Subject: Facetten van de AI, Group D, Vera Stebletsova
Faculty of Sciences: Artificial Intelligence
Vrije Universiteit Amsterdam
De Boelelaan 1081a
1081 HV Amsterdam

Autonomous Navigation in an Unknown Environment

Marten Klencke, 1271229, mklencke@cs.vu.nl

12th May 2005

This paper provides an overview of the issues that arise when designing a system that is capable of autonomous navigation in an unknown environment. In addition to presenting the necessary components for such a system and ways to model the environment, dynamic path planning algorithms are treated from the ground up.

The applicability of different algorithms depends on different factors. Required performance and correctness may make the usage of a particlar algorithm unfeasible. However, much research remains to be done in this domain of AI.

1 Introduction

Successfully landing an unmanned vehicle on the face of the planet Mars is one in a series of recent breakthroughs in space exploration and technical possibilities. This accomplishment has received wide media coverage, which is understandable: it feeds our imagination. Somewhere out there, at least 55 million kilometers away from earth[1], a robotic vehicle autonomously roams the planet's surface, collecting data, taking photographs, and sending them back to us. This machine is behaving in a seemingly intelligent fashion.

One of the crucial capabilities of Marslanders is autonomous navigation. These multibillion Euro projects can not be allowed to fail because the vehicle stumbled and fell down a ravine. Unfortunately, the distance between Earth and Mars is too long to allow for direct control: around 11 lightminutes. Therefore, a navigational system had to be engineered that enables the rover to travel from A to B while detecting and avoiding obstacles and other hazards.

Similar requirements exist in systems that are more down-to-earth. One particular example is computer games. Optimal game-

[1] http://en.wikipedia.org/wiki/Mars

play is achieved when simulated opponents or other computer-controlled entities in virtual worlds behave in an intelligent, perhaps even human-like, fashion. As many games are modelled around some traversable terrain (first-person shooters, real time strategy games, etc.), navigational systems are applicable here too. Countless other systems, like car navigation systems, mobile security units, autonomous vacuum cleaners and lawn mowers, etc. employ the same, or related, techniques.

This paper attempts to provide an overview of the available methods for autonomous navigation, with an emphasis on path planning algorithms and their specific characteristics and benefits. Of importance is the ability to incrementally build a terrain map and adjust the plan accordingly (hence, the "unknown environments" part of the title.)

First, the discretization of world information is addressed. Then, path finding algorithms are introduced starting with Dijkstra's Algorithm and subsequently discussing heuristics and incremental techniques. Finally, some assorted techniques for improvement are briefly explained after which conclusions are drawn.

People with an interest in AI-techniques and path finding, but no strong expertise in the latter (among these are students of computer science and game developers), are the main intended audience for this paper. Most algorithms, such as Dijkstra's Algorithm, will be briefly but sufficiently explained. However, formal proofs of correctness are not within the scope of this text. Readers are referred to the corresponding literature for this, and more, advanced information.

2 Issues in Navigation

There are four main steps that have to be taken in order for a physical vehicle to navigate autonomously:

1. Collect data from sensors.
2. Build an internal representation of the environment.
3. Calculate the shortest path to the goal.
4. Perform physical movement

Of course, virtual vehicles work in a slightly different way. They have no sensors and do not move in the physical world. However, the principles remain the same. A part of the simulation program provides information about the virtual environment (step 1) and another part performs the virtual movement.

This paper is mainly concerned with step 3. It is assumed that a goal location is available. This could be a location on the surface of Mars, the tile behind an enemy in a computer game, or anything else.

Steps 1 and 4 will not be treated as they are the field of other engineering disciplines. Step 2 will be briefly touched upon because of its relevance to the algorithms the algorithms that are discussed subsequently.

2.1 Terrain Representation

Because of their formal nature and practical reasons (like computational power, system complexity, etc.), algorithms need a simplified model of the physical world to work on. Ideally, the representation of the terrain in computer memory is small and can be accessed quickly to optimize performance.

2.1.1 Partitioning

The terrain needs to be partitioned into tiles that can be individually visited by the vehicle. Smaller tiles mean more tiles and therefore a higher amount of required resources (computational power, memory, etc.) On the other hand, smaller tiles allow for greater flexibility: for

example, there is less chance that a tile will contain an obstacle and the vehicle has more freedom of movement.

Typically, the tile size will be related to physical size of the vehicle. A Mars rover with dimensions $50cm * 90cm$ might opt for tiles of 1 square meter. Likewise, tiles in computer games will be approximately the size of their vehicles.

Hybrid methods for tile size are also possible. See section 3.5.

Another design choice is the shape of the tiles and possible directions of movement. In the simplest form (for a 2-dimensional terrain map) they are squares allowing only vertical or horizontal movement. Diagonal movement would add complexity, but also flexibility. Hexagonal tiles may be another option. Key factors in making a decision in regard to this are terrain characteristics and vehicle shape and movement capabilities.

2.1.2 Traversability & Obstacles

One of the requirements of autonomous navigation is obstacle avoidance. These obstacles need to be represented in the map, so there are at least to types of terrain: traversable and untraversable.

Reality is more complex and so multiple terrain types, which are all considered traversable, might also be necessary. For example, a terrain might consist of dirt areas with small pebbles and areas with fine sand. A vehicle can traverse both, but movement over sand requires more effort (but see section 2.2).

2.1.3 Discretization

Fed by the data from sensors, a discretization module creates or updates a representation of the world that the algorithms can operate on. Actual implementations may vary greatly, but this paper uses graph theory for illustrational purposes. This is also the representation used in most texts about algorithms.

We define a graph G in which every node represents a tile on the terrain map. The source node, usually the current location of the vehicle, is denoted by the letter s. Similarly, the destination (or target) node is denoted by the letter d. Paths from one tile to another are represented as edges between nodes in G. Weights of edges specify movement costs (see next section.)

How such a graph is actually implemented in computer systems depends highly on several factors like the architecture of the platform, the programming language used, etc. Naturally, the most efficient solution (in terms of performance, maintainability, etc.) will be preferred in most cases.

2.2 Movement Cost

Each edge in the terrain graph is accompanied by a weight; the movement cost. This is an approximate cost for the vehicle to move from one particular tile to another. Depending on the nature of the pathfinder, this cost could be physical distance, fuel usage, danger, or anything else.

Movement cost may be influenced by factors such as terrain type (paved road vs. mud) and available energy sources (sun vs. shade.)

Because of these many influential factors, one may choose to create several terrain maps that are used simultaneously by the path finding algorithms. For example, a terrain class map, a heigh map, an energy map, etc. Alternatively, these values may be added and represented in one map. These are implementation details that this paper will not elaborate on.

2.2.1 Vertical Movement

Vertical movement occurs when there are slopes on the terrain. It is interesting because

it adds a third dimension to the terrain.

One way to cope with this third dimension is to make the terrain map 3D as well. The vehicle might even be able to move vertically in an arbitrary fashion.

Another, simpler, solution is to introduce extra movement costs to the 2D terrain map. An ascent (positive slope) increases costs, a descent (negative slope) decreases costs.

3 Path Planning Algorithms

Once the necessary information about the terrain is available, the navigational system can kick in.

When planning a path from an arbitrary location A to location B, the goal is usually to find a path that has the lowest cost. A combination of variables is also possible.

3.1 Dijkstra's Algorithm

Dijkstra's Algorithm [4, 6] is a very well-known and widely used algorithm which is guaranteed to find a shortest path (there may be multiple paths of the same length.) It is also referred to as the "Single-Source Shortest Paths" algorithm because, starting from one source location, it will find multiple shortest paths to different nodes in all directions.

Fundamentally, the algorithm works as a breadth-first search, starting out at the source node s and working outward from there until the target node d has been found.

Initially, the accumulated movement cost is set to 0 for s (the vehicle is already there) and is undefined for all other nodes. At each step, the algorithm picks a node (X) that has the lowest accumulated cost and explores its incident nodes (Y). They have two possible states:

1. Undefined. In this case, assign the cost of moving from X to Y, added to the cost of X, to Y.

	1	2	3	4	5	6	7	8	9	10
A	5	4	5	6	7	8	9	10	11	12
B	4	3	4	5	6	■	10	11	12	
C	3	2	3	■	■	■	11	12		
D	2	1	2	■			12			
E	1	S	1	■				12		
F	2	1	2	■	■	■	■	11	D	
G	3	2	3	4	5	6	■	10	11	12
H	4	3	4	5	6	7	8	9	10	11
I	5	4	5	6	7	8	9	10	11	12
J	6	5	6	7	8	9	10	11	12	

Figure 1: Cost Values in Dijkstra's Algorithm

2. $\leq cost(X) + weight(X,Y)$: Leave the current value intact. A shorter path to Y was already found.

A situation where a node Y is encountered that has a higher accumulated cost than the cost of X plus $weight(X,Y)$ will not occur because X is always picked from the list of nodes with lowest cost that still have unexplored incident nodes. Therefore, the explored region will always grow in circular fashion.

Once the target node has been reached, the shortest path can be found by greedily tracing optimal values back from d to s.

Consider Figure 1 . All nodes are represented as squares and edges exist between vertically or horizontally neighbouring squares[2]. The cost of moving from one node to another is 1. Obstacles are depicted as black squares, the source square is marked with an S and the destination square is marked with a D.

There are multiple shortest paths to d of (which could randomly be chosen when tracing back towards s), but they all traverse nodes below the obstacle. Their cost is 12.

Unfortunately, because all nodes with a cost less than or equal to the target cost are processed before the target node is found, Dijk-

[2]This is called a gridworld

stra's algorithm is slow. The are some ways to improve performance (see section 3.5), which mostly depend on the characteristics of the terrain. However, for consistent improvement, altered algorithms are needed. Two ways in which to do this are:

1. Use heuristics to guide the search. A* employs this technique.
2. Assuming that a search has been done earlier, for example when the vehicle was in a different position, reuse earlier search results but alter them to fit the current state. This is done by incremental algorithms such as DYNAMICSWSF-FP.

Still, Dijkstra's Algorithm is very robust. Therefore, there may be situations where this robustness is chosen over performance. For instance, the Mars robot which was mentioned in the introduction needs to navigate very accurately. Autonomous navigation using Dijkstra's Algorithm will not be optimally fast, but is still a great improvement over the 11 minute delay from earth.

3.2 Heuristics

3.2.1 Best-First Search

Best-First Search is very similar to Dijkstra's Algorithm, but instead of using accumulated cost values, node values are calculated by a heuristic function $h(X)$ that approximates the remaining cost to travel to the goal from node X. More importantly, this approach does not necessarily result in an optimal path. It depends on the quality of the heuristic. However, it is very fast.

One such heuristic is the Manhattan Method (probably named after the blocks in New York that one must travel around) [3]. The heuristic value for distance to a node is the distance when only vertical or horizontal movement is possible: $d = |x_1 - x_2| + |y_1 - y_2|$

	1	2	3	4	5	6	7	8	9	10
A	13	12	11	10	9	8	7	6	5	6
B	12	11	10	9	8		6	5	4	5
C	11	10	9				5	4	3	4
D	10	9	8		6	5	4	3	2	3
E	9	S	7		5	4	3	2	1	2
F	8	7	6					1	D	1
G	9	8	7	6	5	4		2	1	2
H	10	9	8	7	6	5	4	3	2	3
I	11	10	9	8	7	6	5	4	3	4
J	12	11	10	9	8	7	6	5	4	5

Figure 2: Manhattan Distances to Target

See figure 2 for the Manhattan values for the gridworld presented in section 3.1. Of course, other heuristics like Euclidian distance are also possible.

The advantage of Best-First Search is that much less calculation needs to be done in order to reach the target. Instead of searching in all directions, the search now consists of greedily finding lower values of neighbouring nodes and travelling there.

There are also caveats. Mainly, the algorithm uses a heuristic in a greedy manner. It does not consider movement costs. Therefore, a route may easily turn out to be much more costly than according to the heuristic.

3.2.2 A*

As was described in the previous sections, both Dijkstra's algorithm and Best-First Search have specific problems: mainly speed and correctness (finding the absolute shortest path) respectively. However, it is possible to combine the two approaches in order to get the best from both worlds. This is what A* does [3, 4]. Instead of spreading out in all directions, the algorithm has a bias towards a certain direction, as determined by a heuristic. Still, because of the use of a heuristic, A* may also not return

the absolute shortest path. Whether this is acceptable depends on the application. In some cases, it may even desirable, for instance to model human-like behavior for opponents in computer games.

The key to determining which nodes to use when searching for a path is the following equation:

$F = G + H$

in which:

- G is the cost to move from the starting node to the current node, as in Dijkstra's Algorithm.
- H is the heuristic for distance to the target, as in Best-First Search.

The algorithm maintains an "open list" of nodes that need to be considered and a "closed list" of nodes that have already been visited. It proceeds as follows:

1. Add the starting node to the open list.
2. Repeat:
   a) Look for the node with the lowest F score in the open list.
   b) Move it to the closed list.
   c) For each incident node, if it is traversable and not in the closed list:
      i. If it is not in the open list, add it to the open list. Make the current node the parent of this incident node. Record the F, G and H values of the node.
      ii. If it is already in the open list, use the G value to determine if this path is better (lower G). If so, change the parent node to the current node and recalculate the G and F values.

The algorithm is terminated in one of the following conditions:

1. The target node has been found (i.e. added to the open list.)
2. The open list is empty but the target node has not been found. In this case, there is no path.

To optimize this algorithm, it is a good idea to keep the open and closed lists sorted.

3.3 Incremental Algorithms

So far, only pathfinding algorithms which require complete knowledge of the terrain have been mentioned. What happens when there is no knowledge of the terrain and only the target location (node) is known?

One strategy is to assume a direct path to the target and attempt to pursue it, discovering obstacles along the way. At each such discovery, the shortest path could be calculated again using one of the above methods. However, some algorithms exist that, in some cases, enable a more efficient method of map updating.

3.3.1 DYNAMICSWSF-FP

This incremental algorithm is based on Dijkstra's algorithm. DYNAMICSWSF-FP [2, 5] basically caches movement cost values that were calculated during previous searches. It uses a clever way of identifying the movement costs that have not changed and recalculates only the ones that have changed. Consequently, it results in highest performance gain (compared to Dijkstra's Algorithm) in situations where only a small number of costs change.

The dynamics of the algorithm can best be explained by example: assuming a gridworld as in Figure 1, where all cost values have been

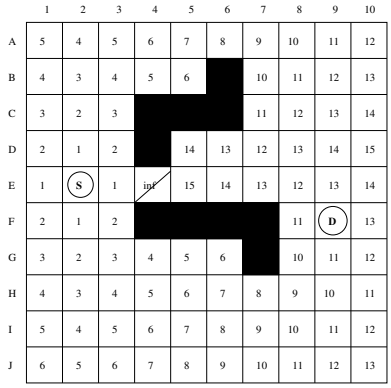

Figure 3: State after first step.

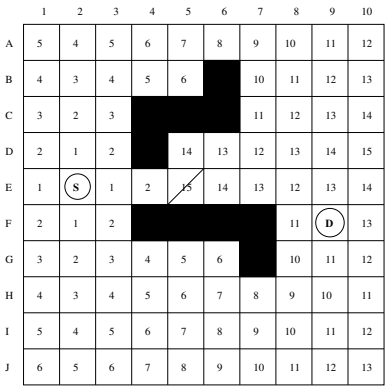

Figure 4: State after second step.

calculated, every traversable node has the following property: the cost value is equal to the minimum of all neighboring nodes plus the cost to move from the particular neighboring node to the current node. For example, the minimum cost value of all neighboring nodes of $C9$ in Figure 1 is 12.[3] Therefore, the cost value of $C9$ is $12+1=13$. It is easy to see that this holds for every node.

This fact can be used when changes in the environment are detected. When an obstacle is removed the tile becomes traversable. Then it is checked for conformance to the fact stated above, which does not hold. The cost value must then be recalculated, after which the fact must be checked for all incident nodes, and so on until no further unconformities are detected. The order in which unconforming incident nodes are updated may be determined randomly or in greedy fashion.

Similarly, when an obstacle is added (making its cost value ∞) its incident nodes may have invalid cost values in the new situation. They are handled in the same way as when an obstacle is removed.

Figures 3 to 6 illustrate the first 3 steps and the last step of DYNAMICSWSF-FP after the

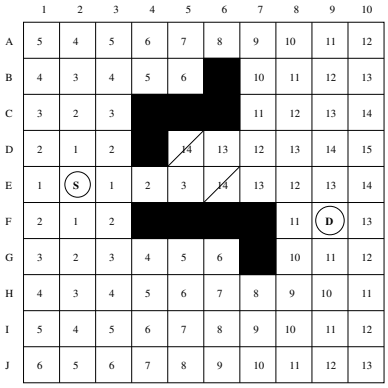

Figure 5: State after third step.

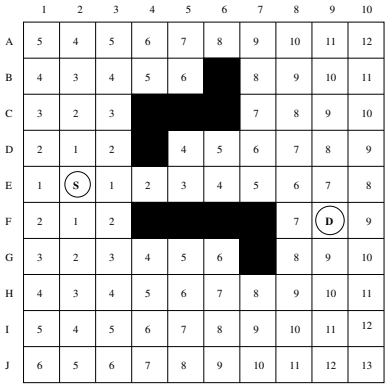

Figure 6: State after last step.

[3]Note that diagonal movement is not possible.

obstacle $E4$ has been removed (assuming that all cost values are available initially.) A diagonal line indicates that a cost value is incorrect and must therefore be recalculated according to the minimum of its neighbor values.

DYNAMICSWSF-FP is most efficient when changes occur near the target node because then the fewest costs have to be updated. If, on the other hand, changes occur near the starting node, it results in recalculation of a large part of the graph. The algorithm has even been known to perform less efficiently than Dijkstra's Algorithm in such cases because of the computational overhead of checked for unconforming nodes.

3.3.2 Lifelong Planning A*

DYNAMICSWSF-FP does not take any heuristics into account, and therefore still relies on a complete path to the target initially, which results in the exploration of many insignificant nodes when Dijkstra's Algorithm is used. When the terrain is not known, this can be an imaginary direct path which is modified along the way, but another possibility is to combine the incremental algorithm with heuristics. One algorithm resulting from such a combination is LPA*, or Lifelong Planning A* [2, 5].

The algorithm is almost identical to DYNAMICSWSF-FP and only differs in calculating the priority of nonconforming nodes to be updated and direction in which to search. Like A*, it uses a heuristic to prioritize nodes which are assumed to be closer to the target.

As with A*, the heuristic approach has both benefits and issues: it is faster, but may not be entirely correct. Additionally, it shares the performance problem of DYNAMICSWSF-FP when nodes near the starting node change. The algorithm's feasibility depends on the situation (where and how often might changes occur, must the path be optimal, etc.)

3.3.3 D*

D* is another incremental algorithm which, as the name suggests, is derived from A*. It uses incremental graph theory techniques to compute a new, optimal path to the target. The algorithm's (advanced) inner workings are beyond the scope of this paper. Interested readers are referred to [8].

However, it may be noted that one particular feature of D* is that it is most efficient when changes are detected near the current starting point in search space, which is the case with, for example, robots equipped with on-board sensors. This is a great benefit over LPA*. According to [8], performance is also very high. Still, this algorithm also relies on heuristic methods and may therefore not be feasible in all situations. Accuracy may be preferred over search speed.

3.4 Other Algorithms

There are methods for path planning that are vastly different from Dijkstra's algorithm and not presented in this paper. Among these are flow fields and network simplex methods (derived from methods for achieving optimal data flow.) They are outside the scope of this paper.

Another interesting fact is that in some cases other methods from AI like evolutionary methods and neural networks may be used to enhance the algorithms presented in this paper. For example, to calculate heuristics.

3.5 Further Techniques

Several techniques exist which can be used to further optimize and improve the ones discussed in the previous sections:

Multiple Resolutions [7] uses multiple resolutions of path planning in order ef-

ficiently calculate a global path (which would be very expensive, both computationally and memory-wise, with a large amount of small tiles) while at the same time performing fine-grained navigation at local level. Of course, this does make the system more complex and therefore computational resources and complexity should be weighed against eachother.

Aging When terrain is being traversed and observed at the same time, some information might be quite old. The vehicle may have been at a particular location days earlier, which means that things could have changed since then. With "aging", the certainty of observed tiles is multiplied by a value that is proportional to the distance the vehicle has travelled since the tile was last examined. This means that new data is preferred, but old data still has impact. A practical example might be that a particular route that has been visited many times recently will be preferred in order to be on the safe side (even if it is slightly longer), but when it is blocked by an obstacle, an older, less certain route, may be taken. [7]

Influence Mapping This is used often in computer games, but may apply to real scenarios as well (particularly where multiple robots are involved.) Certain information on environmental influence is recorded in an "influence map" that fits the world map. One example of influence mapping is death by enemy real time strategy games. When the enemy has built an ambush area in which many units perish, this effect will be taken into account. The units will, in time, prefer a different route.[4] [1]

Deadends Some areas of the map may not lead anywhere. In order to avoid recalculating the corresponding tiles at each step, they may be marked as unimportant. Of course, care has to be taken when using this technique. A tile within the deadend area may become the target at some point in time, which means that the vehicle will have to be able to go there anyway.

4 Conclusion

Path planning is the core activity of autonomous navigation. It depends on support tasks such as collecting terrain information and discretization of this information.

Several choices must be made for the discretizing task, among which: choosing an internal representation, choosing a tile size and shape, determining terrain types and specifying movement costs.

Several techniques exist for path planning, but the most predominant today are based on Dijkstra's Algorithm. Improvements over this original algorithm are generally in the area of faster search or the ability to reuse previous information. Faster search is mainly achieved by heuristics, as with A*. However, it is important to realize that the use of heuristics may cause a system to not always return the absolute shortest path. There is a tradeoff between speed and accuracy.

Incremental algorithms such as DYNAMICSWSF-FP, LPA* and D* enable reuse of previous search information. Depending on where in the world a change has occured, this may improve speed. Both LPA* and D* use heuristics as well.

When assuming a direct path to the target in the beginning, incremental algorithms may be

[4]The artificial player in the game "Red Alert" sometimes suffer greatly from not employing this technique. Units keep coming along the same route, only to meet their death.

used to navigate and explore unfamiliar terrain simultaneosly.

The existing algorithms may be further improved by using additional techniques like multiple resolutions and aging.

Finally, the amount of research that has been done on incremental search methods is quite limited [7]. There remains much to be explored and understood about the problem domain.

References

[1] F. Markus Jonsson. An optimal pathfinder for vehicles in real-world digital terrain maps. Master's thesis, The Royal Institute of Technology, Stockholm, Sweden, 1997.

[2] Sven Koenig, Maxim Likhachev, Yaxin Liu, and David Furcy. Incremental heuristic search in ai. *AI Mag.*, 25(2):99–112, 2004.

[3] Patrick Lester. A* pathfinding for beginners, April 2004. Retrieved May 2005 from http://www.policyalmanac.org/games/aStarTutorial.htm.

[4] Kyle Loudon. *Mastering Algorithms with C*, chapter 16, pages 472–485. O'Reilly & Associates, Inc., 1999.

[5] Kelly Manley. Pathfinding: From a* to lpa. University of Minnesota.

[6] Amit J. Patel. Amit's game programming information, 2004. Retrieved May 2005 from http://www-cs-students.stanford.edu/ amitp/gameprog.html.

[7] Sanjiv Singh, Reid Simmons, Trey Smith, Anthony (Tony) Stentz, Vandi Verma, Alex Yahja, and Kurt Schwehr. Recent progress in local and global traversability for planetary rovers. In *Proceedings of the IEEE International Conference on Robotics and Automation, 2000*. IEEE, April 2000.

[8] Anthony (Tony) Stentz. Optimal and efficient path planning for partially-known environments. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA '94)*, volume 4, pages 3310 – 3317, May 1994.