VARIABLE RESOLUTION A*

Kyle Walsh
RemoTV, Inc.
258 Bradley Street, Suite 2F
New Haven, CT 06510
kwalsh@remotv.com

Bikramjit Banerjee
University of Southern Mississippi
Hattiesburg, MS 39406
Bikramjit.Banerjee@usm.edu
http://www.cs.usm.edu/~banerjee

KEYWORDS: Game AI, Path-finding, A*

ABSTRACT

We argue that A*, the popular technique for path-finding for NPCs in games, suffers from three problems that are pertinent to game worlds: (a) the grid maps often restrict the optimality of the paths, (b) A* paths exhibit wall-hugging behavior, and (c) optimal paths are more predictable. We present a new algorithm, VRA*, that varies map-resolution as needed, and repeatedly calls A*. We also present an extension of an existing post-smoothing technique, and show that these two techniques together produce more realistic looking paths than A*, that overcome the above problems, while using significantly less memory and time than A*.

INTRODUCTION

Path-finding for intelligent Non-Playing Characters or NPCs (henceforth agents) is one of the classic problems in interactive games. Traditionally, the predominant approaches are either offline precomptuation (e.g., Floyd-Warshall's all-pair-shortest paths) or on-line path-finding with A* [5]. In this paper, we focus on the latter approach, and address some issues with A*, pertaining to the game community.

There are at least three issues with A*, particularly relevant to gaming, that have not been addressed adequately, to the best of our knowledge. These are:

- Although A* paths are theoretically *optimal*, the underlying grid structure of the walkable surface often limits the optimality of the resulting path. For instance, even if a straight line path exists between the source and the goal nodes, the A* path may be segmented (see Figure 1, left & middle). The resulting paths look unrealistic [8]. The game industry handles this problem by post-processing the A* path, by techniques such as rubber-banding and smoothing [1, 9]. Our position on this approach is that if we are to rely on post-processing, then it might be sensible to spend less time on the A* search. Other relevant approaches, such as Theta* [8] and Field D* [4] solve this problem by propagating information along the edges of the grid without restricting the paths to grid edges, but they are no faster than basic A*, and also suffer from other problems, noted below.

- A* (as well as Theta* and Field D*) produces the so-called *wall-hugging* behavior. Shortest paths tend to skirt walls or other obstacles while passing as close to them as possible. This leads to agents *hugging* walls while navigating around them. See Figure 1 (right) for an illustration. This problem is typically dealt with by surrounding obstacles with a pseudo-obstacle band where agents are not allowed to tread. However, this solution calls for tedious manual augmentation of the maps, that we seek to avoid.

- A major consequence of the optimality of A* paths is that they tend to be unique, and hence *predictable*. Game players who can predict the possible paths that AI agents can take, can find it easy to lay ambushes or otherwise utilize that predictability to their advantage, ultimately leading to monotonicity and a reduction in the players' interest over time. However, if the AI agents can make their paths less predictable, it can produce more challenging and interesting game play on the part of the players.

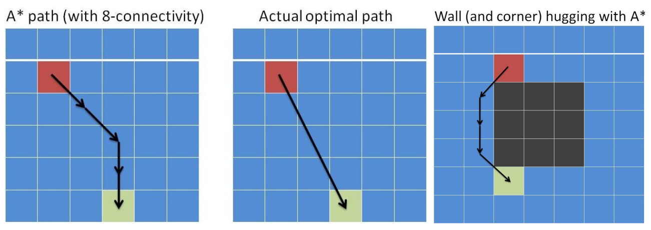

A* path (with 8-connectivity)
Actual optimal path
Wall (and corner) hugging with A*

Figure 1: Some problems with A* in game environments. Left & Middle: Optimality of A* paths are constrained by the underlying grid; Right: Wall hugging with A*.

Clearly, less predictable paths are unlikely to be optimal. We prescribe sacrificing optimality for greater variation in game-play and longer-lasting player interest. It is noteworthy that forsaking optimality does not make path-finding trivial. A game agent still needs to find a *realistic looking* [8] path between the start and goal locations. In this paper, our objective is to prescribe a technique that

- is faster than A* in producing a quick, dirty (but valid) path,
- employs improved rubber-banding technique to refine this path into a *realistic looking* final path,

- is highly unlikely to produce wall-hugging behavior,
- can produce paths that are unpredictable to the players.

We introduce the Variable Resolution A* (VRA*) algorithm for 2D grid surfaces. The idea is to gradually raise the resolution of the map as needed, to do A* searches, instead of applying A* to the maximum resolution map right-away. The idea is similar to *iterative deepening A** (IDA*) [10], but instead of increasing cutoff traversal costs, we use increasing resolution in each iteration. It is likely that a lower resolution will yield a valid path, and the total number of expansions of all A* searches up to that resolution may be fewer than a full-blown A*. The justification comes from existing analysis of IDA*, where the last iteration is usually almost as expensive as the total cost. So if VRA* manages to find a path with less than the full resolution, then the saving should be substantial. The resultant path may or may not be optimal. For instance, in Figure 1(a), VRA* will produce a valid path with the lowest possible resolution (viz. two cells/nodes, one including the start point, and the other including the goal), and this path will *also* be optimal compared to the longer path produced by applying A* on the highest resolution map. However, in many cases, the path produced by VRA* could be longer than A*, and less predictable.

A second contribution of this paper is to extend the rubber-banding approach from [1] to paths that contain line segments, rather than paths that are sequences of cells on a grid map. Due to lack of space in this paper, we handcraft just one map, that showcases the characteristics of VRA*, and show that this rubber-banding technique produces a reasonable final path.

BACKGROUND: A* SEARCH

A* [5] is one of the most popular path-finding techniques in interactive games. It is fundamentally an informed search technique [10], using problem-specific knowledge to find solutions more efficiently than uninformed/blind search. Given a graph, a source/start node and a goal node, A* attempts to find a minimal-cost path between the source and the goal nodes, by using an evaluation function to select the lowest scoring nodes for expansion, i.e., the nodes that are most promising to be on the optimal path. Expansion of a node produces its children (i.e., adjacent nodes), and an accumulation of such nodes that have not been expanded themselves, is called the "fringe". Usually the fringe is stored in ascending score values in a suitable data structure, such as a min-heap. Heuristic functions (constructed from domain knowledge) are often used as the evaluation functions ($h(n)$ for node n). They yield the estimated cost of the cheapest path from a given node to a goal node. An admissible heuristic [10] never overestimates the cost to reach a goal. It assumes the cost of solving a problem is less than it actually is. Using an admissible heuristic in an informed search algorithm prevents exploration of paths that are costlier than the optimal path. The total cost (or the evaluation function) of a node n is given by $f(n) = g(n) + h(n)$, where $g(n)$ is the actual cost of the path from the start node to n. Simply put, the cost $g(n)$ takes into consideration moves made up to the current point, while the heuristic attempts to estimate the future cost, usually considering the proximity of the current location to the destination. The heuristic can be calculated in several different ways, but the commonest in game programming is the Euclidean distance, since it is guaranteed to never overestimate the actual path length, whether 4-connectivity or 8-connectivity is considered.

COMPLEXITY REDUCTION OF PATH-FINDING

In path-finding, there are many cases in which a large number of neighboring nodes in a region do not carry much distinctive path information; e.g., if the map is such that an NPC must cross a swamp on the way to its goal, several paths across the swamp may have only slightly different costs. It may be wasteful to analyze the swamp at a fine resolution, but this cannot be avoided in an A* search. Although A* is a fast and popular method for traversing these types of spaces, there is still room for improvement. In fact, hierarchical A* [6] exploits this characteristic to abstract such similar nodes into larger zones, to reduce the number of nodes, and consequently, the complexity of A* in a bottom-up fashion. However, this requires either the prior knowledge, or prior exploration of the terrain. In contrast, we propose a top-down approach, called VRA*, that exploits low resolution wherever possible, and only uses higher resolution where necessary (such as in the vicinity of irregular obstacles). VRA* only acquires enough terrain knowledge to find a valid path.

Similar top-down approaches have been used in the past. Tozour [11] has used *quad-trees* for efficient path-finding. In this approach, the map is divided into four rectangles, and then each rectangle that includes any obstacle is further subdivided into four rectangles, and this process continues until no further subdivision is necessary. Since paths are constrained to pass through the centroids of any quad, they often look unrealistic. One solution to this problem has been known in robotic navigation. Framed quad-trees [3] impose high resolution cells along the borders of large quads, but the resulting improvement in path quality comes at the price of increased number of search nodes. Moreover, the quad-tree approach also requires prior analysis of the terrain, much of which may be unnecessary unless a search looks through these regions. In contrast, VRA* does not require any prior analysis. It creates a variable resolution map on the fly, and only resolves those regions that are pertinent to the path being searched. On the flip-side, VRA* does suffer from the same limitation to the path quality as quad-trees, but we propose an extended rubber-banding approach to mitigate this problem, instead of increasing the search-cost as in framed quad-trees.

The basic idea of increasing the search resolution comes from the Parti-game algorithm [7]. This algorithm exploits techniques from game theory and computational geometry to adaptively partition a high dimensional space in variable resolution, for fast reinforcement learning. To the best of our knowledge, this idea has never been applied in conjunction with A*. We exploit a line rasterization technique from computer graphics for this adaptation, and show that along with our proposed rubber-banding for post-smoothing, we can produce reasonable-looking paths at a lower cost than A*.

VRA*

In this section we present an algorithmic overview of VRA*. The details of the individual steps are presented in subsections later. We call the search space that A* would normally search on, the *highest resolution search space* (or HRSS), and it can be of any size. Before running VRA*, as is customary in A*, a cost table is generated based on the connectivity of the graph at the highest resolution. By using connectivity data from the highest resolution as used by A*, it is ensured that detecting obstacles at lower resolutions will be consistent between A* and VRA*. After generation of the cost table, VRA* splits the search space into two nodes: one containing the origin point, and the other, the goal point. The area of these cells need not be identical; the important part is only that there are two cells. This lowest resolution gives the *current resolution search space* (or CRSS) at the start.

Following these preprocessing steps, an A* search is performed on the CRSS (i.e., the two node search space). The cost of traveling between nodes at the CRSS cannot be simply looked-up as in A*, because the cost table corresponds to the HRSS, not the CRSS. To solve this problem, we use a line rasterization approach at the HRSS, to compute the link costs at the CRSS. If beginning at the start point, the rasterized cost starts from that point; otherwise, the rasterized cost starts from the centroid of the current cell. Similarly, if aiming for the goal, the rasterized cost ends at that point; otherwise, the rasterization aims for the centroid of the target cell.

The rasterization produces both the link costs between traversable nodes at the CRSS, as well as an indication of which nodes do not have any link between them (i.e., infinite cost links). Thus it produces a graph with all link costs at the CRSS. If A* on this graph fails to return a path, then one or more cells are split, to produce a revised CRSS. Since the new CRSS is only slightly different from the previous CRSS, some rasterized costs that are unchanged can be reused, but others have to be re-computed. Splitting of selected nodes in the search space would continue until either a path is found, or the highest resolution is reached, i.e., CRSS = HRSS. If no path can be found at the highest resolution, then no path exists. However, we expect to find a valid path, if it exists, long before the highest resolution is reached.

Cost Table Generation

Before any code is run related to VRA* or A*, obstacles must be placed on the HRSS grid, as well as the start and goal points selected. After this is complete, the HRSS is processed into a two-dimensional array that serves as a cost table. The array indices corresponds to the (x,y) coordinates of a tile of the HRSS. A tile is either traversable, or an obstacle tile. Traversing between open tiles carries unit (or user-defined) cost. Tiles that have obstacles on them are given infinite cost. Since all the obstacle checks are done in this preprocessing step, computation time is saved during runtime because the algorithm need only index into the cost table to check for collisions, instead of running an obstacle test several times.

Generating the Start and Goal Nodes

Start and goal nodes are generated by computing the midpoint between the start and goal points, and then comparing the x and y distances between the start and goal points to determine which axis to split on. If the x distance is greater, then the split line will be generated at the x coordinate of the midpoint, and likewise if the y distance is greater. This partitions the map into two regions, and creates our initial nodes in the search space. A* will be run on this CRSS but in all likelihood, a straight path will not exist between the start and the goal nodes, unless all obstacles are out of sight between these nodes.

Rasterized Link Cost

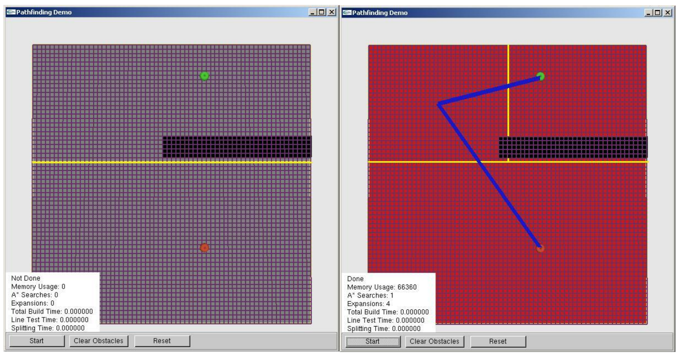
The computation of the rasterized link cost follows the well-known Bresenham's line rasterization technique [2], to find a rasterized path between the centroids of two nodes on the CRSS. Bresenham's line algorithm has been popular in raster graphics, to render a line on the screen pixel by pixel. In our case, the cells in the HRSS act as the pixels.

The rasterized path is a sequence of cells on the HRSS that the agent would have to step through, to travel between two nodes in the CRSS, in an approximately straight line. However, there could be obstacle cells on this path, but this can be easily checked with the cost table. If the cost look-up at any point is infinite, this means we have encountered an obstacle, and the nodes being tested are not connected. The test returns failure, and the nodes are marked for splitting. If the cost is not infinite, then the test succeeds, and a path between the centroids of the two nodes (in the CRSS) exists. The cost at each cell on the rasterized path, from the HRSS cost-table, is summed and used as the overall cost of traversal between the two points.

Splitting Cells for Variable Resolution

As mentioned before, the cell splitting method was inspired by the work of Moore and Atkeson in their Parti-game algorithm [7]. Their algorithm would start with the lowest possible resolution of the search space, and increase resolution of cells when and where necessary, by splitting cells. These splits would occur around obstacles, or as they described, on the borders of *winning and losing* cells. Winning cells were cells that were traversable, and losing cells were cells of infinite cost. Only splits that were needed were performed, and the algorithm continued on its way until the goal was reached.

VRA* puts this same logic to use with a little variation: it only splits one cell. The choice to split just one cell was made as an optimization because splitting several cells is often unnecessary in VRA*, because we are performing A* searches on each search space instead of just continuously navigating the same search space like Parti-game does. Cells marked for splitting by the line tests that occurred in the previous A* search are put into a list. If a path is not found, then before the next A* search is called, only the first cell on the list is split. This cell is either a cell along an obstacle, or a cell containing an obstacle. The cell is split into two, along its longest axis, and the two new cells are added to the list of cells to produce the new CRSS for the next A* search. The centroids of each new cell are computed and stored so the rasterization test has a target point to start from and aim for, in its execution. Figure 2 depicts an example of a cell split



```
Not Done
Memory Usage: 0
A* Searches: 0
Expansions: 0
Total Build Time: 0.000000
Line Test Time: 0.000000
Splitting Time: 0.000000
```

```
Done
Memory Usage: 66360
A* Searches: 1
Expansions: 4
Total Build Time: 0.000000
Line Test Time: 0.000000
Splitting Time: 0.000000
```

Figure 2: Illustration of the cell-splitting process.

before and after an A* search. On the left is the initial VRA* search space: a start point (in red), a goal point (in green), and an obstacle (in black). The start and goal have their own cells (outlined in yellow), and only two cells are present at this point. Notice that the two cells are not of equal areas, rather the partition occurs along the middle of the y-distance between them (because the y-distance is larger than their x-distance). Since a straight path does not exist between these two points, both cells will be marked for splitting, but ultimately only one will be split. Assuming that this is the top cell, on the right, the end result of the split is shown. As explained above, the non-start and non-goal cells' centroids are used as the points of interest for the rasterization test.

Finding Neighbors

In A* on a uniform grid, finding the neighbors of a cell is simple. In 4-connectivity cases, this calls for simply checking the 4 neighboring directions of the current node. If they are reachable, then they are linked up to the parent node as neighbors. The same holds for the 8-connectivity cases, when incorporating diagonal neighbors. In VRA*, however, nodes can be of varying sizes, and there may no longer be any simple relationships at their junctions. For instance, a cell could have one or many neighbors to its left. So the edges of nodes must be checked for overlap, to identify neighbor status, and then the rasterization test is called afterwards as the final check on neighbor connectivity. Both checks perform very fast in our VRA* implementation, and account for very little in the overall path-finding times reported.

POST-SMOOTHING

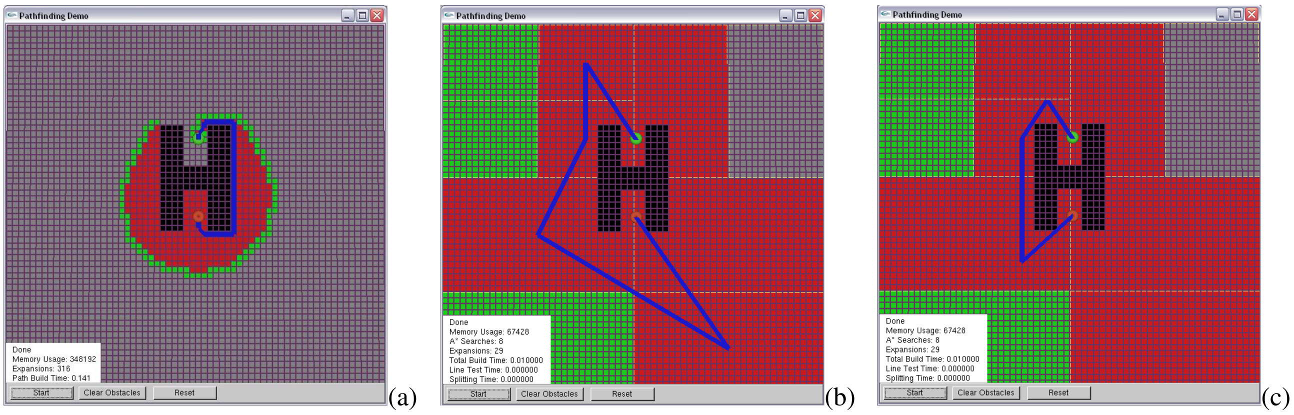
The paths produced by VRA* are non-optimal. In most cases, it is possible to improve the paths through the process of *post-smoothing* as applied to A* [1]. We use the basic idea of this *post-smoothing* algorithm (which was developed for the highest resolution grid) and extend it to VRA* where the final grid may have different resolutions in different regions. Assuming that VRA* produces the path $\langle p_1, p_2, \ldots, p_n \rangle$, the post-smoothing algorithm accepts this path as input and returns an edited path where p_1 and p_n (i.e., the start and goal points) are left unchanged, but some intermediate points are possibly changed or deleted. The main logic is similar to [1], where if a line-of-sight exists between p_i and p_{i+2} then p_{i+1} can be eliminated, to produce a shorter path (rubberbanding). We replace the line-of-sight test with the rasterization test.

If p_{i+1} cannot be deleted as stated above, then unlike [1], we try to shift this point as close to p_{i+2} as possible, using a binary-search on the line segment (p_{i+1}, p_{i+2}). This also reduces the path length by the triangle inequality. It is also possible to search for a replacement of p_{i+1} on the line segment (p_i, p_{i+1}), or just call the post-smoothing procedure twice, once with the original path $\langle p_1, p_2, \ldots, p_n \rangle$, and then again with the *reversed* processed path $\langle p_n, , \ldots, p_1 \rangle$. In the next section, we show the final path with and without post-smoothing, to demonstrate its effect.

EXPERIMENTS

We have tested VRA* on several handcrafted maps. We present the results of applying just A*, VRA* without post-smoothing, and VRA* with post-smoothing on one of these maps (due to space constraint), along with the associated numbers such as total memory used, number of A* searches invoked by VRA*, total number of all expansions (over all A* searches) used by VRA* as well as regular A*, total path build time, and also the rasterization test and splitting times (including neighbor finding) used by VRA*, for comparison of A* and VRA*. The difference in the quality of path produced by VRA*, without and with post-smoothing, testifies to the efficacy of our extended rubber-banding approach.

In the map shown in Figure 3 (for both A* and VRA*), the start point is shown in red, the goal point in green, the obstacles in black, the nodes in the closed list in red, the nodes in the open list (fringe) in green, and the final path in blue. In Figure 3, although the A* path is truly optimal, its wall-hugging behavior is most acute. In part (b), it is clear that VRA* has gotten rid of the wall-hugging behavior, but the paths are clearly not realistic-looking for intelligent agents. However, after applying the post-smoothing steps, the paths from VRA* (part (c) in the figure) looks more realistic. This map highlights that the result of VRA* can be *less predictable* than A*, since the path follows the opposite arm of the 'H', compared to A*. If a player had laid traps along



```
Done
Memory Usage: 348192
Expansions: 316
Path Build Time: 0.141
```

```
Done
Memory Usage: 67428
A* Searches: 8
Expansions: 29
Total Build Time: 0.010000
Line Test Time: 0.000000
Splitting Time: 0.000000
```

```
Done
Memory Usage: 67428
A* Searches: 8
Expansions: 29
Total Build Time: 0.010000
Line Test Time: 0.000000
Splitting Time: 0.000000
```

Figure 3: The result of applying A* (a), VRA* without post-smoothing (b), and VRA* with post-smoothing (c).

Table 1: Comparison of figures of merit between A* and VRA*, for Figure 3.

A*			VRA*		
Memory (bytes)	Expansions	Total time (sec)	Memory (bytes)	Expansions	Total time (sec)
348192	316	0.141	67428	29	0.01

the expected arm, (s)he would be surprised that the agent has snuck up behind him/her. We believe this element of surprise can be sufficiently appealing to players, to compensate for the loss of optimality.

A comparison of the figures of merit for the H-map in Figure 3, is shown in Table 1. We see that VRA* uses an order of magnitude less memory than A*, and produces a post-smoothed path in time that is an order of magnitude lower than A*. Although not shown in this paper, the result was similar in many other maps. Note that the number of expansions for VRA* is the total over all calls to A* that it makes. Also note that the total time under VRA* is the total time it takes to produce a path, and it includes all of the rasterization tests, splitting time and the time to compute neighbors in an irregular, variable resolution map. Moreover, the time difference between unprocessed and post-smoothed VRA* paths is minuscule. The key to the time saving with VRA* is the low number of cells that it has to process.

SUMMARY

In this paper, we have have presented a new algorithm, VRA*, for path-finding on game maps, exploiting a variable resolution, in a top-down fashion. We augment this technique with a post-smoothing approach that extends an existing approach. Experiments on some hand-crafted maps show that VRA* uses significantly less time and memory, and along with the post-processing technique suggested, it produces realistic-looking paths that overcome some of the problems with A*. In the future, we intend to study VRA* more systematically on maps from actual games (such as Baldur's Gate), and compare its performance to several variants of A*.

ACKNOWLEDGMENTS

The authors thank the reviewers for helpful comments. This work was supported in part by a start-up grant from the University of Southern Mississippi.

REFERENCES

[1] A. Botea, M. Muller, and J. Schaeffer. Near optimal hierarchical path-finding. *Journal of Game Development*, 1(1):1–22, 2004.

[2] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[3] D. Chen, R. Szczerba, and J. Uhran. A framed-quadtree approach for determining euclidean shortest paths in a 2-d environment. *IEEE Transactions on Robotics and Automation*, 13(5):668–681, 1997.

[4] D. Ferguson and A. Stentz. Using interpolation to improve path planning: The Field D* algorithm. *Journal of Field Robotics*, 23(2):79–101, 2006.

[5] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC*, 4(2):100–107, 1968.

[6] R. C. Holte, M. B. Perez, R. M. Zimmer, and A. J. MacDonald. Hierarchical A*: Searching abstraction hierarchies efficiently. In *AAAI/IAAI, Vol. 1*, pages 530–535, 1996.

[7] A. Moore and C. Atkeson. The Parti-game algorithm for variable resolution reinforcement learning in multidimensional state-spaces. *Machine Learning*, 21, 1995.

[8] A. Nash, K. Daniel, S. Koenig, and A. Felner. Theta*: Any-angle path planning on grids. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*, volume 2, pages 1177–1183, Vancouver, BC, Canada, 2007.

[9] S. Rabin. *Game Programming Gems*, chapter A* Aesthetic Optimizations. Charles River Media, 2000.

[10] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[11] P. Tozour. Search algorithms and search space demo 1.0. http://www.ai-blog.net, 2005.