57	22.55	**23.11**	0.92	26119	90777	23387	972	378.844	**D* Lite Opt.**
		0.13	27.64	27.77	1.11	32621	92960	0	32621	378.031	MPGAA*
rooms	0.88	6.58	42.66	**49.24**	1.00	81348	189873	72708	34510	891.093	**D* Extra Lite**
		1.51	51.75	53.27	1.08	89149	250085	40511	16464	891.093	D* Lite Opt.
		0.30	176.85	177.15	3.60	277328	834970	0	277328	909.645	MPGAA*
wc3	0.75	2.19	19.93	**22.12**	1.00	33384	75607	28994	10502	461.109	**D* Extra Lite**
		0.66	24.28	24.94	1.13	35867	104626	19348	6241	461.109	D* Lite Opt.
		0.13	69.44	69.57	3.15	100582	296381	0	100582	463.155	MPGAA*
sc	0.41	10.90	87.56	**98.46**	1.00	163964	381328	149447	59992	1621.865	**D* Extra Lite**
		2.60	114.67	117.28	1.19	186328	541962	96924	49940	1621.865	D* Lite Opt.
		0.49	586.66	587.16	5.96	947837	2770506	0	947837	1616.717	MPGAA*
random 40%	0.09	22.04	89.26	**111.31**	1.00	225760	564389	232905	126502	6613.036	**D* Extra Lite**
		10.27	153.53	163.80	1.47	332119	898930	129432	164981	6613.036	D* Lite Opt.
		1.75	416.04	417.79	3.75	902292	2692626	0	902292	6531.051	MPGAA*
mazes	0.02	80.85	365.23	**446.08**	1.00	714783	1843893	826248	441713	18254.245	**D* Extra Lite**
		26.75	839.49	866.24	1.94	1093015	3328033	535517	1041180	18254.245	D* Lite Opt.
		4.99	4013.32	4018.31	9.01	6806971	19593377	0	6806971	18396.122	MPGAA*

Lite, including for random maps. In the case of random maps with a fill-ratio of 40% (Fig. 6(e)), which due to obstacle density are more similar to mazes, D* Extra Lite is 1.47 times faster than D* Lite. MPGAA* seems more case-sensitive than D* Lite and D* Extra Lite; total-time plots for MPGAA* in Figs. 6(b)–(e) are more uneven than are corresponding time plots for the other algorithms.

5.2. Planning on maps with shortcuts and barriers.

A characteristic property of D* Lite, D* Extra Lite and MPGAA* is that they reveal different behaviors when confronted with an action-cost increase and decrease, therefore, we propose a new test. Considering that in the first half of the problems, obstacles were added only (barriers), and in the second half, obstacles were removed only (shortcuts), problems were resolved simply by solving the first half with the freespace assumption, and in the second half, assuming the opposite, such that, although to begin with the agent knows the entire map, as the true map is empty, obstacles can only disappear. Using this approach, for each of three representative map sets (namely *random_10*, *rooms* and *wc3*), 5,000 problems with shortcuts and 5,000 problems with barriers were solved. In Fig. 7, results with a negative path cost overhead correspond to problems with shortcuts, while results with a positive path cost overhead correspond to problems with barriers.

In Figs. 7(a), (d) and (g) represent the total time for the *random_10*, *wc3* and *rooms* map sets, respectively. D* Extra Lite outperforms D* Lite and MPGAA* algorithms for barriers (positive path cost overheads in Figs. 7 (d) and (g)). Moreover, the superiority of D* Extra Lite increases with the difference between true path-length and initial path-length. Next to the total time charts, the search time Figs. 7(b), (e), (h) and the reinitialization time plots for each map set are presented Figs. 7(c), (f), (i). The search

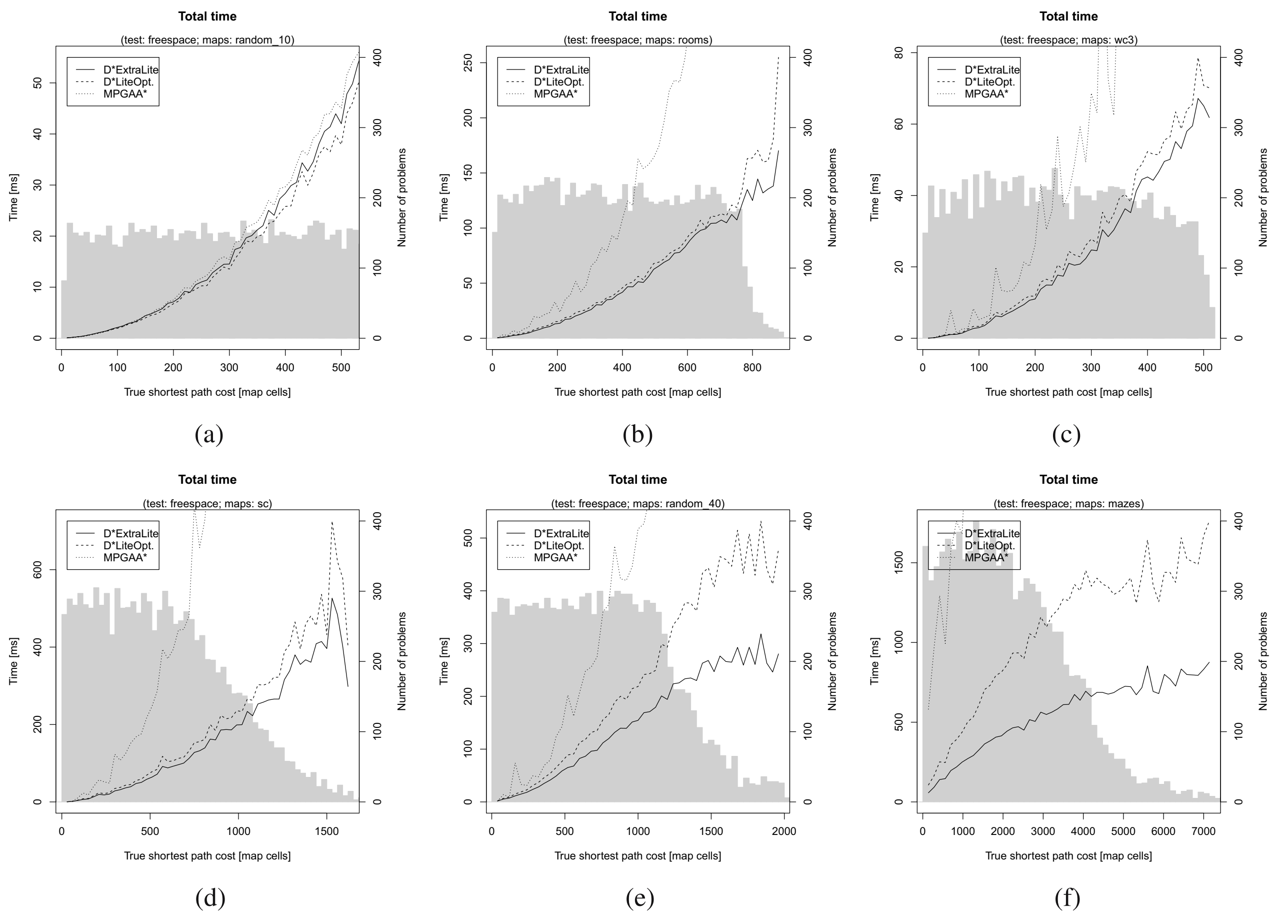

Fig. 6. Total running time for planning with the freespace assumption, for the *random_10* (a), *rooms* (b), *wc3* (c), *sc* (d), *random_40* (e) and *mazes* (f) map sets; in the background the histogram of problems plotted in gray.

time is a dominating component for all three algorithms; however in the case of MPGAA* and D* Extra Lite, the reinitialization time is also meaningful.

A property of the D* Extra Lite algorithm is that the reinitialization time is high only in the case of added obstacles. This is because under-consistent nodes have to be made over-consistent, which is achieved using the CUTBRANCH() procedure (Algorithm 3). In contrast to D* Extra Lite, MPGAA*'s reinitialization time is higher when obstacles are removed. This is owing to its reestablishing procedure for the h value consistency. In the case of shortcuts, D* Lite and D* Extra Lite only reopen affected nodes, and perform regular searches in the same manner. Therefore, the total time for these two algorithms is almost equal.

In Table 2 average parameters for the shortcuts and barriers setting are reported. The average values of particular parameters for barriers are similar to the ones presented in Table 1. Where results illustrate the effect of shortcuts, it can be seen, as expected, that the total time, as well as the other parameters, are similar for D* Lite and D* Extra Lite, and noticeably larger for MPGAA*.

5.3. Summary of experimental results. The experiments were conducted within two settings, in which 512×512 maps sized from six different map sets were used. For each map set 10,000 randomly selected problems were solved. In most experiments, D* Extra Lite performed on the average from 1.08 to 1.94 times faster than D* Lite (optimized version), and from 1.11 up to 9.01 times faster than MPGAA*. Only in tests on random maps with a 10% fill-ratio was D* Lite 1.08 times faster than D* Extra Lite.

The weakness of D* Extra Lite is the number of iterations over predecessors and successors, which for all map sets except *mazes* was higher than for the other two algorithms. This property should be taken into consideration when selecting an algorithm for particular domain. In domains with finite search spaces, such as 2D video game maps, it is possible to initialize the entire search spaces at the beginning. An iteration over a node's neighbors is then a simple operation

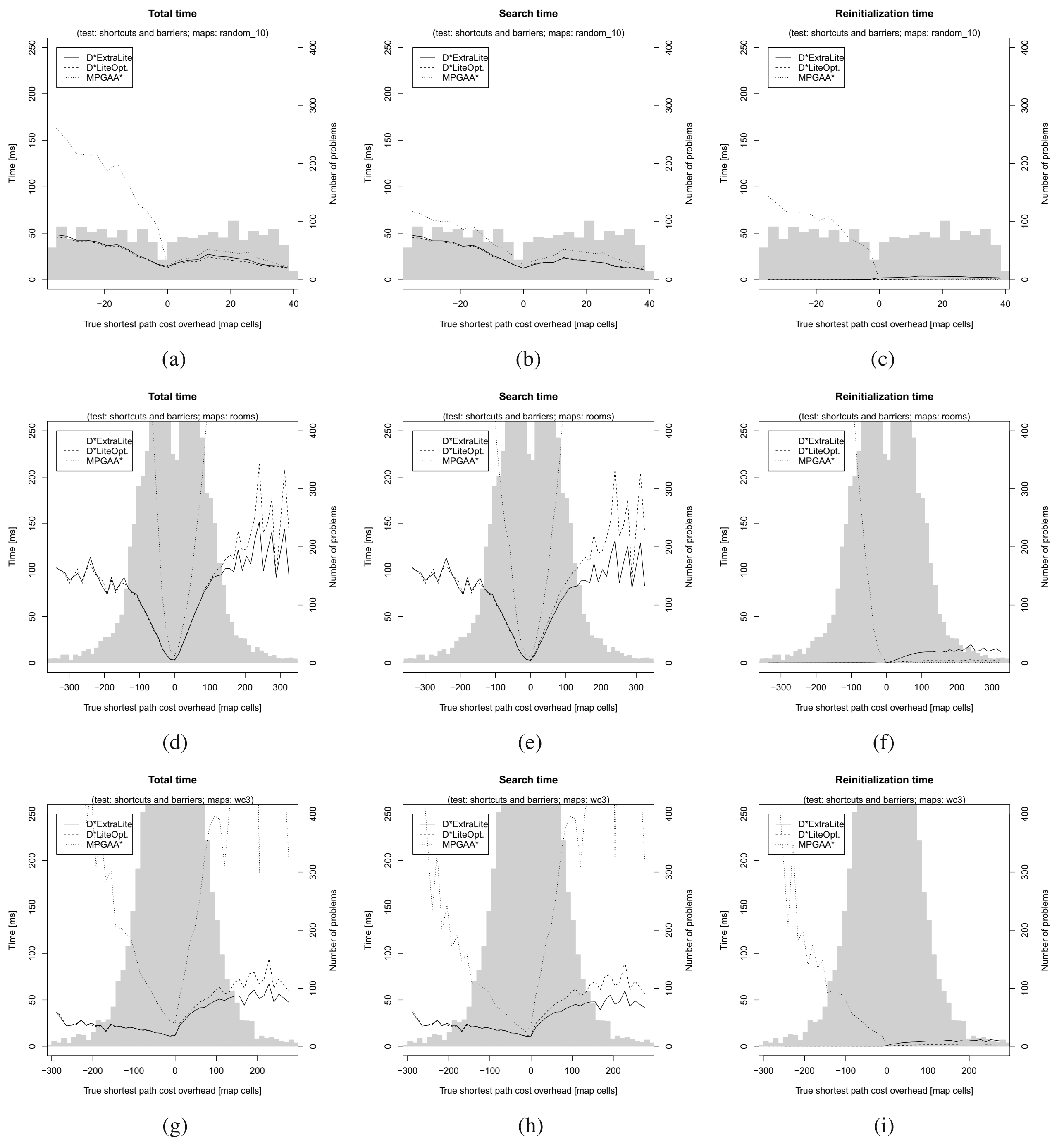

Fig. 7. Running times for planning with shortcuts and barriers, for the *random_10* (a)–(c), *rooms* (d)–(f) and *wc3* (g)–(i) map sets; in the background the histogram of problems plotted in gray.

with pointers. However, as in domains with a large or infinite search-space nodes are expanded bit-by-bit, domain-dependent implementations of $Pred(s)$ and $Succ(s)$ have to be called instead of operations with pointers; therefore, results may differ. Although it should be noted that the common technique of caching; can amortize the running time. Nevertheless, the results of the present study show that in the worst-case scenario D* Extra Lite performs fewer operations than D* Lite. Therefore, irrespective of the implementation, D* Extra Lite is the best choice for difficult, dynamic problems.

The experimental results presented in the paper were gained from tests conducted with an observation range

Table 2. Average experimental results for planning with shortcuts and barriers. T_r: reinitialization time [ms], T_s: search time [ms], T_t: total time [ms], R_t: total time ratio, #S.Steps: number of search steps.

Map set	Shortcuts					Barriers					Algorithm
	T_r	T_s	T_t	R_t	#S.Steps	T_r	T_s	T_t	R_t	#S.Steps	
random 10%	0.64	42.38	43.02	1.00	42337	3.44	20.29	23.73	1.00	23295	D* Extra Lite
	0.46	41.92	**42.38**	0.99	42097	0.76	20.83	**21.59**	0.91	23991	**D* Lite Opt.**
	79.08	63.17	142.25	3.31	219139	0.19	28.68	28.87	1.22	31006	MPGAA*
rooms	0.34	44.28	**44.62**	1.00	46932	7.21	46.70	**53.91**	1.00	79038	**D* Extra Lite**
	0.25	45.23	45.48	1.02	46815	1.63	56.71	58.34	1.08	86192	D* Lite Opt.
	213.40	155.09	368.49	8.26	558437	0.35	164.00	164.35	3.05	250108	MPGAA*
wc3	0.17	22.68	**22.85**	1.00	27166	4.41	36.99	**41.40**	1.00	71476	**D* Extra Lite**
	0.12	23.50	23.62	1.03	27085	1.43	48.71	50.14	1.21	79794	D* Lite Opt.
	70.53	68.81	139.34	6.10	234615	0.30	179.73	180.03	4.35	305136	MPGAA*

Table 3. Average total time [ms] (T_t) and total-time ratios with regard to D* Extra Lite (R_t) across different observation ranges [map cells]; the test conducted with the freespace assumption on 100 problems from the *wc3* map set.

Observation range	D* Extra Lite	D* Lite Opt.		MPGAA*	
	T_t	T_t	R_t	T_t	R_t
10	22.42	24.83	**1.11**	125.41	**5.59**
20	24.82	26.45	**1.07**	129.08	**5.20**
50	23.19	25.48	**1.10**	147.59	**6.36**
100	18.66	20.51	**1.10**	94.34	**5.06**

of 10 map cells. In Table 3 average total running times are presented for the planning with freespace assumption on *wc3* maps with different observation ranges. Across all three algorithms, the running time changed slightly with longer observation ranges. This was the result of more observed changes in the environment, which increased the extent to which the search tree became inconsistent. However, with a longer observation range, more information was gathered. Nevertheless, the observed relationship between the total running time of each of the analyzed algorithms was maintained across the different observation ranges.

6. Conclusions and future work

In this paper, we analyzed several incremental path-planning algorithms, including the recent MPGAA* (Hernández *et al.*, 2015), the popular Focussed D* (Stentz, 1995), and the currently state-of-the-art D* Lite (Koenig and Likhachev, 2005b). We also revisited older ideas, such as Differential A* (Trovato, 1990). In order to gain a deeper insight into properties of incremental heuristic search algorithms, we proposed a new benchmark scenario. This scenario involved planning for both shortcuts and barriers. As for the results of our analysis, we proposed a novel D* Extra Lite algorithm that is both robust and simple. In typical two-dimensional navigation problems D* Extra Lite outperforms both D* Lite (optimized version) and MPGAA*. In addition to comprehensive tests, we have conducted the worst-case complexity analysis which showed that, independently of a particular domain and implementation, D* Extra Lite is faster than D* Lite.

D* Extra Lite is a general purpose, incremental shortest-path algorithm able to work on directed and undirected graphs. It is almost as simple as a regular A* algorithm, only extended with search-tree cutting and frontier-gap repairing. A strong advantage of the D* Extra Lite algorithm over D* Lite is that it performs branch cutting as a simple, recursive operation that makes nodes unvisited without the use of complex operations on the open list. Moreover, from our observations, D* Extra Lite is less vulnerable to numerical errors than MPGAA* and D* Lite.

Additionally, the D* Extra Lite algorithm can be extended easily. We have already extended D* Extra Lite to anytime version (currently unpublished), similarly to that of Anytime D* (Likhachev *et al.*, 2005), and it could also be extended to a truncated version, similar to Anytime Truncated D* (Aine and Likhachev, 2016).

A natural application for D* Extra Lite is navigation of a mobile robot. Currently, we are working on the application of D* Extra Lite for path planning in environments with moving objects using the time-layered search-space architecture presented by Przybylski and Siemiątkowska (2012). In the future we plan to test D* Extra Lite in various domains, especially in hierarchical planning with the use of semantic maps that comprise topological and metrical information (Przybylski *et al.*, 2015) combined with grid-based maps (Belter *et al.*, 2016).

References

Aine, S. and Likhachev, M. (2016). Truncated incremental search, *Artificial Intelligence* **234**: 49–77.

Belter, D., Łabecki, P., Fankhauser, P. and Siegwart, R. (2016). RGB-D terrain perception and dense mapping for legged robots, *International Journal of Applied Mathematics and Computer Science* **26**(1): 81–97, DOI: 10.1515/amcs-2016-0006.

Hart, P.E., Nilsson, N.J. and Raphael, B. (1968). A formal basis for the heuristic determination of minimum cost paths, *IEEE Transactions on Systems Science and Cybernetics* **4**(2): 100–107.

Hernández, C., Asín, R. and Baier, J.A. (2015). Reusing previously found A* paths for fast goal-directed navigation in dynamic terrain, *19th AAAI Conference on Artificial Intelligence, Austin, TX, USA*, pp. 1158–1164.

Hernández, C., Baier, J.A. and Asín, R. (2014). Making A* run faster than D*-Lite for path-planning in partially known terrain, *Proceedings of the 24th International Conference on Automated Planning and Scheduling, Portsmouth, NH, USA*, pp. 504–508.

Hernández, C., Meseguer, P., Sun, X. and Koenig, S. (2009). Path-Adaptive A* for incremental heuristic search in unknown terrain, *Proceedings of the 19th International Conference on Automated Planning and Scheduling, Thessaloniki, Greece*, pp. 358–361.

Hernández, C., Sun, X., Koenig, S. and Meseguer, P. (2011). Tree Adaptive A*, *10th International Conference on Autonomous Agents and Multiagent Systems, Taipei, Taiwan*, Vol. 1, pp. 123–130.

Koenig, S. and Likhachev, M. (2001). Improved fast replanning for robot navigation in unknown terrain, *Technical Report GIT-COGSCI-2002/3*, Georgia Institute of Technology, Atlanta, GA.

Koenig, S. and Likhachev, M. (2005a). Adaptive A*, *Proceedings of the 4th International Joint Conference on Autonomous Agents and Multiagent Systems, Utrecht, The Netherlands*, pp. 1311–1312.

Koenig, S. and Likhachev, M. (2005b). Fast replanning for navigation in unknown terrain, *IEEE Transactions on Robotics* **21**(3): 354–363.

Koenig, S., Likhachev, M. and Furcy, D. (2004). Lifelong planning A*, *Artificial Intelligence* **155**(1): 93–146.

Koenig, S. and Sun, X. (2009). Comparing real-time and incremental heuristic search for real-time situated agents, *Autonomous Agents and Multi-Agent Systems* **18**(3): 313–341.

Likhachev, M., Ferguson, D.I., Gordon, G.J., Stentz, A. and Thrun, S. (2005). Anytime Dynamic A*: An anytime, replanning algorithm, *Proceedings of the 15th International Conference on Automated Planning and Scheduling, Monterey, CA, USA*, pp. 262–271.

Podsędkowski, L. (1998). Path planner for nonholonomic mobile robot with fast replanning procedure, *1998 IEEE International Conference on Robotics and Automation, Lueven, Belgium*, Vol. 4, pp. 3588–3593.

Podsędkowski, L., Nowakowski, J., Idzikowski, M. and Vizvary, I. (2001). A new solution for path planning in partially known or unknown environment for nonholonomic mobile robots, *Robotics and Autonomous Systems* **34**(2): 145–152.

Przybylski, M., Koguciuk, D., Siemiątkowska, B., Harasymowicz-Boggio, B. and Chechliński, Ł. (2015). Integration of qualitative and quantitative spatial data within a semantic map for service robots, *in* R. Szewczyk *et al.* (Eds.), *Progress in Automation, Robotics and Measuring Techniques. Volume 2: Robotics*, Springer, Cham, pp. 223–232.

Przybylski, M. and Siemiątkowska, B. (2012). A new CNN-based method of path planning in dynamic environment, *in* L. Rutkowski *et al.* (Eds.), *Artificial Intelligence and Soft Computing, ICAISC 2012*, Lecture Notes in Computer Science, Vol. 7268, Springer, Berlin/Heidelberg, pp. 484–492.

Stentz, A. (1994). Optimal and efficient path planning for partially-known environments, *Proceedings of the 1994 IEEE International Conference on Robotics and Automation, San Diego, CA, USA*, Vol. 4, pp. 3310–3317.

Stentz, A. (1995). The Focussed D* algorithm for real-time replanning, *Proceedings of the 14th International Joint Conference on Artificial Intelligence, Montreal, Quebec, Canada*, Vol. 2, pp. 1652–1659.

Sturtevant, N.R. (2012). Benchmarks for grid-based pathfinding, *IEEE Transactions on Computational Intelligence and AI in Games* **4**(2): 144–148.

Sun, X. and Koenig, S. (2007). The Fringe-Saving A* search algorithm—a feasibility study, *Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India*, pp. 2391–2397.

Sun, X., Koenig, S. and Yeoh, W. (2008). Generalized Adaptive A*, *Proceedings of the 7th International Joint Conference on Autonomous Agents and Multiagent Systems, Estoril, Portugal*, Vol. 1, pp. 469–476.

Trovato, K.I. (1990). Differential A*: An adaptive search method illustrated with robot path planning for moving obstacles and goals, and an uncertain environment, *International Journal of Pattern Recognition and Artificial Intelligence* **4**(2): 245–268.

Trovato, K.I. and Dorst, L. (2002). Differential A*, *IEEE Transactions on Knowledge and Data Engineering* **14**(6): 1218–1229.

van Toll, W. and Geraerts, R. (2015). Dynamically Pruned A* for re-planning in navigation meshes, *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), Hamburg, Germany*, pp. 2051–2057.

Maciej Przybylski received the MSc degree in robotics from the Warsaw University of Technology (2008). He has been a PhD student and a research assistant at the Institute of Automatic Control and Robotics at the WUT since 2008. He was developing planning algorithms and a software architecture for mobile robots at the robotic startup company Versabox (2014–2016). His interests are in motion and action planning in dynamic environments, the integration of motion and action planning, and software architectures for robots.

Barbara Putz has been working at the Institute of Automatic Control and Robotics (IAiR), Warsaw University of Technology, since 1973. She received her MSc (in industrial automation), PhD and DSc (in machine design and exploitation) degrees from the same university. Currently, she is an associate professor at the IAiR and the head of the Division of Diagnostics and Monitoring of Processes. Her research areas include machine vision, multimodal image fusion, video tracking and geometric modeling.

Received: 27 July 2016
Revised: 16 January 2017
Re-revised: 7 March 2017
Accepted: 25 March 2017