Optimizations of data structures, heuristics and algorithms for path-finding on maps

Tristan Cazenave
Labo IA
Dept. Informatique
Université Paris 8, Saint-Denis, France
cazenave@ai.univ-paris8.fr

***Abstract*— This paper presents some optimizations of A* and IDA* for pathfinding on maps. The best optimal pathfinder we present can be up to seven times faster than the commonly used pathfinders as shown by experimental results. We also present algorithms based on IDA* that can be even faster at the cost of optimality. The optimizations concern the data structures used for the open nodes, the admissible heuristic and the re-expansion of points. We uncover a problem related to the non re-expansion of dead-ends for sub-optimal IDA*, and we provide a way to repair it.**

Keywords: path-finding, game maps, mazes, A-star, IDA*

I. Introduction

Path-finding is an important part of many applications, including commercial games and robot navigation. In games it is important to use an optimized path-finding algorithm because the CPU resources are also needed by other algorithms, and because many games are real-time. The particular problem addressed in this paper is grid-based path-finding. It is often used in real-time strategy games for example, in order to find the shortest path for an agent to its $goal$ location on the map.

A* [1] is the standard algorithm for finding shortest paths. The usual heuristic associated to A* is the Manhattan heuristic. We present data structures and heuristics that enable A* to be up to seven times faster than the usual implementation.

The contributions of this paper are :

- It is better to use an array of stacks than a priority queue for maintaining the open nodes of an A* search.
- The $ALTBest_P$ heuristic is introduced, and is shown to perform better than the Manhattan heuristic and than the ALT heuristic [2].
- It is useful for IDA* to maintain a two-step lazy cache of the length of the shortest paths found so far.
- Adding a constant to the next threshold of IDA* enables large speed-ups at the cost of optimality. However the lengths of the paths are within 2% of the optimal. Sub-optimal IDA* can be competitive with A*.
- Recording the minimum f for each searched location is useful to find dead-ends, but it can cut valid paths when used with a sufficiently large added constant to the threshold of IDA*. Our program can detect and repair the problem.

Section two describes related work. Section three presents optimizations related to the choice of the best open node. Section four deals with the re-expansion of points. Section five presents the different admissible heuristics we have tested. Section six details experimental results. Section seven concludes and outlines future work.

II. Related work

A* [1] is a search algorithm that finds shortest paths. It uses a fast heuristic function that never over-estimates the length of the path to the goal state, i.e. an admissible heuristic often named h. For each node, it knows g the cost of the path from the root of the search to the node, and it computes f = g + h. The f function is used to develop the tree in a best first manner: A* expands the node with the smallest possible f. IDA* [5] is the iterative deepening version of A*. It can be enhanced with a transposition table that detects positions that have already been searched [6].

Fringe Search [4] is a hybrid of A* and IDA* that reduces slightly the computation time compared to A*. The Fringe Search paper also presents useful optimizations of IDA* for game maps.

The use of map abstractions can be combined with A* to make it faster. For example, Path-Refinement A* [7] builds high level plans and progressively refines them into low-level actions.

The admissible heuristic used for path-finding on road maps, which are much more constrained than game maps, can be improved (i.e. give greater admissible values) using the ALT heuristic [2]. The heuristics used on road maps can be reused for game maps, especially when the game maps are complex and are close to mazes.

III. Choosing the best open node

Each time it expands a node, A* chooses the node with the smallest f function. Therefore the cost of finding the node with the smallest f is very important for A*. In this section we present three methods and the associated data structures used to find the best open node. The first method is using a list of open nodes. The second method is widely used and implements the open list as a priority queue. The third method is faster than the two others and uses an array of stacks.

A. Maintaining a list

The naive method for finding the open node with the smallest f is to go through all the open list to find it. When the open list grows up to more than ten thousand nodes as it can be the case for game maps, it becomes quite time consuming.

B. Maintaining a priority queue

A commonly used optimization for maintaining the open list is to use a priority queue [8], [3], [4], [7]. If N is the number of elements in the priority queue, the insertion and the extraction of an element both take O(log N) which is faster than a constant time for inserting and a linear time for extracting as in the list implementation.

C. Maintaining an array of stacks

Given that the f values are bounded by a relatively small value, it is possible to implement a data structure that inserts nodes in constant time, and that extracts the best node in a short time. This structure is an array of stacks. The index of a stack in the array is the common f value of all the nodes in the stack.

The node class has a $next$ field which is a pointer to another node, and which can be used to put the node on top of a stack of nodes. The code is as follows:

```
Node Open [MaxLength + 1];
int currentf;

void insert (Node *node) {
  Node * tmp = & Open [node->f ()];
  node->next = tmp->next;
  tmp->next = node;
}

Node * best () {
  while (Open [currentf].next == NULL &&
         currentf < MaxLength)
    currentf++;
  return Open [currentf].next;
}
```

The insert function inserts a node in constant time in the array of stacks. The best function is used to extract the node with the best f, and takes very little time too.

The property used to extract the best node starting at currentf in the *best* function is that moves in the map domain never decrease the f function, given the h heuristics used by the program. In domains with possibly decreasing f values, it is straightforward to adapt the code by maintaining the currentf variable in the insert function, taking the minimum of the f of the inserted node and of currentf.

The space complexity of the array is the maximum length of a shortest path, which is low. The space complexity of the stacks is proportional to the number of open nodes.

IV. Avoiding re-expansion of points

Avoiding re-expansion of points is important for path-finding on maps because there are many paths that goes through a point. Among these paths, many arrive at the point with a path longer than the shortest path to the point and must not be expanded. Among the paths that arrive with a length equal to the length of the shortest path, it is necessary to expand only one, and it saves time to cut the others.

A. Checking of the open and closed nodes

A simple method to avoid re-expanding points that have already been expanded with a shorter path is to go through the open and closed lists and verify if the point has already been seen. However when the number of open and closed nodes grows up, this method is quite inefficient.

B. The lazy cache optimization

As game maps fits in memory, it is more efficient to use an array of the size of the map. Each point in the map has an associated index, and the value stored at this index in the array is the shortest path found so far to the point. As it is time consuming to reinitialize the whole array before each search, an optimization is to use a lazy initialization [4]. It consists in having an integer named the $marker$ initially set to zero, and another array of the size of the map initially set to zero too. Let $seen$ be the name of the array that contains the length of the shortest path found, and $mseen$ be the name of the array that is used for the lazy initialization. Before each search, the only thing to do to reinitialize the arrays is to increment the $marker$. To verify if the length of the shortest path to a point has been stored, the program verifies that the $mseen$ array has the value of the $marker$ at the index of the point. To update a shortest path, the program stores the length of the shortest path in the $seen$ array at the index of the point, and put the value of $marker$ at the index of the point in the $mseen$ array. This optimization can be used both for the A* and the IDA* algorithms.

C. Maintaining the best path for each visited point

For each point the program keeps the length of the shortest path to this point that has been found yet. Each time the search passes through the point, the value of g is compared to the stored value for the point. If g is strictly smaller, the value is replaced with g and the search continues. If g is greater or equal to the value of the point, the search is stopped. When using the lazy cache optimization for A*, a single lazy initialization is sufficient. When using it for IDA*, two lazy initializations are better. A first lazy initialization is used before the first search of IDA*, as in A*, in order to reinitialize the values. However, before performing the second search of the IDA* algorithm, we are faced with a dilemma: if we reinitialize the values the program loses the valuable information of the length of the shortest paths found so far, and if we do not reinitialize the values, the program won't expand the nodes that have already been searched coming from a shortest path and then the search will fail. The solution is to use two lazy arrays and two markers: $mseen$ and $marker$ to differentiate the searches between different points, and $mming$ and $markerming$ to differentiate between different searches between the same points. Note that the later differentiation is only useful and used for IDA*.

Before making a move, the program calls the $Seen$ function that tells him if the move leads to a re-expansion. To make things even clearer, we give the code of the function that uses the two lazy arrays:

```
bool Seen (int pos, int g) {
  if (mseen [pos] != marker) {
    seen [pos] = g;
    mseen [pos] = marker;
    mming [pos] = markerming;
    return false;
  }
  if (g < seen [pos]) {
    seen [pos] = g;
    mming [pos] = markerming;
    return false;
  }
  if (g == seen [pos])
    if (mming [pos] != markerming) {
      mming [pos] = markerming;
      return false;
    }
  return true;
}
```

The points on the map are represented by an integer, and the map and its associated arrays are represented by unidimensional arrays.

D. Maintaining the minimum f for each visited point

A lazy cache can also be used to memorize for each node the minimum f found over all leaves under the node. It can be used to detect dead ends of the map [4].

The code of IDA* with the memorization of the minimum f, and the related cut of dead-ends is:

```
bool IDA (int g, int pos, int & minf) {
 // minf is passed by reference, it can
 // be changed in the calling function
 nodes++;
 int f = g + h (pos);
 // currentfIDA is the threshold of
 // the iterative deepening search
 if (f > currentfIDA) {
  if (f < minf)
    minf = f;
  return false;
 }
 if (pos == goalPos)
  return true;
 int tmpminf = MaxLength;
 for (newpos in all neighbors) {
  if (!occupied (newpos))
   if (!Seen (newpos, g)) {
    if (Seenminf (newpos) &&
        Minf (newpos) == MaxLength)
     continue;
    if (IDA (g + dg, newpos, tmpminf))
     return true;
    Setminf (newpos, tmpminf);
   }
 }
 if (tmpminf < minf)
  minf = tmpminf;
 return false;
}
```

The lazy cache is modified using the $Setminf$ function. A cut occurs when a location has already been searched ($Seenminf(newpos)$ returns true) and when no minimum f has been found with this search ($Minf(newpos)$ returns $MaxLength$).

E. Thresholds for IDA*

The minimum f found over all the leaves of the root node ($minf$) can be used for the next threshold of the IDA* search, instead of simply incrementing the threshold. It saves the searches with the thresholds between the last threshold and minf.

A problem with IDA* on maps is that the number of nodes of a search with a threshold is not small in comparison with the number of nodes of the search with the next threshold. This is the main reason why IDA* is not competitive with A* on maps. If we accept to find slightly sub-optimal paths, it is possible to improve the speed of IDA* by taking a threshold slightly greater than minf at each iteration of IDA*. We call this algorithm $IDA\Delta_D$ when the threshold for the next iteration is minf + D.

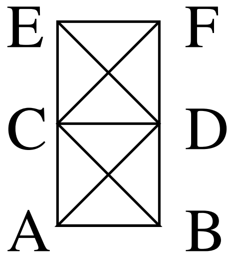

Fig. 1. Problem cutting with the minimum f

However, there is a problem using the minimum f optimization with increased thresholds. Let's look at the possible paths from A in figure 1. We choose for this example that going horizontally and vertically costs two, and that going diagonally costs three. If the program starts searching paths ADE and ADF, the $seen$ value of D is three, the $seen$ value of E is six, and the $seen$ value of F is five. Now if the program search the paths ABC, it arrives at C with a g of five, and therefore the tree is cut when it continues to E (g is seven and $seen(E)$ is six), to F (g is eight and $seen(F)$ is five), and to D (g is seven and $seen(D)$ is three). The consequence is that the minimum f stored at C is MaxLength and that C is considered a dead-end. So when the program looks at the path that starts with AC (and which may be the shortest path), the tree is cut.

In our experiments the problem does not appear when using the minf threshold with IDA* or when using tiles. However it appears on complex maps using octiles and a large delta for $IDA\Delta_D$.

V. Improving the admissible heuristic

The usual heuristic for path-finding on maps is the Manhattan heuristic. The ALT heuristic usually finds better values than the Manhattan heuristic, but it takes more memory since for each point of the heuristic the pre-computations take a memory proportional to the size of the map, and it also takes more time to compute. The ALTBest$_P$ heuristic gives better values than the Manhattan heuristic but takes more time than it, and worse values than the ALT heuristic but takes less time than it. This section presents these three heuristics.

A. The Manhattan heuristic

The Manhattan heuristic is a very popular heuristic. It is used for path-finding on maps, but also for other games such as the 15-Puzzle or Rubik's cube. It consists in considering that the path to the goal is free of obstacles, which allows a very fast computation of a lower bound on the length of the shortest path. For maps, the heuristic is different if the moves are restricted to the four horizontal and vertical neighbors (tiles), or if the eight neighbors including diagonals are allowed (octiles). The code for the Manhattan heuristic on maps is:

```
int h (int pos) {
  int dx = abs (x (goalPos) - x (pos));
  int dy = abs (y (goalPos) - y (pos));
  if (nbNeighbors == 8)
    return CostDiag * min (dx, dy) +
           Cost * (max (dx, dy) -
                   min (dx, dy));
  else
    return dx + dy;
}
```

where $Cost$ is the cost of moving to a horizontal or vertical neighbor, and $CostDiag$ the cost of moving to a diagonal neighbor.

B. The ALT heuristic

ALT is a heuristic that works well on road maps [2]. It consists in pre-computing the distance to all points from a given point, and then in using these pre-computed distances to calculate an admissible heuristic. The heuristic is based on the triangular inequality. For example if $d(X, Y)$ is the length of the shortest path between X and Y, if the distances are pre-computed from $pPos$, the current node is at $cPos$, and the goal position is at $gPos$, we have the following inequalities:

$$d(cPos, pPos) \leq d(cPos, gPos) + d(gPos, pPos) \quad (1)$$

$$d(cPos, gPos) \leq d(cPos, pPos) + d(pPos, gPos) \quad (2)$$

$$d(gPos, pPos) \leq d(gPos, cPos) + d(cPos, pPos) \quad (3)$$

From 1 and 3 we can show, respectively:

$$d(cPos, gPos) \geq d(cPos, pPos) - d(gPos, pPos) \quad (4)$$

and

$$d(gPos, cPos) \geq d(gPos, pPos) - d(cPos, pPos) \quad (5)$$

Given that $d(gPos, cPos) = d(cPos, gPos)$ we have (abs is the absolute value):

$$d(gPos, cPos) \geq abs(d(gPos, pPos) - d(cPos, pPos)) \quad (6)$$

Therefore, an admissible heuristic, which only uses pre-computed values, is:

$$h = abs(d(gPos, pPos) - d(cPos, pPos)) \quad (7)$$

If distances are pre-computed for multiple points, the heuristic chooses for h the maximum value over the ALT values given by each point.

C. The ALTBest$_P$ heuristic

The ALTBest$_P$ uses pre-computed distances from P points. Instead of taking the maximum value over the h values computed with the P points at each node of the search, it selects among the P points the one that gives the highest h value at the root node. For all nodes of the search, it chooses as h value the maximum of the ALT value computed with the selected point and of the Manhattan heuristic. The h values found with this heuristic are worse than the h values found with the ALT heuristic, therefore the search will develop more nodes. The advantage of ALTBest$_P$ is that it takes less time at each node and that it is not much worse.

D. Other use of pre-computed distances

Pre-computed distances can also be used to find if a goal is impossible. If the starting location has a distance to a pre-computed point, and that the goal location has an infinite distance to the pre-computed point, the program knows it is useless to search a path, and it can find it without search. This is also true for the reverse situation where there is an infinite distance to the starting location, and a finite distance to the goal location.

The symmetric use of pre-computed points is to find if a path is possible. This can be useful for the $IDA\Delta_D$ algorithm when it has problems due to the minimum f dead-end cuts. If the algorithm finds a path is impossible when a pre-computed point finds there is one, the program can revert to a slower but more safe algorithm such as IDA* without the minimum f dead-end cut optimization, or to a simple IDA* with a D set to zero.

VI. Experimental results

The maximum number of nodes per search is set to 10,000,000. Experiments are performed on a Celeron 1.7 GHz with 1GB of RAM. When the program only considers four neighbors for each point on the map, the cost of a move to one of the four neighbor is set to one. When it considers eight neighbors, the cost of going to a vertical or a horizontal neighbor is set to two, and the cost of going to a diagonal neighbor is set to three. If the horizontal and vertical cost is two, the real diagonal cost is 2.8 which is close to three. The reason we choose three is that the array of stacks optimization works more simply with integer costs.

A. Memory allocation

Memory allocation is one of the instructions that takes the most time with current operating systems. The standard A* algorithm allocates memory for each new node of the search. In order to optimize further A*, we have used pre-allocation. An array of 10,000,000 nodes is allocated once for all the searches at the beginning of the program, and when a new node is needed it is taken from the array. When a search is over, to reinitialize memory, the only thing to do is to put the integer associated to the array back to the top of the array. This optimization is linked to the buffering optimization used in [3], but we find the use of a pre-allocated array more simple and more efficient.

B. The tested algorithms

The different algorithms that have been tested are:

- $M(N)$ is A* with the Manhattan heuristic, each point has N neighbors. The structure used to maintain the open list sorted is an array of stacks, and new nodes are taken from a pre-allocated array of nodes.
- $Mmem(N)$ is M (N) with memory allocation at each node.
- $Mqueue(N)$ is M (N) with a STL (Standard Template Library) priority queue for finding the best open node.
- $ALT_P(N)$ is A* with the ALT heuristic, points on the map have N neighbors, and P points chosen at random are used for computing the distances for the ALT heuristic. All the distances are pre-computed, and their pre-computation is not taken into account for the timing of the algorithm. The admissible heuristic consists in computing, for all the pre-computed P points, the absolute value of the difference of the distance between the point and the goal position, and of the distance between the point and the node's position. It then chooses the maximum value over all the absolute values and the Manhattan heuristic.
- $ALTBest_P(N)$ is A* with the ALTBest$_P$ heuristic, points on the map have N neighbors. P points are randomly chosen and the distances to each point of the map are pre-computed for each of the P points. At the root of the search, the program selects the pre-computed point which has the highest h value. During the remainder of the search, it computes h as the maximum of the ALT value computed with this point and of the Manhattan heuristic.
- $IDA(N)$ is iterative deepening A* with the Manhattan heuristic. Each point has N neighbors.
- $IDABest_P(N)$ is iterative deepening A* with the ALTBest$_P$ heuristic. Each point on the map has N neighbors.
- $IDABest_P\Delta_D(N)$ is iterative deepening A* with the ALTBest$_P$ heuristic. Each point on the map has N neighbors. At each iteration of the iterative deepening search, instead of taking minf as the next threshold for the search, the programs takes the minimum f over the leaves of the previous search (minf) plus D.
- $IDA\Delta_D(N)$ is iterative deepening A* with the Manhattan heuristic. At each iteration, the next threshold is minf + D.
- $IDAnof(N)$ is $IDA(N)$ except that the program does not use the recorded minf at each node to cut the search.

C. The experimental testbed

The algorithms have been tested with different values for the number of neighbors, the number of pre-computed points, and the delta threshold. There are three experiments, all with 300x300 maps. The experiments use maps with respectively two hundred walls of size twenty, four hundred walls, and six hundred walls. A set of one hundred maps has been generated for each experiment. The walls of a map are generated by taking at random an unoccupied point and one of the eight directions, for vertical and horizontal directions the wall is generated as a line with a thickness of one, and for diagonal directions it is generated as a line with a thickness of two in order to avoid a path that goes diagonally through a diagonal wall. For each map, two unoccupied points are chosen at random and the algorithm searches for a shortest path between these two points. For each algorithm, we give the sum of the number of nodes of all the searches, the time spent, the number of problems where the algorithm found a path, and the sum of the lengths of the found paths.

D. Results on simple maps

Table I gives the nodes, time, number of solved problems and sum of lengths of paths found for different algorithms on 300x300 maps with 200 random walls of size 20.

The best algorithm for speed is $IDABest_{10}\Delta_{10}$ both for tiles and octiles. The length of the paths it finds is close to the shortest path as can be seen comparing the sums of the lengths.

The best exact algorithm is $ALTBest_{10}$. For tiles it is interesting and surprising to note that $IDABest_{10}(4)$ always finds the shortest path and is faster than $ALTBest_{10}(4)$.

Another result is that using an array of stacks ($M(8)$) is twice as fast as using priority queues ($Mqueue(8)$) for octiles, and three times faster for tiles ($M(4)$ vs $Mqueue(4)$). The number of nodes is different in the favor of priority queue for the two algorithms because they do not expand nodes in the same order due to their different algorithms for selecting the best node.

TABLE I

300X300 MAPS WITH 200 WALLS OF SIZE 20.

$Algorithm$	$nodes$	$time$	$solved$	sum
$IDABest_{10}\Delta_{10}(8)$	1,102,592	0.79s	98	36,387
$ALTBest_{10}(8)$	480,407	1.60s	98	35,998
$IDABest_{10}(8)$	2,146,234	1.98s	98	35,998
$M(8)$	764,262	2.10s	98	35,998
$ALT_{10}(8)$	335,418	3.21s	98	35,998
$IDA(8)$	4,340,651	3.42s	98	35,998
$ALT_5(8)$	500,989	3.80s	98	35,998
$Mqueue(8)$	722,344	4.16s	98	35,998
$IDABest_{10}\Delta_{10}(4)$	479,046	0.26s	98	21,300
$IDABest_{10}(4)$	690,457	0.49s	98	21,072
$ALTBest_{10}(4)$	399,839	0.77s	98	21,072
$M(4)$	611,595	0.91s	98	21,072
$IDA(4)$	1,628,550	1.08s	98	21,072
$ALT_{10}(4)$	256,310	1.23s	98	21,072
$Mqueue(4)$	664,504	2.96s	98	21,072

TABLE II

300X300 MAPS WITH 400 WALLS OF SIZE 20.

$Algorithm$	$nodes$	$time$	$solved$	sum
$IDABest_{10}\Delta_{10}(8)$	2,601,120	2.05s	95	39,362
$IDABest_{10}\Delta_{20}(8)$	2,773,262	2.07s	95	39,969
$ALTBest_{10}(8)$	950,862	2.93s	95	38,911
$M(8)$	1,455,092	4.24s	95	38,911
$ALT_5(8)$	866,041	4.85s	95	38,911
$Mmem(8)$	1,455,092	6.30s	95	38,911
$IDABest_{10}(8)$	7,901,822	7.92s	95	38,911
$Mqueue(8)$	1,358,262	8.36s	95	38,911
$ALT_{10}(8)$	646,554	11.74s	95	38,911
$IDA(8)$	21,837,340	19.71s	95	38,911
$IDAnof(8)$	133,089,043	98.93s	95	38,911

$IDA(8)$ and $IDA(4)$ are worse than $M(8)$ and $M(4)$ as already found in other studies [4], but the difference between the two is much smaller than what was found before (20% to 60% more time instead of six to twenty times more time).

The ALT heuristic searches half of the nodes of the Manhattan heuristic but is slower.

E. Results on moderately complex maps

Table II gives the results for different algorithms on 300x300 maps with 400 random walls of size 20.

The $IDABest_{10}\Delta_{10}(8)$ is the fastest algorithm, it solves all the solvable problems, and the sum of the lengths of the found paths is within 2% of the sum of the shortest paths.

The best exact algorithm is again $ALTBest_{10}(8)$, it develops less nodes in less time than the Manhattan heuristic. $ALT_5(8)$ and $ALT_{10}(8)$ also develop less nodes than the two previous algorithms, but take more time due to the overhead of computing the ALT heuristic at each node.

Comparing $Mqueue(8)$ and $M(8)$, we can see that maintaining an array of stacks enables a speed-up of two compared to maintaining a priority queue. Pre-allocating nodes in an array gives a speed-up of 1.5 as can be seen when comparing $Mmem(8)$ and $M(8)$.

Concerning the $IDA(8)$ algorithm, it is almost five times slower than the $M(8)$ algorithm. Even if the minimum f dead-end cut can be harmful with $IDABest_{10}\Delta_{10}(8)$ or with $IDABest_{10}\Delta_{20}(8)$, it is not the case here as they solve all the problems. However we tested the usefulness of the minimum f dead-end cut by removing it, and we see that $IDAnof(8)$ is five times slower than $IDA(8)$ with the optimization. Using the $ALTBest_{10}$ heuristic with IDA* improves it since $IDABest_{10}(8)$ is more than twice as fast as $IDA(8)$.

TABLE III

300X300 MAPS WITH 600 WALLS OF SIZE 20.

$Algorithm$	$nodes$	$time$	$solved$	sum
$IDABest_{10}\Delta_{10}(8)$	1,880,053	1.42s	61	36,513
$IDABest_{10}\Delta_{20}(8)$	2,262,738	1.71s	60	36,008
$ALTBest_{10}(8)$	604,461	1.81s	61	36,208
$IDABest_{10}\Delta_{20}nof(8)$	3,302,562	2.36s	61	36,785
$IDABest_{10}\Delta_{10}nof(8)$	3,318,593	2.50s	61	36,469
$ALT_5(8)$	542,129	2.71s	61	36,208
$ALT_{10}(8)$	347,189	3.00s	61	36,208
$IDABest_{10}(8)$	6,683,647	6.33s	61	36,208
$M(8)$	2,438,460	6.67s	61	36,208
$IDA\Delta_{20}(8)$	12,111,234	9.45s	56	33,087
$Mmem(8)$	2,438,460	10.05s	61	36,208
$IDA\Delta_{10}(8)$	12,233,668	10.93s	61	36,637
$Mqueue(8)$	2,296,036	13.67s	61	36,208
$IDA(8)$	67,483,398	70.27s	61	36,247
$IDABest_{10}\Delta_{10}(4)$	614,865	0.37s	44	15,889
$IDABest_{10}\Delta_{20}(4)$	693,650	0.37s	44	16,161
$ALTBest_{10}(4)$	401,287	0.61s	44	15,711
$IDABest_{10}(4)$	2,954,173	1.81s	44	15,711
$M(4)$	1,886,487	2.94s	44	15,711
$IDA\Delta_{20}(4)$	7,575,321	4.66s	44	16,199
$IDA\Delta_{10}(4)$	7,839,990	5.49s	44	15,959
$IDA(4)$	233,428,544	131.24s	44	15,711

F. Results on complex maps

Table III gives the nodes and time for different algorithms on 300x300 maps with 600 random walls of size 20.

The first observation is that on complex maps that look more like mazes or road maps, the ALT_5 and the ALT_{10} heuristics are now faster than the Manhattan heuristic. The overhead of computing the ALT values at each node is compensated by a greater number of cut nodes. However the $ALTBest_{10}(8)$ is still the best exact algorithm and it is more than three times faster than the Manhattan heuristic, and more than seven times faster than $Mqueue(8)$ which is the standard implementation for path-finding on game maps.

$M(8)$ with arrays of stacks is still twice as fast as $Mqueue(8)$, and 1.5 faster than $Mmem(8)$.

The fastest algorithm is still $IDABest_{10}\Delta_{10}$ and it is within 2% of the optimal sum of lengths. However $IDABest_{10}\Delta_{20}(8)$ and $IDA\Delta_{20}(8)$ do not find all the paths, due to the problem with the minf dead-end cut when employed with a sufficiently large delta. This problem appears on octiles and not on tiles. A possible repair to the problem is to use the pre-computed distances. If the program knows a path is possible, and $IDA\Delta_{20}(8)$ does not find it, it can revert to $IDABest_{10}\Delta_{20}nof(8)$ or $IDABest_{10}(8)$ only for this problem.

Unlike $IDAnof(8)$ which was five times slower than $IDA(8)$ in the previous experiment, here $IDABest_{10}\Delta_{10}nof(8)$ is less than two times slower than $IDABest_{10}\Delta_{10}(8)$, and $IDABest_{10}\Delta_{20}nof(8)$ finds all the paths when $IDABest_{10}\Delta_{20}(8)$ misses one.

The $ALTBest_{10}$ heuristic is even more useful on complex maps than on more simple maps. $IDABest_{10}(8)$ is more than ten times faster than $IDA(8)$, and it is even better with tiles since $IDABest_{10}(4)$ is more than seventy times faster than $IDA(4)$.

VII. Conclusions and future work

New data structures, heuristics and algorithms for fast path-finding have been described and tested.

We showed that maintaining an array of stacks enables A* to be faster than usual implementations of A* that use a priority queue.

We presented the $ALTBest_P$ heuristic and we experimentally proved it is better than the Manhattan heuristic and the ALT heuristic for maps of different complexities, both for A* and IDA*. The best exact algorithm we have presented, based on $ALTBest_P$ and A* with arrays of stacks, is up to seven times faster than the usual algorithm for path-finding on maps.

Another result is that IDA* can be competitive with A* on maps. We presented an algorithm based on IDA* ($IDABest_{10}\Delta_{10}$) that finds close to optimal paths faster than our best implementation of A* ($ALTBest_{10}$). We also observed that the speed up of traditional A* over traditional IDA* depends on the complexity of the map.

We have also shown that it is useful to have a two-step lazy cache strategy for remembering the length of the shortest path to visited points of the maps. A potential problem with the use of a recorded minimum f for each visited point, when it is used to cut the search, has been uncovered. A repair strategy has been proposed when this problem happens. It uses the pre-computed distances from the ALT heuristic to detect the problem, and it falls back on a safe algorithm when the problem occurs.

The $ALTBest_P$ heuristic as well as the lazy cache optimizations increase the space requirements of the algorithms by an ammount proportional to the size of the map.

For future work, it is interesting to find a way to deal better with the problems encountered while cutting nodes due to the memorized minimum f. In particular, it would be valuable to keep the current power of memorizing the minimum f-value for each node and of cutting dead-end nodes, while enabling to increase the threshold of IDA* by more than the minimum f, without losing some paths.

It is also interesting to have a better selection of the points used for the ALT heuristic, like for example selecting the points which are the farthest away from already existing points [2].

Combining the optimizations presented in this paper with optimizations due to abstractions of the maps, or optimizations related to way-points can also be of interest.

References

[1] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybernet.*, vol. 4, no. 2, pp. 100–107, 1968.

[2] A. V. Goldberg and C. Harrelson, "Computing the shortest path: A* search meets graph theory," in *SODA'05*, 2005.

[3] P. Kumar, L. Bottaci, Q. Mehdi, N. Gough, and S. Natkin, "Efficient path finding for 2D games," in *CGAIDE 2004*, Reading, UK, 2004, pp. 263–267.

[4] Y. Bjornsson, M. Enzenberger, R. C. Holte, and J. Schaeffer, "Fringe search: beating A* at pathfinding on game maps," in *IEEE CIG'05*, Colchester, UK, 2005, pp. 125–132.

[5] R. E. Korf, "Depth-first iterative-deepening: an optimal admissible tree search," *Artificial Intelligence*, vol. 27, no. 1, pp. 97–109, 1985.

[6] A. Reinefeld and T. Marsland, "Enhanced iterative-deepening search," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 16, pp. 701–710, 1994.

[7] N. Sturtevant and M. Buro, "Partial pathfinding using map abstraction and refinement," in *AAAI 2005*, Pittsburgh, 2005.

[8] B. Stout, "Smart moves: Intelligent path-finding," *Game Developper Magazine.*, pp. 28–35, October 1996.