Pathfinding: From A* to LPA

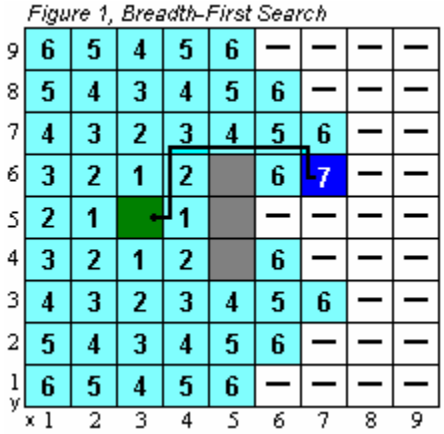
Kelly Manley
University of Minnesota
705 Imperial Dr. Apt. 102
Morris, MN 56267
1-320-589-2953
manl0026@mrs.umn.edu

Abstract
This paper discusses the movement from one-time computation algorithms like A* [5] to incremental search algorithms such as Lifelong Planning A* (LPA*) [1] and their relation to pathfinding. Starting with breath-first search, the most basic form of method used to find the shortest path between two points. Then to reduce the amount of computation required from a breadth-first search, using the A* algorithm. However, A* must reconstruct a new path every time the state of the environment changes. To combat this, A* was modified from a replan strategy to a reuse strategy using an incremental search algorithm named DynamicSWSF-FP. [4] This algorithm reconstructs only the areas affected by the changes to the environment's state. However, alone DynamicSWSF-FP is of not much use in pathfinding. If combined with A*, the result is an algorithm that can find the shortest path in less time then either of them could do on their own. This algorithm is named Lifelong Planning A*.

Keywords
Pathfinding, Breadth-First Search, A*, and LPA*, DynamicSWSF-FP.

1 Introduction
Pathfinding is the ability to find the shortest path between two points. It is a cornerstone of the artificial intelligence in fields such as interactive entertainment and robotics. There are several key elements that exist is most forms of pathfinding. These elements range from where to start and where to end. Also some way to represent an environment in which to roam is needed. This paper uses a grid of cells that which can be seen in Figure 1, to represent its environment. The start position is the cell in green, the goal cell is in blue, and non-traversable cells are in gray. The teal colored cells represent cells that had been computed or are currently being computed in order to locate the goal cell. Cells containing dashes are cells that did not have to be computed. Another key element found in pathfinding environment is movement cost. For example, if the gridworld was to represent terrain. A cost value of one could be given to a cell in order to represent flat terrain and the cost of a five could be given to a cell to represent an elevated terrain. All non-traversable cells are given an infinite cost. The cost to move from one cell to another in this paper is one. In this paper only horizontal and vertical moves are allowed. In several instances there are ties between two cost values. The solution to this problem is to simply set a bias. Throughout this paper ties are broken by a bias to horizontal movements to that of vertical movements. Between vertical movements the bias is to move upward and to move right in horizontal movements.

2 Breadth-First Search Algorithm
Figure 1, Breadth-First Search



Shown above is a 9x9 gridworld of cells. The numbers within each cell represent the cost to move from the start cell to itself. This gridworld is used to help explain how the breadth-first search algorithm obtains the shortest path.

Breadth-first search works simply by branching out from the start cell until the goal cell is located. The search first begins by evaluating the cost of the cells adjacent to the start cell. The cell(s) with the lowest cost get(s) computed first. Next it recursively computes the cell(s) with the lowest costs adjacent to itself and so on until the goal cell is computed. In Figure 1 the cost to get from the start cell to the goal cell is 7. This means all cells with a value less then 7 have to be computed in order to reach the goal cell. Once the goal cell has been located a path from the start cell to the goal cell is created. The dark line represents the path in the gridworld. To create this path, simply backtrack from the goal cell to the start cell. However, as the size of the path

grows so did the amount of computation required. In order to ease these requirements a heuristic search method called A* was used.

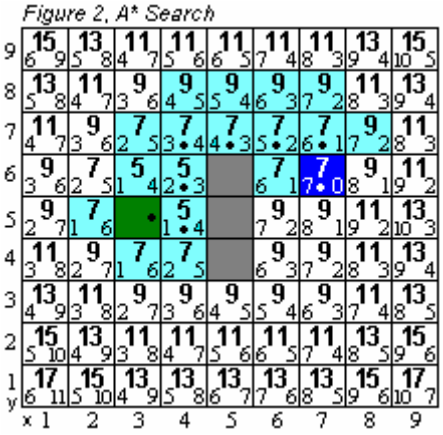
3 A-Star Search

Traditionally, pathfinding research in artificial intelligence has focused on one-time computation of paths. The leading algorithm in one-time computation is A*. Even today A* is still the most used algorithm for pathfinding in interactive entertainment. Unlike the breadth-first search algorithm, A* only has to compute the cells it comes in contact with during the search.

The equation "F = G + H" is a key element of A*. The variable G represents the cost of moving from the start cell to another cell on the grid. H is the estimated cost to arrive at the goal cell. To obtain this estimation a heuristic is used. For the purpose of this paper the Manhattan method is applied. [5] The Manhattan method uses the following equation to calculate H:

H = abs (current X – target X) + abs (current Y – target Y)

Figure 2, A Search*



An example of A* search algorithm can be seen in Figure 2. The numbers within each cell represent the cell's F, G, and H values. The top number represents F, the lower left number represents G, and the lower right number represents H. Since F is the sum of G and H, its value is used to traverse the grid to arrive at the goal cell. The dots depict the shortest path found between the start and goal cells.

The structure of A* is that of two lists: Open and Closed. The Open list is a list of all cells that have yet to be saved. The Close contains the list of all cells that have been saved. Once the goal is located the shortest path can easily be obtained. This is done by traversing backwards, using G values, from the goal cell to the start cell.

3.1 A* Algorithm

The algorithm for A* is as follows:

1. Add starting cell to open list.
    a. Set starting cell to current cell.
    b. Add starting cell to closed list.
2. Add the adjacent cells of the current cell to the open list if:
    a. The adjacent cell is traversable.
    b. If adjacent cell currently doesn't exist within the open list then set adjacent cell to be the child of the current cell and save the child's F, G, and H cost values.
    c. If adjacent cell currently exists in open list then compare the old cell's G value to the new cell's G value. If the G value of the new cell is lower then the old, change the parent of the adjacent cell to the current cell and recalculate the G and F values of the adjacent cell.
3. Place the current cell into closed list.
4. Find and set the cell with the lowest F value to current cell and return to step 2 until:
    a. The goal cell is added to the open list.
    b. The open list becomes empty, or in other words, no path to the goal cell exists.
5. Upon adding the goal cell to the open list. Save the path by going from the goal cell back to the start cell.

3.2 Issues with A*

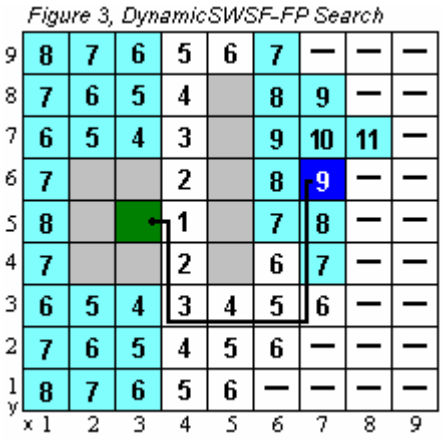
Finding the adjacent cell with the lowest F cost is time consuming in large paths. There are several ways to reduce this consumption. One solution is to sort the F values. Sorting using a binary heap structure has the potential to reduce the amount of computation greatly.

A* is well suited for programs in which the state changes at an acceptable rate, so recreating the path at every state change isn't a serious problem. However, if placed in an environment where the state can change quickly, having to replan with every modification to state becomes unacceptable. A solution to this problem is to change A* from a replan strategy to a reuse strategy.

4 DynamicSWSF-FP

DynamicSWSF-FP stands for "Dynamic Strict Weakly Superior Function – Fixed Point Problem". Since DynamicSWSF-FP builds off of previous searches, the breadth-first gridworld will be used to depict the original state of the gridworld.

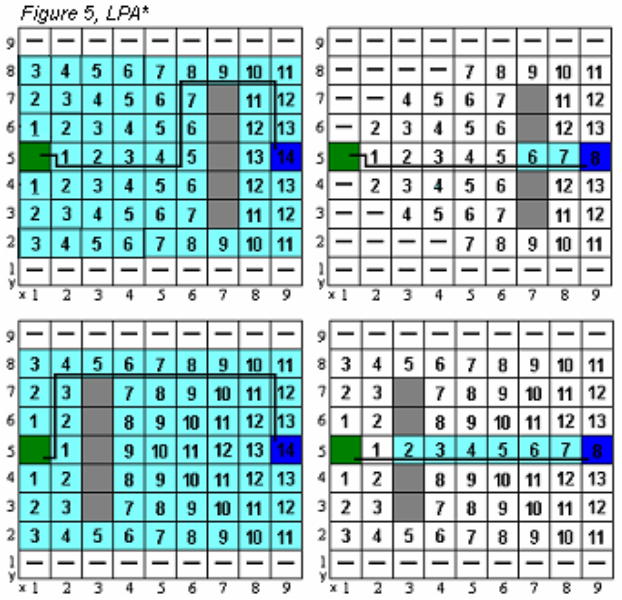
Figure 3, DynamicSWSF-FP Search



The above gridworld is an example of DynamicSWSF-FP after the state of the gridworld changed. Cells (4, 2), (4, 3), (5, 2), (6, 2), (7, 5), and (8, 5) have become non-traversable. Because of this change the path must be recomputed. All cells that have been affected must be recomputed. More in-depth information on DynamicSWSF-FP can be found in [4] paper.

5 Lifelong Planning A*

Figure 5, LPA*

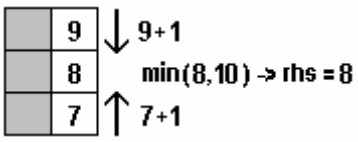


Incremental search methods are not commonly used in artificial intelligence. The combination of the heuristic search method A* and an incremental search method similar to that of DynamicSWSF-FP has been found to decrease computation while locating the shortest path. Instead of starting from scratch every time a path must be located, as in A*, it reuses data found in previous searches to save on computation time. This algorithm was given the name Lifelong Planning A* because of its ability to reuse data from previous searches. Initially LPA* constructs a path to the goal cell that is identical to that of A*. After the initial path has been stored, any future paths can refer to it in order to reduce the amount of computations required to compute a new path.

Figure 5 illustrates this ability. The upper left gridworld represents the initial search. In the upper right gridworld cell (5, 7) has become traversable. This resulted in only cells (5, 8) and (5, 9) having to be recomputed, while all of row 5 would have to be recomputed using A*. The two lower gridworlds are there to show a possible problem in using LPA*. This case being if the changes made to the gridworld are close to the start cell. It often results in the majority of the previous cells having to be recomputed. In some cases LPA* may even take longer then A* in finding the new path.

5.1 RHS Values

A new variable called rhs-values is introduced in order to evaluate changes to cost values. To obtain a cell's rhs value, first find the minimum g-value between the cells adjacent to the current cell. Then take that value and add the movement cost of that cell to it. The figure below should help explain it more clearly.



Using the rhs value, the consistency of a cell can be determined. In the example above the G value is equal to the rhs value. This means the cell is locally consistent and recompilation isn't necessary. However if the G value is not equal to the rhs value then it is deemed locally inconsistent and may need to be recomputed. If the G value is overconsistent or greater then the rhs-value, the cell can be made locally consistent by setting the G value equal to the rhs-value. If the G value is underconsistent or less then the rhs value, then the cells G value is set to infinity. This can either result in the cell becoming locally consistent, or it can turn it into locally overconsistent. An example of it becoming locally consistent is if non-traversable cells surrounded the cell. These effects can be seen in Figure 6, which is based on figures contained within reference [3]. Figure 6 shows a step-by-step reconstruction of a path using the LPA* algorithm. The first gridworld shows the initial path located. In the next gridworld cell B3 has become non-traversable. This triggers the path to be reconstructed. The first cell to be checked for consistency is C3. Upon evaluation C3 is determined to be locally underconsistent. Because of this its G value is set to infinity and its values [4; 2] are added to the priority queue. These values are found by applying the following function to the cell.

[Min (G, rhs) + H); min (G, rhs))]

Since C3's G value is now set to infinity cell D3 now needs to be evaluated. The same steps as in C3 are taken to evaluate D3. This continues until the goal state becomes locally consistent once again as seen in Figure 6.

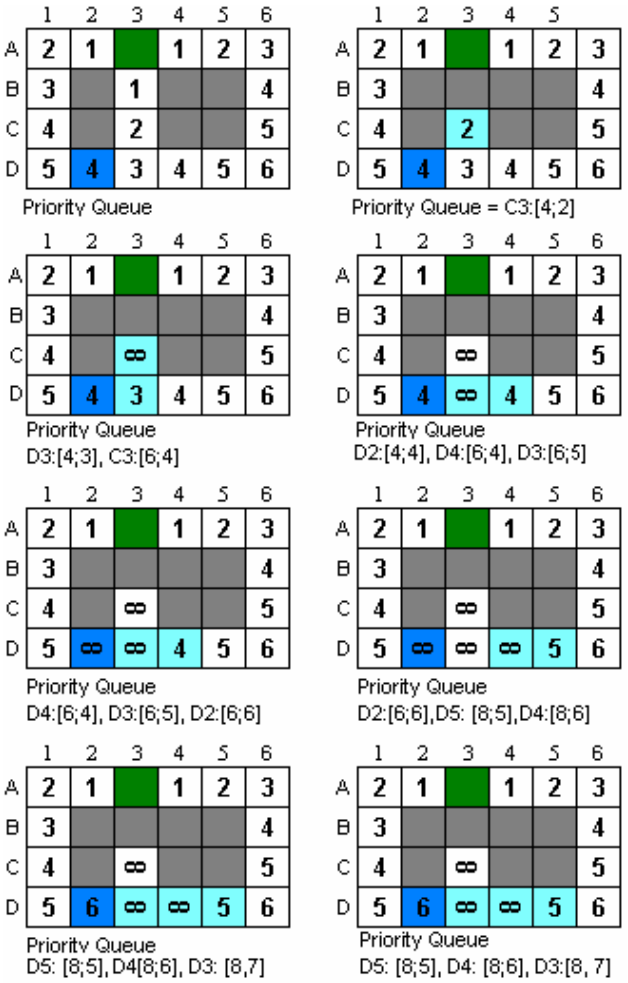


Figure 6, Priority Queue

5.2 LPA* Search Algorithm

First step is to set the G values of every cell to infinity. Next set all the cells rhs values using equations 3 and 4 above. This makes the start cell locally inconsistent and adds it to the priority queue. These steps are necessary in order to construct an initial search identical to that of A* when ComputeShortestPath() is called.

The next step of the algorithm is to wait until a change occurs to the gridworld. Once a change does occur, UpdateVertex() is then called. This method updates the rhs values affected do to the change. ComputeShortestPath() is once again called after enough cells have been updated.

```
procedure CalculateKey(s)
{01} return [min(g(s), rhs(s)) + h(s, s_goal); min(g(s), rhs(s))];
procedure Initialize()
{02} U = ∅;
{03} for all s ∈ S rhs(s) = g(s) = ∞;
{04} rhs(s_start) = 0;
{05} U.Insert(s_start, CalculateKey(s_start));
procedure UpdateVertex(u)
{06} if (u ≠ s_start) rhs(u) = min_{s' ∈ Pred(u)}(g(s') + c(s', u));
{07} if (u ∈ U) U.Remove(u);
{08} if (g(u) ≠ rhs(u)) U.Insert(u, CalculateKey(u));
procedure ComputeShortestPath()
{09} while (U.TopKey() < CalculateKey(s_goal) OR rhs(s_goal) ≠ g(s_goal))
{10}   u = U.Pop();
{11}   if (g(u) > rhs(u))
{12}     g(u) = rhs(u);
{13}     for all s ∈ Succ(u) UpdateVertex(s);
{14}   else
{15}     g(u) = ∞;
{16}     for all s ∈ Succ(u) ∪ {u} UpdateVertex(s);
procedure Main()
{17} Initialize();
{18} forever
{19}   ComputeShortestPath();
{20}   Wait for changes in edge costs;
{21}   for all directed edges (u, v) with changed edge costs
{22}     Update the edge cost c(u, v);
{23}     UpdateVertex(v);
```

Shown above is pseudocode taken from [2], which is used to explain the LPA algorithm below.*

6 Conclusion

Pathfinding has changed over the years from searching an entire environment for the goal state to narrowing searches down though the use of heuristics. In recent years several new algorithms have emerged that reduce the computation to find the shortest path by reusing previous searches. Research in search algorithms for pathfinding is still an active area of research. Due to increasing demands for better pathfinding algorithms in interactive entertainment environments and robotics, it is likely there will be people researching it for many more years to come.

7 References

[1] Koenig, S. and Likhavec, M. 2001. Incremental A*. In proceedings of the Neural Information Processing Systems.

[2] Koenig, S. and Likhavec, M. 2002. D* Lite, Technical report, College of Computing, Institute of Technology, Atlanta (Georgia).

[3] Koenig, S. 2002. Fast Optimal Replanning; Georgia Tech; College of Computing, Institute of Technology

[4] Ramalingam, G. and Reps, Thomas, An Incremental Algorithm for a Generalization of the Shortest-Path Problem

[5] Lester, P. 2003 A* Pathfinding for Beginners. http://www.policyalmanac.org/games/aStarTutorial.htm