

How do the state-of-the-art pathfinding algorithms for changing graphs (D*, D*-Lite, LPA*, etc) differ?

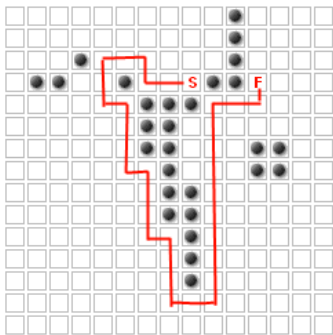
A lot of pathfinding algorithms have been developed in recent years which can calculate the best path in response to graph changes much faster than A* - what are they, and how do they differ? Are they for different situations, or do some obsolete others?

These are the ones I've been able to find so far:

- D* (1994)
- Focused D* (1995)
- DynamicSWSF-FP (1996)
- LPA (1997)
- LPA*/Incremental A* (2001)
- D* Lite (2002)
- SetA* (2002)
- HPA* (2004)
- Anytime D* (2005)
- PRA* (2005)
- Field D* (2007)
- Theta* (2007)
- HAA* (2008)
- GAA* (2008)
- LEARCH (2009)
- BDDD* (2009 - I cannot access this paper :|)
- Incremental Phi* (2009)
- GFRA* (2010)
- MTD*-Lite (2010)
- Tree-AA* (2011)

I'm not sure which of these apply to my specific problem - I'll read them all if necessary, but it would save me **a lot** of time if someone could write up a summary.

My specific problem: I have a grid with a start, a finish, and some walls. I'm currently using A* to find the best path from the start to the finish.



The user will then **move one wall**, and I have to recalculate the entire path again. The *"move-wall/recalculate-path"* step happens many times in a row, so I'm looking for an algorithm that will be able to quickly recalculate the best path without having to run a full iteration of A*.

Though, I am not necessarily looking for an alteration to A* - it could be a completely separate algorithm.

ds.algorithms shortest-path

edited Jul 13 '12 at 20:54

community wiki
21 revs, 2 users 100%
BlueRaja - Danny Pflughoeft

-
- 7 @Kaveh: Why isn't asking for an overview of the state-of-the-art in dynamic pathfinding algorithms "research-level?" Field-D* is less than 5 years old, and LEARCH is less than 3.. — [BlueRaja - Danny Pflughoeft](#) Jun 28 '12 at 4:38
-
- 7 I'm not sure why this isn't an appropriate question for this forum. this is by no means a settled and old topic — [Suresh Venkat](#) ♦ Jun 28 '12 at 5:20
-
- 5 @BlueRaja-DannyPflughoeft I also think this is a good, research-level question. I'll add an answer based on my comment, and expand it a little bit. — [Juho](#) Jun 28 '12 at 8:00
-
- 4 @BlueRaja-DannyPflughoeft Got linked on reddit. — [U2EF1](#) Feb 24 '13 at 11:55
-

6 Answers

So, I skimmed through the papers, and this is what I gleamed. If there is anyone more knowledgeable in the subject-matter, please correct me if I'm wrong (*or add your own answer, and I will accept it instead!*).

Links to each paper can be found in the question-post, above.

• Simple recalculations

- **D* (aka *Dynamic A**)** (1994): On the initial run, D* runs very similarly to A*, finding the best path from start to finish very quickly. However, as the unit moves from start to finish, if the graph changes, D* is able to very quickly recalculate the best path from that unit's position to the finish, much faster than simply running A* from that unit's position again. D*, however, has a reputation for being extremely complex, and has been completely obsoleted by the much simpler D*-Lite.
- **Focused D*** (1995): An improvement to D* to make it faster/"more realtime." I can't find any comparisons to D*-Lite, but given that this is older and D*-Lite is talked about a lot more, I assume that D*-Lite is somehow better.
- **DynamicSWSF-FP** (1996): Stores the distance from every node to the finish-node. Has a large initial setup to calculate all the distances. After changes to the graph, it's able to update only the nodes whose distances have changed. Unrelated to both A* and D*. Useful when you want to find the distance from multiple nodes to the finish after each change; otherwise, LPA* or D*-Lite are typically more useful.
- **LPA*/Incremental A*** (2001): LPA* (*Lifelong Planning A**), also known as Incremental A* (*and sometimes, confusingly, as "LPA," though it has no relation to the other algorithm named LPA*) is a combination of DynamicSWSF-FP and A*. On the first run, it is exactly the same as A*. After minor changes to the graph, however, subsequent searches from the same start/finish pair are able to use the information from previous runs to drastically reduce the number of nodes which need to be examined, compared to A*. This is exactly my problem, so it sounds like LPA* will be my best fit. LPA* differs from D* in that it always finds the best path from the same start to the same finish; it is not used when the start point is moving (*such as units moving along the initial best path*). However...
- **D*-Lite** (2002): This algorithm uses LPA* to mimic D*; that is, it uses LPA* to find the new best path for a unit as it moves along the initial best path and the graph changes. D*-Lite is considered much simpler than D*, and since it always runs *at least* as fast as D*, it has completely obsoleted D*. Thus, there is never any reason to use D*; use D*-Lite instead.

• Any-angle movement

- **Field D*** (2007): A variant of D*-Lite which does not constrain movement to a grid; that is, the best path can have the unit moving along any angle, not just 45- (or 90-)degrees between grid-points. Was used by NASA to pathfind for the Mars rovers.
- **Theta*** (2007): A variant of A* that gives better (shorter) paths than Field D*. However, because it is based on A* rather than D*-Lite, it does not have the fast-replanning capabilities that Field D* does. *See also.*
- **Incremental Phi*** (2009): The best of both worlds. A version of Theta* that is incremental (*aka allows fast-replanning*)

• Moving Target Points

- **GAA*** (2008): GAA* (*Generalized Adaptive A**) is a variant of A* that handles moving target points. It's a generalization of an even earlier algorithm called "Moving Target Adaptive A*"
- **GRFA*** (2010): GFRA* (*Generalized Fringe-Retrieving A**) appears (?) to be a generalization of GAA* to arbitrary graphs (*ie. not restricted to 2D*) using techniques from another algorithm called FRA*.
- **MTD*-Lite** (2010): MTD*-Lite (*Moving Target D*-Lite*) is "an extension of D* Lite that uses the principle behind Generalized Fringe-Retrieving A*" to do fast-replanning

moving-target searches.

- **Tree-AA*** (2011): (???) Appears to be an algorithm for searching unknown terrain, but is based on Adaptive A*, like all other algorithms in this section, so I put it here. Not sure how it compares to the others in this section.

- **Fast/Sub-optimal**

- **Anytime D*** (2005): This is an "Anytime" variant of D*-Lite, done by combining D*-Lite with an algorithm called *Anytime Repairing A**. An "Anytime" algorithm is one which can run under any time constraints - it will find a very suboptimal path very quickly to begin with, then improve upon that path the more time it is given.
- **HPA*** (2004): HPA* (*Hierarchical Path-Finding A**) is for path-finding a large number of units on a large graph, such as in RTS (*real-time strategy*) video games. They will all have different starting locations, and potentially different ending locations. HPA* breaks the graph into a hierarchy in order to quickly find "near-optimal" paths for all these units much more quickly than running A* on each of them individually. See also
- **PRA*** (2005): From what I understand, PRA* (*Partial Refinement A**) solves the same problem as HPA*, but in a different way. They both have "similar performance characteristics."
- **HAA*** (2008): HAA* (*Hierarchical Annotated A**) is a generalization of HPA* that allows for restricted traversal of some units over some terrains (ex. a small pathway that some units can walk through but larger ones can't; or a hole that only flying units can cross; etc.)

- **Other/Unknown**

- **LPA** (1997): LPA (*Loop-free path-finding algorithm*) appears to be a **routing-algorithm** only marginally related to the problems the other algorithms here solve. I only mention it because this paper is confusingly (and incorrectly) referenced on several places on the Internet as the paper introducing LPA*, which it is not.
- **LEARCH** (2009): LEARCH is a combination of machine-learning algorithms, used to teach robots how to find near-optimal paths on their own. The authors suggest combining LEARCH with Field D* for better results.
- **BDDD*** (2009): ??? I cannot access the paper.
- **SetA*** (2002): ??? This is, apparently, a variant of A* that searches over the "binary decision diagram" (BDD) model of the graph? They claim that it runs "several orders of magnitude faster than A*" in some cases. However, if I'm understanding correctly, those cases are when each node on the graph has many edges?

Given all this, it appears that **LPA*** is the best fit for my problem.

edited Jul 13 '12 at 21:05

answered Jun 28 '12 at 16:20



BlueRaja - Danny
Pflughoeft

1,173 2 10 19

Well.. I also found [this paper](#) by @lhrios which compares some of the algorithms. – mg007 Jun 8 '13 at 17:57

I know this is old, but I think it's worth noting a small flaw in your description of Field D*. Regular D* isn't constrained to "a grid", it's constrained to a discrete graph. The point the paper makes is that in order to make A*, D* etc work you have to discretize a continuous space into chunks, which limits the angles you can move at. Field D* eliminates that constraint and allows you treat successor states in a continuous, rather than discrete manner (more or less, trickery is involved). It simply uses a 2D grid as an example, D*/A* etc are by no means constrained to a grid. – LinearZoetrope Dec 8 '13 at 8:36

I should mention that Field D* is limited to a grid, though the paper mentions that they did work on a 3D version. This is due to the interpolation it uses. It still stands that A* and D* work on graphs with arbitrary numbers of successor states, Field D* is just an improvement for scenarios where D* uses grid-based planning. – LinearZoetrope Dec 8 '13 at 8:47

An important difference between Field D* and Theta*/Incremental Phi*, is that Field D* can have unique weights for each square, whereas Theta* and Incremental Phi* are limited to having the same weight for all squares that can be visited. Hence, Incremental Phi* is not superior to Field D*. – HelloGoodbye May 7 '15 at 12:33

1 @Jsor: Here is a 3D version of Field D*: [3D Field D - JPL Robotics](#) – HelloGoodbye May 7 '15 at 12:49

There's a big caveat when using D*, D*-Lite, or any of the incremental algorithms in this category (and it's worth noting that this caveat is seldom mentioned in the literature). These types of algorithms use a reversed search. That is, they compute costs outwards from the goal node, like a ripple spreading outwards. When the costs of edges change (e.g. you add or remove a wall in your example) they all have various efficient strategies for only updating the subset of the explored (a.k.a. 'visited') nodes that is affected by the changes.

The big caveat is that the location of these changes with respect to the goal location makes an enormous difference to the efficiency of the algorithms. I showed in various papers and my thesis that it's entirely possible for the worst case performance of any of these incremental

algorithms to be worse than throwing away all the information and starting afresh with something non-incremental like plain old A*.

When the changed cost information is close to the perimeter of the expanding search front (the 'visited' region), few paths have to change, and the incremental updates are fast. A pertinent example is a mobile robot with sensors attached to its body. The sensors only see the world near the robot, and hence the changes are in this region. This region is the starting point of the search, not the goal, and so everything works out well and the algorithms are very efficient at updating the optimum path to correct for the changes.

When the changed cost information is close to the goal of the search (or your scenario sees the goal change locations, not just the start), these algorithms suffer catastrophic slowdown. In this scenario, almost all the saved information needs to be updated, because the changed region is so close to the goal that almost all pre-calculated paths pass through the changes and must be re-evaluated. Due to the overhead of storing extra information and calculations to do incremental updates, a re-evaluation on this scale is slower than a fresh start.

Since your example scenario appears to let the user move any wall they desire, you will suffer this problem if you use D*, D*-Lite, LPA*, etc. The time-performance of your algorithm will be variable, dependent upon user input. In general, "this is a bad thing"...

As an example, Alonzo Kelly's group at CMU had a fantastic program called PerceptOR which tried to combine ground robots with aerial robots, all sharing perception information in real-time. When they tried to use a helicopter to provide real-time cost updates to the planning system of a ground vehicle, they hit upon this problem because the helicopter could fly ahead of the ground vehicle, seeing cost changes closer to the goal, and thus slowing down their algorithms. Did they discuss this interesting observation? No. In the end, the best they managed was to have the helicopter fly directly overhead of the ground vehicle - making it the world's most expensive sensor mast. Sure, I'm being petty. But it's a big problem that no one wants to talk about - and they should, because it can totally ruin your ability to use these algorithms if your scenario has these properties.

There are only a handful of papers that discuss this, mostly by me. Of papers written by the authors or students of the authors of the original papers listed in this question, I can think of only one that actually mentions this problem. Likhachev and Ferguson suggest trying to estimate the scale of updates required, and flushing the stored information if the incremental update is estimated to take longer than a fresh start. This is a pretty sensible workaround, but there are others too. My PhD generalizes a similar approach across a broad range of computational problems and is getting beyond the scope of this question, however you may find the references useful since it has a thorough overview of most of these algorithms and more. See http://db.acfr.usyd.edu.au/download.php/Allen2011_Thesis.pdf?id=2364 for details.

answered Feb 23 '13 at 23:11

community wiki
Schwolop

1 Thank you for adding these details :) In my application, the wall gets moved towards the beginning just as often as towards the end. I implemented BFS, A*, and LPA*; A* was actually slightly slower than BFS (*my spaces tend to be confined, so A* only searches slightly fewer nodes than BFS; meanwhile BFS only needs a queue, which can be implemented to be faster than a priority-queue can*), but using LPA* averaged to be over twice as fast. – BlueRaja - Danny Pflughoeft Feb 24 '13 at 1:44

The main idea is to use an incremental algorithm, that is able to take advantage of the previous calculations when the initial calculated route gets blocked. This is often investigated in the context of robots, navigation and planning.

Koenig & Likhachev, *Fast replanning for Navigation in Unknown Terrain*, IEEE Transactions on Robotics, Vol. 21, No. 3, June 2005 introduces D* Lite. It seems safe to say that D* is outdated in a sense that D* Lite is always as fast as D*. In addition, D* is complex, hard to understand, analyze and extend. Figure 9 gives the pseudocode for D* Lite, and Table 1 shows experimental results with D* Lite compared to BFS, Backward A*, Forward A*, DynamicSWSF-P and D*.

I do not know the newer algorithms you list (Anytime D*, Field D*, LEARCH). Very recently I saw a robot that used D* Lite for planning in an environment with random walkers in it. In this sense, I don't think D* Lite is outdated by no means. For your practical problem, I guess there's no harm in trying the usual engineering way: take some approach, and if it doesn't fit your needs, try something else (more complex).

answered Jun 28 '12 at 8:14



Juho

2,581

2 18 31

I'd like add something about rapidly-exploring randomized trees, or RRTs. The basic idea has good discussion all over the internet, but it's probably safe to start with the links off the Wikipedia page and with Kuffner and LaValle's original papers on the topic.

The most important feature of RRTs is that they can deal with real-valued spaces of *extremely* high dimension without choking. They can handle dynamics, are not optimal but are probabilistically complete (guaranteed to succeed if possible as computation time goes to infinity), and are capable of handling moving targets. There are some extensions that let them work in non-static spaces, the best of which looks to be the Multipartite RRT work that I've linked below.

- http://en.wikipedia.org/wiki/Rapidly_exploring_random_tree
- http://swing.adm.ri.cmu.edu/pub_files/pub4/zucker_matthew_2007_1/zucker_matthew_2007_1.pdf

answered Feb 24 '13 at 5:27

community wiki

Saul Reynolds-Haertle

Data structures called ***distance oracles*** handle such problems. However, most research results are for *static* graphs only.

If the graphs are grids (and thus planar), some *dynamic* data structures exist (unclear whether the constants are small enough for the application in question):

Exact shortest paths:

Jittat Fakcharoenphol, Satish Rao: Planar graphs, negative weight edges, shortest paths, and near linear time. J. Comput. Syst. Sci. 72(5): 868-889 (2006)

Approximate shortest paths:

Philip N. Klein, Sairam Subramanian: A Fully Dynamic Approximation Scheme for Shortest Paths in Planar Graphs. Algorithmica 22(3): 235-249 (1998)

Ittai Abraham, Shiri Chechik, Cyril Gavoille: Fully dynamic approximate distance oracles for planar graphs via forbidden-set distance labels. STOC 2012: 1199-1218

answered Feb 24 '13 at 1:45

community wiki

Christian Sommer

Sorry, but if they only work on static graphs, then what do you mean by "they handle such problems?" The problem asked is *specifically* about non-static graphs. – BlueRaja - Danny Pflughoeft Feb 24 '13 at 1:47

let me change the emphasis: *most* results are for static graphs only. *some* dynamic data structures exist. followed by a list of those dynamic data structures. – Christian Sommer Mar 18 '13 at 4:30

In school I dabbled with ant colony optimization. In some texts it was touted as a solution for continuously changing graphs, routing networks, etc. It's not an engineered solution, but an adaptation from how ants navigate around obstacles by spreading pheromones to mark good and bad paths. I have no idea if it works for your problem, but i think it's an interesting point of view.

edited Feb 25 '13 at 18:29

community wiki

2 revs, 2 users 67%
silversplinter
