Optimizing Motion-Constrained Pathfinding

Nathan R. Sturtevant
Department of Computing Science, University of Alberta
Edmonton, Alberta, Canada T6G 2E8
nathanst@cs.ualberta.ca

Abstract

High-quality motion planning can be quite expensive to perform, yet is critical for many video games. As a result, many approximations are employed to reduce costs. Common approximations include pathfinding on a grid or on a navigation mesh instead of in a real-valued world. As a result, the paths that are found are not immediately appropriate for use, and often require post-processing, such as smoothing. An alternate approach is to incorporate pathfinding constraints directly into A* search. This approach is expensive and so it hasn't been widely applied in games. In this paper we analyze the complexity of planning with complex motion constraints and suggest optimizations which make such planning feasible for use in modern games. We propose the use of two ideas, abstract perimeter heuristics and intermediate search truncation, which can reduce the cost of search by an order of magnitude or more.

Introduction and Background

In this paper we are concerned with the study of pathfinding with more realistic motion models. As pathfinding is often quite expensive, especially considering the need for many different units to successively plan, the costs for planning with more realistic motion models is often considered too expensive for practical use. We analyze three motion models and propose a variety of optimizations that can be applied to make planning with more realistic motion models feasible.

There are a number of approaches that are currently used for pathfinding in games. For many games that can afford the memory costs, grids are attractive, as they are easy to compute and update. But, grids can have large memory overheads, and so a variety of other data structures have been used for pathfinding such as navigation meshes (Tozour 2002) or waypoint graphs. Navigation meshes and waypoint graphs are essentially an abstract representation of the underlying state space which reduce the cost of pathfinding. When grids are used, similar abstractions are often created as well (Sturtevant 2007). In general, paths are planned in the abstract state space, and then refined into paths in the actual state space.

These abstract representations do not adequately represent motion constraints which may be present with vehicular or creature movement. Thus, once an abstract path (eg one that follows obstacles constraints but not motion constraints) has been created, it must be transformed into a path which can be followed (and animated) by a character in the game.

A comprehensive introduction to pathfinding with more realistic motion constraints was described by Pinter (2001). This article has many practical suggestions which are necessary for building a good pathfinding system, but also contains many ad-hoc techniques which require engineering by hand. Our goal here is not to duplicate this work, but to complement it with the suggestion of additional techniques as well as a full study of these approaches.

Some of the ideas suggested by Pinter have been subsumed by the recent θ^* (Theta*) algorithm (Nash et al. 2007). θ^* attempts to do smoothing as part of the planning process instead of as a post-processing step. While this reduces some of the need for post-processing, it does not eliminate it completely. For instance, θ^* does not take motion constraints into account during planning. However, it is possible that θ^* can be combined with the ideas discussed here for even more efficient search.

Researchers in robotics have also been concerned with motion-constrained pathfinding, as motion planners must generate plans that can be executed by a robot. Thus, some of the heuristic-building ideas here overlap both with work in robotics (Knepper and Kelly 2006; Likhachev and Ferguson 2009) as well as problems like road layout (Mandow and de-la Cruz 2004), which has similar constrains.

This work makes the following contributions: We perform a systematic study of search with motion constraints, demonstrating that this search is not only feasible, but that it can be, given a proper heuristic, no more expensive than normal A* search. This is made possible first by breaking long paths into shorter paths using abstraction and refinement. The shorter paths can then be solved cheaply due to *abstract perimeter heuristics* and *intermediate search truncation*. These techniques will be described in more detail in the following sections.

Problem Definition

The grid-based pathfinding problem is defined as finding a sequence of legal actions from a start state (x_s, y_s) to a goal state (x_g, y_g), where x and y are positive integers. The agent can move to any of the immediate 8 neighboring cells, but

can only pass through a cell if it is not blocked. In this paper we extend the state of an agent to include heading and speed. Thus, the goal is to find a path from the current state, $(x_s, y_s, \theta_s, v_s)$, to a goal state, $(x_s, y_s, \theta_s, v_s)$, where θ and v are the current heading and velocity. In this case the movement must obey both map constraints (blocked cells) and movement constraints, which are implied by the given motion model. It is assumed that an underlying grid exists and planning occurs on this grid, although this does not require that a grid is used as the underlying representation of the map. Motion is restricted to grid centers, although this can be relaxed slightly in practice.

We define and use three motion models. As there are a wide variety of design decisions that can be made regarding movement, these are used to show the generality of the techniques introduced, and to understand better how design decisions affect the computation costs for planning in a game.

Our three models are:

- **Humanoid model**: The humanoid model only moves forward, but can turn when stopped. There are two speeds, walking and running. Turns are discretized into 16 directions (22.5°), as shown in Figure 1, and characters using the model can move up to 67.5° in a single action.
- **Tank model**: The tank model can move forward and backwards, but only has a single speed. Turns for the tank model are discretized into 24 directions (15° resolution) and characters using the model can turn up to 45° in a single action.
- **Vehicle model**: This is the most complicated model. Vehicles can move in forward or reverse, but cannot turn when stopped. In reverse the model can only move slowly, but has three speeds when moving forward. The vehicle can only increase or decrease speed by two levels per action. Turns are discretized into 16 directions. The vehicle can turn faster at slow speeds than at high speeds. At high speeds no turning is allowed. At a medium speed 22.5° turns are allowed, and at low speeds up to 67.5° turns are allowed. In reverse turns of up to 45° are allowed.

It is important when designing such models that there is always a way to reach even difficult locations in the world. If a model can turn when stopped, this will always be possible. As a design issue, it may be worthwhile to add a turn action even to units which are not able to turn in place. If this action has high cost, then a search will only use this action if absolutely necessary. In this way the use of the action by the planner will help indicate that the model is being asked to perform a maneuver which is too difficult for the model constraints.

State-space analysis

Adding motion constraints can be seen as adding extra dimensions to the search space, however these extra dimensions are relatively constrained. Assuming there are N states in the original state space before adding motion constraints, we compute the size of the state space that results from imposing the motion model onto search.

In the humanoid model there are three speeds (stopped, walking and running) and 16 possible headings. This means

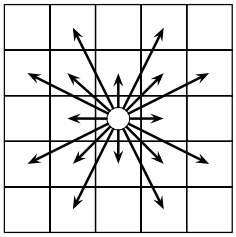

Figure 1: Humanoid and vehicle models: 16 possible movements staying on an underlying grid.

that the size of the state space increases to $48N$. The tank model also has three possible speeds. But, there are 24 possible headings, so the new state space has $72N$ states. Finally, in the vehicle model there are five speeds (backwards, stopped, and three forward speeds) as well as 16 possible headings, meaning the state space grows by a factor of 80 over the original state space.

While these factors significantly increase the size of the state space, the cost of storing the representation of the underlying map does not change. They can, however, increase the average branching factor. As long as we can restrict the number of states explored during search adding motion constraints does not add significant other costs.

Search Optimizations

There are a variety of optimizations required for an efficient implementation of directional pathfinding. We describe these optimizations here and then will perform experiments to measure their effectiveness.

We will search with each of these models using A* (Hart, Nilsson, and Raphael 1968). We assume the reader is familiar with A*, but a few features of the algorithm are particularly relevant to this work. A* expands nodes by f-cost, where $f(n) = g(n) + h(n)$. A common variant on A* is weighted A*, which places a weight, w, on the heuristic value. Weighted A* searches with $f(n) = g(n) + w \cdot h(n)$. Weighted A* tends to make greedy moves early in the search when the h-cost is decreasing. Later in the search when many nodes have similar h-costs, more emphasis is placed on minimizing the g-cost.

A* maintains two lists, an open list of nodes being considered for expansion and a closed list of nodes which have been expanded. A* will never re-open a node off the closed list if the heuristic being used is consistent. In an undirected domain, an consistent heuristic is one for which the heuristic between two nodes never changes more than the edge cost (Martelli 1977). The heuristic we propose here is not always consistent, so it is important to understand this, although in practice the heuristic is nearly always consistent.

Most simple implementations of A* do not consider the possibility of re-opening nodes. But, some combinations of motion models, heuristics, and weighted A* search make it possible for a shorter path to a node to be discovered after the node was closed. In some cases it is advantageous to re-open closed nodes while in other cases it is not. While not

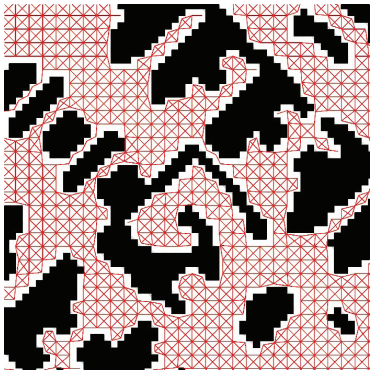

Figure 2: Abstraction overlaying a portion of a map.

discussed in detail here, it is an important to be aware of.

High-Level Optimizations

The first set of optimizations are relatively high-level and apply to any pathfinding operations, not just directional pathfinding.

Finding long paths is expensive and so, as much as possible, a pathfinding engine should only be used to find short paths in the world. This is most often done with an abstract representation of the world for which high-level plans can be built. We give an example of this in Figure 2. This figure shows a map with open spaces in white. The lines across the map show the abstract graph processed from the underlying state space. A path is originally found in this graph. Then, each edge in the graph can individually be refined into a path in the underlying state space, with intermediate goals at the nodes in the graph.

A high-level plan usually defines a corridor in the world in which a model can be expected to travel, although when turning the abstract path into a walkable path, it may or may not be desirable to constrain the model to stay within the corridor (Bulitko et al. 2007). Additionally, the high-level path does not necessarily need to take into account the directional constraints. If there are areas in a map which are problematic for a particular model, they can be tagged and avoided by the high-level planner.

The second optimization we call *intermediate search truncation*. When refining abstract paths, it is not necessary to reach the goal state, as it is only an intermediate step on the complete path. Instead, any state near the goal is sufficient. Thus, the search can be terminated not when the goal is reached, but when any state near the goal is expanded. Particularly with directional constraints, the final steps of orienting a model on the goal location can be quite difficult. Doing so optimally may require actions early in the search which are not immediately obvious, which increases the cost of search. This work can be avoided by terminating the search when it nears the intermediate goal but does not exactly reach it. In a navigation mesh this can be handled, for instance, by stopping the search when it enters the same polygon as the intermediate goal. In a grid-based abstraction this can be handled by terminating the search when it reaches the same sector (square) as the intermediate goal.

Finally, many searches can be improved simply by using weighted A*. Weighted A* allow small suboptimalities early in the search and focuses its effort on finding good paths near the goal.

Heuristic Optimizations

It is difficult to build a good heuristic for directional search problems. A simple heuristic would be to take the straight-line distance between the start and goal locations, dividing this by the maximum speed. This is the minimum time possible to travel between the start and the goal, but it is clearly only a lower bound if the model is beginning and ending at rest, as it takes time to accelerate and decelerate. However, straight-line distance is a weak heuristic in a problem without directional movement. When a motion model causes the state space to grow by an order of magnitude, it becomes quite costly to have a poor heuristic.

The first possibility for improving the search is to do a reverse A* search and use it for heuristic values, as has been done for cooperative search between multiple agents (Silver 2005). In this way, the distance portion of the heuristic will be exact, and only the directional constraints of the search will not be taken into account by the heuristic. This approach offers improvement, but the most difficult part of the search is the restrictions imposed by the model dynamics, so this is most important to model with the heuristic.

It is possible to analyze the dynamics of the world by hand in order to build a better heuristic function. This process, however, is difficult and error prone. Once completed, it makes it difficult to modify the motion parameters, as the heuristic analysis will have to be redone. Instead, it is more desirable for this process to be performed automatically.

An idea which has been used in other domains, such as the sliding tile puzzle, is perimeter search (Manzini 1995). In perimeter search, a perimeter of exact heuristic values is built though reverse search around the goal state. Then, instead of searching to the goal state, the search is performed towards the perimeter nodes. When a perimeter node is found, the exact distance to the goal is known and the search can optionally terminate. (There are termination conditions for optimality which we do not discuss here.) This idea has been applied in domains where there is only a single goal state, as opposed to pathfinding, where there are many different goal states.

We use a modified version of the idea that we call an *abstract perimeter heuristic*. First, we abstract the state space by assuming that the goal is in open space with no barriers. In this case, we can build a perimeter by doing a breadth-first search in all directions to a fixed distance, storing the resulting distances efficiently in an array. This is analogous to an endgame database in a two-player game, which stores exact information on how to finish off a game. The memory required by the perimeter depends on the model being used, but, given a number optimizations, small perimeters can be stored in reasonably amounts of memory.

The memory required for building a heuristic radius 15 (31×31 table) with no optimization is shown in the first line

	Humanoid	Tank	Vehicle
Full Table	8.6MB	19.6MB	24MB
Speed-Limited	2.9MB	6.5MB	4.8MB
Symmetry-Reduced	360k	810k	600k
Rotated	90k	135k	140k

Table 1: Memory required by radius 15 extended perimeter heuristics.

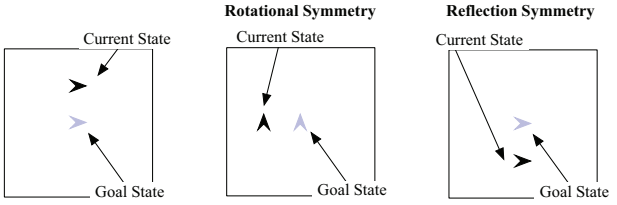

Figure 3: Symmetric lookups in abstract perimeter heuristic.

of Table 1. As an example, the memory for the humanoid model depends on the number of x/y cells in the table (961), each of which must have information for all possible headings and speeds (48). Additionally, one table is needed for all possible goals and headings (48). Assuming 4 bytes per entry, the total memory is $961 * 48 * 48 * 4 = 8.6$MB.

Memory Reductions

In an open world this perimeter provides a perfect heuristic for any possible goal, although the memory overhead is costly. The overhead can be reduced in several ways. First, assuming that the goal is always to stop at a particular location, a separate perimeter is only needed for each angle possible in a goal state, not each speed. As shown in the second line of Table 1 this reduces the memory needed by a factor of three for the humanoid and tank models, and a factor of five for the vehicle model. For intermediate segments of a longer path the intermediate path truncation, which can be enhanced by further trimming, helps to avoid unnecessary slowing in the middle of a path.

The tables also contain rotational and reflectional symmetries. These are illustrated in Figure 3. The heuristic distance for the left-most state is clearly the same as both the rotated and reflected states. For problems with a grid aligned (45° or 90°) goal heading, the reflectional symmetry can be used to eliminate half the entries in the abstract perimeter heuristic.

With 16 headings, a single perimeter heuristic is needed as well as two half-sized perimeters due to reflectional symmetries. With 24 headings, two full perimeter heuristics are needed along with two half-sized tables. Taking these symmetries into account reduces the total memory needed to under 1MB.

There is one final optimization which further reduces memory. Notice that the perimeter heuristic assumes an empty map, and that the grid is arbitrarily overlaid on an underlying map. Therefore, instead of building a heuristic for each possible goal heading, it is possible to (virtually) rotate the map for every search problem so that the heading at the goal is always the same. Then, a single table can be used for any search problem. This is the final entry in Table 1, and should be affordable in most games.

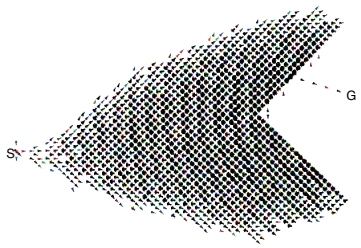

Figure 4: Visualization of search with perimeter heuristic.

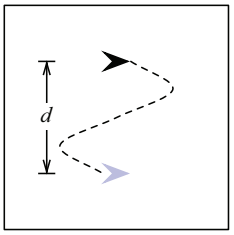

Figure 5: Experiment 1 setup.

Extended Abstract Perimeter Heuristics

The abstract perimeter heuristic is a perfect heuristic for open space, but when obstacles are present it will be less accurate. Additionally, it is possible that many nodes may have to be expanded which are beyond the heuristic perimeter. This is seen in Figure 4. There are many nodes expanded up until the perimeter, with relative few after that. The solution to this is to scale the perimeter beyond the initial area. Suppose that a unit is initially 22m from the goal, while the perimeter is only built to 15m from the goal. Then, the heuristic value at 22m will be roughly twice the heuristic value from 11m. But there is a margin of error. So, we choose a custom weight, w_m, for extended distances for each model, and use this to scale heuristic values to a distance twice the size of the actual heuristic table. We call this an *extended* abstract perimeter heuristic.

This approach is now general and is quite effective in most situations. It only requires a small pre-computation and a small memory footprint, but significantly reduces the search overhead in most situations. The one time when this approach fails is when the goal is highly constrained. In this case it may worthwhile to dynamically build a perimeter or to reverse the direction of the search, always searching the direction that is least constrained. However, we leave this concept for future work.

Experimental Results

We have proposed a number of methods for searching with directional constraints. Our goal is to evaluate each of these

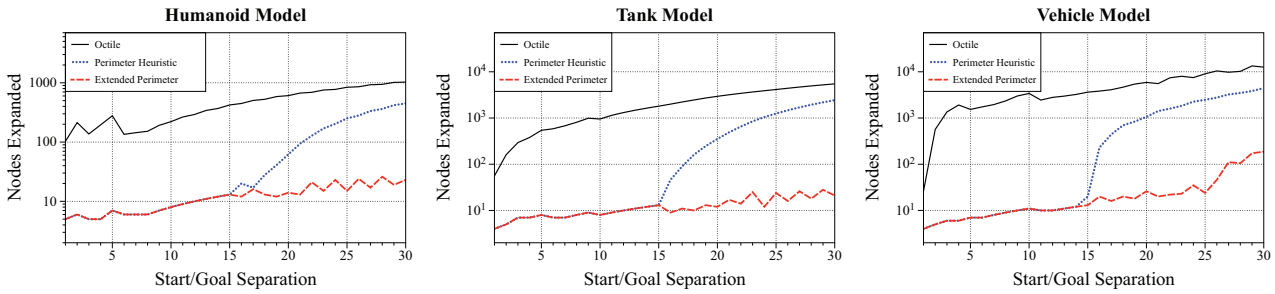

Figure 6: Work done planning for the humanoid, tank, and vehicle models for empty-map planning.

methods independently and as part of a complete search implementation. We begin with optimal search.

Search in an Empty Map

The first problem we test is pathfinding in a open world. This is trivial in normal grid-based pathfinding, but much more difficult in an environment with directional constraints. With each of our model types we being with a model facing due east, and ask the model to move a fixed distance south, arriving facing due east. This is illustrated in Figure 5. We then vary the distance travelled and the combination of techniques used, measuring the cost of search.

We use three possible heuristics with each model: The first, (octile) is the straight-line distance divided by the maximum speed. The second is the abstract perimeter heuristic. The third is the extended abstract perimeter heuristic. We scale the heuristic by a factor of two in the tank and humanoid models, but by a factor of 1.5 for the vehicle model.

Figure 6 shows the results with the humanoid, tank and vehicle models. The x-axis is the distance between the start and goal state. The y-axis is the number of nodes expanded. Note that the y-axis is on a logarithmic scale, meaning that even for small distances, the octile heuristic can expand thousands of nodes.

The perimeter heuristic is very effective when searching from inside the perimeter, limiting the search to around approximately 10 nodes. Beyond the perimeter, the extended heuristic is also effective at minimizing the amount of work. We tried adding weighted A* on top of these techniques, but the gains were insignificant, so we do not report them here. Weighted A* on its own without an improved heuristic is also effective, but not nearly as effective as the perimeter heuristics.

We show a visualization of the effect of the non-extended perimeter in Figure 4. In this model a vehicle starts at S and plans towards G. It is clear where the heuristic values are available, as they immediately cutoff the search. Only a single path is explored within this region, which leads directly to the goal.

Full Pathfinding on Real Maps

In this section we consider the cost of full-path planning. We took a set of 75 maps from a commercial game and asked an agent to move to and return from 200 pairs of waypoints

Figure 7: Example map used in experiments.

on each map. This results in 30k total paths, over which 300-500k planning steps were performed, depending on the model. The underlying search was performed with weighted A* with a weight of 1.2. Abstract paths were formed over sectors of size 6 (Sturtevant 2007), and we experimented with refining abstract paths of length 3 and 4, which means each low-level pathfinding task was length 12-18 or 18-24 respectively, and low-level searches were cut off when an agent was within 3 steps of the goal.

We compare the work done by the octile heuristic and the extended abstract perimeter heuristic with and without intermediate search truncation for each of the models. The results from the shorter paths are found in Table 2 and results from longer paths are in Table 3. We report the median cost, as well as the 90 and 99th percentiles. These percentiles are important, because 99% of all pathfinding operations will cost less than this. Thus, if the 99th percentile cost is small enough, the technique is affordable in a game.

Without intermediate search truncation all costs are too high too be feasible in real-time games. With shorter paths and the extended perimeter heuristic, all models should be affordable. But, as the path lengths increase to the 18-24 range, the models become less feasible. This is primarily due to obstacles in the world, as they are not accounted for in the heuristic. The longer the path, the more room for obstacles.

Regardless, the techniques described represent a major reduction in planning costs for motion constrained path planning. For many of the models work is reduced a factor of 100 or more over planning with an octile heuristic and no intermediate search truncation.

	Humanoid		Tank		Vehicle	
	Octile	Perimeter	Octile	Perimeter	Octile	Perimeter
No intermediate search truncation						
Median (50th perc.)	6874	2062	1705	79	9638	595
90th percentile	13197	4373	8138	776	30164	1051
99th percentile	24284	6538	12429	1308	34394	12429
Intermediate search truncation						
Median (50th perc.)	25	13	7	7	82	14
90th percentile	82	30	17	10	248	22
99th percentile	1638	260	981	105	3065	104

Table 2: Results from full planning task, planning segments length 12-18.

	Humanoid		Tank		Vehicle	
	Octile	Perimeter	Octile	Perimeter	Octile	Perimeter
No intermediate search truncation						
Median (50th perc.)	12325	4171	4591	137	14234	909
90th percentile	28387	8189	22482	1290	41325	1750
99th percentile	50741	17185	36023	4764	48279	3452
Intermediate search truncation						
Median (50th perc.)	56	162	11	11	157	64
90th percentile	338	565	103	98	770	273
99th percentile	5486	2541	4030	1683	7454	746

Table 3: Results from full planning task, planning segments length 18-24.

Conclusions

This work has shown that pathfinding with motion constraints is feasible under several conditions. First, an abstract path must be used to divide the planning problem into smaller, more feasible components. Then, intermediate search truncation and extended abstract perimeter heuristics can be used to reduce the search effort to reasonable levels. The final cost is well within commercial game constraints.

There are a number of open questions. The quality and cost of low-level paths is dependent on the high-level paths from which they are derived. Work is needed to investigate how a high-level path can be optimized for low-level use. Additionally, the heuristics assume the world is empty. It is likely that information about static obstacles in the map can be used to improve performance and dynamically increase the accuracy of the heuristics. These techniques can be applied to any type of motion constraints for movement. We hope to see games using these technologies and plan to develop playable prototypes which use them.

Acknowledgements

This work was supported by the Alberta Informatics Circle of Research Excellence (iCORE). Thanks to Oliver Dou for early discussion and prototyping of this work.

References

Bulitko, V.; Sturtevant, N.; Lu, J.; and Yau, T. 2007. Graph abstraction in real-time heuristic search. *Journal of Artificial Intelligence Research (JAIR)* 30:51 – 100.

Hart, P.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4:100–107.

Knepper, R. A., and Kelly, A. 2006. High performance state lattice planning using heuristic look-up tables. In *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*, 3375 – 3380.

Likhachev, M., and Ferguson, D. 2009. Planning long dynamically-feasible maneuvers for autonomous vehicles. *International Journal of Robotics Research (IJRR) (to appear)*.

Mandow, L., and de-la Cruz, J.-L. P. 2004. Model and heuristics for the shortest road layout problem. In *ECAI*, 740–744.

Manzini, G. 1995. Bida: an improved perimeter search algorithm. *Artif. Intell.* 75(2):347–360.

Martelli, A. 1977. On the Complexity of Admissible Search Algorithms. *Artificial Intelligence* 8(1):1–13.

Nash, A.; Daniel, K.; Koenig, S.; and Felner, A. 2007. Theta*: Any-angle path planning on grids. In *AAAI*, 1177–1183.

Pinter, M. 2001. Toward more realistic pathfinding. In *gamasutra.com*.

Silver, D. 2005. Cooperative pathfinding. In *AIIDE*, 117–122.

Sturtevant, N. R. 2007. Memory-efficient abstractions for pathfinding. In *AIIDE*, 31–36.

Tozour, P. 2002. Building a near-optimal navigation mesh. In *AI Game Programming Wisdom. (S. Rabin, ed.)*, 171–185.