Soft Comput
DOI 10.1007/s00500-015-1750-1

METHODOLOGIES AND APPLICATION

Relaxed Dijkstra and A* with linear complexity for robot path planning problems in large-scale grid environments

Adel Ammar[1] · **Hachemi Bennaceur**[1] · **Imen Châari**[3,4] · **Anis Koubâa**[4,5,6] · **Maram Alajlan**[2,4]

Abstract Although there exist efficient methods to determine an optimal path in a graph, such as Dijkstra and A* algorithms, large instances of the path planning problem need more adequate and efficient techniques to obtain solutions in reasonable time. We propose two new time-linear relaxed versions of Dijkstra (RD) and A* (RA*) algorithms to solve the global path planning problem in large grid environments. The core idea consists in exploiting the grid-map structure to establish an accurate approximation of the optimal path, without visiting any cell more than once. We conducted extensive simulations (1290 runs on 43 maps of various types) for the proposed algorithms, both in four-neighbor and eight-neighbor grid environments, and compared them against original Dijkstra and A* algorithms with different heuristics. We demonstrate that our relaxed versions exhibit a substantial gain in terms of computational time (more than 3 times faster in average), and in most of tested problems an optimal solution (in at least 97 % of cases for RD and 82 % for RA*) or a very close one is reached (at most 9 % of extra length, and less than 2 % in average). Besides, the simulations also show that RA* provides a better trade-off between solution quality and execution time than previous bounded relaxations of A* that exist in the literature.

Communicated by V. Loia.

✉ Imen Châari
imen.chaari@coins-lab.org

Adel Ammar
adel.ammar@ccis.imamu.edu.sa

Hachemi Bennaceur
hachemi@ccis.imamu.edu.sa

Anis Koubâa
akoubaa@coins-lab.org

Maram Alajlan
maram.ajlan@coins-lab.org

1 College of Computer and Information Sciences, Al Imam Mohammad Ibn Saud Islamic University (IMSIU), Riyadh, Kingdom of Saudi Arabia

2 Research Unit of Sciences and Technology, Al-Imam Mohamed bin Saud University, Riyadh, Saudi Arabia

3 University of Manouba, National School of Computer Science, Manouba, Tunisia

4 Cooperative Intelligent Networked Systems (COINS) Research Group, Riyadh, Saudi Arabia

5 College of Computer and Information Sciences, Prince Sultan University, Riyadh, Saudi Arabia

6 CISTER/INESC-TEC, ISEP, Polytechnic Institute of Porto, Porto, Portugal

1 Introduction

Dijkstra's algorithm and A* search (Dijkstra 1959; Hart et al. 1968; Choubey and Gupta 2013) are often used to find the shortest path in various real-life applications such as robot navigation, routing problems and games. A*, with a consistent heuristic, is usually faster than Dijkstra, in practice, for finding the shortest path between two nodes, but Dijkstra algorithm has the advantage of calculating all distances from one node to all other nodes on the graph, in a single run. Nevertheless, these two algorithms may be inefficient in terms of computation time for large-scale grid environments (Potamias et al. 2009; Jigang et al. 2010; Kanoulas et al. 2006; Sven Peyera and Rautenbachb 2009), since the Dijkstra's algorithm has a quadratic time complexity $\mathcal{O}(n^2)$ and processes the whole grid to find the optimal path and A* algorithm may be time consuming to reach the optimal solution for hard instances (such as mazes) depending on the density of obstacles. In order to overcome these drawbacks, we

Published online: 23 July 2015

propose two novel and efficient approaches called Relaxed Dijkstra (RD) and Relaxed A* (RA*) to improve the basic Dijkstra and A* algorithms, in the specific case of regular grid environments. The core idea consists in exploiting the grid-map structure, typically used as the environment model for the global path planning problem in mobile robotics (Tiwari et al. 2012), and we establish an accurate approximation of the cost of the optimal path between two cells. Namely, we provide lower and upper bounds of the cost of the optimal path and we show that this cost could be approximated in terms of the number of moves on that path. Based on that approximation, we designed a linear relaxed variant of the basic Dijkstra algorithm RD and an efficient linear relaxed variant RA* of A* technique.

In this paper, we are particularly interested in the global path planning problem for mobile robot navigation. Indeed, global path planning is a central problem in robotics. It aims to find the shortest obstacle-free path allowing the robot to move between two given locations. In many robotics systems, the grid-map is the common representation of the operational environment of the robot. The grid-map is divided into cells in which the robot has to traverse a set of free cells to reach its goal. In the literature, several heuristic solutions and approaches have been proposed to solve the global path planning problem such as Ant Colony Optimization (ACO) (Chaari et al. 2012), Genetic Algorithms (GA) (Alajlan et al. 2013), Particle Swarm Optimization (PSO) (Shiltagh and Jalal 2013) and Tabu Search (TS) (Masehian and Amin-Naseri 2006). Other techniques are based on exact methods, such as Dijkstra and A*, which have the advantage of being complete and guarantee finding the optimal solution if it exists. This diversity raises the complex challenge of choosing the best algorithm for solving the path planning problem. We are addressing this research question in the iroboapp project (Iroboapp 2015a), aiming at understanding the capabilities and performance of existing approaches for solving the global path planning problem and design new hybrid algorithms for efficient path search in large-scale environments. In previous papers (Alajlan et al. 2013; Chaari et al. 2012), we have investigated heuristic methods including GA, Tabu Search and ACO for solving the global path planning problem. In this paper, we focus on exact methods, namely Dijkstra and A* algorithms, to design new fast and exact algorithms for the global path planning problem in large-scale grid environments.

In grid environments, there are typically two approaches for moves: either the moves of the robot are restricted to horizontal and vertical adjacent cells in the grid (four possible neighbor moves), or additional diagonal moves are allowed (eight possible neighbor moves).

Although the new relaxed approaches might find sub-optimal solutions when diagonal moves are allowed, the simulation results demonstrate their excellent performance in terms of execution time and solution quality (path cost), since in most cases the relaxed approaches find optimal solution in much lower time (three times faster in average) than traditional Dijkstra and A* algorithms. Namely, in case of the restricted neighborhood with four moves, RD finds always the optimal solution and RA* finds it in 99 % of cases. In case of the neighborhood with eight moves RD and RA* algorithms find 97 and 63 %, respectively. Moreover, the gap between the found solution and the optimal solution never exceeds 9 and 7 %, respectively.

The paper is structured as follows. In Sect. 2 we review and discuss some relaxed Dijkstra and A* algorithms. Section 3 presents the RD and RA* algorithms and their theoretical foundation. Section 4 provides an extensive simulation study and performance evaluation results. In Sect. 5, we describe how we integrated our RA* planner in global path planner module of the ROS framework and then we present the experimental study to evaluate the performance of the RA* compared to the ROS default global planner $navfn$ which implements a Dijkstra algorithm. Finally, Sect. 6 concludes the paper and presents future works.

2 Related works

Dijkstra's algorithm (Dijkstra 1959) is the basic technique for finding the shortest path in a directed graph. In general, its time complexity is of the order $\mathcal{O}(n^2)$ where n is the number of vertices of the graph. Various techniques have been proposed for enhancing the original Dijkstra algorithm. An implementation of this algorithm using Fibonacci heap (Fredman and Robert 1984) improved the time complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}((n+m)\log(n))$, where m is the number of edges of the graph. Note that this implementation becomes worse if the graph is dense. In the following, we describe the principle of some proposed techniques enhancing the basic Dijkstra algorithm. These techniques as well as our proposed algorithms aim to exploit the specific structure of the addressed problem to reduce the computational time. The goal is to solve large-scale size problems in reasonable time. In order to deal with large networks, some papers (Peyer et al. 2009; Jigang et al. 2010; Sven Peyera and Rautenbachb 2009) introduced a divide-and-conquer strategy to partition the graph into small subgraphs and materialize the shortest-paths between border nodes in different subgraphs. In Peyer et al. (2009), the authors proposed a generalization of Dijkstra's shortest path algorithm to deal with applications of VLSI routing. The VLSI routing problem aims to find millions of shortest paths in partial grid graphs with billions of vertices. This paper exploits the specific instance structure of the specific problem to decompose the initial graph into subgraphs in such a way the shortest path problems restricted to these subgraphs is computationally easy. The

performances of this algorithm are significantly better when the number of involved subgraphs is small compared to the order of the original graph. Hadlock's Algorithm (Hadlock 1977) explores the search space with linear time and space complexities. The structural properties of the grid map representation of the problem are exploited to find the optimal path. Namely, it takes advantage of the concept of direction present in grid graphs by counting a detour number, which is a penalty expressing the number of grid cells on a path P directed in the opposite direction of the target. This penalty is added to the classical Manhattan distance to obtain the cost of P. The detour number is a kind of penalty that is incremented when the path P moves away from the goal.

Both Hadlock's Algorithm and our RD algorithm are of linear complexity, but our approach offers a different original way exploiting the structural properties of grid environments, and its main advantage, in this regard, is its ability to process both four and eight neighbor grid environments, while Hadlock algorithm can not deal with diagonal moves, since penalties are not simple to calculate in a eight-neighbor environment. Simulation results (IV) show that RD always provides the optimal path in 100 % of cases when dealing with four-neighbor grids (like Hadlocks algorithm), and in more than 97 % of cases when dealing with eight-neighbor grids. Another feature of our approach is that the same idea can be also applied to A* algorithm. A* (Hart et al. 1968) is probably one of the most well-known Artificial Intelligence algorithms. Several bounded relaxations of A* were proposed to enhance its performance. The main idea used to build such variants of A* is the relaxing of the admissibility criterion ensuring an optimal solution path. These relaxations are often bounded by weighting the admissible heuristic to guarantee near-optimal solutions no worse than $(1+\epsilon)$ times the optimal solution path. This heuristic is referred to as ϵ-admissible. For instance, the relaxation Weighted A* (Judea 1984) uses the ϵ-admissible heuristic $hw(n) = \epsilon \times h(n)$, $\epsilon \geq 1$ is shown to be faster than the classical A* using the admissible heuristic h since fewer nodes are expanded. Moreover, in the worst case the gap between the path solution found by that algorithm and the optimal solution is of order $c * (\epsilon - 1)$ where $c*$ is the cost of the optimal solution. The Static Weighting relaxation (Ira 1970) uses the cost function $f(n) = g(n) + (1 + \epsilon) \times h(n)$. The dynamic weighting Pohl (1973) uses the cost function $f(n) = g(n) + (1 + \epsilon \times w(n)) \times h(n)$, where $w(n)$ is a function of the depth of the search. Sampled Dynamic Weighting Andreas and Kaindl (1992) uses sampling of nodes to better estimate and debias the heuristic error. Many other relaxed A* approaches were proposed, most of them relax the admissibility criterion. In Harabor and Grastien (2011), the authors proposed an intelligent exploration of the search space, called jump points, by selectively expanding only certain nodes in a grid map. They defined a strategy to prune a set of nodes between two jump points so that intermediate nodes on a path connecting two jump points are never expanded. They showed that this approach is optimal and can significantly speed up A*.

Also to enhance the performance of A*, different optimizations were proposed in Cazenave (2006). The optimizations are related to the data structure used to implement A*, the heurisitics and the re-expansion of nodes . Indeed, the authors proved that the use of an array of stacks instead of a list to implement the open and the closed lists speeds up the A* algorithm. They also tested different heurisitcs including the Manhattan heuristic, the ALT heuristic and the ALTBestp heuristic and they showed that ALTBestp heuristic is better than the two other heuristics for both A* and IDA* with an array of stacks. To avoid re-expanding nodes already expanded by going through the open and the closed lists, a two-step lazy cache optimization strategy has been proposed. In Antsfeld et al. (2012), the authors present a performance evaluation of TRANSIT routing technique. TRANSIT is a technique for computing shortest path in road networks. They tested it on a set of grid-based video-game benchmarks. They claimed that TRANSIT is strongly and negatively impacted, in terms of running time and memory, by uniform-cost path symmetries. To solve this problem, the authors designed a new technique for breaking symmetries using small additive epsilon-costs to perturb the weights of edges in the search graph. Simulation results showed that the modified TRANSIT technique reduces the execution time as compared to A* and CPD technique. It also reduces the number of expanded nodes. Maxim Likhachev and Thrun (2004) proposed the Anytime Repairing A* (ARA*) algorithm for path planning. ARA* runs a series of A* with weighted heuristics but it reuses information from previous executions. ARA* uses a weighting value (ϵ) (like explained above) to rapidly find an initial suboptimal path and then reduces ϵ to improve path quality over time. ARA* finds an initial solution for a given initial value of ϵ and continues the search with progressively smaller values of ϵ to improve the solution and reduce its suboptimality bound. The value of ϵ is decreased by a fixed amount each time an improved solution is found, and the current-best solution is proven to be ϵ-suboptimal. The $f(s)$-values of the states s $\in$ OPEN list are updated to account for the new value of ϵ. The initial value of ϵ and the amount by which it is decreased in each iteration are parameters of the algorithm. The same team later presented the AD* algorithm (An Anytime, Replanning Algorithm) Likhachev et al. (2005), an algorithm that combines the benefits of anytime and replanning algorithms to solve path planning in dynamic, relatively high-dimensional state spaces. The AD* combines the D* lite (Koenig and Likhachev 2002) and the ARA* algorithm (Maxim Likhachev and Thrun 2004). The core idea of the algorithm consists in generating a set of solutions with improved bound after executing a series of searches while decreasing the inflation factors. When there are changes in

the environment affecting the cost of edges in the graph, locally affected states are placed on the OPEN queue with priorities equal to the minimum of their previous key value and their new key value, as with D* Lite. States on the queue are then processed. To evaluate the performance of AD*, they compared it against ARA* and D* lite on real-world robotic application, and they proved that the novel technique is very efficient. In van den Berg et al. (2005), a new Anytime A* algorithm called Anytime Nonparametric A* (ANA*) is proposed. The algorithm improves on ARA* in five ways: (1) ANA* does not require parameters to be set, (2) ANA* is maximally greedy to find an initial solution, (3) ANA* is maximally greedy to improve the current-best solution, (4) ANA* gradually decreases the suboptimality bound of the current-best solution and (5) ANA* only needs to update the keys of the states in the OPEN queue when an improved solution is found.

The main difference of these algorithms with our variant RA* of A* algorithm is that the relaxing is carried out on the exact cost g of the evaluation function f ($f = g+h$) instead of the heuristic h as usually done in the existing relaxations of A*. The amount g is computed at most only once for every cell during the entire search; it is determined when a node is visited the first time and then $f = g+h$ remains constant during the rest of the search. The advantage of our approaches is the saving of processing time to find the (near)-optimal solution. The efficiency of our approaches is proved theoretically as their time complexities are linear (see Theorem in paragraph 3.4) and it is also proved virtually as the simulation results show substantial gain in terms of computational time with finding in most case optimal solutions.

3 Relaxed Dijkstra and A* approaches

The classical Dijkstra and A* algorithms may be time inefficient with respect to large-scale grid instances (Potamias et al. 2009; Jigang et al. 2010; Kanoulas et al. 2006; Sven Peyera and Rautenbachb 2009), since Dijkstra algorithm has a quadratic time complexity $\mathcal{O}(n^2)$ and process the whole grid to find the optimal path. In addition, A* algorithms requires powerful and consistent heuristics to reach the optimality. By exploiting the grid-map structure representing the environment model of the global path planning problem, we establish an accurate approximation of the cost of the optimal path between two cells. Based on this approximation, we propose a relaxed variant RD of the basic Dijkstra algorithm with a linear time complexity and an efficient relaxed variant RA* of A* technique.

3.1 Approximation of the optimal path

Let G4-grid and G8-grid denote a regular grid where each cell has at most four and eight neighbors, respectively. Precisely, a path in a G4-grid allows only horizontal and vertical moves while a path in a G8-grid allows in addition diagonal moves. In a regular grid map, the cost of a move is defined to be equal to one unit for horizontal and vertical moves. For diagonal moves, the cost is equal to $\sqrt{2}$.

Let P be a path from a start cell to a goal cell, where $|P|$ denotes the number of moves on P and $\text{cost}(P)$ denotes the sum of move costs of P. The following property provides bounds of a given path P in a regular grid.

Property 1 For any path P in a regular grid map we have

$$|P| \leq \text{cost}(P) \leq \sqrt{2}\,|P|$$

Proof Let x be the number of horizontal or vertical moves in P and let y be the number of diagonal moves; then we have

$$|P| = x + y.$$
$$\text{cost}(P) = x + \sqrt{2}\,y$$

We deduce that

$$|P| = x + y \leq \text{cost}(P) = x+\sqrt{2}y \leq \sqrt{2}x+\sqrt{2}y=\sqrt{2}|P|$$

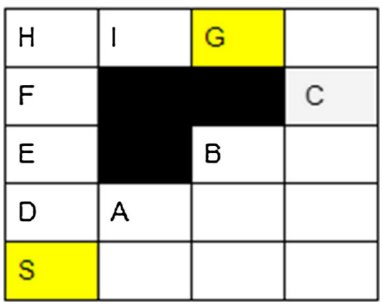
Thus it is clear that the property holds.

In G4-grid we establish the following stronger property:

Property 2 For any path P in a G4-grid, $\text{cost}(P) = |P|$.

Proof Using the same notations as in the proof of Property 1, since in a G4-grid $y = 0$, $\text{cost}(P) = |P| = x$.

Let $P(S, G)$ be the set of feasible (obstacle-free) paths from cell S to cell G. As a direct consequence of property 2, in G4-grid we have that $\text{argmin}_{P\in P(S,G)}|P|$ is the optimal cost path from S to G, While this statement is not necessarily true in G8-grids. For instance, in the example below $\text{argmin}_{P\in P(S,G)}|P| = (S, A, B, C, G)$. For this path, $|(S, A, B, C, G)| = 4$, and $\text{cost}(S, A, B, C, G) = 4\sqrt{2} = 5.6$.



While there exists a path (S, D, E, F, I, G) that has greater number of moves ($|(S, D, E, F, I, G)| = 5$) but a better cost equal to $1 + 1 + 1 + 1 + \sqrt{2} = 5.4$. The gap between the optimal path (S, D, E, F, I, G) and the path (S, A, B, C, G) which has the minimum number of moves is 0.2

(+3.7 %). Thus, our proposed method favours the number of moves as the cost metric over the actual distance (i.e. unit cost in all 8 directions). In the extreme case, where all moves are diagonal, there might be a $\sqrt{(2)} = 1.41$ ratio between the solution found by the proposed methods and the actual shortest path found using the horizontal and vertical edges on the graph. The simulation results will show that the gap is much smaller in practice and that in most cases the path with the smallest number of moves is actually the shortest path even in G8-grids.

3.2 Relaxed Dijkstra algorithm

Based on Property 2, we designed two variants of Dijkstra and A* algorithms, called RD and RA*. Their main advantage is their linear time complexity $\mathcal{O}(n)$ where n is the size of the grid (number of cells). Unlike the classical Dijkstra's algorithm, RD algorithm visits and processes each cell only once. The optimality of the new variant of the Dijkstra's algorithm is guaranteed only in G4-grids (see Property 2). Indeed according to Property 2, the optimal path coincides with the minimum-move path. So, the first time when a cell is processed it is assigned definitively the cost of the optimal path to reach it, whereas, in G8-grids, the algorithm may find a suboptimal solution (see Property 3 below). Nevertheless, the simulation results will show that in most cases, the optimal solution is found. So we conclude that the cost of the minimum-move path is overall a good approximation of the cost of the optimal path.

3.2.1 The algorithm

In what follows, we first recall the pseudocode of the well-known classical Dijkstra algorithm (Algorithm 1). Since our aim is to obtain the optimal path between a start node and a goal node (and not to calculate all distances on the map), we stop the algorithm as soon as the goal node is visited (line 8):

In order to obtain the relaxed Dijkstra version RD, specific to grid environments, we simply skip some instructions of Dijkstra algorithm that are either useless (in G4-grids), or that are time-consuming without significant gain in solution quality (in G8-grids). The main idea is to process each expanded node only once. Skipped instructions are highlighted in Algorithm 1; we remove the visited label (lines 3, 8 and 17), since a node will be processed only once. A node can be recognized as visited or not according to the value of its distance (non visited nodes are those presenting an infinite distance). Besides, to save time and memory, we do not keep track of the previous node at each expanded node (line 4 and 16). Instead, after reaching the goal, and quitting the main loop, the path can be reconstructed, from goal to start by selecting, at each step, the neighbor having the minimum distance. Moreover, we do not search for the minimum distance in the queue (line 9). We can simply take the queues head at each step (the queue will be treated as a FIFO structure). In fact, the propagation of distances in the Dijkstra algorithm (when dealing with a grid environment) makes that a node $n1$ situated at a number of moves $k1$ from the start node cannot be expanded before a node $n2$ situated at a number of moves $k2$ from the start node, if $k2 > k1$. This means that if, at each step, expanded nodes are added at the tail of the queue, it will be always sorted according to the number of moves from the start goal. Hence, instead of searching for the minimum, we simply pop the queues head. This guarantees the obtention of the optimal path in terms of number of moves, which is the shortest path in the case of *G4-grids* (please refer to Property 2), and a near-optimal path in the case of *G8-grids* (as will be assessed in the Sect. 4).

Finally, we do not need to compare the newly calculated distance to the current distance (line 14) since the first calculated distance will be considered as definite (see Property 3).

To replace the removed instructions from Standard Dijkstra algorithm, some distinctive features are added in RD algorithm (Algorithm 2); instead of labelling nodes as visited, nodes having a finite distance value are considered as visited

```
   input  : Grid, Start, Goal
   output : dist
1  for each vertex v in Grid do
2  |  dist(v)= infinity // Mark distance from
   |  start to each node v as not yet
   |  computed;
3  |  visited(v)= false // Mark all nodes as
   |  unvisited;
4  |  previous(v)=undefined // Previous node in
   |  optimal path;
5  end
6  dist(Start)=0;
7  insert Start into Q // Initially, only the
   start node is in the Queue;
8  while (Q is not empty) and (not visited(Goal)) do
9  |  u = not visited vertex in Q with smallest distance;
10 |  remove u from Q;
11 |  visited(u)=true // mark this node as
   |  visited;
12 |  for each neighbor v of u do
13 |  |  g = dist(u) + distEdge(u, v) // distance
   |  |  from Start;
14 |  |  if g ≤ dist(v) then
15 |  |  |  dist(v)=g;
16 |  |  |  previous(v)=u ;
17 |  |  |  if !visited(v) then
18 |  |  |  |  insert v into Q // Add unvisited
   |  |  |  |  node into the Queue to be
   |  |  |  |  processed;
19 |  |  |  end
20 |  |  end
21 |  end
22 end
```

Algorithm 1: Classical Dijkstra

```
   input  : Grid, Start, Goal
   output : dist
1  for each vertex v in Grid do
2  |  dist(v)= infinity // Mark distance from
   |  start to each node v as not yet
   |  computed;
3  end
4  dist(Start)=0;
5  insert Start into Q // Initially, only the
   start node is in the Queue;
6  while (Q is not empty) and (dist(Goal)==infinity) do
7  |  u = head of Q;
8  |  remove u from Q;
9  |  for each neighbor v of u do
10 |  |  if dist(v)==infinity then
11 |  |  |  insert v at the tail of Q // Add
   |  |  |  unvisited node into the Queue
   |  |  |  to be processed;
12 |  |  end
13 |  |  dist(v)=dist(u)+distEdge(u, v);
14 |  end
15 end
```

Algorithm 2: Relaxed Dijkstra

(line 6 and line 10). Besides, as explained above, instead of searching for the minimum distance in the queue, we simply take the queue's head at each step (line 7).

Property 3 The new variant RD of Dijkstra's algorithm is optimal in G4-grids and suboptimal in G8-grids.

Proof The RD algorithm visits and processes every cell only once. When a cell is processed, it is assigned definitively a cost equal to the shortest path to reach it. Based on Property 2, this cost corresponds to the optimal path to reach that cell in G4-grids. Thus, it is not necessary to process again that cell to update its cost as in the classical Dijkstra algorithm. Since the RD algorithm determines $\text{argmin}_{P\in P(S,G)}|P|$ where S and G are the start and goal cells, respectively, then based on Property 1, it is equal to the cost of the optimal path.

In fact, in G4-grids, the distance d of a newly expanded node n will be first calculated after having expanded all the free nodes situated at distances $d' = 1, 2, d-1$. So, nodes that will be expanded after the node n, will have a distance greater or equal than d. Consequently, it is useless to wait for an update of the firstly calculated distance d. In G8-grids, the RD algorithm may discard the optimal path. As every cell is visited and processed only once the algorithm does not consider all the paths reaching that cell. So, if the first path found reaching that cell is not the optimal one then the optimal path is discarded. Thus, in this case, RD algorithm may find a suboptimal path. Indeed in G8-grids, Property 2 does not hold, it means that there is a gap between the cost of $\text{argmin}_{P\in P(S,G)}|P|$ returned by RG algorithm and the cost of the optimal path. According to Property 1, in the worst case, this gap never exceeds $(\sqrt{2}-1)\times|P|$. Moreover, simulations will show that it saves an important amount of time without significantly degrading the solution quality.

3.3 Relaxed A* algorithm

A* algorithm expands the nodes of the tree-search according to the evaluation function $f(n) = g(n)+h(n)$ where $g(n)$ is the exact cost of the path from the root (start node) to the node n of the tree-search and $h(n)$ is a heuristic function estimating the remaining path to the goal. Hence, $f(n)$ is an estimation of the optimal path from the root to a goal traversing the node n. It is proven that A* finds the optimal solution when the heuristic h is consistent (Russell and Norvig 2009). Notice that the exact cost $g(n)$ of a node n may be computed many times, namely it is computed for each path reaching node n from the root. Based on Property 2, the entity $g(n)$ may be approximated by the cost of the minimum-move path from the start cell to the cell associated with node n. The idea of the RA* algorithm is to approximate the exact costs of all paths leading to node n from the root of the tree-search by the exact cost of the first path reaching n. Thus, unlike standard A* algorithm, $g(n)$ is computed only once since the optimal path reaching node n from the start cell S is approximated by the value of $\text{argmin}_{P\in P(S,n)}|P|$. Since both terms $g(n)$ and $h(n)$ of the evaluation function of the RA* algorithm are not exact, there is no guaranty to find an optimal solution, but we believe that the performance of A* in terms of computational time will be improved. The simulation results show likewise that in most cases (at least 63 %) the optimal solution is found; thus this confirms the above conclusion stating the accuracy and the validity of the approximation of the optimal path.

3.3.1 The algorithm

Algorithm 3 presents the standard A* pseudocode. In order to obtain the Relaxed version RA*, following the same idea of the RD algorithm presented above, we simply skip some instructions (highlighted in the algorithm above) that are time-consuming with relatively low gain in terms of solution quality.

The main idea in the relaxed version is that a node is never processed more than once. Consequently, we no longer need a closed set (lines 1, 13 and 15). Besides, as explained for the RD algorithm, to save time and memory, we do not use a $came\_from$ map (lines 3 and 20). Instead, after reaching the goal, the path can be reconstructed, from goal to start by selecting, at each step, the neighbor having the minimum g_score value. Moreover, we do not need to compare a tentative g_score to the current g_score (line 19) since the first calculated g_score is considered definite. This is an approximation that can be correct or not. Simulations will later assess the quality of this approximation. Finally, we do not need to

```
   input  : Grid, Start, Goal
   // Initialisation:
1  closedSet = empty set // Set of already evaluated
   nodes;
2  openSet = Start // Set of nodes to be
   evaluated;
3  came_from = the empty map // map of
   navigated nodes;
4  tBreak = 1+1/(length(Grid)+width(Grid));
   // coefficient for breaking ties;
5  g_score[Start] = 0; // Cost from Start along
   best known path;
   // Estimated total cost from Start to
   Goal:
6  f_score[Start] = heuristic_cost(Start, Goal);
7  while openSet is not empty do
8      current = the node in openSet having the lowest
       f_score;
9      if current = Goal then
10         return reconstruct_path(came_from, Goal);
11     end
12     remove current from openSet;
13     add current to closedSet;
14     for each free neighbor v of current do
15         if v in closedSet then
16             continue;
17         end
18         tentative_g_score = g_score[current] +
           dist_edge(current, v);
19         if v not in openSet or tentative_g_score <
           g_score[v] then
20             came_from[v] = current;
21             g_score[v] = tentative_g_score;
22             f_score[v] = g_score[v] + tBreak *
               heuristic_cost(v, Goal);
23             if neighbor not in openSet then
24                 add neighbor to openSet;
25             end
26         end
27     end
28 end
29 return failure;
```

Algorithm 3: Standard A*

check whether the node is in the open list (line 23). In fact, if its g_score value is infinite, it means that it has not been processed yet and hence is not in the open list.

The tie-breaker factor (slightly >1) which multiplies the heuristic value (line 4) is used so that the algorithm favours a certain direction in case of ties. If we do not use tie-breaking (tBreak = 1), the algorithm would explore all equally likely paths at the same time, which can be very costly, especially when dealing with a grid environment, as will be confirmed by the simulations. Figure 1 shows a simple example of several equivalent optimal paths between two nodes in a G4-grid. Actually, in this example, any obstacle-free path containing three moves up and ten moves right, in whatever order, is an optimal path. A similar example can be shown in a G8-grid. If we do not use a tie-breaking factor, A* would explore

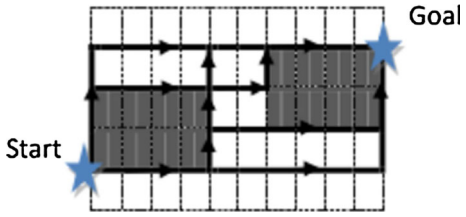


Fig. 1 Example of several equivalent optimal paths between two nodes in a G4-grid. Obstacles are in *grey*

all equivalent paths at the same time. In fact, at a given iteration, several nodes belonging to different paths would have the same f_score, which means that, in the following iterations, each time one of these nodes would be chosen randomly with no preference given to any of these paths. While with the tie-breaker factor, when two nodes have the same value of the sum ($g\_score + h\_score$), the node having a minimum h_score (presumably closest to the goal) would be expanded first. Thus, the algorithm will follow only one of the paths until it either hits the goal, or finds an obstacle. We chose tBreak $= 1 + \epsilon$, with ϵ a small positive equal to 1/(mapWidth + mapLength), both for G4 and G8-grids. This assures that the new heuristic function $(1 + \epsilon) * h$ remains consistent. In fact, whether h is the Manhattan distance or the shortcut distance (defined in Sect. 4.2) it is bounded by (mapWidth + mapLength), which means that ϵ is always smaller than $1/h$. Consequently, $(1+\epsilon)*h \leq h+1$, which means that the solution cost is guaranteed to be strictly less than optimal cost + 1 (see Sect. 2). The tie-breaking technique has already been used in variants of A* algorithm. The purpose here is to measure its influence in a regular grid environment for both A* and RA*.

In order to assess the influence of tie-breaking on both standard and relaxed algorithms, we tested the two versions for each algorithm (with or without tie-breaking). In the following, we will use these notations:

- A^*_{wot}: standard A* algorithm without tie-breaking (tBreak = 1).
- RA^*_{wot}: Relaxed A* algorithm without tie-breaking.
- A^*_t: standard A* algorithm with tie-breaking (tBreak = 1 + 1/(length(Grid) + width(Grid))).
- RA^*_t: Relaxed A* algorithm with tie-breaking.
- $A^*_{\epsilon=x}$: bounded relaxation of A*, as those used in Judea (1984) and Ira (1970), and are also behind the basic idea of Maxim Likhachev and Thrun (2004), Likhachev et al. (2005) and van den Berg et al. (2005) (see Sect. 2), where the heuristic function h is multiplied by a fixed factor x. This is implemented similarly to the tie-breaking versions, except that the value of $(x - 1)$ is not negligible, so that the new heuristic function $x \times h$ is no longer consistent. This speeds up A*, but the algorithm is no longer guaranteed to provide the optimal solution.

```
   input  : Grid, Start, Goal
1  tBreak = 1+1/(length(Grid)+width(Grid));
   // Initialisation:
2  openSet = Start // Set of nodes to be
   evaluated;
3  for each vertex v in Grid do
4  |  g_score(v)= infinity;
5  end
6  g_score[Start] = 0;
   // Estimated total cost from Start to
   Goal:
7  f_score[Start] = heuristic_cost(Start, Goal);
8  while openSet is not empty and g_score[Goal]==
   infinity do
9  |  current = the node in openSet having the lowest
   |  f_score;
10 |  remove current from openSet;
11 |  for each free neighbor v of current do
12 |  |  if g_score(v) == infinity then
13 |  |  |  g_score[v] = g_score[current] +
   |  |  |  dist_edge(current, v);
14 |  |  |  f_score[v] = g_score[v] + tBreak *
   |  |  |  heuristic_cost(v, Goal);
15 |  |  |  add neighbor to openSet;
16 |  |  end
17 |  end
18 end
19 if g_score(goal) ! = infinity then
20 |  return reconstruct_path(g_score) // path will
   |  be reconstructed based on g_score
   |  values;
21 else
22 |  return failure;
23 end
```

Algorithm 4: Relaxed A*

RA* pseudocode is presented in Algorithm 4. Highlighted instructions are specific to the relaxed version. Instead of having a closed set, nodes having a finite g_score value are considered as processed and closed (lines 3–5, 8, 12, 19). And after reaching the goal, and quitting the main loop, the path is reconstructed (line 20), from goal to start, based on g_score values (at each step, we add to the head of the path the neighbor node presenting the minimum g_score value).

3.4 Complexity analysis

RD and RA* algorithms explore the search space of the searching problem with a linear time bounded by the number of cells of the grid. The following theorem demonstrates this statement:

Theorem *The time complexities of RD and RA* are linear* $\mathcal{O}(n)$ *where n is the number of cells in the grid.*

Proof Let c be any cell of the grid and P be a path from the start cell S to c, $|P|$ denotes the number of moves on P and $\text{cost}(P)$ denotes the sum of move costs of P. RD assigns a cost g equal to $\text{cost}(\text{argmin}_{P\in P(S,c)}|P|)$. Since for every path P' from the start cell S to c we have $g \leq |P'|$ and by Property 1, we have $g \leq |P'| \leq \text{cost}(P')$. This means that RD algorithm guarantees assigning a minimum cost to every cell when it is processed the first time. So, the cost assigned to every cell is constant and every cell is processed at most only once during the entire search. Thus the number of processed cells till reaching the goal cell is n in the worst case. In the same manner, RA* assigns a cost f equal to $g + h$. Since $g + h = \text{cost}(\text{argmin}_{P\in P(S,c)}|P|) + h$ then for every path P' from the start cell S to c we have $g + h \leq |P'| + h \leq \text{cost}(P') + h \leq g' + h$. This means that all paths reaching the cell c from S will have a greater cost so the cost of cell c will be constant equal to $g + h$ during the entire search. Thus each cell will be expanded only once and so the number of expanded cells till reaching the goal cell is n in the worst case.

4 Simulation results

The simulations were conducted using Matlab R2012a, on a Laptop equipped with an Intel core i7 processor (2.4 Ghz) and a 16-GB RAM. The benchmark used for testing the algorithms consists of four categories of maps:

1. Maps with randomly generated rectangular-shaped obstacles: this category contains nine (100×100) maps, ten (500×500) maps, six (1000×1000) maps and one (2000×2000) map (since the processing of such large maps is very time-consuming), with different obstacle ratios (from 0.1 to 0.3) and obstacle sizes (ranging from 2 to 50 grid cells).
2. Mazes: All maps in this set are of size 512×512. We used six maps that differ by the corridor size in the passages (1, 2, 4, 8, 16, or 32).
3. Rooms: The room maps were initially uniformly filled with square rooms of fixed size. Then, with a probability of 0.8, a random cell between two adjacent rooms was un-blocked. We used four maps in this category, with different room sizes (8×8, 16×16, 32×32 and 64×64). All maps in this set are of size 512×512.
4. Video games and real maps: six maps extracted from the set of video-game grid maps were made available by BioWare company. We tried to select maps of various difficulty (based on the average number of expanded nodes by A* algorithm). There are four maps of size 512×512, one map of size 512×768 and one map of size 1024×1024. We added to this category one real map: the Willow Garage map (Fig. 24), of size 526×584, commonly used for simulations in ROS robotic framework.

We have designed and generated the random maps of the first category, while the three remaining categories (mazes,

Table 1 Percentage of optimal paths (when a path exists) for the different algorithms, per environment size, in G4-grids, for the six tested algorithms

Algorithm	100 × 100 (%)	500 × 500 (%)	1000 × 1000 (%)	2000 × 2000 (%)	Mazes (512 × 512) (%)	Rooms (512 × 512) (%)	VideoGames (512 × 512 to 1024 × 1024) (%)	All (%)
Dijkstra	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100
RD	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100
A^*_{wot}	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100
RA^*_{wot}	98.1	99.3	100.0	100.0	100.0	98.3	97.5	98.9
A^*_t	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100
RA^*_t	88.7	91.3	95.0	96.7	46.7	68.3	72.9	80.1
$A^*_{\epsilon=1.05}$	97.0	69.2	50.6	46.7	92.8	80.0	71.9	76.6
$A^*_{\epsilon=1.1}$	86.8	58.5	36.1	36.7	67.8	49.2	53.8	60.4

Table 2 Average cost for the different algorithms, per environment size, for a G4-grids

Algorithm	100 × 100	500 × 500	1000 × 1000	2000 × 2000	Mazes (512 × 512)	Rooms (512 × 512)	VideoGames (512 × 512 to 1024 × 1024)	All
Dijkstra	72.68	340.45	750.20	1261.67	1750.71	380.56	455.07	585.46
RD	72.68	340.45	750.20	1261.67	1750.71	380.56	455.07	585.46
A^*_{wot}	72.68	340.45	750.20	1261.67	1750.71	380.56	455.07	585.46
RA^*_{wot}	72.72	340.46	750.20	1261.67	1750.71	380.61	455.12	585.48
A^*_t	72.68	340.45	750.20	1261.67	1750.71	380.56	455.07	585.46
RA^*_t	73.03	340.84	750.38	1261.80	1758.80	381.96	458.89	587.53
$A^*_{\epsilon=1.05}$	72.74	342.03	754.63	1268.00	1750.86	381.29	456.28	586.90
$A^*_{\epsilon=1.1}$	73.02	344.76	761.29	1277.20	1751.73	383.34	458.36	589.40

rooms, and video games) were considered from the large set of benchmarking maps available at Sturtevant (2012) and are available at Maps:benchmark (2015).

For each map, we conducted 30 runs with different positions of start and goal nodes, randomly fixed at each run. For each run, we executed the six algorithms described in Sect. 3 (Dijkstra, RD, A^*_{wot}, RA^*_{wot}, A^*_t and RA^*_t). In order to compare our algorithms to existing work on relaxed A* algorithms, we also tested two versions of bounded relaxed algorithms ($A^*_{\epsilon=1.05}$ and $A^*_{\epsilon=1.1}$, for G4-grids, and $A^*_{\epsilon=1.1}$ and $A^*_{\epsilon=1.2}$, for G8-grids). All these algorithms were tested on the same configuration (same map and same start and goal positions). This makes 1290 (43 maps × 30) total runs for each algorithm both for G4-grids and for G8-grids. In G4-grids, random configurations of start and goal positions yield 17 situations (over 1290) with no connected path between start and goal cells. As for G8-grids, the goal is unreachable in only 11 configurations. Obviously, these configurations were not included in the statistics presented below. We will show the results of the tested algorithms on each one of the four categories of maps. We will also detail the results of the four subcategories of the first category of random maps (100 × 100, 500 × 500, 1000 × 1000 and 2000 × 2000), given that the map size is an important parameter affecting the performance of the algorithms.

4.1 Results in G4-grids

4.1.1 RD algorithm

As has been proven theoretically, the simulation shows that, when dealing with G4-grids, RD algorithm gives always the same path cost as Standard Dijkstra, which is the optimal path cost (see Tables 1, 2).

Figure 2 shows a scatterplot of the execution time (for finding a path) versus the optimal path cost, for each map size of the category of random maps, for both Dijkstra and RD. Figure 3 shows similar results for the three other categories of maps (mazes, rooms and video games). While the path cost is the same between the two algorithms, we can observe on Figs. 2, 3 and 4, that RD algorithm is always markedly faster than Standard Dijkstra. The minimum execution time ratio between the two algorithms is 1.4, the maximum is 3.6 and the average ratio is 2.6. So, we can conclude that it is always advantageous to use RD algorithm instead of standard Dijkstra when we deal with G4-grids.

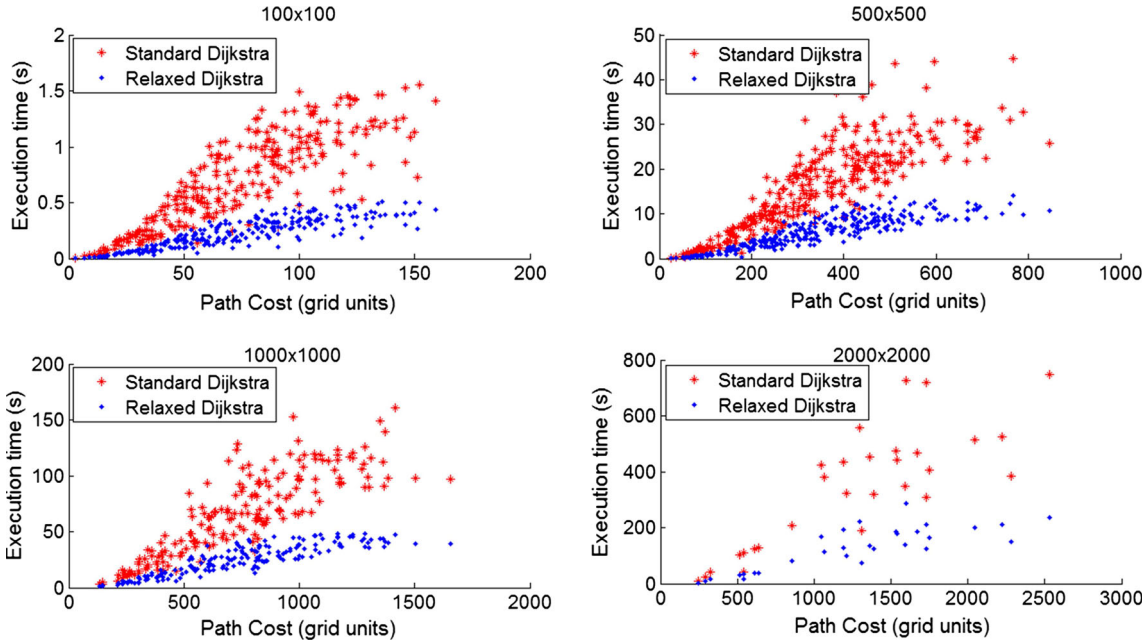
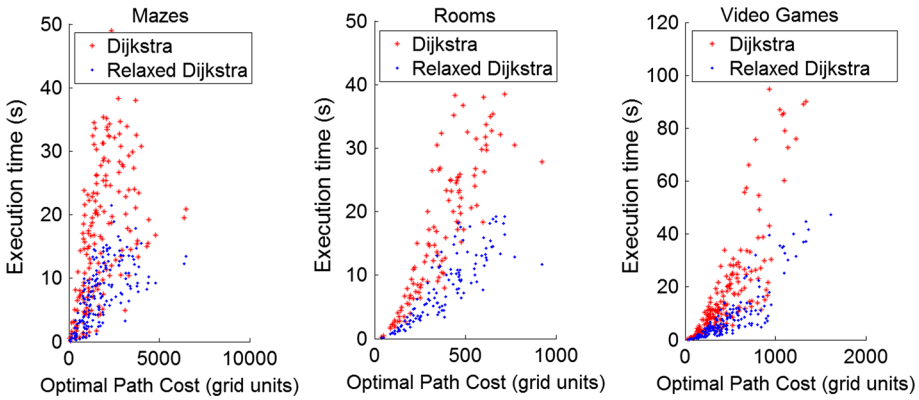


Fig. 2 Comparison of execution time between Dijkstra and RD algorithms, for different environment sizes, randomly generated, in G4-girds

Fig. 3 Comparison of execution time between Dijkstra and RD algorithms, for different structured environments, in G4-girds



*4.1.2 Relaxed A**

The heuristic used for A* is the manhattan distance. It can be easily demonstrated that, in a four-neighbour connectivity, this heuristic is consistent. Figures 5 and 6 compare the execution time between A^*_{wot} and RA^*_{wot}, in the different categories of maps. The horizontal axis represents the cost of the optimal path so that a same configuration (map, start and goal positions) for the two algorithms corresponds to a same value on this axis. We observe that the evolution of the execution time of RA^*_{wot} seems to be linear with regards to the path cost, while the evolution of A^*_{wot} is seemingly quadratic. Notice that this is not an exact relationship, since the execution time depends not only on the path length but also on the presence of obstacles between start and goal positions.

There are six configurations over 1273 (0.5 %) in which RA^*_{wot} is slightly slower than A^*_{wot}. All these six configurations correspond to very short paths between start and goal cells (optimal path cost between 3 and 16 units), for which the execution time is less than 10 ms. In fact, for very

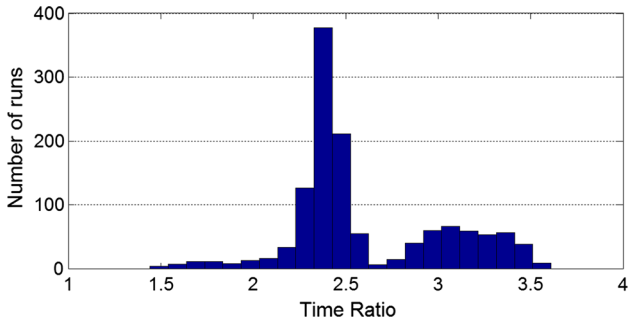


Fig. 4 Histogram of the execution time ratio between Dijkstra and RD algorithms, in G4-grids

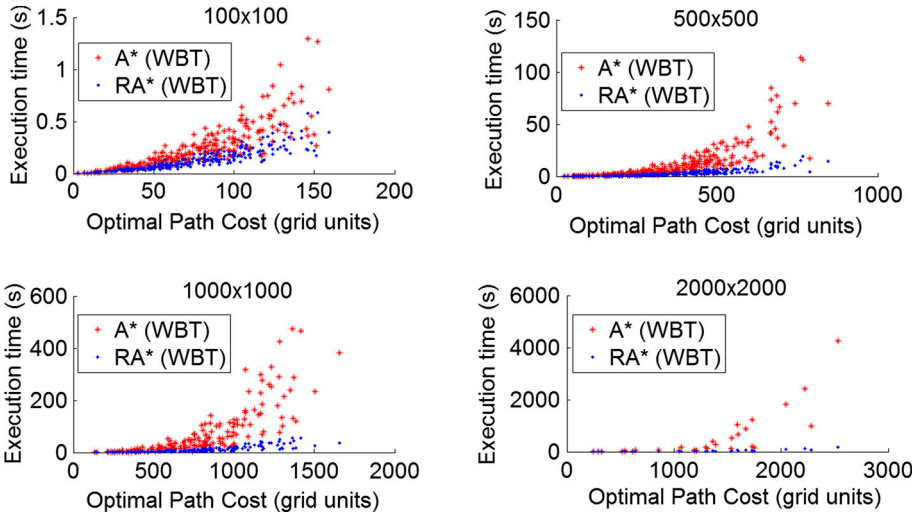


Fig. 5 Comparison of execution time between A^*_{wot} and RA^*_{wot} algorithms, for different environment sizes, randomly generated, in G4-grids

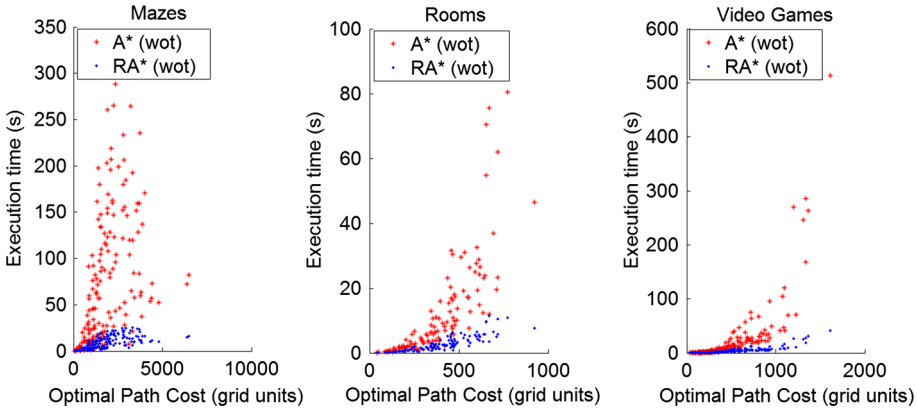


Fig. 6 Comparison of execution time between A^*_{wot} and RA^*_{wot}, for different structured environments, in G4-grids

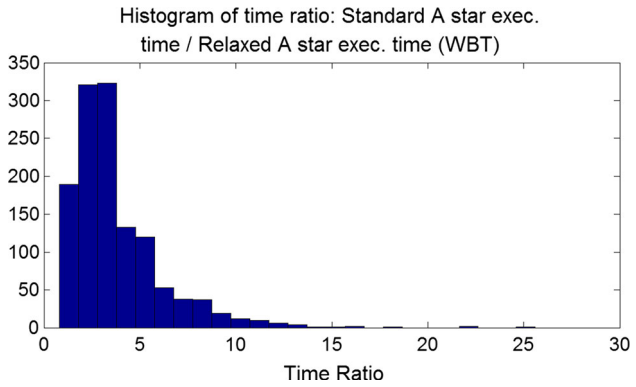


Fig. 7 Histogram of the execution time ratio between A^*_{wot} and RA^*_{wot} algorithms, in G4-grids

short paths, RA* algorithm can present an expensive overhead due to the fact that the entire f value array must be initialised to infinity before every run. The minimum time ratio ($Standard\_exec\_time/Relaxed\_exec\_time$) is 0.82. The maximum time ratio is 25.6, and the average ratio is 3.81. The histogram of time ratios is depicted in Fig. 7. On another hand, as can be seen in Table 3, introducing a tie-break factor speeds up more the standard version (average execution time divided by 2.2), than the relaxed version (average execution time divided by 1.5), which makes the difference between the two algorithms less important (Fig. 8). This is because, with the use of tie-breaking A* explores less concurrent paths, which makes that the g_score value for a given node is updated less frequently, whereas, for RA*, with or without break-ties, the g_score value is calculated only once. We notice also in Table 3 that the effect of tie-breaking is much more important in the category of random maps than in the categories of structured maps. Tie-breaking has even (a slight) negative effect in maze and room maps. This can be explained by the fact that in these types of maps, we rarely find equivalent paths, which makes tie-breaking useless and does not compensate the (slight) calculation overhead of multiplying the heuristic function by $1 + \epsilon$. On the contrary, in random maps, the tie-breaking factor sharply reduces the domain of explored nodes due to the presence of much more free space which implies the presence of many equivalent paths (Fig. 9). This is clearly shown in Fig. 10 which compares the g_score map yielded by A^*_{wot} and A^*_t, on a sample 1000×1000 random map.

Table 3 Average execution time (in seconds) for the different algorithms, per environment size in G4-grids

Algorithm	100 × 100	500 × 500	1000 × 1000	2000 × 2000	Mazes (512 × 512)	Rooms (512 × 512)	VideoGames (512 × 512 to 1024 × 1024) %	All %
Dijkstra	0.69	15.91	66.67	350.88	16.21	18.77	19.10	28.62
RD	0.23	5.89	25.25	126.36	7.46	8.00	7.99	11.03
A^*_{wot}	0.23	10.24	67.06	529.04	72.49	12.09	21.28	39.12
RA^*_{wot}	0.12	2.58	10.66	31.77	9.03	2.64	3.45	4.95
A^*_t	0.11	1.93	14.29	32.96	73.22	11.49	18.77	17.64
RA^*_t	0.07	0.68	5.30	10.50	9.05	2.76	3.34	3.23
$A^*_{\epsilon=1.05}$	0.11	1.60	9.75	27.54	72.00	9.07	16.17	15.99
$A^*_{\epsilon=1.1}$	0.10	0.97	9.79	23.05	70.79	7.60	13.89	15.07

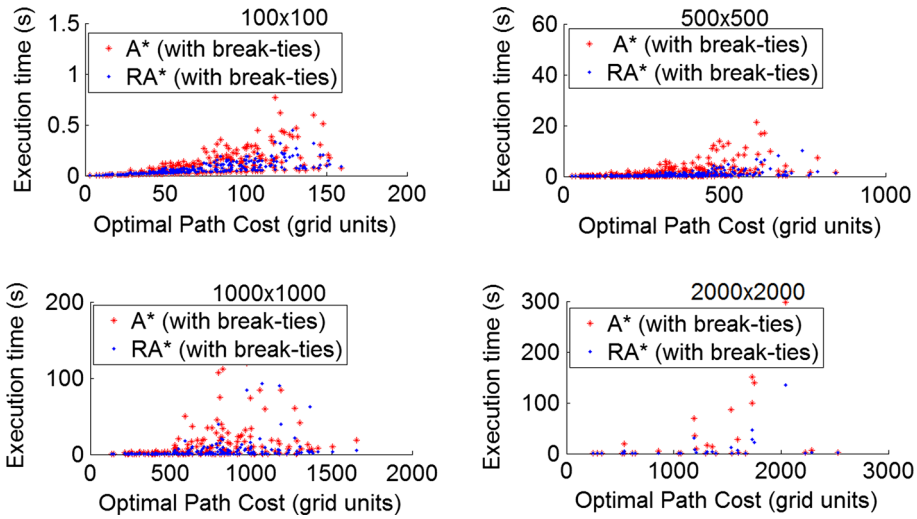


Fig. 8 Comparison of execution time between A^*_t and RA^*_t algorithms, for different environment sizes, randomly generated, in G4-grids

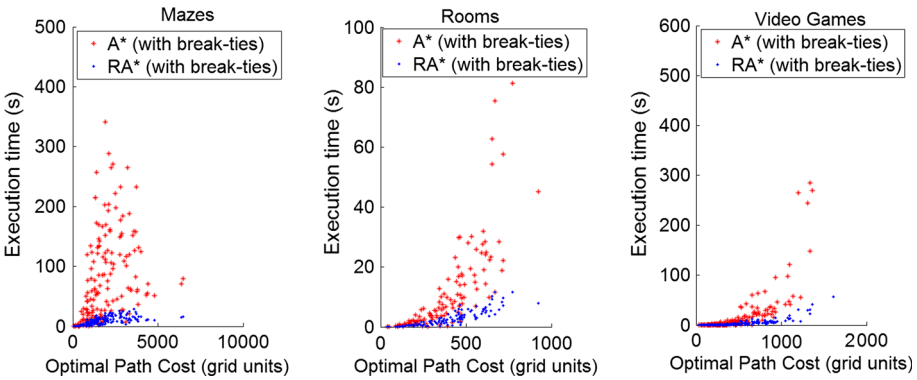


Fig. 9 Comparison of execution time between A^*_t and RA^*_t algorithms, for different structured environments, in G4-grids

The effect of tie-breaking can also be noticed in Table 4. This table contains the average ratio between the number of nodes explored by each algorithm and the actual number of nodes in the optimal path. An explored node is counted only once, even if the algorithm visits it several times (following different paths). An explored node is counted only once, even if the algorithm visits it several times (following different paths). This ratio would be equal to 1 for a perfect algorithm that does not explore any node outside the optimal path. It can actually be very close to 1, in certain cases, for A^*_t algorithm when there are no obstacles between start and goal cells. But we notice in Table 4 that all the tested algorithms explore much more nodes than those contained in the path. Dijkstra algorithm visits the largest number of nodes because its exploration is made in all directions, contrary to A* which explores first the nodes that are in the direction

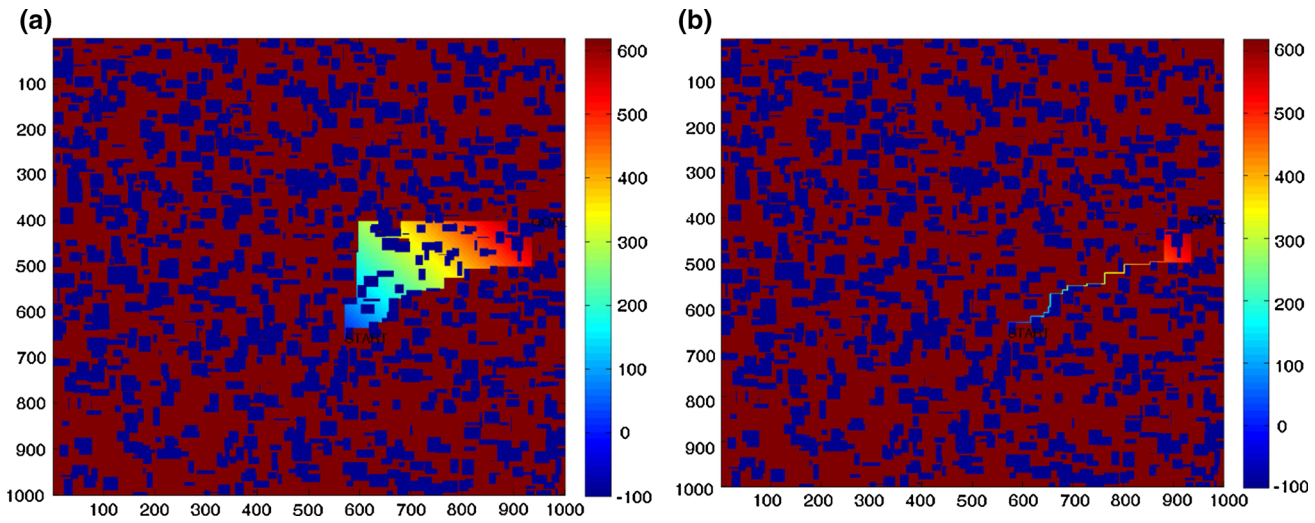


Fig. 10 g_score values in a 1000 × 1000 map. **a** When we apply A^*_{wot} algorithm. **b** When we apply A^*_t algorithm. Obstacle values are fixed at −100, and non visited free nodes have infinite values. Visited nodes are those colored from *clear blue* to *clear red*

Table 4 Average ratio between the number of nodes explored by each algorithm and the actual number of nodes in the optimal path, in G4-grids

Algorithm	100 × 100	500 × 500	1000 × 1000	2000 × 2000	Mazes (512 × 512)	Rooms (512 × 512)	VideoGames (512 × 512 to 1024 × 1024)	All
Dijkstra	45.2	258.8	506.2	1144.4	65.8	259.8	207.4	235.0
RD	44.0	257.4	505.1	1142.8	65.7	258.5	206.5	233.9
A^*_{wot}	9.5	41.4	81.9	149.0	46.2	45.8	43.9	44.5
RA^*_{wot}	9.5	41.4	81.9	149.0	46.2	45.8	43.8	44.5
A^*_t	5.1	10.4	26.6	25.0	45.3	42.0	32.6	23.3
RA^*_t	5.3	11.1	27.3	25.1	45.5	45.4	34.1	24.2
$A^*_{\epsilon=1.05}$	5.0	8.6	20.2	20.6	44.8	35.1	29.9	20.7
$A^*_{\epsilon=1.1}$	4.7	7.3	17.4	16.8	44.1	30.7	27.3	19.0

of the goal. We observe that our relaxed versions of Dijkstra and A* do not significantly reduce the number of explored nodes. In fact, the gain in time is rather obtained through the suppression of multiple visits to a given node. The number of explored nodes is even slightly smaller (in average) in A^*_t than in its relaxed version RA^*_t. This is due to the fact that the relaxed version may overestimate the value of g_score for a given node. Consequently, some paths that are not explored in A^*_t (because of higher cost) may be explored in RA^*_t. This also explains the fact that RA^*_t is, in some cases (34 %), slower than A^*_t, as can be observed in Fig. 11.

On the contrary, bounded relaxation algorithms ($A^*_{\epsilon=1.05}$ and ($A^*_{\epsilon=1.1}$) provide a gain in time mainly through the reduction of the domain of explored nodes (Table 4), at the risk of missing the optimal path and obtaining longer paths.

The value of the tie-break factor (tBreak $= 1 + 1/(\text{MapLength} + \text{MapWidth})$) is chosen such that the new heuristic function tBreak $* h$ is still consistent. Consequently,

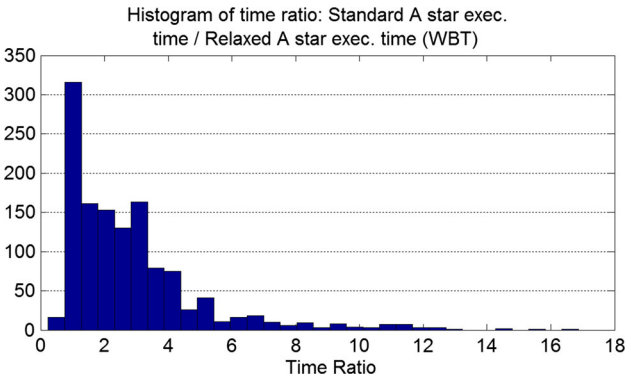


Fig. 11 Histogram of the execution time ratio between A^*_t and RA^*_t algorithms, in G4-grids

A^*_t still gives the optimal solution in all cases, while the quality of the solution yielded by RA^*_t is degraded compared to RA^*_{wot}, as can be noticed in Tables 1, 2 and 5. The percentage

Table 5 Average percentage of extra length compared to optimal path, calculated for non optimal paths, in G4-grids

Algorithm	100 × 100	500 × 500	1000 × 1000	2000 × 2000	Mazes (512 × 512)	Rooms (512 × 512)	VideoGames (512 × 512 to 1024 × 1024)	All
RA^*_{wot}	2.58	0.82	0.00	0.00	0.00	0.47	0.54	1.30
RA^*_t	4.09	1.64	0.55	0.23	0.80	1.26	2.13	1.61
$A^*_{\epsilon=1.05}$	2.16	1.24	1.03	0.81	0.13	0.81	0.72	1.00
$A^*_{\epsilon=1.1}$	2.83	2.51	2.07	1.74	0.24	1.24	1.3	1.77

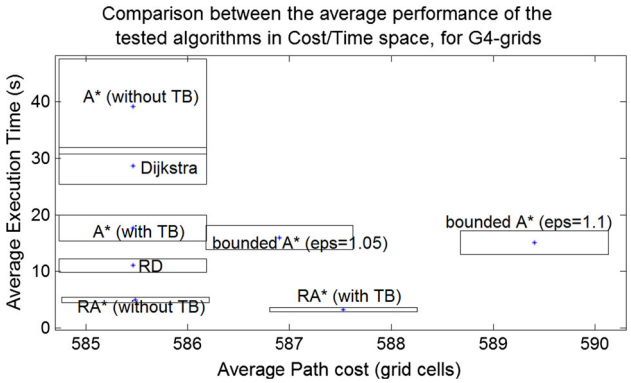


Fig. 12 Comparison between the tested algorithms, in G4-grids, in cost/time space, in terms of average and standard deviation. The *width of each rectangle* is proportional to the path cost standard deviation, and its *height* is proportional to the execution time standard deviation. The average is represented by a *star at the center of each rectangle*

of optimal solutions decreases from 98.9 to 80.0 % and the percentage of extra length for non optimal solutions increases from 1.3 to 1.6 %. This is due to the fact that tie-breaking reduces the domain of explored paths, thus leading to a higher probability that the first calculated g_score value for a given cell in RA* is not the correct one.

Nevertheless, our RA* algorithms (with or without tie-breaking) give a better trade-off between solution quality and execution time than tested versions of bounded relaxed A* ($A^*_{\epsilon=1.05}$ and $A^*_{\epsilon=1.1}$). First, RA^*_{wot} and RA^*_t provide the optimal path in more than 98.9 and 80.1 %, respectively, while $A^*_{\epsilon=1.05}$ and $A^*_{\epsilon=1.1}$ fail to find the optimal solution in 23.4 and 39.6 % of cases, respectively (Table 1). Second, RA^*_{wot} gives a better average path cost than both $A^*_{\epsilon=1.1}$ and $A^*_{\epsilon=1.2}$ (Table 2; Fig. 12), even though the percentage of average extra length in non optimal solution is lowest for $A^*_{\epsilon=1.05}$, as can be noticed in Table 5 (since optimal solutions are not counted in this table).

Finally, RA^*_t necessitates a much lower execution time (3 to 5 times faster, in average over all maps) than both $A^*_{\epsilon=1.05}$ and $A^*_{\epsilon=1.1}$ (Table 3), while yielding a better or very near solution quality. Our algorithms are even almost eight times faster in the category of mazes, where the maps are so complicated with a high ratio of obstacles that $A^*_{\epsilon=1.05}$ and $A^*_{\epsilon=1.1}$ provide very little gain in time compared to standard A*. Of course, if we take greater values of ϵ, we would obtain a lower execution time, but at the cost of degrading further the solution quality. While if we use values smaller than 1.05, we would have a better solution quality but the gain in execution time would be further reduced. This also raises the problem of choosing the best value of ϵ, which is not evident. In fact, we notice in Tables 1, 2, 3 and 5 that the performance of $A^*_{\epsilon=1.05}$ and $A^*_{\epsilon=1.1}$ varies significantly depending on the map size and type.

Table 6 shows the percentage of runs for which each one of the eight tested algorithm appears in rank 1 to 8, with regards to the execution time. The sum of each column and each row equals 100 %. We see that RA^*_t is the fastest in 50 % of the total 1273 runs (1290 minus 17 cases for which the goal is unreachable), and it is among the two fastest in more than 75 % of cases. As expected, Dijkstra is the slowest algorithm because it explores paths in all directions. But it should be understood that the scope of Dijkstra and A* algorithms is not exactly the same, since Dijkstra can calculate the shortest distance between a start node and all reachable nodes on the map, while A* focuses only on the distance between a pair of nodes.

Figure 13 depicts the box plots of the ratio between execution time and optimal path length for each algorithm. We divided by the optimal path length (which is a fixed value for a given run) to have a better visibility of the plot. On each box, the central mark is the median, the edges of the box are the 25th and 75th percentiles, the whiskers extend to the most extreme data points not considered outliers, and outliers are plotted individually (red crosses). We observe on this figure that our relaxed versions of Dijkstra and A* not only reduce the median execution time, but all the range of values is reduced. Bounded relaxation versions $A^*_{\epsilon=1.05}$ and $A^*_{\epsilon=1.1}$ present also a reduced range of execution time values but with some extreme values.

Nevertheless, considering only execution time values is not sufficient, since the tested algorithms also differ in terms of solution quality. For this purpose, Fig. 13 presents a comparison between the tested algorithms in cost/time space, in

Table 6 Percentage of runs for which each algorithm appears in rank 1 to 8, with regards to the execution time, in G4-grids

Algorithm	Fastest (%)	v2nd fastest (%)	3rd fastest (%)	4th fastest (%)	5th fastest (%)	6th fastest (%)	7th fastest (%)	Slowest (%)
RA^*_t	50.0	25.5	13.8	9.7	0.8	0.2	0.0	0.0
A^*_t	13.6	13.6	14.8	14.1	20.1	11.3	9.0	3.5
$A^*_{\epsilon=1.1}$	14.3	15.6	26.4	20.2	15.6	5.2	1.6	1.1
RA^*_{wot}	11.1	30.4	11.5	4.3	41.0	1.7	0.0	0.0
$A^*_{\epsilon=1.05}$	0.7	14.1	19.9	32.2	13.7	11.9	5.5	2.0
RD	10.1	0.9	13.6	2.7	4.8	23.0	44.9	0.0
A^*_{wot}	0.2	0.0	0.0	0.4	3.4	46.1	32.1	17.8
$A^*_{\epsilon=1.05}$	0.9	23.8	30.4	29.2	10.9	4.1	0.6	0.1
Dijkstra	0.0	0.0	0.1	16.4	0.6	0.5	6.8	75.6

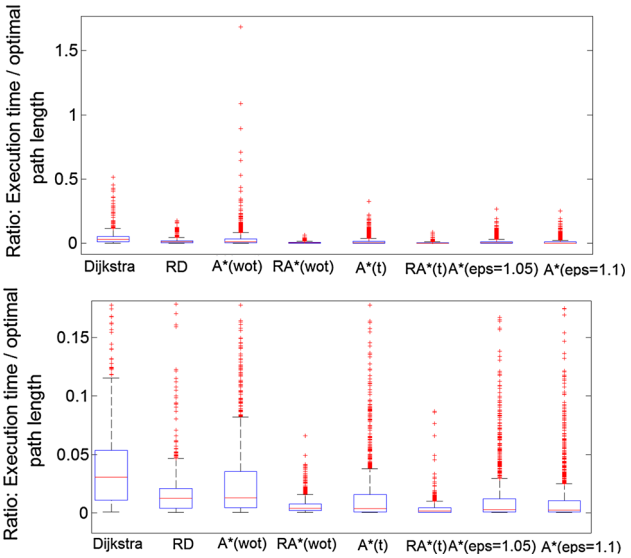


Fig. 13 Box plot of the ratio between execution time and optimal path length for each algorithm, in G4-grids. *Up* representing all the points (runs). *Down* a zoom of the same plot that discards the most extreme outliers

terms of average and standard deviation. The average is represented by a star at the center of each rectangle, and the size of the rectangle is proportional to the path cost and the execution time standard deviations. Since the path cost and the execution time values have a large variance (because the tests are made on various map sizes and path lengths), and for clarity purposes, we multiplied the path cost and the execution time standard deviations by 0.001 and 0.05 (respectively) to obtain the width and height of each rectangle. The more a rectangle is close to the left down corner, the better the performance. This figure shows clearly the superiority of our relaxed algorithms (RA^*_t and especially RA^*_{wot}) compared to bounded relaxations of A* ($A^*_{\epsilon=1.05}$ and $A^*_{\epsilon=1.1}$) when considering the trade-off between path cost and execution time. Notice that taking smaller values than 1.05 for ϵ, or higher values than 1.1 would not provide a better tradeoff, since, in the first case, it would move towards the position of A^*_t on the plot, and in the second case, it would move towards the right down corner (smaller execution time but larger path cost).

4.2 Results in G8-grids

When dealing with G8-grids, a path between start and goal nodes always exists for 1279 runs over 1290 (there 11 impossible paths, all of them in the category of video-game maps). In G8-grids, the manhattan distance is no longer a consistent heuristic (nor even admissible). Instead, we used the diagonal shortcut distance, which consists in taking the diagonal line when possible then completing the path with a horizontal or vertical line. The diagonal shortcut distance between two grid points $(x1, y1)$ and $(x2, y2)$ is $ds = \min(|x1-x2|, |y1-y2|) * \sqrt{2} + \max(|x1-x2|, |y1-y2|) - \min(|x1-x2|, |y1-y2|)$.

It can be easily demonstrated that the diagonal shortcut distance is a consistent heuristic in a G8-grid. As expected, Standard Dijkstra and Standard A* (with or without broken ties) provide always optimal paths (Tables 7, 9), while our three relaxed versions are near-optimal, but with a reduced execution time, as can be seen in Figs. 14, 15, 16, 17, 18, 19, 20 and 21. More precisely, Table 8 shows that even when the relaxed algorithms do not provide the optimal path, they provide a near-optimal path which is in average 1.8 % longer for RD, 1.7 % longer for RA^*_{wot} and 4.9 % longer for RA^*_t. As explained in the previous section, introducing a tie-break factor degrades the solution quality of RA*. Besides, Table 10 shows that, in G8-grids, RA^*_t is only 6 % faster, in average, than RA^*_{wot}, in terms of execution time. In fact, the introduction of the tie-break factor proves to be more beneficial for the standard A* algorithm than for RA*, for the same reasons explained in the previous section.

When comparing our RA* algorithms to bounded relaxed A* versions ($A^*_{\epsilon=1.1}$ and $A^*_{\epsilon=1.2}$), in G8-grids, we notice that RA^*_{wot} gives the best trade-off between solution quality and

Table 7 Percentage of optimal paths (when a path exists) for the different algorithms, per environment size, in G8-grids

Algorithm	100 × 100	500 × 500	1000 × 1000	2000 × 2000	Mazes (512 × 512)	Rooms (512 × 512)	VideoGames (512 × 512 to 1024 × 1024) %	All %
Dijkstra	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100
RD	98.9	98.3	96.7	100.0	100.0	85.8	98.0	97.3
A^*_{wot}	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
RA^*_{wot}	79.6	85.0	78.9	95.7	21.7	25.0	47.2	62.9
A^*_t	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0
RA^*_t	39.3	27.0	10.0	21.7	18.9	1.7	11.6	21.0
$A^*_{\epsilon=1.1}$	67.8	37.0	18.9	20.0	31.7	29.2	31.2	38.2
$A^*_{\epsilon=1.2}$	54.4	32.3	15.0	16.7	25.6	10.0	26.6	30.3

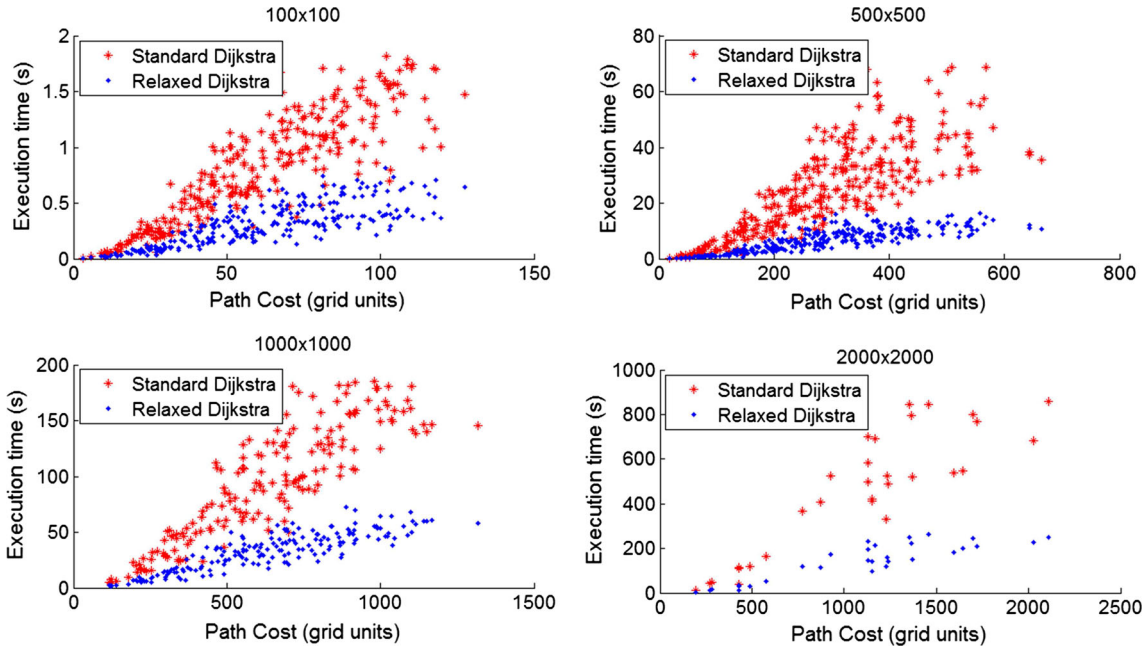
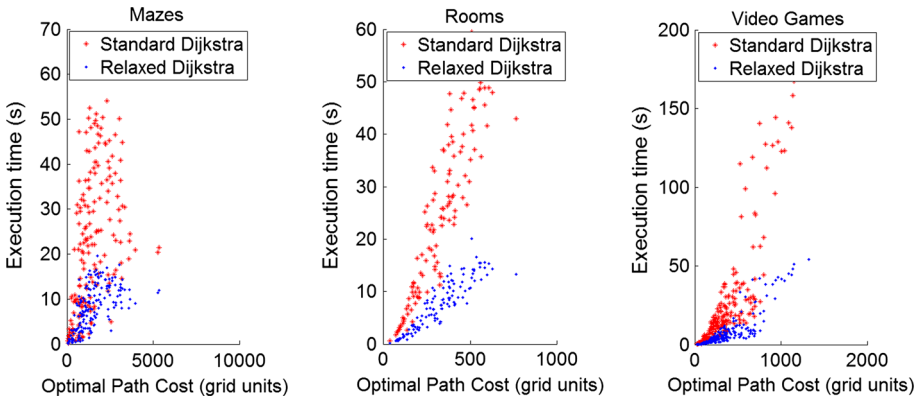


Fig. 14 Comparison of execution time between Dijkstra and RD algorithms, for different environment sizes, randomly generated, in G8-girds

Fig. 15 Comparison of execution time between Dijkstra and RD algorithms, for different structured environments, in G8-girds



execution time. First, it provides the optimal solution in 63 % of cases, while this percentage does not exceed 40 % for $A^*_{\epsilon=1.1}$ and $A^*_{\epsilon=1.2}$ (Table 7). Second, it gives a better solution quality than $A^*_{\epsilon=1.2}$, and a very near solution quality to $A^*_{\epsilon=1.1}$ (Tables 8, 9), while being around five times faster, in average, than both $A^*_{\epsilon=1.1}$ and $A^*_{\epsilon=1.2}$. This time ratio reaches even

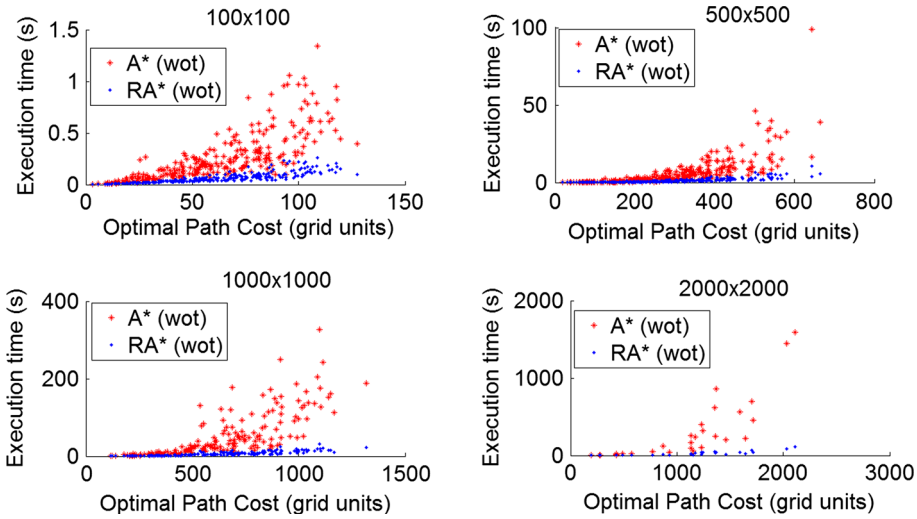


Fig. 16 Comparison of execution time between A^*_{wot} and RA^*_{wot} algorithms, for different environment sizes, randomly generated, in G8-grids

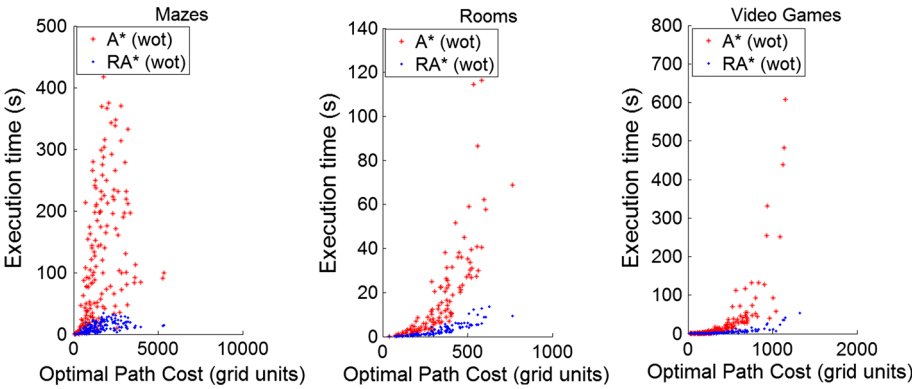


Fig. 17 Comparison of execution time between A^*_{wot} and RA^*_{wot} algorithms, for different structured environments, in G8-grids

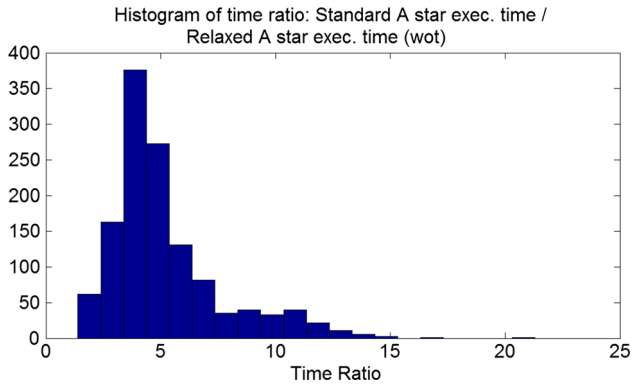


Fig. 18 Histogram of the execution time ratio between A^*_{wot} and RA^*_{wot} algorithms, in G8-grids

nearly ten times in maze maps (Table 6). It is understood that if we use greater values of ϵ, the solution quality would degrade further, while if we use values smaller than 1.1, the gain in execution time would be further reduced.

Table 11 contains the average ratio between the number of nodes explored by each algorithm and the actual number of nodes in the optimal path. We notice that all the tested algorithms explore much more nodes than those contained in the path. As in G4-grids, we observe that Dijkstra algorithm visits the largest number of nodes because its exploration is made in all directions. We also notice that the number of explored nodes is smaller (in average) in A^*_t than in its relaxed version RA^*_t , for the same reason explained in the previous section. Again, bounded relaxation algorithms ($A^*_{\epsilon=1.1}$ and $A^*_{\epsilon=1.2}$) explore the least number of nodes, at the risk of missing the optimal path and obtaining longer paths.

Table 12 shows the percentage of runs for which each one of the eight tested algorithm appears in rank 1 to 8, with regards to the execution time. We see that RA^*_t is the fastest in 37 % of cases, followed by RA^*_{wot}. RA^*_t and RA^*_{wot} are markedly faster than A^*_{wot} and A^*_t, which are even slower than RD, in many cases, especially in the structured maps, as can be seen in Table 6. Again, standard Dijkstra algorithm is the slowest algorithm because it explores paths in all directions. Nevertheless, such an exploration makes it faster than A* in the case of complex environments such as mazes (Table 6). That is because, in such maps with high obstacle ratio, A* gets obstructed each time it explores nodes in the direction of the goal.

Figure 22 depicts the box plots of the ratio between execution time and optimal path length for each algorithm. As

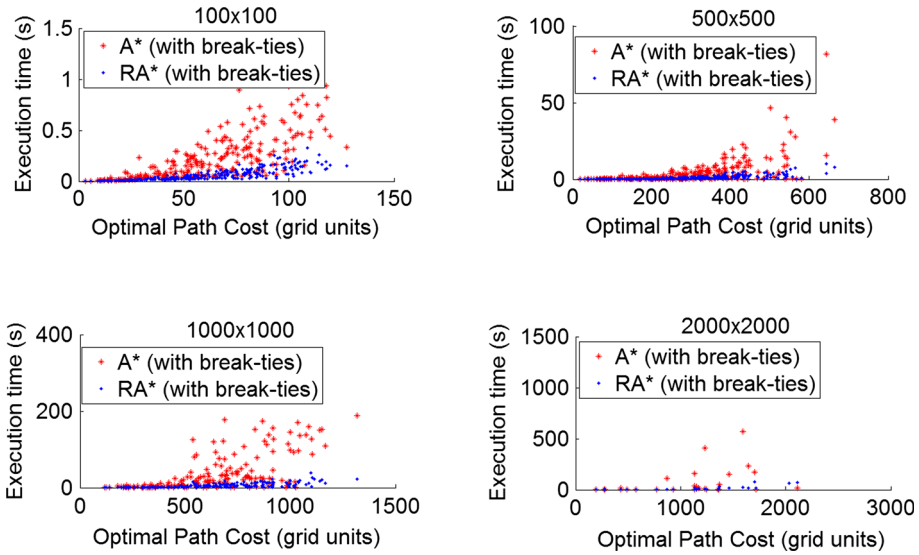


Fig. 19 Comparison of execution time between A_t^* and RA_t^* algorithms, for different environment sizes, randomly generated, in G8-grids

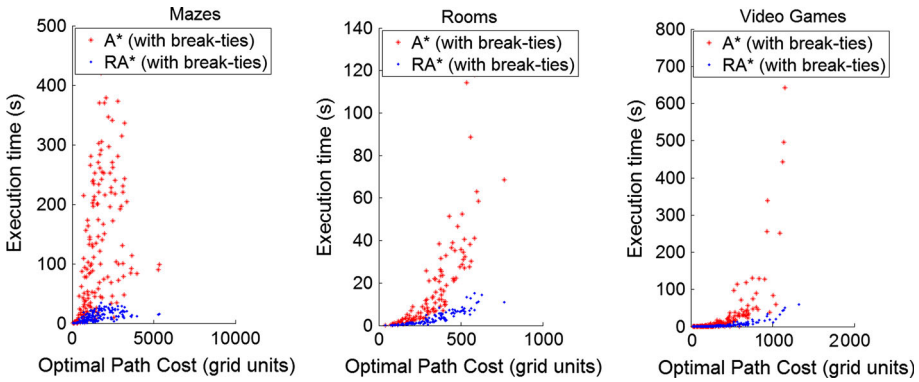


Fig. 20 Comparison of execution time between A_t^* and RA_t^* algorithms, for different structured environments, in G8-grids

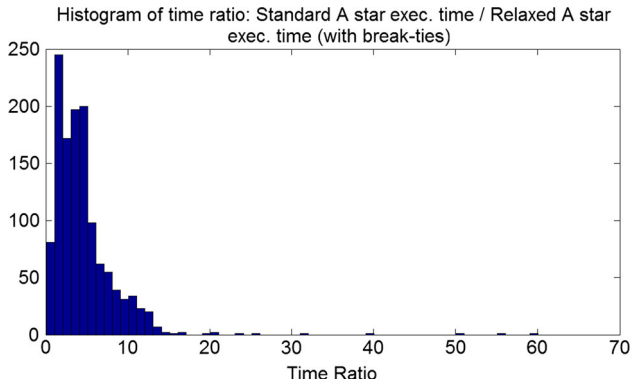


Fig. 21 Histogram of the execution time ratio between A_t^* and RA_t^* algorithms, in G8-grids

Table 8 Percentage of extra length compared to optimal path, calculated for non optimal paths, in G8-grids

Algorithm	Mean (%)	Std (%)	Max (%)
Dijkstra	0.0	0.0	0.0
RD	1.8	2.2	9.4
A^*_{wot}	0.0	0.0	0.0
RA^*_{wot}	1.7	1.6	7.4
A_t^*	0.0	0.0	0.0
RA_t^*	4.9	3.4	18.4
$A^*_{\epsilon=1.1}$	1.4	1.1	6.8
$A^*_{\epsilon=1.2}$	2.4	1.9	13.1

in G4-grids, we observe that our relaxed versions of Dijkstra and A* not only reduce the median execution time, but also all the range of values. Bounded relaxation versions $A^*_{\epsilon=1.05}$ and $A^*_{\epsilon=1.1}$ present also a reduced range of execution time values but with some extreme values.

Besides, Fig. 23 presents a comparison between the tested algorithms in cost/time space, in terms of average and standard deviation. For clarity purposes, we multiplied the path cost and the execution time standard deviations by 0.003 and 0.07 (respectively) to obtain the width and height of each rectangle. The more a rectangle is close to the left down corner, the better the performance. We observe that our RA^*_{wot} algorithm assures the best trade-off between execution time and solution quality. As for RA_t^*, while having the minimum execution time, it has also the maximum path cost.

Table 9 Average path cost (in grid unities) of the six algorithms, in G8-grids

Algorithm	100 × 100	500 × 500	1000 × 1000	2000 × 2000	Mazes (512 × 512)	Rooms (512 × 512)	VideoGames (512 × 512 to 1024 × 1024)	All
Dijkstra	60.43	284.80	631.23	1086.25	1479.08	317.14	375.80	490.26
RD	60.44	284.90	631.91	1086.25	1479.08	317.74	376.13	490.49
A^*_{wot}	60.43	284.80	631.23	1086.25	1479.08	317.14	375.80	490.26
RA^*_{wot}	60.69	285.03	632.28	1086.29	1500.99	321.00	381.32	494.82
A^*_t	60.43	284.80	631.23	1086.25	1479.08	317.14	375.80	490.26
RA^*_t	62.31	293.37	652.36	1114.25	1540.31	341.60	400.05	510.98
$A^*_{\epsilon=1.1}$	60.86	288.58	640.51	1101.86	1481.98	320.37	379.85	494.25
$A^*_{\epsilon=1.2}$	61.46	290.95	646.24	1105.06	1485.50	326.06	383.94	497.48

Table 10 Average execution time (in seconds) of the six algorithms, in G8-grids

Algorithm	100 × 100	500 × 500	1000 × 1000	2000 × 2000	Mazes (512 × 512)	Rooms (512 × 512)	VideoGames (512 × 512 to 1024 × 1024)	All
Dijkstra	0.82	24.92	95.91	459.91	22.52	23.54	27.59	39.97
RD	0.30	6.70	33.03	141.54	7.86	7.42	8.59	12.74
A^*_{wot}	0.20	5.92	50.07	296.28	111.60	17.67	30.40	37.53
RA^*_{wot}	0.06	1.15	6.33	24.82	11.04	2.87	3.50	4.12
A^*_t	0.23	3.71	39.27	113.96	113.00	17.76	29.97	31.35
RA^*_t	0.06	1.01	5.51	13.55	11.19	3.47	4.11	3.88
$A^*_{\epsilon=1.1}$	0.12	1.75	16.69	7.96	109.44	11.50	22.46	22.95
$A^*_{\epsilon=1.2}$	0.10	1.20	9.34	4.06	107.42	7.52	16.62	20.12

This can be explained by the fact that in G8 grids, the probability that the first calculated g_score is not the correct one is greater than in G4-grids (because of the additional diagonal move possibility), and this probability becomes even higher when we use tie-breaking, since it limits further the domain of explored paths. So, we conclude that in G8-grids, it is definitely preferable to use RA* algorithm without tie-breaking. On another hand, RD algorithm presents the second better overall trade-off between solution quality and execution time: while yielding the optimum path in 97 % of cases, it is 2.5 times faster than A* in average over all maps. But when we look closer, it appears that RD is only 5 % faster than A^*_t over the first category of random maps, while it is seven times faster in average over the three other categories of structured maps, which are more complex. This time ratio reaches even 14.4 over mazes.

5 Integration in ROS

To demonstrate the feasibility and effectiveness of our proposed path planners in real-world scenarios, we have integrated the RA* algorithm as global path planner in the Robot Operating System (ROS) (2015) as possible replacement of the default $navfn$ path planner (based on a variant of the Dijkstra's algorithm). In what follows, we present an overview on ROS, we describe the integration process of RA* as global path planner and evaluate its performance against the default global path planner.

ROS is a free and open-source robotic middleware for the large-scale development of complex robotic systems. it acts as a meta-operating system for robots as it provides hardware abstraction, low-level device control, inter-processes message-passing and package management. The main advantage of ROS is that it allows manipulating sensor data of the robot as a labelled abstract data stream, called topic, without having to deal with hardware drivers. This makes the programming of robots much easier for software developers as they do not have to deal with hardware drivers and interfaces.

5.1 Simulation model

5.1.1 ROS navigation stack

Mobile robot navigation generally requires solutions for three different problems: mapping, localization and path planning. In ROS, the *Navigation Stack* plays such a role to integrate

Table 11 Average ratio between the number of nodes explored by each algorithm and the actual number of nodes in the optimal path, in G8-grids

Algorithm	100 × 100	500 × 500	1000 × 1000	2000 × 2000	Mazes (512 × 512)	Rooms (512 × 512)	VideoGames (512 × 512 to 1024 × 1024)	All
Dijkstra	65.2	362.0	699.7	1569.2	88.4	366.0	298.1	327.1
RD	62.8	358.7	697.6	1628.5	88.2	362.3	296.1	326.2
A^*_{wot}	8.6	29.4	69.7	106.0	61.9	63.3	49.1	43.3
RA^*_{wot}	8.4	28.3	66.7	105.0	62.1	63.3	48.5	42.5
A^*_t	7.6	19.3	53.1	44.5	61.6	62.5	44.6	36.1
RA^*_t	8.1	24.1	56.1	52.0	62.2	75.9	50.9	40.3
$A^*_{\epsilon=1.1}$	6.0	10.2	27.9	11.4	59.9	43.2	34.5	25.7
$A^*_{\epsilon=1.2}$	5.2	7.8	18.6	8.3	58.3	30.3	28.7	21.3

Table 12 Percentage of runs for which each algorithm appears in rank 1 to 8, with regards to the execution time, in G8-grids

Algorithm	Fastest	2nd fastest	3rd fastest	4th fastest	5th fastest	6th fastest	7th fastest	Slowest
$A^*_{\epsilon=1.2}$	24.2	12.1	28.1	17.0	16.2	1.5	0.7	0.2
RA^*_t	37.1	30.1	20.6	10.1	2.0	0.2	0.0	0.0
$A^*_{\epsilon=1.1}$	1.3	17.1	17.7	33.6	13.3	13.8	2.7	0.5
RA^*_{wot}	24.5	34.0	17.6	11.8	12.1	0.0	0.0	0.0
A^*_t	2.0	5.6	3.8	6.1	33.5	19.2	16.9	13.0
RD	11.1	1.1	12.0	4.7	12.8	7.3	50.9	0.0
A^*_{wot}	0.0	0.0	0.1	0.5	9.7	55.5	27.8	6.4
Dijkstra	0.0	0.0	0.0	16.2	0.5	2.4	1.0	79.9

together all the functions necessary for autonomous navigation. The path planning is performed in the *move_base* package and is divided into *global* and *local* planning modules which is a common strategy to deal with the complex planning problem.

The *global path planner* is responsible for generating a long-term plan from the start or current position to the goal position before the robot starts moving. A grid-based global planner that uses Dijkstra's algorithm to compute shortest collision-free path for a robot is used by default in the *navfn* package. The current implementation of the global planner in ROS assumes a circular-shape robot. This results in generating an optimistic path for the actual robot footprint, which may be infeasible path. Besides, the global planner ignores kinematic and acceleration constraints of the robot, so the generated path could be dynamically infeasible.

On the other hand, the *local path planner* (also called the controller) is seeded with the plan produced by the global planner and attempts to follow it as closely as possible while taking into account the kinematics and dynamics of the robot as well as the obstacle information. In order to generate safe velocity commands, a package named *base_local_planner* provides implementations of the Trajectory Rollout (Gerkey and Konolige 2008) and the Dynamic Window Approach (DWA) (Fox et al. 1997) to perform forward simulation and the selection among potential commands.

Also, the *move_base* package maintains two costmaps, *global_costmap* and *local_costmap*, to be used with the global and local planners, respectively. The costmap provides a configurable structure to maintain information about where the robot should navigate in the form of occupancy grid. The costmap uses sensor data and information from the static map to store and update information about obstacles in the world.

5.1.2 Integration of a new global planner to ROS navigation stack

In what follows, we present the main guidelines for integrating a new global path planner to ROS navigation stack. For more detailed step-by-step instructions, the reader is referred to our ROS tutorial available on this link Iroboapp (2015b) and ROS Wiki (2015). For any global or local planner to be used with the *move_base* it must first adhere to some interfaces defined in *nav_core* package, which contains key interfaces for the navigation stack; then it must be added as a plugin to ROS.

All the methods defined in *nav_core* :: *BaseGlobal Planner* class must be overridden by the new global path planner. The main methods in the *nav_core* :: *BaseGlobal Planner* class are *initialize* and *makePlan*. The *initialize* is an initialization function, which initializes the costmap for the planner. We use this function to get the costmap; then we

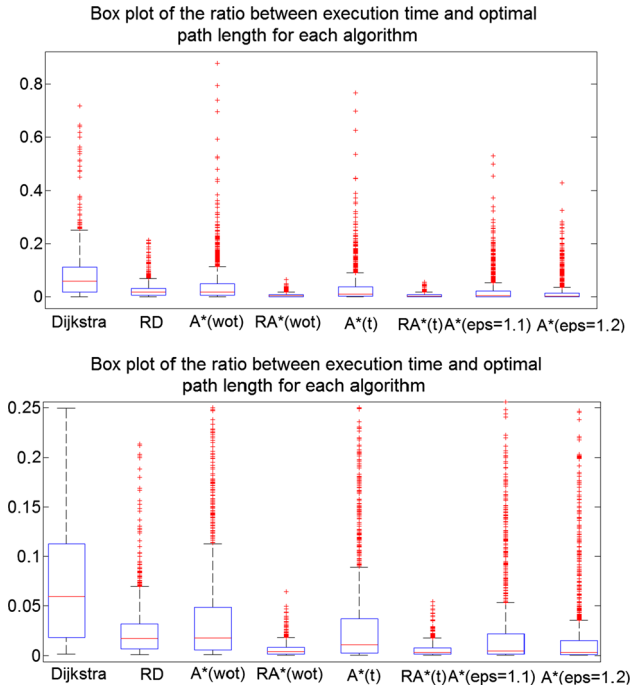


Fig. 22 Box plot of the ratio between execution time and optimal path length for each algorithm, in G8-grids. *Up* representing all the points (runs). *Down* a zoom of the same plot that discards the most extreme outliers

store it in an array of *bools* where *true* means a free cell and *false* otherwise. The *makePlan* function is responsible for computing the global path. It takes the start and goal positions as an input. In this function, we first convert the start and goal coordinates into cells ID. Then, we pass those IDs with the map array to the RA* planner. To implement RA* planner we used the sorted *multiset* data structure to maintain the open set, and it sorted the cells based on their *f_score* values. This allows a significant decrease of the execution time as we only need to pick up the first element in the sorted set (having the lowest *f_score*) instead of search for it in each iteration. When the planner completes its execution, it returns the computed path to the *makePlan*. Finally, the path will be converted to x and y coordinates and sent back to the *move_base* which will publish it as a new path to ROS ecosystem.

5.2 Performance evaluation

For the experimental evaluation study using ROS, we have used the real-world Willow Garage map (Fig. 24), with dimensions 584 × 526 cells and a resolution 0.1 m/cel. In this map, the white color represents the free area, the black color represents the obstacles or walls and the grey color represents the unknown area.

We considered 30 different scenarios, where each scenario is specified by the coordinates of randomly chosen start and goal cells. Each scenario, with specified start/goal cells, is repeated 30 times (i.e. 30 runs for each scenario). In total,

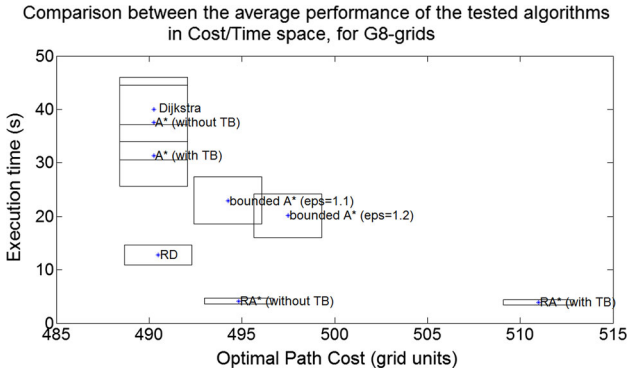


Fig. 23 Comparison between the tested algorithms in cost/time space, in terms of average and standard deviation, in G8-grids. The *width of each rectangle* is proportional to the path cost standard deviation, and its *height* is proportional to the execution time standard deviation. The average is represented by a *star at the center of each rectangle*

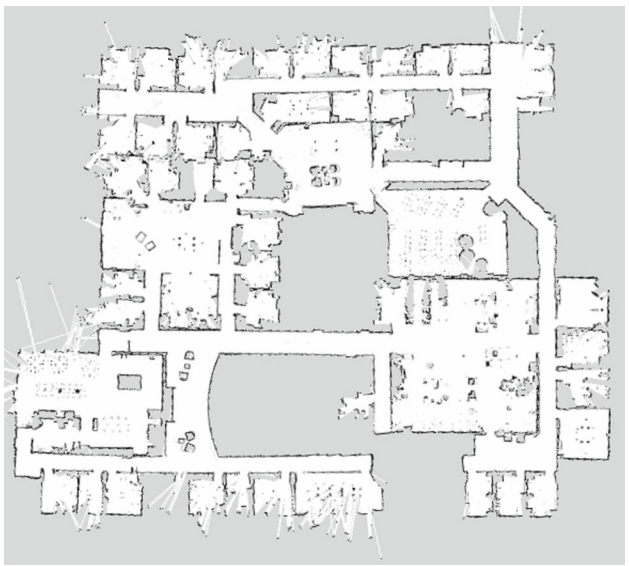


Fig. 24 Willow Garage map

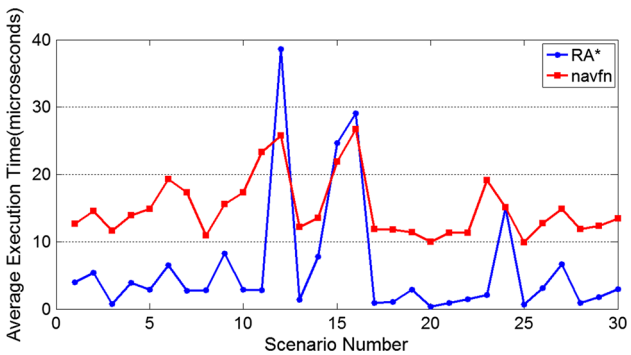


Fig. 25 Average execution time (microseconds) of RA* and navfn

Table 13 Execution time in (microseconds) and path length in (meters) of RA* and navfn

Planner	Execution time Total	Average	Path length Total	Average
RA*	185.1265156	6.170884 ± 0.504922	582.34962	19.41165
navfn	448.6488877	14.95496 ± 1.174413	564.9527	18.83176

900 runs for the Willow Garage map are performed in the performance evaluation study for each planner.

Two performance metrics are considered to evaluate the global planners: (1) *the path length*: it represents the length of the shortest global path found by the planner, (2) *the execution time*: it is the time spent by an algorithm to find its best (or optimal) solution.

Figure 25 shows that the RA* is faster than *navfn* in 87 % of the cases. Table 13 shows that, in average, the RA* is much faster than *navfn*, with execution time less than the half of that of *navfn*. RA* provides near-optimal paths, which are in average only 3.08 % longer than *navfn* paths. These results confirm the MATLAB simulation results about the efficiency of the RA*. Moreover, the small loss in terms of path quality does not matter in practice as this path will be used only as a guide for *local path planner*, which will generate the actual path that the robot will follow (which may be different from the global path) considering the kinematics and dynamics of the robot as well as moving obstacles, if any.

6 Conclusion

In this paper, we have presented two grid-specific relaxed versions RA* and RD of traditional A* and Dijkstra algorithms. We exploited the structural properties of the problem to efficiently explore the solution space. The goal is to deal with large-scale path planning problems in grid environments in reasonable time. We have conducted extensive simulations to evaluate the two proposed algorithms, and we have demonstrated that they explore the solution space in linear time. Besides, RD is optimal in G4 grids and near-optimal in G8 grids (optimal in 97 % of cases), while RA* is near-optimal in both G4 and G8 grids, yielding optimal solutions in 99 and 63 % (for RA^*_{wot}) of cases, respectively. Then, we can conclude that the searching strategies used in RA* and RD manage to cover a large part of the solution space, while significantly reducing the execution time. We have also compared our algorithms against previous bounded relaxation versions of A* and showed that RD and RA* provide better trade-offs between solution quality and execution time.

Notice that the proposed approaches are not restricted to robotic applications. They are generic and may be exploited in other applications such as games and routing problems. As future work, we plan to conduct experiments on real structured environments and implement our algorithms on robots to evaluate their practical behaviour.

Acknowledgments This work is supported by the iroboapp project "Design and Analysis of Intelligent Algorithms for Robotic Problems and Applications" Iroboapp (2015a) under the grant of the National Plan for Sciences, Technology and Innovation (NPSTI), managed by the Science and Technology Unit of Al-Imam Mohamed bin Saud University and by King AbdulAziz Center for Science and Technology (KACST). This work is partially supported by Prince Sultan University.

References

Alajlan M, Koubaa A, Chaari I, Bennaceur H, Ammar A (2013) Global path planning for mobile robots in large-scale grid environments using genetic algorithms. In: 2013 international conference on individual and collective behaviors in robotics ICBR'2013, Sousse

Andreas K, Kaindl H (1992) A new approach to dynamic weighting. In: Proceedings of the 10th European conference on artificial intelligence (ECAI-92), Vienna, pp 16–17

Antsfeld L, Harabor DD, Kilby P, Walsh T (2012) Transit routing on video game maps. In: AIIDE

Cazenave T (2006) Optimizations of data structures, heuristics and algorithms for path-finding on maps. In: CIG, pp 27–33

Chaari I, Koubaa A, Bennaceur H, Trigui S, Al-Shalfan K (2012) Smartpath: a hybrid ACO-GA algorithm for robot path planning. In: 2012 IEEE congress on evolutionary computation (CEC), Brisbane, pp 1–8

Choubey N, Gupta MBK (2013) Analysis of working of Dijkstra and A* to obtain optimal path. Int J Comput Sci Manag Res 2:1898–1904

Dijkstra EW (1959) A note on two problems in connexion with graphs. Numer Math 1(1):269–271

Fox D, Burgard W, Thrun S (1997) The dynamic window approach to collision avoidance. IEEE Robotics Autom Mag 4(1):23–33

Fredman ML, Robert TE (1984) Fibonacci heaps and their uses in improved network optimization algorithms. In: 25th IEEE annual symposium on foundations of computer science, pp 338–346

Gerkey BP, Konolige K (2008) Planning and control in unstructured terrain. In: Workshop on path planning on costmaps. Proceedings of the IEEE international conference on robotics and automation (ICRA)

Hadlock FO (1977) A shortest path algorithm for grid graphs. Netw Int J 7:323–334

Harabor D, Grastien A (2011) Online graph pruning for pathfinding on grid maps. In: Proceedings of association for the advancement of artificial intelligence

Hart PE, Nilsson NJ, Raphael B (1968) A formal basis for the heuristic determination of minimum cost paths. IEEE Trans Syst Sci Cybern 4:100–107

Ira P (1970) First results on the effect of error in heuristic search. Mach Intell 5:219–236

Iroboapp (2015a) Design and analysis of intelligent algorithms for robotic problems and applications. http://www.iroboapp.org. Accessed 3 Feb 2015

Iroboapp (2015b) Adding a global path planner as plugin in ROS. http://www.iroboapp.org/index.php?title=Adding_A_Global_Path_Planner_As_Plugin_in_ROS. Accessed 27 Apr 2015

Jigang W, Han P, Jagadeesh GR, Srikanthan T (2010) Practical algorithm for shortest path on large networks with time-dependent edge-length. In 2010 2nd international conference on computer engineering and technology (ICCET), vol 2, Chengdu, pp 57–60

Judea P (1984) Heuristics: intelligent search strategies for computer problem solving. Addison-Wesley, New York

Kanoulas E, Du Y, Xia T, Zhang D (2006) Finding fastest paths on a road network with speed patterns. In: Proceedings of the 22nd international conference on data engineering, ICDE'06, pp 10–19

Koenig S, Likhachev M (2002) D* lite. In: Proceedings of the 18th national conference on artificial intelligence (AAAI), pp 476–483

Likhachev M, Ferguson D, Gordon G, Stentz A, Thrun S (2005) Anytime dynamic A*: an anytime, replanning algorithm. In: Proceedings of the international conference on automated planning and scheduling (ICAPS)

Maps:benchmark (2015). http://movingai.com/benchmarks. Accessed 3 Mar 2015

Masehian E, Amin-Naseri MR (2006) A tabu search-based approach for online motion planning. In: IEEE international conference on industrial technology, Mumbai, pp 2756–2761

Maxim Likhachev GG, Thrun S (2004) Ara*: Anytime A* with provable bounds on sub-optimality. In: Advances in neural information processing systems (NIPS), vol 16. MIT Press, New York

Peyer S, Rautenbach D, Vygen J (2009) A generalization of Dijkstra's shortest path algorithm with applications to VLSI routing. J Discrete Algorithms 7:377–390

Pohl I (1973) The avoidance of (relative) catastrophe, heuristic competence, genuine dynamic weighting and computational issues in heuristic problem solving. In: Proceedings of the third international joint conference on artificial intelligence (IJCAI-73), California, pp 12–17

Potamias M, Bonchi F, Castillo C, Gionis A (2009) Fast shortest path distance estimation in large networks. In: Proceedings of the 18th ACM conference on information and knowledge management (CIKM 09), Hong Kong, pp 867–876

Robot Operating System (ROS) (2015). http://www.ros.org. Accessed 15 Apr 2015

ROS Wiki (2015) Writing a global path planner as plugin in ROS. http://wiki.ros.org/navigation/Tutorials/Writing%20A%20Global%20Path%20Planner%20As%20Plugin%20in%20ROS. Accessed 4 June 2015

Russell S, Norvig P (2009) Artificial intelligence: a modern approach, 3nd edn. Prentice Hall, New York

Shiltagh NA, Jalal LD (2013) Optimal path planning for intelligent mobile robot navigation using modified particle swarm optimization. Int J Eng Adv Technol (IJEAT) 2(4):260–267

Sturtevant N (2012) Benchmarks for grid-based pathfinding. Trans Comput Intell AI Games 4(2):144–148. http://web.cs.du.edu/sturtevant/papers/benchmarks.pdf

Sven Peyera JV, Rautenbachb D (2009) A generalization of Dijkstra's shortest path algorithm with applications to VLSI routing. J Discrete Algorithms 7:377–390

Tiwari R, Shukla A, Kala R (2012) Intelligent planning for mobile robotics: algorithmic approaches

van den Berg J, Shah R, Huang A, Goldberg K (2005) ANA*: anytime nonparametric A*. In: Annual conference of the association for the advancement of artificial intelligence (AAAI)