Hindawi Publishing Corporation
International Journal of Computer Games Technology
Volume 2008, Article ID 873913, 11 pages
doi:10.1155/2008/873913

Research Article

Hierarchical Pathfinding and AI-Based Learning Approach in Strategy Game Design

Le Minh Duc, Amandeep Singh Sidhu, and Narendra S. Chaudhari

School of Computer Engineering, Nanyang Technological University, 50 Nanyang Avenue, Singapore, Singapore 639798

Correspondence should be addressed to Le Minh Duc, minhducle@pmail.ntu.edu.sg

Received 10 October 2007; Accepted 26 February 2008

Recommended by Kok Wai Wong

Strategy game and simulation application are an exciting area with many opportunities for study and research. Currently most of the existing games and simulations apply hard coded rules so the intelligence of the computer generated forces is limited. After some time, player gets used to the simulation making it less attractive and challenging. It is also costly and tedious to incorporate new rules for an existing game. The main motivation behind this research project is to improve the quality of artificial intelligence- (AI-) based on various techniques such as *qualitative spatial reasoning* (Forbus et al., 2002), *near-optimal hierarchical pathfinding* (HPA*) (Botea et al., 2004), *and reinforcement learning* (RL) (Sutton and Barto, 1998).

1. INTRODUCTION

Although strategy games have been around for above ten years, AI is still the biggest challenge in games with many unsolved problems. In this research, RL is chosen to further develop AI techniques. RL is learning from interaction with an environment, from the consequences of action, rather than from explicit teaching. In order to apply RL successfully, some of qualitative spatial reasoning techniques and HPA* are employed to design a better framework. In addition, real-time strategy (RTS) genre is selected for implementing the game to demonstrate the result.

Here are the milestones of this research work. Firstly, a game idea is brainstormed and implemented into a complete game demo called StrikeXpress with all the basic characteristics of a RTS. Secondly, the game demo is optimized with more expressive spatial representations, better communication of intent, better pathfinding, and reusable strategy libraries [1]. Finally, the RL is applied to the game's AI module. The paper is organized as follows. In Section 2, we review important concepts used in this project and briefly outline the development platform and tools used to create the game demo. In Section 3, we discuss the approaches for pathfinding and qualitative spatial reasoning techniques. We describe the framework for RL in Section 4 and conclude in Section 5.

2. LITERATURE REVIEW

2.1. Game design process

Game is made of many components, and game design process has to go through many steps as discussed in detail in [2]. However, making a complete commercial game is not our intention; our main focus is to build a basic game to demonstrate the research idea. For this purpose, we follow a simple game design process as shown in Figure 1. In *Concept phase*, we have to brainstorm the game story, look for concept arts, and choose the development platform. *Design phase* is mainly to design models, game levels based on design documents. *Components Implementation phase* is to implement components such as user interface, visual and audio effects, game mechanics, and AI. The next steps are *integration*, *fine tuning, and testing* before launching the game demo. The most challenging issue is how to implement machine learning feature nicely without affecting the game flow. Figure 3 shows the overall project architecture where the left part is game design process, and the right part is the framework for RL.

2.2. Qualitative spatial reasoning

Qualitative representations carve up continuous properties into conceptually meaningful units [3]. Qualitative spatial

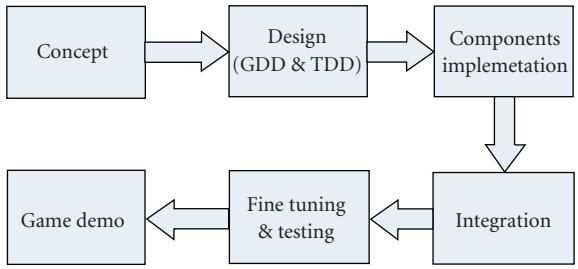


Figure 1: Game design process.

representations carve space into regions [4], based on a combination of physical and task-specific constraints.These techniques can provide a more humanlike representation of space and can help overcome the difficulties of spatial reasoning. This will let us build strategy AIs that more effectively use terrain to achieve their goals, take orders, and find their way around. Moreover, decoupling the spatial representation in the AI from the spatial implementation in the game engine constitutes a large step toward making reusable AIs, driving down development costs while improving AIs' subtlety [1]. The approach is discussed in detail in Section 3 together with pathfinding.

2.3. Near-optimal hierarchical pathfinding

The popular solution for pathfinding is A* algorithm. However, as A* plans ahead of time, the computational effort and size of the search space required to find a path increase sharply. Hence, pathfinding on large maps can result in serious performance bottlenecks. Therefore, HPA* [5] is used to overcome the limitations of A*. The main idea is to divide and conquer—break down a large task into smaller specific subtasks. HPA* will be discussed in detail in Section 3.

2.4. Hierarchal AI-based learning

In Figure 2, we use simple American hierarchal military structure to demonstrate the idea. An Army Lieutenant typically leads a platoon-size element (16 to 44 soldiers) to perform specific tasks given by higher commissioned officer such as Captain, Major, Colonel, or General. Similarly in the game, platoon represents *the lowest level agents* (LLA-) which perform real actions like move, run, shoot, and guard. Lieutenant represents *the middle level agent* (MLA) which decides the best strategy for the platoon such as to find the optimal paths, split the platoon into subgroups to move on different paths, decide the suitable time for engagement, retreat or call for reinforcement, and report results to higher officer. Higher commissioned officer represents *the highest level agent* (HLA) in the game that uses RL to learn from the environment and the consequences of actions performed by lower level agents to decide next actions such as to send forces to engage the enemy at coordinate (x, y, z), to agree or

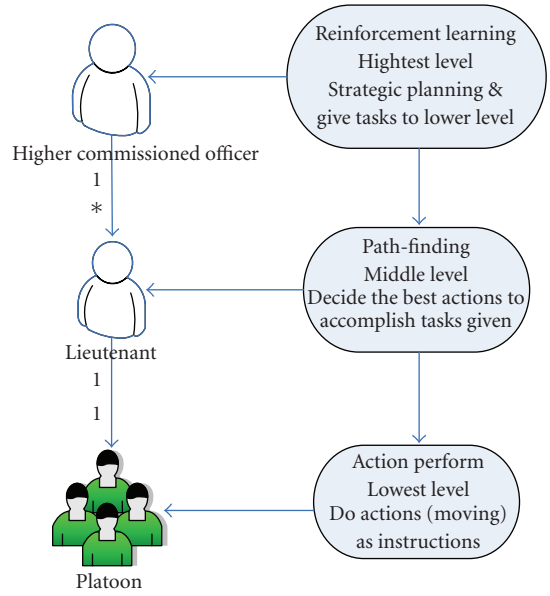


Figure 2: AI-based learning structure.

reject to send reinforcement, and to set up a series of strategic actions.

We notice that LLA is the easiest to implement as most of the actions are primitive and can be taken care of them by the game engine's built-in functions. HLA can be realized based on proven and established RL algorithm, provided that there is sufficient information for decision making—as shown in Figure 3; machine learning structure is already designed for RL. It is realized that the most difficult and bottleneck part is MLA where the game can be slow down noticeably, or the AI can become stupid due to improper, nonoptimized pathfinding, and data structure. Besides, most of the strategic actions planed by HLA involve some kind of movement. Without an efficient MLA, RL may not work properly.

2.5. Development tools

The game engine used to create game demo, StrikeXpress, is 3D Gamestudio (http://www.3dgamestudio.com/). MySQL is used for database storage (http://www.mysql.com/), and Matlab is used for running RL function (http://www.mathworks.com/products/matlab/). In addition, to connect between these tools, we must use some DLL extensions supported by 3D Gamestudio which is basically an external library that adds functions to the game engine. A piece of code written in DLL form may run faster than that written in game scripting language due to its precompilation. Theoretically everything, MP3 or MOD players, a physics engine, another 3D engine, or even another scripting language, can be added to the engine this way. Therefore, in order to make connection between the game engine and MySQL, we use a DLL extension called A6mySQL which is written in C++ (http://www.plants4games.com/hmpsplit/files/A6MySQL_Public_Release.rar).

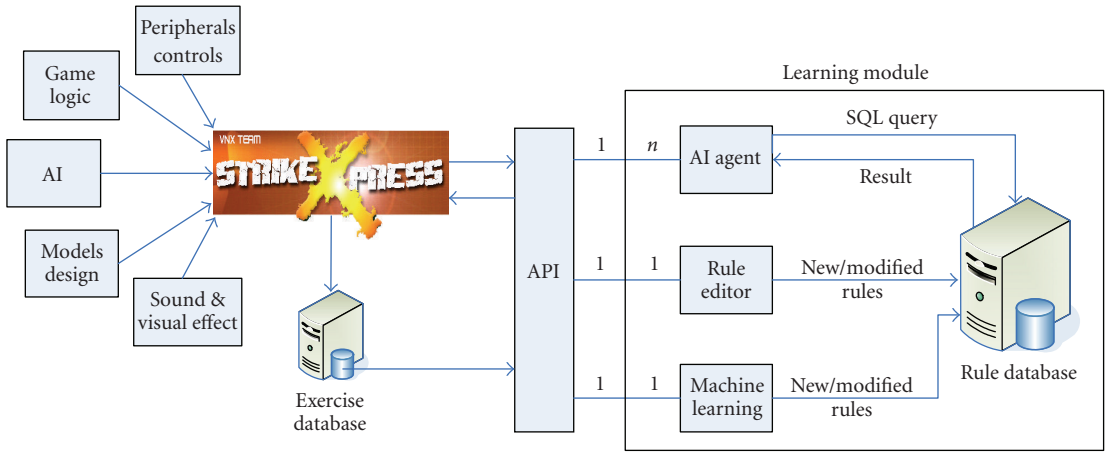


Figure 3: The overall project architecture.

To make connection between MySQL and Matlab, the plugin created by Robert Almgren is used (http://mmf.utoronto.ca/resrchres/mysql/).

3. PATHFINDING APPROACHES

Pathfinding on RTS games is complex because the environment is dynamic; there are lots of units which continuously move around the map and its scope equals the size of the level. This section is to demonstrate the use of two pathfinding approaches in the game: *Points of visibility* [6] and HPA*.

3.1. *Points of visibility*

Points of Visibility algorithm uses waypoints scattered around the level. A set of waypoints are connected together to create a network of movement directions. As shown in Figure 4, this network alone is sufficiently enough to guide a unit to transverse every obvious location of the map. In this approach, for simplicity, all the waypoints are placed manually, but the connections between those waypoints are done automatically like in Figure 5. How the waypoint is placed will make or break the underlying pathfinding code. The idea is to build a connected graph which will visit all places of our level. In human architecture, particularly tight corridors and other areas where the environment constrains the agents' movement into straight lines, waypoints should be placed in the middle of rooms and hallways, away from corners to avoid the wall-hugging issues. However, in large rooms and open terrain, waypoints should be placed at the corners of obstacles in a game world with edges between them. It will help generate paths almost identical to the optimal one.

In the graph making process, we select one entity to be responsible for creating and loading the graph data file. What the graph making process actually does is to let the selected entity move from one waypoint to another. A waypoint A is said to be connected with waypoint B if the selected entity is able to walk from A to B in straight line. The size of this selected entity will be considered when creating the graph so we must choose it wisely. After the graph making process is completed, all the connections will be stored in a data file. In the game, when pathfinding is invoked, it will process on the graph loaded from the file, and we can use any search algorithms to find the way. In this project, for simplicity, we deploy Dijkstra search algorithm.

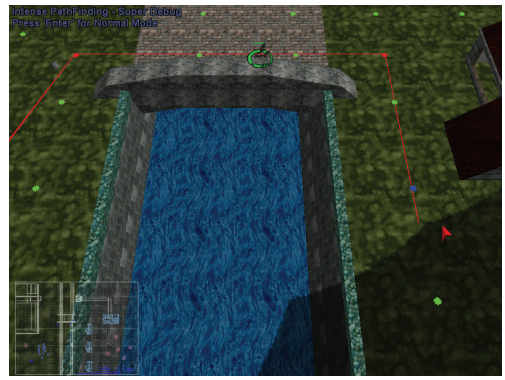


Figure 4: A path shown in Debug mode.

Advantages

Points of Visibility are being used today by more than 60% of modern games. It is simple and efficient thank to node-based structure. It is particularly useful when the number of obstacles is relatively small and they have a convex polygonal shape. When encountering slopes, hills, and stairs, we will get better results if placing a waypoint every short distance to fully cover it. Also, the graph does not need to be fully connected. The algorithm can handle the case where a level is split into two parts, and the player teleports from one part to the other. We can apply any search algorithm to this approach. We also can smooth the paths to make it look

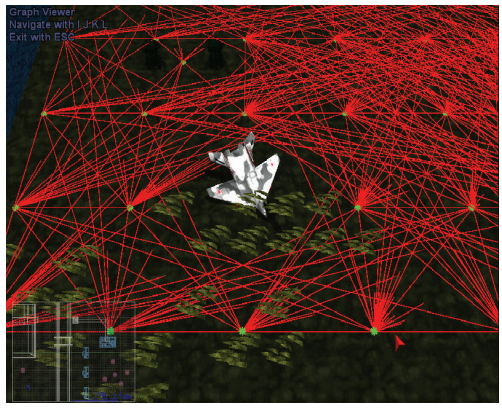


Figure 5: Waypoints placed manually and connections generated automatically.

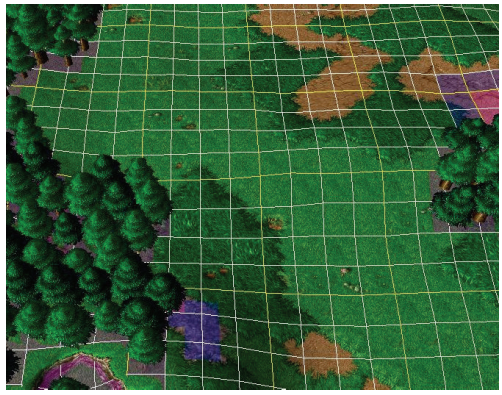


Figure 6: Transform terrain to large grid [screenshot taken from Warcraft III map editor].

more natural. The graph making process is also useful for debugging as all the waypoints and connections are displayed explicitly like Figure 5.

Disadvantages

The efficiency of the method decreases when many obstacles are present and/or their shapes are not a convex polygon or the level is open terrain with dense collection of small size obstacles. Modeling such a topology with this approach would result in a large graph with short edges. Therefore, the key idea of traveling long distances in a single step would not be efficiently exploited. The need for algorithmic or designer assistance to create the graph is also troublesome. In addition, the movement needs a lot of adjustment to be realistic, and the complexity increases fast when dealing with multiagents.

3.2. The HPA* algorithm

This technique is highly recommended based on its efficiency and flexibility to handle both random and real-game maps with a dynamically changing environment using no domain-specific knowledge. It is also simple and easy to implement. If desired, more sophisticated, domain-specific algorithms can be plugged in to increase the performance.

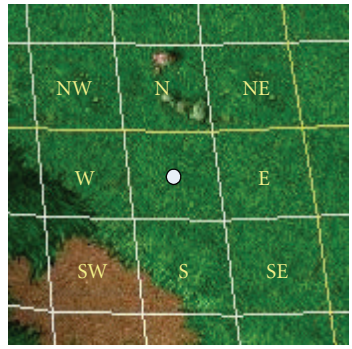


Figure 7: Representation of the first 8 neighbor cells.

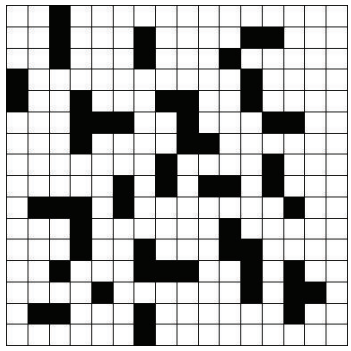


Figure 8: More expressive representation level grid.

3.2.1. HPA* preprocessing phase (offline)

Transform the level to large grid

The entire level is transformed into large grid with equal cells' size as shown in Figure 6. All the cells will be scanned. From all the accessible cells, we will check the height and any special values of each cell to determine its cost to use in A* algorithm. Hence, each cell can be treated as a node similar to waypoint in previous algorithm. All the cells' information will be put into an array for further processing.

Prelink the cell array

After transforming the level to large grid, we scan through each cell to see what surrounding cells can actually link to (NE, N, NW, E, W, SE, S, SW) as shown in Figure 7. For a surface with many cliffs, a cell on a cliff may not be reachable from its neighbor if the slope is too great. As a result, the level is transformed from the original in Figure 6 to more expressive representation grid like in Figure 8 where the black cell is totally inaccessible and white cell is accessible by some of its neighbors. The cost of each white cell may be different.

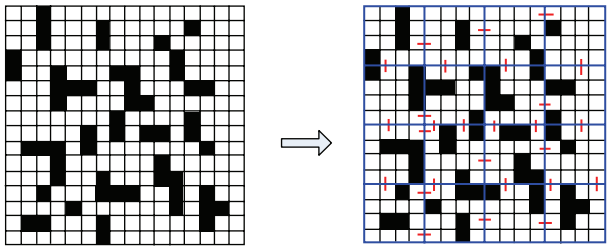


Figure 9: Grid to 16 subgrids.

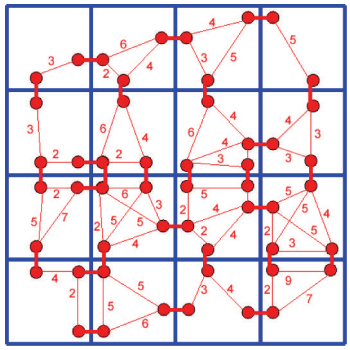


Figure 10: Abstract subgrid connectivity graph.

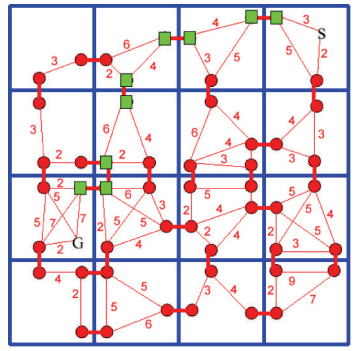


Figure 11: Use A* to find path from S to G with cost 29.

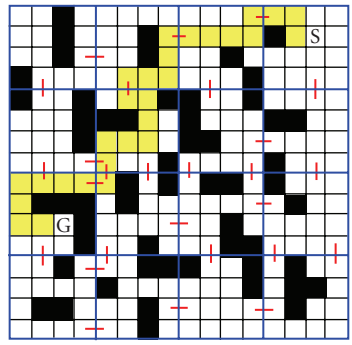


Figure 12: Path refinement with cost 29.

Divide a large grid into smaller clusters and find entrances between these clusters

The grid in Figure 8 can be divided into subgrids (clusters) in many ways, as shown in Figure 9. An entrance is a maximal obstacle-free segment along the common border of two adjacent clusters c1 and c2 [5]. Entrances are obtained for each subgrid in the same manner as larger grid and the red lines connect the resulting entrance nodes.

Build abstract subgrid connectivity graph

Transitions are used to build the abstract problem graph. For each transition, we define two nodes in the abstract graph and an edge that links them. The edge represents a transition between two clusters is called interedge. Each pair of nodes inside a cluster is linked by an edge called intraedge. The length of an intraedge is computed by searching for an optimal path inside the cluster area. We only cache distances between nodes and discard the actual optimal paths corresponding to these distances. If desired, the paths can also be stored, for the price of more memory usage [5]. After building the abstract graph like Figure 10, this graph is saved into a precompiled node list file for that level.

3.2.2. Pathfinding phase (online)

Add S and G to abstract graph and use A search*

When the game is loaded, we will also load its precompiled node list. The first phase of the online search connects the starting position S to the border of the cluster containing S by temporarily inserting S into the abstract graph. Similarly, connecting the goal position G to its cluster border is handled by inserting G into the abstract graph. After S and G have been added, A* is used to search for a path between S and G in the abstract graph. This is the most important part of the online search where heapsort and heap structure are used. It provides an abstract path, the actual moves from S to the border of S's cluster, the abstract path to G's cluster, and the actual moves from the border of G's cluster to G [5] as shown in Figure 11. In case S and G change for each new search, the cost of inserting S and G is added to the total cost of finding a solution. After a path is found, we remove S and G from the graph. Consider the case when many units have to find a path to the same goal, we insert G once and reuse it. If among these units there are some units close to each other, this group of units can share the same search operation. In the case the destination can be reached without obstacles in the way, a simple linear path should be chosen instead. The cost of inserting G is amortized over several searches. In general, a cache can be used to store connection information for popular start and goal nodes.

Refine path as needed

Path refinement translates an abstract path into a low-level path. Each cluster crossing in the abstract path is replaced by an equivalent sequence of low-level moves as shown in Figure 12. If the cluster preprocessing cached these move

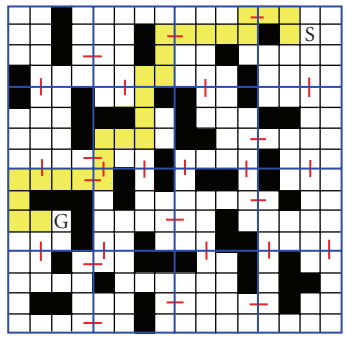
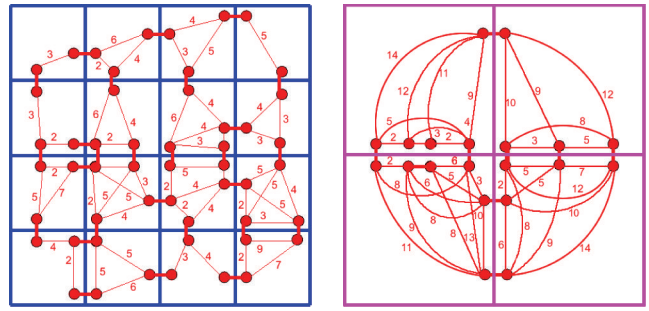


Figure 13: Path smoothing with cost 27.



(a) (b)

Figure 14: Multi-level abstract graphs with 16 "1-clusters" and 4 "2-clusters".

sequences attached to the intraedges, then refinement is simply a table look-up. Otherwise, we perform small searches (using A*) inside each cluster along the abstract path to rediscover the optimal local paths. Consider a domain where dynamic changes occur frequently, after finding an abstract path, we can refine it gradually as the character navigates toward the goal. If the current abstract path becomes invalid, the agent discards it and searches for another abstract path. There is no need to refine the whole abstract path in advance [5].

Apply smoothing

The topological abstraction phase defines only one transition point per entrance and gives up the optimality of the computed solutions. Solutions are optimal in the abstract graph but not necessarily in the initial problem graph. Therefore, we perform a postprocessing phase for path smoothing to improve the solution quality. The main idea is to replace local suboptimal parts of the solution by straight lines. Starting from one end of the solution, for each cell, we check whether we can reach a subsequent cell in the path in a straight line. If this happens, then the linear path between the two cells replaces the initial suboptimal sequence between these cells [5]. This step could be done one frame after applying A*. If the entity begins to walk in the same frame as the proper A* or one frame later, it can hardly be recognized by the player.

3.2.3. Multilevel hierarchy

Additional levels in the hierarchy can reduce the search effort, especially for large mazes. In a multilevel graph, nodes and edges have labels showing their levels in the abstraction hierarchy. Pathfinding is performed using a combination of small searches at various abstraction levels. We build each new level on top of the existing structure. The clusters for level l are called l-clusters [5]. To search for a path between S and G, we search only at the highest abstraction level and will always find a solution, assuming that one exists. The result of this search is a sequence of nodes at the highest abstraction level. If desired, the abstract path can repeatedly be refined until the low-level solution is obtained.

3.2.4. Data structure representation

Looking at HPA*, we notice that the number of search operations in one pathfinding can be up-to $l + 1$ times: one search for the highest abstraction level and l searches for recursive path refinement. Even when caching and unit grouping are used, HPA* is still slow if the A* search operation is not efficient. To optimize A* search, we focus on improving the data structure representation.

Node and cell structure

The elements for pathfinding in this approach are nodes (for multilevel abstract graphs) and cells (for low-level graph). For simplicity, we can call these elements as cells. In A* search, the algorithm has the choice of connected cells from the current entity position. When it decides to go to a direction, it can choose again out of its connected cells and can calculate again. It goes on and on until one of the cells leads direct to the goal. Once the search reaches G, the algorithm has to trace back to S. Heuristic that could help us with probabilities, but an exact statement about which cells lead to the goal could not be made. That means all the cells have to be saved and to be recallable every time. Alternatively, we could search a path from G to S so that the path can be used immediately. Otherwise, the saved path has to be reversed. As the number of cells is large, it would be useful when our algorithm could process as much cells as possible to find even longer and complex ways. Here is some information a cell must contain.

(i) The position of the cell to calculate the distance to G, we may take the coordinate of its center point.

(ii) The heuristic to determine how probable it is to reach G from the current state of position.

(iii) A reference to the previous (parent) cell to trace back.

(iv) A unique ID: an individual number of identification for access every cell later on. It has to be approachable.

For example, with the low-level graph, every terrain consists of vertices which are numbered consecutively so that each vertex has its own unique number. Besides, most of the

engines have function to access the vertex directly based on its number. Therefore, the solution is to assign the unique number of the cell's center vertex to the cell's unique ID. There are alternative ways when we do not want to analyze the terrain in our game. However, we believe that pathfinding based on analyzing the terrain has better quality. Here is an example of defining cell:

$$\begin{aligned} \text{CELL[ID]} &= \text{cell\_center\_vertex\_number;} \\ \text{CELL[waycosts]} &= \text{PARENT\_CELL[waycosts]} + 1; \\ \text{CELL[cellcosts]} &= \text{CELL[waycosts]} \\ &\quad + \text{distance(current\_pos, goal\_pos);} \\ \text{CELL[parent]} &= \text{parent\_cell\_ID.} \end{aligned}$$

The information about the position of the cell can be found out through cell ID. Every time new cells get created, the waycosts increases by 1. The sum out of many heuristic values gets normally summarized as cellcosts. As an array represents a single cell, multidimensional array is used to represent the level grid. On the basis of the cellcosts, the algorithm has to go for a cell with the lowest cellcosts inside the array where the pathfinding continues. It would be very ineffective to let the algorithm search again in its saved cells for the best one since it already searched and saved the cells that lead to G. It would be more luxurious if the array with the saved cells is prearranged so that the presently cheapest cell is always at the first array entry. Among all the sort algorithms, the *heapsort* is the most efficient.

Heapsort

According to Williams [7], who invented Heapsort and the heap data structure, Heapsort is a combination of tree sort developed by Floyd [8] and tournament sort described, for instance, by Iverson [[9], Section 6.4] (see also [10]). A heap is a binary tree (a tree structure, whose knots have only two edges), whose roots/knots have a lesser (or greater, depending on heap attribute) value than their direct succession roots/knots. The heap attribute is determined by the heap order. When roots/knots have a lesser value than their successors, it is an increasing heap order (the deeper you go down the binary tree, the greater the value gets).

At a heap with increasing heap order like the example in Figure 15, the smallest value of this data structure always inside the root that is pretty practical because our array with the cell entries could sort the cellcosts that way—the presently cheapest cell (the cell with the least cell costs) would always be at the root. To represent the array as a heap structure, first, we put the first cell at CELL_LIST array on position CELL_LIST[1]; then, the successors of a cell in CELL_LIST[i] are saved at the positions CELL_LIST[2*i] and CELL_LIST[2*i + 1]. Reversely, the parent cell can be found by dividing the position of the current cell by 2: CELL_LIST[i/2]. In array shape, a heap would look like Figure 16.

We use the heap from the start as a data structure. The heap is not empty at the beginning; the heapsort sorts a new value directly after the entry. Also, changing and deleting

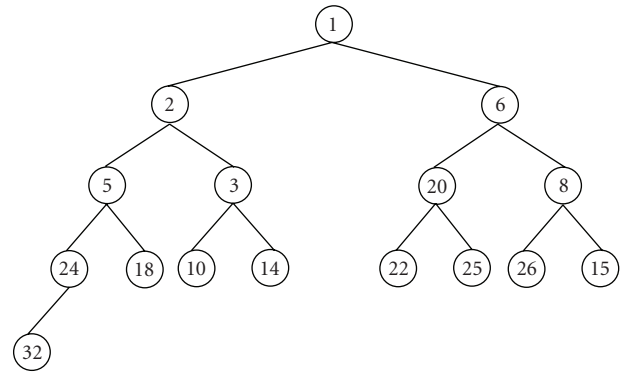


Figure 15: A heap with an increasing heap-order.

an entry (and the combined rearrangement) have to be managed by the heapsort. A heap that is used in such a kind of heapsort is called priority queue. The priority lays on the cellcosts that shall be possibly low. To add new cell or modify the value of a cell lesser, a procedure called *up-heap* [7] is used. The new cell is added as leafs at the end of the array, and the heapsort starts bottom-up. In case our defined heap order is overridden, the modified value of a cell is greater than the value of one of its child nodes, we have to use *down-heap* [7] procedure, sort top-down after the up-heap. We may optimize the sort by using other variants of heapsort such as weak heapsort [11] or ultimate heapsort [12].

3.2.5. Experimental results

In [5], experiments were performed on a set of 120 maps extracted from BioWare's game, BALDUR'S GATE, varying in size from 50 × 50 to 320 × 320. For each map, 100 searches were run using randomly generated S and G pairs where a valid path between the two locations existed. The experimental results show a great reduction of the search effort. Compared to a highly-optimized A*, HPA* is shown to be up to 10 times faster, while finding paths that are within 1% of optimal.

Figure 18 compares low-level A* to abstract search on hierarchies with the maximal level set to 1, 2, and 3. The left graph shows the number of expanded nodes and the right graph shows the time. For hierarchical search, the total effort is displayed, which includes inserting S and G into the graph, searching at the highest level and refining the path. The real effort can be smaller since the cost of inserting S or G can be amortized for many searches, and path refinement is not always necessary. The graphs show that, when complete processing is necessary, the first abstraction level is good enough for the map sizes that we used in this experiment. We assume that, for larger maps, the benefits of more levels would be more significant. The complexity reduction can become larger than the overhead for adding the level. More levels are also useful when path refinement is not necessary, and S or G can be used for several searches. Figure 19 shows how the total effort for hierarchical search is composed of the

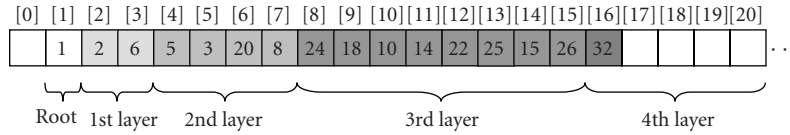


Figure 16: Representation of a heap in an array.

abstract effort, the effort for inserting S and G, and the effort for solution refinement. The cost for finding an abstract path is the sum of only the main cost and the cost for inserting S and G. When S or G is reused for many searches, only part of this cost counts for the abstract cost of a problem. Considering these, the figure shows that finding an abstract path becomes easier in hierarchies with more levels.

4. AI-BASED LEARNING

In Section 3, we discuss the approach to pathfinding used in MLA (Figure 2). In this section, we describe an AI-based learning design (Figure 3) to be used in all the agents. The purpose of AI-based learning is to capture and consolidate expert knowledge to achieve realistic game, evaluate the scenarios and strategies with greater accuracy. It will help the player experience increasing level of intelligence with every interaction in the game.

As RL is rule based, all the rules of the game will be extracted and stored in a *Rule Database*. During the game play, HLA would query the rules through *Rule API* from time to time. These rules will be used by computer's forces to play against the player. The detailed environment parameters and the result of action performed by agents are captured and logged in an *Exercise Database* which will be used for RL. In an offline situation where the game is not running, *Machine Learning* module analyzes the data from the *Exercise Database* based on RL functions and creates new rules or modifies existing rules for *Rule Database*. The modification of rules will increase AI gradually. It means that the level of difficulty rises up, and the player will find it harder to beat the computer [13]. Another function for the offline situation is the *Rule Editor* that has the capabilities to display, create, modify, and delete rules.

4.1. Rule API and rule database

Rule API is the interface for all operations. The most important functions are to attach *Rule Database* and to query the rules. When the game is loaded, each entity will be attached to its corresponding rule database through its agent.Subsequently, the entities can query for the rules in the rule database.The rules have to decide the actions to be carried out by the entities based on the information provided. Each query of the rule database will return one action. After the execution of that action, query the database for the next action will base on new information. As querying the database may become speed bottleneck, we may cache the entire rule database if the memory is large enough.

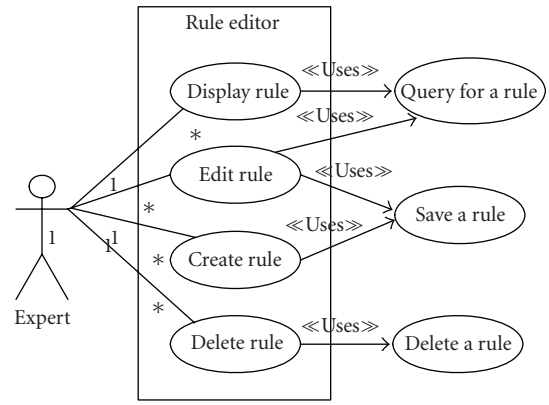


Figure 17: Use case for rule editor.

Otherwise, we only cache some of the frequently accessed rules.

In *Rule Database* , there are rules to define the *mission* of the forces which is the overall objective of HLA. This mission contains a set of *submissions* (SM) that is to be carried out by lower agents in order to accomplish the mission. For each command or overall mission, there is a set of SM that would be directly related to the mission stored in the database. The SM has to be assigned to forces to execute or complete the task. Information regarding the SM, for example, parameters, type of forces to be assigned, and priorities will also be provided in the database. Hence, there will be a rule database for HLA to assign the SM to forces. The assignment is under these conditions: after a main command (or overall mission) is given, a force has finished its assigned SM, new force is created, or a situation occurs, for example, enemy situation, operational situation, or obstacle situation. In the situation awareness, the mission, situation and its parameters are required by the rules. When a force encounters a situation, it will immediately react based on its rules of situation awareness. Hence, a rule database for MLA and LLA would also be necessary to respond accordingly. At the same time, the encountered situation and actions taken will also be reported to HLA which would then evaluate the situation and react appropriately. New SM can be reassigned to other forces whenever necessary. If the situation is not resolvable by the rules, user intervention may be requested.

4.2. AI agent

Every entity that is said to have AI will have an *AI agent* assigned to it. At LLA, AI called *unit agent* is used to control

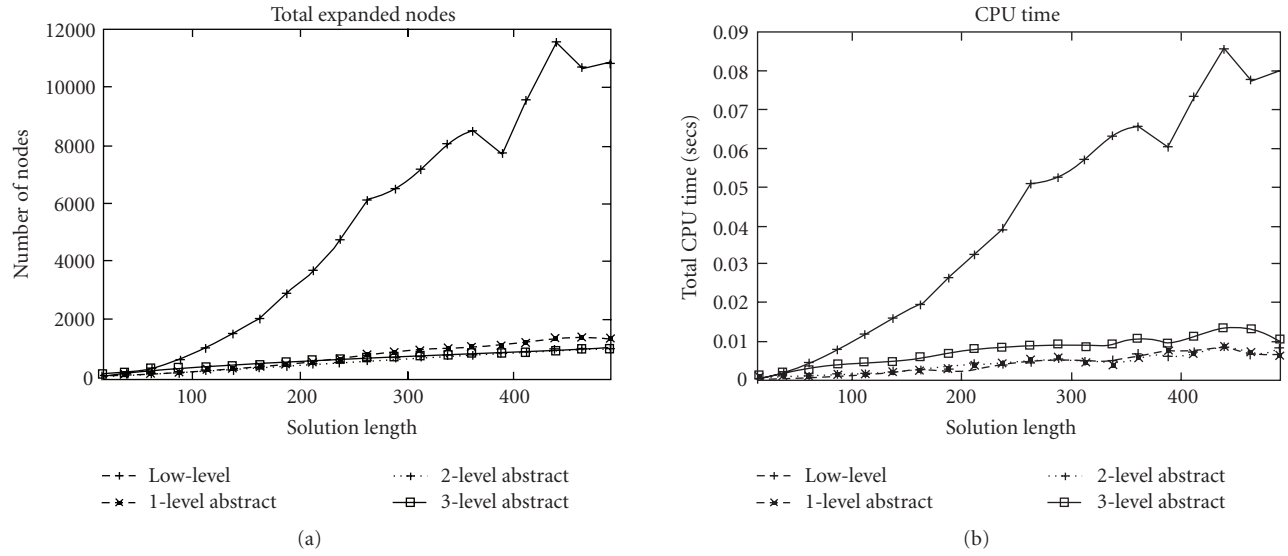


Figure 18: Low-level A* versus hierarchical pathfinding.

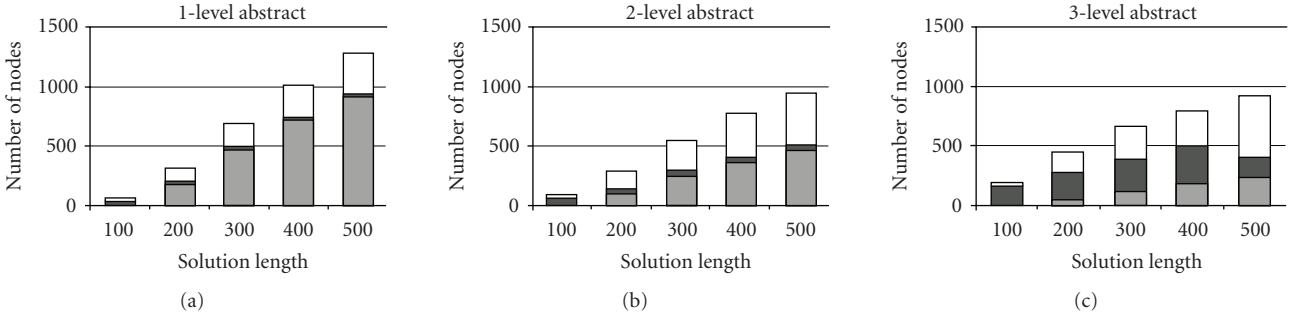


Figure 19: The effort for hierarchical search in hierarchies with one abstract level, two abstract levels, and three abstract levels. We show in what proportion the main effort, the SG effort, and the refinement effort contribute to the total effort. The gray part at the bottom of a data bar represents the main effort. The dark part in the middle is the SG effort. The white part at the top is the refinement effort.

detailed behaviors of different units. Unit agent is attached to every unit entity and consisted of various state machines to handle the detail movement and strategic reactions when it carries on the task given. Detail movement of an entity is determined by the game mechanics such as stand, guard, run, shoot, and throw grenade. Strategic reactions consist of individual reaction and group reaction.

4.2.1. Individual reaction

The unit agent will consider its survival probability as well as the present of enemy force in its line of sight to act according to the situation.For example, consider the case when a unit is at state *stand*, it detects enemy within its range of fire. If no task is given by higher agent, the unit agent has choices to 50% switch to state *shoot*, or to 30% switch to state *retreat*, or to 20% remain in state *stand*. The probability parameters are specified in the rule database and are loaded into the game at initialization process. We notice that game difficulty level could be adjusted simply based on some factors that affect the "skill" of the unit agent. For example, the reaction time, update cycle speed, health level, fire power of the enemy forces could be increased to add in challenges. The opponent could also have a "cheat" factor, that is, it will be given more units than the player.

4.2.2. Group reaction

Agents will also be attached to capture the hierarchy, that is, battalion, company, and the group behaviors. These agents will communicate with unit agents to get the status of different units. This status will help the hierarchical agent to make a better decision. For example, group formation in movement is useful to ensure that all the units keep their original formations upon reaching their targets. To achieve group formation, we use a simple approach: calculate the center position of all the selected units (a point that is roughly the middle of where they currently are). From that point, we get the offset for each unit, for example, if the center point is at [5,1] and one unit is at [6,1] then the offset would be [1,0]. The offset is, then, added to the destination point and that would be the point to move the selected

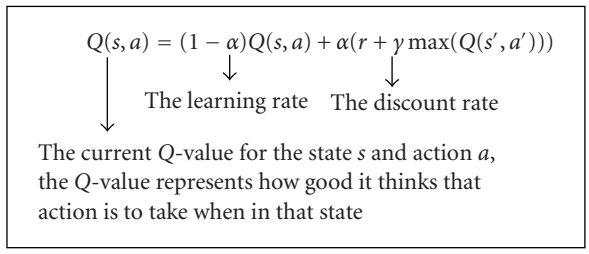


Figure 20

unit to. This would ensure all the units keep their original formations upon reaching their targets.

Another example is *coordinated behavior* in the enemy situation. In case our units surround the enemies, we want them to shoot the enemies without shooting at each other. Some of the games make it simple by letting the bullet go through ally to reach the enemies. We also can implement a small procedure to avoid friend's line of fire. In the enemy engagement, if a unit has line of sight to the target, it can shoot immediately. Otherwise, if obstructing object is an ally, request him to move away. If the ally is busy, or obstruction is not an ally, the unit moves itself to another place until it has the line of sight to the target. Another possible solution would be flocking which lets units repulse from each other and arrives at different offsets from the destination. However, we believe that flocking is overkill for RTS game, unless we really want to mimic the behavior of flocks.

4.3. Rule editor

The main responsibility of *Rule editor* is to edit the rule database. It has functions to add, delete, and modify any rule. It follows the model-view-controller design pattern. The Editor facilitates to change the state of the database on receiving instructions. In Figure 17, *Display Rule* allows the experts to view the rules displayed sequentially within a rule database. The expert has the ability to skip the current database and view the rules from another one. *Edit Rule* is to edit an existing rule. *Create Rule* allows the expert to create new rule from scratch. The expert, then, key-in or select the desired values for the parameters as well as the actions or output the rule would return. The completed rule will be stored in respective rule database. While the existing rules are displayed, the expert is given the option to delete the rule using *Delete Rule*.

4.4. Machine learning

This module is used to learn from environments, scenarios, and unsuccessful attempts. Based on the information obtained, it would try to extract new rules. The module checks with the rule database to ensure that the learnt rules are not present in the database. Newly learnt rules would be saved to the database [13].

RL function, Q-Learning [14], uses "rewards" and "punishments" so that an AI agent will adapt and learn appropriate behaviors under some conditions. In the experience tuple (s, a, r, s'), s is the start state, a is the action taken, r is the reinforcement value, and s' is the resulting state. The exploration strategy is to select the action with the highest Q-value from the current state.

There are two types of learning: supervised learning and unsupervised learning [15]. *Supervised Learning (SL)* is when machine learns from the user through user's input or adjustment of parameters. SL occurs when the rules fail to decide on an appropriate reaction to a situation and request for user's intervention or when the user decides to intervene. This intervention and its result will be logged in the *Exercise Database* for the offline learning. During the offline learning process, the effectiveness of user intervention is analyzed, and a new rule is generated.

Unsupervised Learning (UL) , in contrast, is to learn new rules without the knowledge or inputs from the user. The learning would be based on the existing set of rules to either generate new rules or enhance the old one. Some rules are specific to be fired by certain situations; some are more generic to be fired by a larger number of situations. The situations that would fire the rules could interest or subset with another one. This may result in several possible rules to be fired for one situation. Hence, these possible rules for a situation need to be prioritized to obtain the most efficient outcome. For example, the assignment of SM to forces can be conducted in several ways or sequences, and UL is to learn the best way to assign the SM. Each possible assignment of forces is valued with a priority or probabilities. Usually the possible assignment with the highest value is selected. If a sequence of assignment fails in a mission, the probabilities of this sequence will be decreased accordingly to reflect the failure. On the other hand, the probability would increase for a successful mission. Similar concept is also applied to the rules that respond to situations. Rules will be rewarded or punished based on the successful or failure executions of the reactions.

5. CONCLUSIONS

This research work is from the development of the basic game with simple AI modules, to the research of the higher-level concepts—advanced AI-based learning algorithms. Using the game demo as an effective tool, we implement various game AI techniques such as finite state machine, group behaviors, and pathfinding algorithms. We, then, work on finding the optimal combination of efficient techniques that are easy to implement and generic enough to be applicable in many games with little implementation changes. Based on this combination, we design the architecture for RL and propose the framework for future developments.

Our approach can have any number of hierarchical levels, making it scalable for large problem spaces. When the problem map is large, a larger number of levels can be the answer for reducing the search effort, for the price of more storage and preprocessing time. We use no application specific knowledge and apply the technique independently of the map properties. We handle variable cost terrains and various topology types such as forests, open areas with

obstacles of any shape, or building interiors without any implementation changes.

This research work has exposed us to new technologies and to current trends in computer game industry. We have explored some of game AI techniques and evaluated their pros and cons as part of the objectives. These technologies have shown to possess great potential in penetrating into the market, and there is plenty of room for improvement.

In the future, we will continue evaluating the proposed RL architecture to prove its effectiveness. We will also explore on some advanced techniques such as fuzzy logic, Bayesian networks, and neural networks, and will modify them to use in strategic game domain. Using these techniques, we will focus on tactical AI, particularly focusing on pathfinding, tactic analysis, and tactical representation. In addition, group dynamics and coordinated behavior are also very interesting to spend time on. At the same time, the underlying cognitive architecture needs to be expanded to make the games even more realistic.

REFERENCES

[1] K. D. Forbus, J. V. Mahoney, and K. Dill, "How qualitative spatial reasoning can improve strategy game AIs," *IEEE Intelligent Systems*, vol. 17, no. 4, pp. 25–30, 2002.

[2] E. Bethke, *Game Development and Production*, Wordware, Plano, Tex, USA, 2003.

[3] K. Forbus, "Qualitative reasoning," in *CRC Handbook of Computer Science and Engineering*, pp. 715–733, CRC Press, Boca Raton, Fla, USA, 1996.

[4] A. G. Cohn, "Qualitative spatial representation and reasoning techniques," in *Proceedings of the 21st Annual German Conference on Artificial Intelligence: Advances in Artificial Intelligence (KI '97)*, vol. 1303 of *Lecture Notes in Computer Science*, pp. 1–30, Springer, Freiburg, Germany, September 1997.

[5] A. Botea, M. Müller, and J. Schaeffer, "Near optimal hierarchical path-finding," *Journal of Game Development*, vol. 1, no. 1, pp. 7–28, 2004.

[6] S. Rabin, "A* speed optimizations," in *Game Programming Gems*, M. DeLoura, Ed., pp. 272–287, Charles River Media, Rockland, Mass, USA, 2000.

[7] J. W. J. Williams, "Algorithm 232: heapsort," *Communications of the ACM*, vol. 7, no. 6, pp. 347–348, 1964.

[8] R. W. Floyd, "Algorithm 113: treesort," *Communications of the ACM*, vol. 5, no. 8, p. 434, 1962.

[9] k. E. Iverson, "A programming Language," John Wiley and Sons, New York, NY, USA, 1962.

[10] E. H. Friend, "Sorting on electronic computer systems," *Journal of the ACM*, vol. 3, no. 3, pp. 134–168, 1956.

[11] R. D. Dutton, "Weak-heap sort," *BIT Numerical Mathematics*, vol. 33, no. 3, pp. 372–381, 1993.

[12] J. Katajainen, "The ultimate heapsort," *Australian Computer Science Communications*, vol. 20, no. 3, pp. 87–95, 1995.

[13] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, Mass, USA, 1998.

[14] I. Millington, *Artificial Intelligence for Games*, Morgan Kaufmann, San Mateo, Calif, USA, 2006.

[15] D. Michie, D. J. Spiegelhalter, and C. C. Taylor, *Machine Learning, Neural and Statistical Classification*, Prentice Hall, Upper Saddle River, NJ, USA, 1994.