Advances in Linear Algebra & Matrix Theory, 2012, 2, 20-24
http://dx.doi.org/10.4236/alamt.2012.22003 Published Online June 2012 (http://www.SciRP.org/journal/alamt)

A Linear Interpolation-Based Algorithm for Path Planning and Replanning on Girds*

Changwen Zheng, Jiawei Cai, Huafei Yin
National Key Laboratory of Integrated Information System Technology Institute of Software,
Chinese Academy of Sciences, Beijing, China
Email: changwen@iscas.ac.cn

Received April 5, 2012; revised May 5, 2012; accepted May 20, 2012

ABSTRACT

Field D* algorithm is widely used in mobile robot navigation since it can plan and replan any-angle paths through non-uniform cost grids. However, it still suffers from inefficiency and sub-optimality. In this article, a new linear interpolation-based planning and replanning algorithm, Update-Reducing Field D*, is proposed. It employs different approaches during initial planning and replanning respectively in order to reduce the number of updates of the *rhs*-values of vertices. Experiments have shown that Update-Reducing Field D* runs faster than Field D* and returns smoother and lower-cost paths.

Keywords: Field D* Algorithm; Path Planning and Replanning; Any-Angle Path; Linear Interpolation; Grid Cell

1. Introduction

In mobile robot navigation, path planning leads a robot from its initial location to some desired goal location. The two most popular techniques for path planning are deterministic algorithms and randomized algorithms [1]. Among deterministic algorithms, A* provides heuristic search in static, known environments [2]. LPA* combines heuristic search and incremental search [3]. D* Lite could replan in unknown environments efficiently [4].

When provided with a grid-based representation of environments, these algorithms are limited by the discrete set of possible headings between gird cells. For example, the eight-connected grids restrict the agent's heading changes by multiples of $\pi/4$. As a result, the paths are suboptimal and unrealistic looking. To alleviate this problem, several methods for any-angle path planning have been investigated. A* PS uses post-smoothing to generate any-angle path [5]. But it does not always work as is showed in **Figure 1** (The path returned by A* PS (in black line) is not the optimal path (in dash line) from *S* to *G*). Basic Theta* algorithm and AP Theta* algorithm (Angle-Propagation Theta*) allow the parent of a node to be a node other than its local neighbor [6]. AA* (Accelerated A*) can plan a shortest any angle paths fast [7]. However, these algorithm are only usable for uniform cost environments.

Based on D* Lite and linear interpolation, Field D* could fast plan and replan any-angle path in grid environments, whatever the environment costs are known or partially-known, uniform or non-uniform [8]. Field D* is employed as the path replanner in a wide range of fielded robotic systems.

However, Field D* still suffers from two major drawbacks: 1) It plans and replans much slower than D* Lite. 2) The path returned by Field D* is not always the optimal solution. Motivated by these observations, a linear interpolation-based planning and replanning algorithm, Update-Reducing Field D* (URFD*), is proposed in this paper. It reduces the number of updates of the *rhs*-values so as to speed up the search. It also employs a method of post-smoothing to generate a lower-cost and smoother path. Besides, a heuristic with a variable factor according to the environments is used. As a result, the novel algorithm could efficiently produce a near-optimal path in non-uniform cost grids.

2. Linear Interpolation-Based Path Planning

2.1. The Idea of Field D*

Field D* stores the *rhs*-value, a one-step look ahead estimate of the goal distance (by the goal distance of a vertex we mean the cost of an optimal path from this vertex to the goal). For vertex *s* it satisfies:

$$rhs(s)=\begin{cases}0, & \text{if } s=s_{goal}\\ \min_{s'\in nbrs(s)}\left(g(s')+c(s',s)\right) & \text{otherwise}\end{cases} \quad (1)$$

*This work was supported in part by the National Natural Science Foundation of China under Grant 60873183.

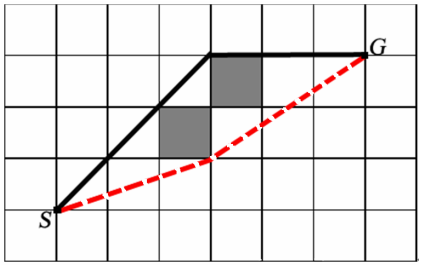

Figure 1. Sub-Optimality of A* PS.

where s_{goal} is the goal vertex. $Nbrs(s)$ denotes the set of all neighboring vertices of s. $g(s')$ is an estimate of goal distance of s'. $c(s',s)$ is the cost of a path between s' and s. In classical grid-based methods, it is assumed that s' and s are two corner vertices and the path between s' and s is a straight line. Field D* relaxes this assumption and takes any point along the boundary of a cell into consideration. To make it possible, Field D* makes an approximation that the path cost of any point s_y residing on the edge between two consecutive corner vertices s_1 and s_2 is a linear combination of $g(s_1)$ and $g(s_2)$:

$$g(s_y) = yg(s_2) + (1-y)g(s_1) \quad (2)$$

where y is the distance from s_1 to s_y (assuming unit cells).

With the form of the optimal path in a unit cell, Field D* could compute $rhs(s)$ and find the point to move by making

$$\mathrm{d}\left(c(s',s) + g(s')\right)/\mathrm{d}v = 0, \quad (3)$$

where v is the variable on which the path cost depends.

2.2. Differences and Inefficiency

A linear interpolation-based replanner performs differently from a classical path replanner such as D* Lite, which leads to its inefficiency. To explain this, we define $g^*(s)$ as the cost of the optimal path from vertex s to the goal with respect to the linear interpolation assumption (so it is slightly different from the cost of the actual optimal path). We call $g(s)$ (or $rhs(s)$) is inaccurate when $g(s)$ (or $rhs(s)$) is not equal to $g^*(s)$. Also we call $g(s)$ is more accurate than $rhs(s)$ when $g(s)$ is more close to $g^*(s)$ than $rhs(s)$. $g^*(s)$ satisfies:

$$g^*(s) = \begin{cases} 0, & \text{if } s = s_{goal} \\ \min_{s' \in nbrs(s)} \left(g^*(s') + c(s',s)\right) & \text{otherwise.} \end{cases} \quad (4)$$

A linear interpolation-based replanner expands vertices in a different way from D* Lite. With a consistent heuristic, a locally overconsistent vertex (whose *g*-value is lager than *rhs*-value) becomes locally consistent (the *g*-value equals *rhs*-value) after selected for expansion and then remains locally consistent until edge cost changes are detected in D* Lite. It implies that D* Lite expands any locally overconsistent vertex at most once. However, a linear interpolation-based replanner tends to expand a locally overconsistent vertex for many times. Besides, the key values (denoting the priorities of vertices in the priority queue) of the vertices selected for expansion are monotonically nondecreasing over time in D* Lite, while it is not naturally the case in a linear interpolation-based replanner. This can be explained as follows: For some vertex *s*, the computation of its *rhs*-value is based on the *g*-value of one neighboring vertex in D* Lite, but the *g*-values of two neighboring vertices in a linear interpolation-based replanner. During the planning and replanning process, it is common that at least one of the two neighboring vertices has an inaccurate *rhs*-value. Relied upon them, *s* will get an inaccurate *rhs*-value. And the vertices relied upon *s* will also be affected, and so on. It is the reason that a linear interpolation-based replanner updates the *rhs*-values of vertices repeatedly to their final results. Such a phenomenon is easily observed particularly in environment consisting mainly of free space since the *g*-value of a vertex is very close to those of its neighbors in free space.

After the cost field created, path extraction for linear interpolation-based replanners is also different from that for D* Lite. When backpointers are not recorded, D* Lite can trace back a lowest-cost path from s_{start} to s_{goal} by always moving from the current vertex *s*, starting at s_{start}, to any neighbor s' that minimizes $c(s',s) + g(s')$ until s_{goal} is reached. So the optimality of the result depends on the accuracy of $g(s')$. But in a linear interpolation-based replanner the *g*-values are not the goal distances exactly, resulting in the sub-optimality of paths.

From the discussion above, we can see that there exist two major drawbacks of Field D*: 1) It plans and replans much slower than D* Lite, especially in the environments consisting mainly of free space. 2) The solution path is not the optimal solution. **Figure 2** shows the second problem. The path returned by Field D* has unnecessary heading changes even if no obstacle exists (see **Figure 2(a)**). To extract a smoother path, [9] gives a gradient interpolation method. The result is showed in **Figure 2(b)**, from which we can see that the unnecessary heading changes still exit. Obstacles can also make the interpolation assumption break down so that affects the quality of extracted paths. To alleviate it, [8] uses a one-step look ahead mechanism. But this method checks very limited steps so that cannot avoid generating a pathologic path between vertices *a* and *b* in **Figure 2(c)**.

3. Update-Reducing Field D*

3.1. Basic Idea

Since the runtime of a linear interpolation-based replanner

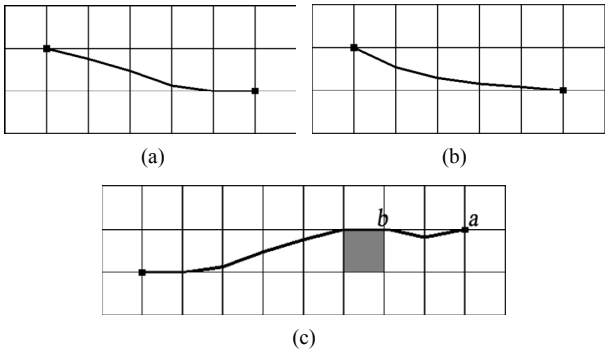

Figure 2. Paths returned by interpolation methods.

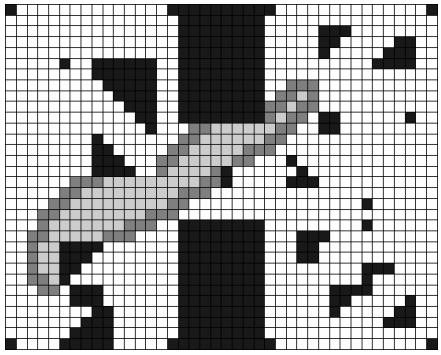

Figure 3. A snapshot during initial planning of Field D*.

depends heavily on the number of updates of the *rhs*-values of vertices, the key to high efficiency of our algorithm is reducing the number of updates. We use the fringe vertices to refer to the vertices on the fringe of the expanded vertices during initial planning. The fringe vertices have not been expanded yet so that their *g*-values are not computed (namely infinite), leading the *rhs*-values computed by the *g*-values of the fringe vertices to inaccuracy. Then the inaccurate *rhs*-values along with the infinite *g*-values affect next vertex expansions. This is the main source of the repeated updates of *rhs*-values. Note that most of the fringe vertices are just locally overconsistent vertices during initial planning, as is showed in **Figure 3**. (The locally consistent vertices (in light grey), which have been expanded at least once, are almost surrounded by the locally overconsistent vertices (in dark grey). Vertices in black are obstacles.) The *rhs*-values of locally overconsistent vertices are better informed and thus more accurate than the *g*-values. So when it is a locally overconsistent vertex, we can use the *rhs*-value instead of the *g*-value to make the computation more close to the g^*-value. When it is a locally consistent vertex, we can also use the *rhs*-value because it equals the *g*-value and thus could get a result at least no poorer than that computed by the *g*-value.

During replanning, if we only encounter cell cost decreases the approach above is still useful. However, when locally underconsistent vertices (whose *g*-values are smaller than *rhs*-values) appear, this approach tends to make the algorithm less efficient and even incomplete. It could be explained as follows: When the *rhs*-value of a vertex becomes larger due to edge cost increases, the old *rhs*-value is out of date and thus to be abandoned. However, the algorithm does not distinguish between the old and the new so that it is possible for the old *rhs*-value to be used to compute the *rhs*-value of another vertex, resulting in "false" relation between these two vertices. For example, there exist two vertices *a* and *b*. After initial planning, *g*(*a*), *rhs*(*a*) and *g*(*b*) are all infinite, *rhs*(*b*) is 50. Then *rhs*(*a*) and *rhs*(*b*) are updated due to cell cost increases. When *rhs*(*a*) is recomputed, old *rhs*(*b*) (namely 50) is used. Then *rhs*(*b*) is updated to a new value of 64. Thus, the computation of *rhs*(*a*) seems to rely on *rhs*(*b*) but in fact this relation possibly dose not exist. Furthermore the wrong *rhs*(*a*) leads to a priority error of *a* (*g*(*a*) is infinite so that does not affect the key value), which possibly makes *a* expand while *b* can never be expanded again. Thus the "false" relation has no chance to be corrected, resulting in the incompleteness of the algorithm.

In order to reduce the updates of the *rhs*-values during replanning, we use a technique similar to which Delayed D* used to speed up D* Lite [10], that is, delaying the processing of locally underconsistent vertices.

During path extraction, A* search, which depends on the accuracy of heuristic less heavily than greedy search does, could avoid errors caused by obstacles (see **Figure 2(c)**). However, based on the linear approximation, A* search still cannot ensure an optimal path even if no obstacles exist (see **Figure 2(b)**). And greedy search needs to be kept for checking solution paths for any loops. Note that the limitation of post-smoothing showed in **Figure 1** can be overcome if it is already an any-angle path before smoothing.

Combined with the methods above, Update-Reducing Field D* (URFD*) is a modified version of Field D*. It redefines the *rhs*-values (denoted by *rhs*'-values to be distinguished from the original) as

$$rhs'(s)=\begin{cases}0 & \text{if } s=s_{goal}\\ \min_{s'\in nbrs(s)}\left(rhs'(s')+c(s',s)\right) & \text{otherwise}\end{cases} \tag{5}$$

where notation follows from (1). URFD* calculates the *rhs*-values of vertices according to (5) during initial planning. During replanning it calculates the *rhs*-values according to (1), which is similar to Field D*, and delays the propagation of cost increases. It checks the consistency of a path in every path extraction and ends with a post-smoothing step.

3.2. Algorithm Description

Figure 4 shows the pseudocode of the URFD* algorithm.


```
Key(s)
  01. return [min(g(s), rhs(s)) + h(s_start, s); min(g(s), rhs(s))];

Initialize(s)
  02. for all s ∈ S  rhs(s) = g(s) = ∞;
  03. rhs(s_goal) = 0;   U = ∅;   raise = false;
  04. Insert(U, s_goal, Key(s_goal));

ComputeCost(s, s_a, s_b, cost)
  05. Use cost(s_a) and cost(s_b) to compute rhs(s);

UpdateState(s)
  06. if(g(s) ≠ rhs(s))
  07.   if(s ∈ U) Remove(U, s);
  08.   Insert(U, s, Key(s));
  09. else if(g(s) = rhs(s) AND s ∈ U) Remove(U, s);

UpdateStateLower(s)
  10. if(g(s) > rhs(s))
  11.   if(s ∈ U) Remove(U, s);
  12.   Insert(U, s, Key(s));
  13. else if(g(s) = rhs(s) AND s ∈ U) Remove(U, s);

ComputeState(s)
  14. if(s ≠ s_goal)
  15.   if(it is inital planning) rhs(s) = min_(s',s")∈connbrs(s) ComputeCost(s, s', s", rhs);
  16.   else rhs(s) = min_(s',s")∈connbrs(s) ComputeCost(s, s', s", g);

ComputeShortestPath()
  17. while(min_(s∈U)(Key(s)) < Key(s_start)) OR rhs(s_start) ≠ g(s_start))
  18.   s = U.Top();
  19.   if(g(s) > rhs(s))
  20.     g(s) = rhs(s);
  21.     Remove(U, s);
  22.     for all s' ∈ nbrs(s)
  23.       ComputeState(s');  UpdateStateLower(s');
  24.   else
  25.     g(s) = ∞;
  26.     for all s' ∈ nbrs(s) ∪ {s}
  27.       ComputeState(s');  UpdateState(s');

FindRaiseStatesOnPath()
  28. raise = false;   s = s_start;   ctr = 0;   loop = false;
  29. while(s ≠ s_goal AND loop = false AND ctr < maxsteps)
  30.   x = arg min_(s'∈succ(s)) (c(s, s') + g(s'));
  31.   for all s' ∈ vicinity(s_c)
  32.     if(s' is locally inconsistent AND s' has never been added
            into U with local underconsistency during this replanning episode)
  33.       ComputeState(s');  UpdateState(s');  raise = true;
  34.   if(s is visited before) loop = true;
  35    s = x;   ctr++;

Main()
  36. Initialize();
  37. ComputeShortestPath();
  38. Path extraction;
  39. Post-Smoothing;
  40. forever
  41.   Wait for changes in cell costs;
  42.   for all cells x with changed costs
  43.     for each state s on a corner of x
  44.       ComputeState(s);  UpdateStateLower(s);
  45.   ComputeState(s_start);  UpdateState(s_start);
  46.   ComputeShortestPath();  FindRaiseStatesOnPath();.
  47.   while(raise)
  48.     ComputeShortestPath();  FindRaiseStatesOnPath();.
  49.   Post-Smoothing;
```

Figure 4. The update-reducing Field D* algorithm.

During initial planning, URFD* calls ComputeShortestPath() to expand vertices. ComputeCost() calculates the *rhs*-value in a way similar to the interpolation-based path cost calculation in Field D*, but every vertex uses *rhs*-values, instead of *g*-values, of its neighbors. ComputeState() then computes the *rhs*-values according to (5) (line 15). During replanning, ComputeState() calls ComputeCost() to compute the *rhs*-values according to (1) (line 16). The *rhs*-values of the start vertex and every vertex immediately affected by the changed edge costs are updated, but only the locally inconsistent start vertex and locally overconsistent vertices are inserted into priority queue *U* for expansion (lines 42 - 45). Then FindRaiseStatesOnPath() (FRSOP) is called. FROSP checks whether locally underconsistent vertices are in the vicinity of the node. All the unprocessed locally underconsistent vertices that are adjacent to this node will be added into priority queue *U* (lines 31 - 33). Here *vicinity*(*s*) refers to the set of all corner vertices in the vicinity of node *s* (*s* is included). When the number of nodes exceeds the given limit *maxsteps*, or a loop is found, which indicates a potential failure of path extraction, FRSOP stops the extraction to expand locally underconsistent vertices in priority queue *U*. After path extraction, the solution path is post-processed by a smoothing step. (lines 39, 49). Given two cell boundary nodes along the path, the post-smoothing replaces the solution path between these two nodes with a straight line path if the latter is less costly. It is done with cell boundary nodes along the solution path iteratively. However, some small techniques are used to avoid a large amount of computation: 1) It only performs a single iteration. 2) It only smoothes the path between cell corners because necessary and sharp heading changes usually occur on them. 3) Before smoothing a path between two cell corners, it checks whether the costs of all grid cells that the original path is through are all same. If they are, the original path is kept.

4. Experimental Results

We compared the performance of URFD*, Field D* and Delayed D*. 800 different 500 × 500 random grid environments were generated: 400 environments with uniform cost grids and 400 with non-uniform cost grids. For uniform cost grid environments, four different initial percentages of obstacle cells were selected: 10%, 20%, 30% and 40%. For non-uniform cost grid environments, we assigned each traversable cell an integer cost between 1 (free space) and 15, and four different initial percentages of free space cells were selected: 90%, 70%, 50% and 30%, while the rest of the cells each got a cost (infinity or an integer between 2 and 15) randomly with the same probability (namely 1/15). For each environment, the initial task was to plan a path from the lower left corner to a randomly selected goal on the right edge. After that, we altered the costs of cells close to the agent with probability 0.1 (1.6% of the cells in the environment were changed) and had each approach repair its solution

Table 1. Performance comparison among URFD*, Field D* and Delayed D* in three kinds of environments.

		Path Cost		*Rhs*-values Updates		Vertex Expansions		Runtime	
		Initial	***Replan***	***Initial***	***Replan***	***Initial***	***Replan***	***Initial***	***Replan***
A	Field D*	0.9538	0.9544	6.1560	64.1851	5.2317	32.3386	5.7784	58.7625
	URFD*	0.9510	0.9519	2.5438	28.5094	2.2528	14.7824	2.3987	21.0833
B	Field D*	0.9619	0.9618	2.4143	10.7327	1.8484	8.9259	2.1848	13.0324
	URFD*	0.9568	0.9567	1.4702	4.9103	1.4780	4.2553	1.8199	5.1806
C	Field D*	0.9639	0.9643	1.3155	1.2895	1.0600	0.9707	1.2174	4.9793
	URFD*	0.9592	0.9596	0.8337	1.4234	0.9218	1.0877	0.7672	2.3122

path. We use the weighted heuristic, which is described in the previous section, for URFD* and Field D*.

We selected the results in three kinds of environments (A: uniform cost grids with 10% obstacle cells. B: uniform cost grids with 30% obstacle cells. C: non-uniform cost grids with 50% free space cells) and showed them in **Table 1**. Four performance measures were used here: the path cost, the total number of *rhs*-value updates (that is, updates of the *rhs*-values), the total number of vertex expansions (updates of the *g*-values) and the runtime. Each value is a ratio of a performance measure of URFD* (or Field D*) to that of Delayed D* averaged over initial planning (or replanning) episodes. Note that in environments with more free space the runtimes of initial planning and replanning of Field D* drastically increased while those of URFD* increased much more stably. The performance in environments C shows the possibility that the number of updates of the *rhs*-values during replanning could be slightly larger than that of Field D* in some scenarios. However, since the number of vertices in the priority queue is limited by selectively processing locally underconsistent vertices, making the priority queue operations less expensive, the runtime of URFD* is still shorter than that of Field D* in those scenarios.

5. Conclusion

We present URFD*, a linear interpolation-based algorithm that plans and replans any-angle paths in dynamic environments with uniform and non-uniform cost grids. It makes efforts in the reduction of updates of the *rhs*-values, which contributes to the gain in efficiency. The solution paths returned by URFD* are smooth and near-optimal. As opposed to Field D*, it performs faster planning and replanning and returns a path with lower cost and fewer heading changes. However, URFD* is not optimal either due to the linear interpolation assumption.

REFERENCES

[1] D. Ferguson, M. Likhachev and A. Stentz, "A Guide to Heuristic-Based Path Planning," *Proceedings of the Workshop on Planning under Uncertainty for Autonomous Systems at the International Conference on Automated Planning and Scheduling*, Monterey, 5-10 June 2005, pp. 9-18.

[2] P. Hart, N. Nilsson and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, 1968, Vol. 4, No. 2, pp. 100-107.

[3] S. Koenig, M. Likhachev and D. Furcy, "Lifelong Planning A*," *Artificial Intelligence Journal*, Vol. 155, No. 1-2, 2004, pp. 93-146.

[4] S. Koenig and M. Likhachev, "Improved Fast Replanning for Robot Navigation in Unknown Terrain," *Proceedings of the IEEE International Conference on Robotics and Automation* (*ICRA* 2002), Washington, 11-15 May 2002, pp. 968-975.

[5] A. Botea, M. Müller and J. Schaeffer, "Near Optimal Hierarchical Path-Finding," *Journal of Game Development*, 2004, Vol. 1, No. 1, pp. 1-22.

[6] A. Nash, K. Daniel, S. Koenig and A. Felner, "Theta*: Any-Angle Path Planning on Grids," *Proceedings of the National Conference on Artificial Intelligence*, 22-26 July 2007, Vancouver, pp. 1177-1183.

[7] D. Šišlák, P. Volf and M. Pěchouček, "Accelerated A* Path Planning," Springer-Verlag, Berlin, 2009.

[8] D. Ferguson and A. Stentz, "The Field D* Algorithm for Improved Path Planning and Replanning in Uniform and Non-Uniform Cost Environments," Technical Report, Carnegie Mellon University, Pittsburgh, 2005.

[9] M. W. Otte and G. Grudic, "Extracting Paths from Fields Built with Linear Interpolation," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, St. Louis, 10-15 October 2009, pp. 4406-4413.

[10] D. Ferguson and A. Stentz, "The Delayed D* Algorithm for Efficient Path Replanning," *Proceedings of the IEEE International Conference on Robotics and Automation*, Barcelona, 18-22 April 2005, pp. 2045-2050.

