Hindawi
Mathematical Problems in Engineering
Volume 2017, Article ID 1850678, 14 pages
https://doi.org/10.1155/2017/1850678

Research Article

Escaping Depressions in LRTS Based on Incremental Refinement of Encoded Quad-Trees

Yue Hu, Long Qin, Quanjun Yin, and Lin Sun

College of Information Systems and Management, National University of Defense Technology, Changsha 410073, China

Correspondence should be addressed to Quanjun Yin; yin_quanjun@163.com

Received 31 July 2016; Revised 9 February 2017; Accepted 28 February 2017; Published 19 March 2017

Academic Editor: Chunlin Chen

In the context of robot navigation, game AI, and so on, real-time search is extensively used to undertake motion planning. Though it satisfies the requirement of quick response to users' commands and environmental changes, learning real-time search (LRTS) suffers from the heuristic depressions where agents behave irrationally. There have introduced several effective solutions, such as state abstractions. This paper combines LRTS and encoded quad-tree abstraction which represent the search space in multiresolutions. When exploring the environments, agents are enabled to locally repair the quad-tree models and incrementally refine the spatial cognition. By virtue of the idea of state aggregation and heuristic generalization, our EQ LRTS (encoded quad-tree based LRTS) possesses the ability of quickly escaping from heuristic depressions with less state revisitations. Experiments and analysis show that (a) our encoding principle for quad-trees is a much more memory-efficient method than other data structures expressing quad-trees, (b) EQ LRTS differs a lot in several characteristics from classical PR LRTS which represent the space and refine the paths hierarchically, and (c) EQ LRTS substantially reduces the planning amount and curtails heuristic updates compared with LRTS on uniform cells.

1. Introduction

In fields like robotics and RTS (Real-Time Strategy) games, lots of real-time applications are becoming prevalent. Situated agents are required to limit the amount of planning per action under a constant upper bound and thus capable of responding instantly to users' commands and environment changes regardless of underlying problem size.

Since static full search (e.g., A* [1] and its variations) cannot cater to the requirements, a host of learning real-time search (LRTS) methods have been developed, including LRTA* [2], ϵ-LRTA* [3], FALCONS [4], SLA* [5], γ-Trap [6], and DTA* [7]. Among these, LRTA* is the seminal and fundamental one. Although LRTS can satisfy the requirement of real time, it suffers from the problem of heuristic depressions.

According to [8], heuristic depressions are bounded regions in search space where the heuristic values of inner states are lower than or equal to those of the states on the depression border, namely, heuristic local minima. Therefore when LRTA*-controlled agents enter the regions, they will get trapped. To escape depressions, agents revisit many states and learn their heuristic values, potentially several times for each. When the values have been updated to be equal to or greater than those of states in the border, agents finally succeed to leave. The state revisitation brings about scrubbing behaviors of agents, which visually look rather irrational and unacceptable to users.

The odd behaviors and slow learning are show-stopping problems of LRTS, keeping it from wide use in many applications. As much attention arises, researchers have proposed several kinds of solutions. There mainly exist two fashions. One, usually operating on uniform cells, focuses on learning strategies or depression avoidance. The other develops new methods of space representation to assist agents get out of depressions.

Among the first class of algorithms, Bulitko and Lee merged some extensions on LRTA* into a unifying framework of LRTS [9]. Apart from them, LSS-LRTA* [10] and RTAA* [11] are the representatives which work on fine-grained local search space and aggressive learning strategies. Literature [12] introduced depression avoidance mechanisms to identify depressions and guide agents towards the borders. Besides, some algorithms [13–15] offline compute subgoals and regard them as exits from local minima at online stages.

The second solution to the problem of heuristic depressions falls on spatial abstraction from original maps such as GVDs [16] and clique abstraction [17]. In real-time search, agents reason about the space and make decisions only according to the local information and previous records of sensors. Methods operated on ground-level cells are susceptible to lack of abstract cognition about the space, so agents are prone to get confused in local traps. Therefore, researchers, who favor the second kind of solution, aim at providing high-level information about search space to enable agents to make decisions more rationally.

Typically, many researchers have been working on hierarchical state abstraction [18, 19] to generalize the updates to heuristic function and speed up the learning. Path Refinement Learning Real-Time Search (PR LRTS) [20, 21] is the most outstanding example. It uses clique abstraction which gradually aggregates states from the ground level into higher levels based on connectivity relationship. Based on the abstraction hierarchy, it refines abstract paths and learns the heuristics from higher levels to the lower ones. However, the hierarchical abstraction possesses inconvenient process of construction, local repair, and memory storage. Also, the planning process trades efficiency to some extent for near optimality.

This paper follows the second idea and applies nonpointer quad-tree abstraction in real-time search. Unlike hierarchical abstraction, we build quad-trees by dividing the space into leaf nodes of different levels, which are completely empty or occupied by obstacles. In areas where obstacles are sparse, nodes of big sizes are sufficient to stand for large regions, whereas more nodes of finer granularity are needed where obstacles are dense. Furthermore, we design several encoding principles to efficiently describe the linkage among tree nodes, to save substantial memory and support convenient inquiries.

This state aggregation method facilitates the planning and heuristic generalization of our EQ LRTS (encoded quad-tree based LRTS). Operated on leaf nodes of trees instead of ground-level grids, the real-time search involves fewer states and consequently generates fewer planning steps. Planning costs naturally get decreased. Similarly, it can generalize heuristic updates of leaf nodes onto corresponding grids. Hence, learning process gets accelerated. As a result, agents who enter depressions can escape more quickly than those only gathering information on uniform cells.

Furthermore, we introduce an efficient local repair mechanism of quad-trees. EQ LRTS therefore possesses the capability of incrementally refining the abstraction during exploration when agents sense new information or underlying environment changes (e.g., obstacle insertion and removal).

The outline of this paper is as follows. Section 2 discusses related work on existing solutions to heuristic depressions in learning real-time search, followed by Section 3 which gives the problem formulation. Section 4 designs a coding method of nonpointer quad-tree and its local repair mechanism. Section 5 shows the framework of EQ LRTS and its implementation. The algorithm analysis and experiments are in Section 6. This paper ends up with conclusions in Section 7.

(1) Initialize the heuristic: $h \leftarrow h_0$;
(2) Reset the current state: $s \leftarrow s_{\text{start}}$;
(3) **while** $s \notin S_{\text{goal}}$ **do**
(4) Generate children one move away from state s;
(5) Find the state s' with the lowest $f = g + h$;
(6) Update $h(s)$ to $f(s')$ if $f(s')$ is greater;
(7) Execute the action to get to s';
(8) **end**

Algorithm 1: LRTA* with a lookahead of one: LRTA* (s_{start}, S_{goal}).

2. Related Work

To the best of our knowledge, Korf in 1990 [2] proposed the first and best-known algorithm of learning real-time search, LRTA*, whose pseudocode is shown in Algorithm 1 [9].

As it is shown, the algorithm follows a plan-learn-act cycle. On each episode, LRTA* first expands the neighbors of the current state s by a one-step lookahead search (line 4). Then it chooses the most promising one s' as the next state that keeps the lowest $f = g + h$ (line 5). The initial heuristic function is usually inaccurate, so LRTA* has to learn a more accurate function for reasonable planning. If $h(s) < f(s')$, then $h(s)$ is incremented to $f(s')$ (line 6). The episode ends up with the movement to s' (line 7). Based on LRTA*, there are three extensions: (1) γ-traps proposed in [6] allows more flexible lookahead selection, enabling agents to see farther. (2) Shimbo and Ishida [3] proposed a weighted version of LRTA* to speed up convergence at the sacrifice of optimality. (3) SLA* [5] supports backtracking to previous states if certain condition is met. These extensions are all included in the framework of LRTS [9].

Compared to A*, LRTS interleaves planning and execution in each trial which results in a much faster first-move lag. On the downside, LRTS can only perceive local information around the agent and lack aggressive learning method, making the agent "myopic." The agent takes on reasonable behaviors in obstacle-scattered areas but behaves irrational in heuristic depressions. To tackle the problem, two main streams of solutions are applied.

2.1. Solutions Operated on Uniform Cells. This category of algorithms works on a more pertinent local search space (LSS), more aggressive update strategies, or a combination of both. Many of state-of-the-art algorithms are able to escape heuristic depressions more quickly than LRTA*, including LRTA*(k) [22], LRTA*$_{\text{LS}}$(k) [12], LSS-LRTA* [10], RTAA* [11], RIBS [23], and f-LRTA* [24]. For example, LRTA*(k) updates heuristic estimate up to k states based on a bounded propagation strategy, which gain better quality of the first trial. Its variation LRTA*$_{\text{LS}}$(k) selects a local update space and updates each interior state once to its final heuristic value.

LRTA* applies a breadth-first search to expand neighbors which forms a disk-shaped LSS. To make it more goal-oriented and thus informative, LSS-LRTA* and RTAA* use a bounded A* and deal with the obtained states in *Closed* list as LSS. This is a fine-grained local space and reduces

redundant amount of search. They differ from each other in update strategies. LSS-LRTA* implements a modified Dijkstra's procedure from the border back to interior states which greedily learns the heuristic values of states in LSS. RTAA* uses a considerably fast way that generalize the sum of g-values and h-values of states in LSS to the lowest f-value of a state in *Open* list. With these elaborate operations, the two algorithms gain fast convergence and fewer revisitations.

Furthermore, [25] applies some useful principles on the two algorithms. As a theoretical basis, it formally gave a more general definition of cost-sensitive heuristic depressions than the notion in [8], which took into account the costs of the actions needed to move from the interior of the depression to the exterior. By its virtue, heuristic depressions can be effectively identified. A simple mechanism of depression avoidance was introduced, which embraces two principles of *mark-and-avoid* and *move-to-border*. They recognize the depressions and mark them and then guide agents to avoid them and then move towards borders. This work gave birth to a class of algorithms: aLSS-LRTA*, daLSS-LRTA*, aRTAA*, and daRTAA*.

The RIBS (Real-time Iterative-deepening Best-first Search) considers the learning problem from a novel perspective. It learns the "cost-so-far" (g-values) instead of "cost-to-go" (h-values) to identify redundant paths and dead-end states. Then RIBS removes these states from search space and thus increases the efficiency of search. As a combination of RIBS and LRTA*, f-LRTA* emphasizes both of the past and the goal information. While g-values are able to avoid repeating the past mistakes, learning h-values follow aggressive strategies to approach the goal.

The algorithms above hammer at informed local search space or efficient heuristic updates, so they possess the capability of quick heuristic increment. The operations to some extent ease the scrubbing behaviors, yet the agents' movements seem that they are fumbling around. The problem stands still severe in deep depressions where the agents behave just like LRTA*-controlled ones. Lack of abstract information about the space makes scrubbing behaviors difficult to avoid.

2.2. Solutions Focusing on Space Representations. The methods of this kind build particular space representations from basic spatial discretization and endow searching and learning with holistic view of the world.

Regardless of the swift response to changes of surroundings and tasks, real-time search on grid decomposition is susceptible to absence of the underlying physics of the world. Agents can only undertake planning and heuristic generalization in a local fashion. As a result, the agents are myopic and take long to escape traps.

On top of that, kinds of space representations are eligible to solve it. A space representation method means extracting structural information of underlying space and inducing a high-level representation based on the ground-level discretization. Many different methods have been developed (e.g., waypoint graphs and navigation meshes) which are used in real-time search. For instance, [26] built Generalized Voronoi Diagrams (GVDs) to represent topological structures and dealt with GVD edges as its search space. Although LRTS on GVDs gains reduced planning cost, maximal clearance to obstacles, and depression avoidance, resultant paths are longer than the desired ones.

Automatic state abstraction is a more common method of space representation. According to [27], an abstraction is a graph homomorphism which maps nodes from low-level graphs to nodes in high-level ones. With the graph abstraction, real-time search is undertaken in a hierarchical way that plans an abstract path and then gets it refined level by level, finally a traversable grid-based route. HPA* in [28] follows this scheme in which it hierarchically abstracts a map into linked local clusters.

Apart from it, [27] presented 5 different mechanisms, clique, sector, radius, line, and nonlimit abstraction, among which clique is the most widely used one. The idea behind it is to extract cliques in each level of the abstraction in which each pair of two states are connected except the orphans. In this way, the abstraction hierarchy will be built recursively from bottom to top.

The best-known algorithm which applies the clique abstraction, to our knowledge, is PR LRTS [21]. As original LRTS framework learns the heuristics in a tabular form in which each entry corresponds to a single state, it converges to the optimal values via individual updates. Therefore significant cost is incurred. Now with the help of cliques, PR LRTS speeds up the learning by propagating the incremented values from abstract cliques, which are of large sizes, to concrete states. As for the planning, it firstly assigns A* or LRTS to abstraction levels and thereupon limits the search space of low-level planning to corridors. The planning eventually goes deep to the grid-based map and generates a traversable path. Besides, to adapt the algorithm to dynamic space, there are several operations to support a local repair mechanism of the abstraction hierarchy.

While PR LRTS and EQ LRTS both adopt the idea of state aggregation and heuristic generalization, there still exist significant differences. Basically, the former builds its levels from bottom to top via hierarchical aggregation, while the latter divides the space into different granularities from top to bottom. More in-depth comparisons will be included in Section 6.

2.3. Other Algorithms. There are some algorithms in real-time search that cannot be easily grouped into any of the two categories discussed so far.

Apart from solutions mentioned above, the idea of subgoaling is another scheme to learning acceleration. Examples include D LRTA* [13], kNN LRTA* [14], LRTA* with subgoals [15], and HCDPS [29].

To illustrate, kNN LRTA* predetermines a fixed number of optimal paths between random states and compresses them into subgoal sequences via a hill-climbing agent. With them in database, it at runtime finds the most similar case to the current one and guides the agent along the related subgoals. HCDPS (Hill-Climbing Dynamic Programming Search) at the offline stage partitions the entire space into regions within which states are mutually hill-climbing reachable to their representatives. And then it employs A* algorithm and

```
(1) if isObstExist(obstList, indexX, indexY, size) ≠ true then
(2)     Set the node state white, and insert it into the hash table;
(3) else
(4)     if isAllOccupied(obstList, indexX, indexY, size) = true then
(5)         Set the node state black, and insert it into the hash table;
(6)     else
(7)         Set the node state grey, and insert it into the hash table;
(8)         Generate four children, and assign newIndexX[4], newIndexY[4], size = size/2;
(9)         for i = 1 : 4 do
(10)            BuildQuadtree(obstList, newIndexX[i], newIndexY[i], size)
(11)        end
(12)    end
(13) end
```

Algorithm 2: Pseudocode of the algorithm to recursively build the quad-trees: **BuildQuadtree**(*obstList, indexX, indexY, size*).

dynamic programming to obtain optimal and suboptimal paths between representatives of each region pair. To its virtue, online planning runs without heuristic updates.

Although the methods act well to avoid depressions, they require substantial time for preprocess and much memory for database storage. Moreover, there is limited coverage for online problems and they are not dynamically adaptive to changing environments.

3. Problem Formulation

Similar to most papers on real-time search, some definitions of related terms are necessary for understanding the problem.

A *search problem* can be defined as a tuple (S, A, c, s_0, S_g, h_0), where S is a finite set of states, A is a finite set of alternative actions regarding the current state, and $c(s, a)$ is the cost function incurred by the action a in state s. The initial state is s_0 and $S_g \in S$ is the goal states. h_0 is the initial heuristic function.

A *path* is a sequence of states $(s_0, \ldots, s_i, s_{(i+1)}, \ldots, s_g)$ which starts with the initial state and ends up with one of the goal states $s_g \in S_g$. Each state on the path should be valid and there is at least a practical action between each pair of states $(s_i, s_{(i+1)})$. The total traveling cost of a path is the sum of the costs between all the adjacent states.

The *current state* s_c is where the agent resides at the moment and can be changed by executing actions and incurring costs. s_c starts at the initial state s_0 and arrives at one of the goal states s_g if a path can be found.

A *trial* is a sequence of states the algorithm visits between s_0 and the first goal state it encounters. The *first-move lag* is the planning amount before the first move. The *final trial* means the first trial on which no learning happens. A *convergence run* can be defined as a sequence of trials from the first to final trial. This paper mainly discusses the performance of algorithms on the first trial.

In most real-time applications, an agent cannot know the problem in its entirety. Instead, it can sense the neighborhood of the current state s_c at each time step. At all times an agent has an internal model of what the search space is like based on the current and past information. The model is updated as the agent discovers the search graph.

4. Quad-Tree Abstraction

Whether a quad-tree is adequate to be adopted by the real-time search depends on the memory usage and efficiency of executing basic operations which will be used frequently during searching and planning (e.g., repairing local nodes and querying neighboring states). Therefore in this section we will design an encoding method to effectively support these requirements. The codes will be used as indexes to map the corresponding tree nodes to a hash table. We then describe the local repair mechanism which will be used in Section 5 by our EQ LRTS algorithm.

4.1. Encoding and Building the Quad-Tree. The construction of a quad-tree can be done in a top-down manner shown in Algorithm 2: a squared map is divided into 2×2 subsquares. If the child nodes contain no obstacle, mark their state as *white* nodes (lines 1-2). If they are completely occupied by obstacles, mark their state as *black* nodes (lines 4-5). If they are not so pure, we call them *grey* nodes and then the partition is repeated to these nonleaf nodes (lines 6–12). In the algorithm, functions of isObstExist and isAllOccupied return Boolean values after obstacle referring. There are several basic inputs: *obstList* is a one-dimensional Boolean array which delineates the initially known obstacle distribution, *indexX* and *indexY* are the spatial coordinates of current states top-left vertex, and *size* reflects its side length.

Based on the construction algorithm, Figure 1 describes the process of mapping tree nodes to the hash table with a hand-traceable example. The left figure depicts the hierarchical spatial decomposition using quad-tree abstraction, in which the same partition process used on the root node is also applied downwards. As for the middle one, it graphically depicts the encoding principle. The red numbers reflect which layer the nodes are on and the black numbers can be called position codes that contain information of node positions and parent-children relationship. Basically, 0, 1, 2, and 3 can be used to identify top-left, top-right, bottom-left, and bottom-right nodes derived from one parent. For instance, node 1-1 here can be decomposed into 4 children: 2-1, 0, 2-1, 1, 2-1, 2, and 2-1, 3. The mapping process goes along with the construction. Whenever the tree generates a new node, its

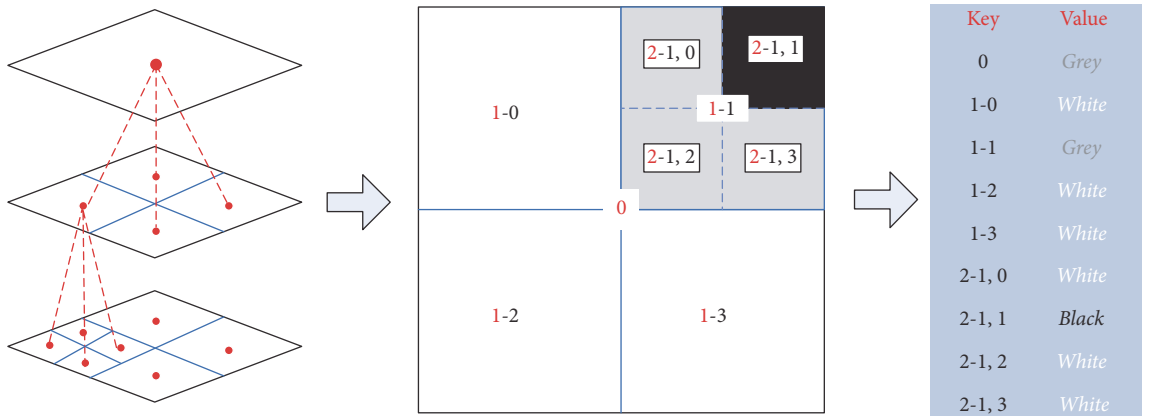


Figure 1: Graphic representation of mapping process. All nodes on different layers have unique codes as indexes in the hash table. Based on the principle, relative information will be maintained in the table.

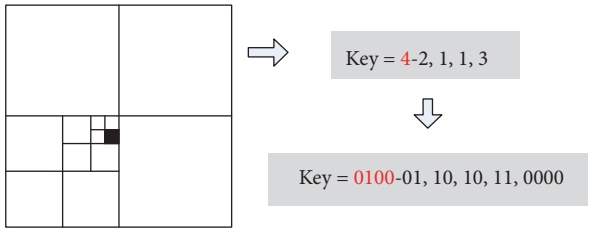


Figure 2: A simple example of tree node encoded by a bit sequence.

code and state will be inserted into the hash table according to the format shown in the right figure.

To yield comparable effect of compressing data storage, the codes stored in the hash table can easily be encoded into bits. Figure 2 shows a concrete example in which the code of the black cell is 4-2, 1, 1, 3 according the encoding principle mentioned above. Here it can be completely expressed by a bit sequence at the length of 16, whose former 4 bits replace the layer number and the remaining 12 bits take the place of position code.

In quad-tree abstraction, basic operations include querying a node state and locally repairing the tree in accordance with the underlying spatial changes. With the help of the encoding principle and hashing technique, these operations can be easily accessed.

4.2. The Mechanism of Local Repair. In real scenarios, what agents always encounter are dynamic environments where obstacle movements, deletions, and insertions are rather common. To guarantee the correctness of reasoning and planning, spatial representation methods are required to be adaptive to environmental changes. Instead of rebuilding the quad-tree from scratch, we can react much more quickly via launching a local repair process, also called local update. Different from the depth-first search used by pointer-based quad-trees, our algorithm accesses the target node by referring to its index, which can be obtained by the aforesaid principle. Algorithm 3 showcases the pseudocode of algorithm to repair and query a node and [30] employs a similar method. In the beginning, the quad-tree is empty and there only exists a leaf node, that is, the root, in the hash table. As the exploration and update process go on, new tree nodes are discovered and added to the table. The main idea of local repair is to merge nodes upwards or clear the subtrees downwards. If the four child nodes with regard to the same parent possess the same state, the merging process will iteratively be invoked (lines 6–12). Besides, if a node which is not a leaf node turns pure owing to the environmental changes, its state will be rewritten and corresponding subtrees will be erased from the hash table (lines 13–18).

The function of RepairANode receives a node code and its state as inputs. If the node has already been inserted into the table, then its state will get updated instantly through a direct access. Then the subtree deletion or backtracking process of merging will be launched if certain conditions in line 3, 6, or 13 are satisfied. And the ClearSubTree function (lines 23–29) will be called in the process of subtree deletion. In the contrast, if the target node does not exist in the tree currently, CreateANode will be called to create corresponding node instance (lines 19–21). The function firstly queries the nearest ancestor of this node (lines 32–34). Regarding the ancestor as a temporary root, the function iteratively divides it into subsquares until the target node is reached (lines 35–44). Finally, line 45 updates its state and inserts it into the hash table. As to the QueryANode function, it is analogous to CreateANode (lines 47–50).

5. Algorithm of Incremental EQ LRTS

In common scenarios, agents are given an initial map of the environment where they will conduct certain missions. The map is usually not so practical that only partial information about the terrains is included and there is no dynamic information provided. To reason about the environment correctly and plan their motions reasonably, agents have to use their sensors to collect surrounding environmental information. Also, agents can gather data transmitted from other agents to facilitate their own actions. Since

```
(1) RepairANode(m, NewState)
(2) if m in the hash table then
(3)    if m is a leaf node then
(4)       m.state = NewState;
(5)       n ← m.parent;
(6)       while all children of n possess the same
          state do
(7)          erase all the children from the table;
(8)          n.state ← NewState, n ← n.parent;
(9)          if n is the root node then
(10)             break the while loop;
(11)         end
(12)      end
(13)   else
(14)      m.state = NewState;
(15)      for each n ← m.child do
(16)         ClearSubTree(n);
(17)      end
(18)   end
(19) else
(20)   CreateANode(m, NewState);
(21) end
(22) ClearSubTree(m)
(23) if m is not a leaf node then
(24)   for each n ← m.child do
(25)      ClearSubTree(n);
(26)   end
(27) else
(28)   erase m from the table;
(29) end
(30) CreateANode(m, NewState)
(31) t ← m;
(32) while m not in the table and not the root node do
(33)   m ← m.parent;
(34) end
(35) if m ≠ t then
(36)   m.child ← new 4 leaf nodes, q ← m;
(37)   for each n ← m.child do
(38)      if n is an ancestor of t or n = t then
(39)         m ← n, n.state = grey;
(40)      else
(41)         n.state ← q.state;
(42)      end
(43)   end
(44) end
(45) t.state = NewState;
(46) QueryANode(m)
(47) while m not in the table and not the root node do
(48)   m ← m.state;
(49) end
(50) return m.state;
```

Algorithm 3: Pseudocode of the algorithm to repair and query a node.

sensors can merely perceive at a limited range, agents always update their spatial cognition incrementally as they are moving.

Based on the local repair mechanism, we in this section build the algorithm framework of incremental EQ LRTS. Figure 3 portrays the flowchart which describes the main steps. For the algorithm undertakes searching on quad-tree nodes, the first step abstracts the initial given maps into quad-trees and finds corresponding tree nodes of the initial state and the set of goal states, which are the basic arguments. Then before the searching loop, it resets the current state and heuristics.

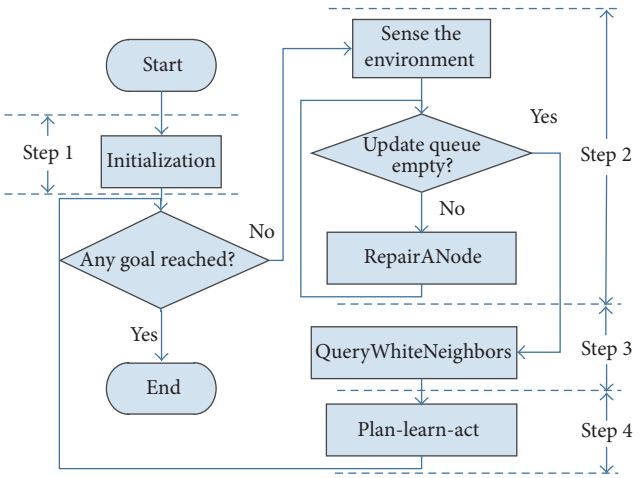


Figure 3: The flowchart showing the process of incremental EQ LRTS.

After initialization operations, agents enter the searching loop. In Step 2, agents first sense the environment and see whether the quad-tree model is supposed to be repaired. Through agents' ongoing exploration, the spatial representation is being refined and updated incrementally. To get passable neighboring states is the next needed step, and the quad-tree querying algorithm will provide related supports. Like basic real-time search methods, agents in Step 4 evaluate the candidates and choose the most promising state to travel to. If some condition is met, the heuristic value of the current state will get refined. This section will explain the main steps in detail.

5.1. Incremental Refinement of Quad-Trees. Since obstacle placement keeps changing, agents are required to keep track of useful information around them. Once situations emerge, which do not accord with the historical records or initial given map, necessary data will be stored in the update queue. Therefore the algorithm iteratively calls RepairANode function to get the quad-tree representation refined and stay adequate. The loop in step 2 will terminate until the agent has dealt with all related changes.

Figure 4 gives a clear example illuminating the process of repairing the incipient quad-tree model during agents' exploration. Figure 4(a) shows the map which the EQ LRTS-controlled agent (here shown as the blue circle) initially has. This map may contain incorrect information or only partially cover the search space. The black and red blocks are obstacles and the set of goal states, respectively. Before path-finding starts, the BuildQuadtree function is called to represent the obstacle distribution in quad-tree abstraction, which is show in Figure 4(b). The larger circle outside the agent stands for its sensing range.

Arriving at the position in Figure 4(c), the agent observes that the representation is needed to reach finer resolutions here reacting the obstacle removal and insertion at region 1. Meanwhile, although obstacle removal also occurs at region 2, there is no need to undertake local updates because the agent has already left there. In Figure 4(d), the agent finds two walls at region 4 that are not reported by the beginning map for they are distant from the start. Again, the agent calls the RepairANode function to enable the quad-tree updated. Besides, enemies have built a fort in region 3 which lies out of the agent's observing scope, thus being set aside by the representation.

5.2. Neighbor-Querying Method. In real-time search, agents standing on the current location look around to see where they can head to (Step 3 of EQ LRTS). If the underlying world is simply partitioned into grids, it is pretty easy to know which neighboring states the agents can transfer to. As for quad-tree methods, querying neighbors of all possible sizes is a relatively troublesome task. The main steps of our neighbor-querying method are as follows:

(1) Compute the codes of all neighbors equally sized to the current node m, which are stored in *equalNeighbors*. This step involves micro bit operations based on the encoding principle.

(2) For each node $n \in$ *equalNeighbors*,

(i) if the node is out of the map range, continue.

(ii) if it is not in the hash table, call the QueryANode function to find its nearest ancestor. If the state of the ancestral node is *white*, see it as an eligible neighbor,

(iii) if the node has already been in the hash table and n.state = *white*, see it as an eligible neighbor,

(iv) else if n.state = *grey*, find its offspring which are adjacent to m, among which the *white* ones will be eligible. If n.state = *black*, continue.

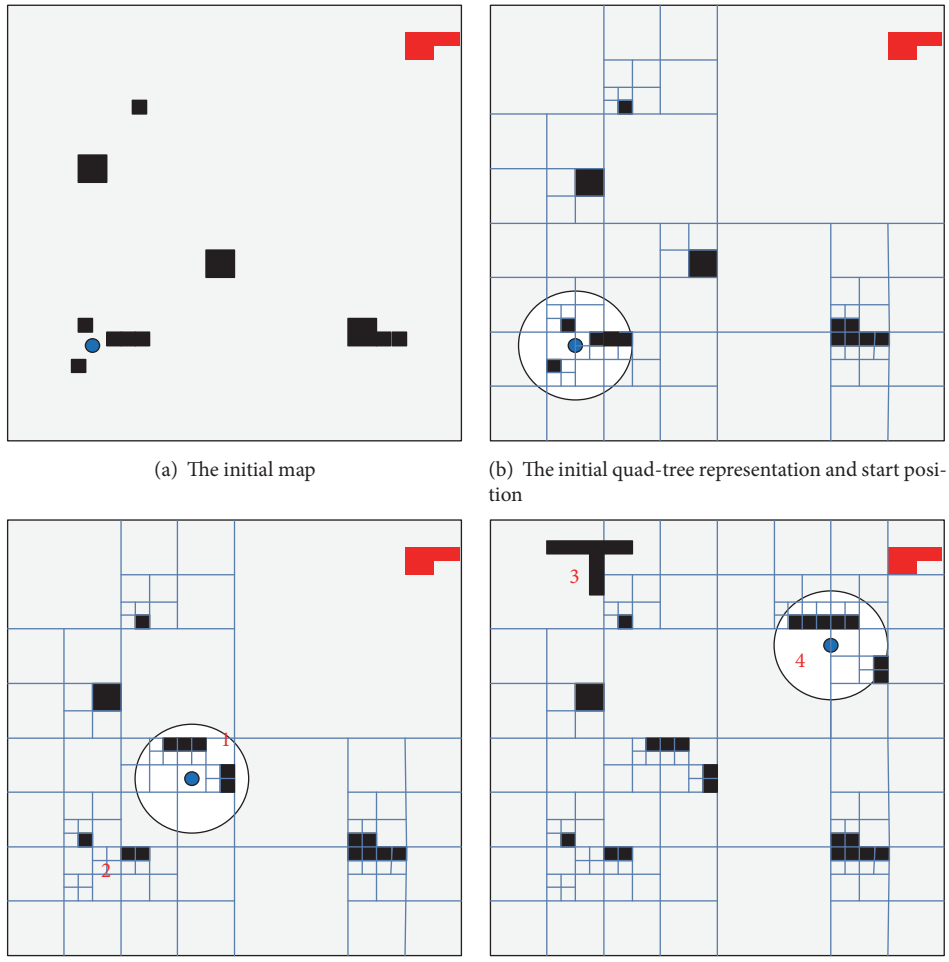
5.3. Planning and Learning. As mentioned above, LRTS suffers unreasonable behaviors from its slow heuristic learning, especially in heuristic depressions. Many methods which combine state abstractions with LRTS, PR LRTS included, speed up the learning via the idea of state aggregation. Similarly, quad-tree based real-time search refines the heuristic function on its tree nodes rather than grids which are of finer granularities.

A local terrain around the agent is given in Figure 5, and we assume that its goal is situated at (28, 28). The corner-shaped walls, shown as the black blocks, form typically a heuristic depression. The left figure labels the coordinates of the current node s_c and its traversable neighbors. The initial heuristic used here is the so-called octile distance defined as follows in two-dimensional space:

$$\text{dist}(A, B) = \sqrt{(x_A - x_B)^2 + (y_A - y_B)^2}. \quad (1)$$

As a result, heuristic values of related nodes in iteration i are shown in the middle figure, where $h_i(s_c) = 21.2$. The agent applies the mini-max search to determine the next state s' in iteration $i + 1$ by the following two formulas:

$$s' = \arg \min_{s \in \text{neighbors}} \left(g_i(s) + h_i(s)\right),$$
$$h_{i+1}(s_c) = \max\left(h_i(s_c), f_i(s')\right). \quad (2)$$



(a) The initial map

(b) The initial quad-tree representation and start position

(c) Certain position close to the start

(d) Certain position close to the goals

FIGURE 4: An example of updating the quad-tree model to reflect environmental changes.

In this given local map, we have

$$f_i\left(s'\right) = g_i\left(s'\right) + h_i\left(s'\right) = 19.8 + 1.6 = 21.4, \tag{3}$$

which is greater than $h_i(s_c)$. Consequently, the heuristic value of s_c gets incremented to 21.4.

To the virtue of quad-trees' multiresolution, 4^N $(N \geq 0)$ grids are extracted to just one quad. On heuristic learning, it enables the generalization from learnt h-value of the current quad s_c to corresponding multiple ground-level grids. Back to the example, the right figure propagates the learnt $h_{i+1}(s_c)$ to four underlying grids at merely one iteration that functions much faster than the grid-discretization does.

6. Analysis and Experiments

Since there exist several different data structures to express quad-trees in memory, it is necessary to make an analysis and comparison. Also we in this section analyze the differences between EQ LRTS and PR LRTS, the representative algorithm applying automated state abstraction to LRTS. Then we conduct a series of experiments to prove that our algorithm hosts the capability of reducing revisits and heuristic updates, compared to LRTS which operates on uniform grids.

6.1. Analysis of Quad-Tree Abstraction. It can be easily argued from Figure 4 that the number of quad-tree nodes is much less than that of grids. According to [30], in the worst case, namely, to represent a map in which each grid possesses a distinct state from its adjacent nodes, all the inner quads of the quad-trees must be expanded to the finest resolution. Assume that the map is of $2^N \times 2^N$ $(N \geq 0)$, where N is the number of existing layers; the total number of quad-tree nodes is

$$1 + 4 + 4^2 + \ldots + 4^N = \frac{4}{3} \times 4^N - \frac{1}{3}. \tag{4}$$

Thereby the worst space complexity is $\mathcal{O}(4^N)$. More commonly, the number of leaf nodes in a quad-tree map having polygonal obstacles is roughly $(2/3)\mathcal{O}(n)$, where n is the total perimeters of polygonal obstacles [31].

We build the quad-trees in terms of hash tables, so the complexity of querying an existing node is in constant time. As for the nodes which have not been expanded, the

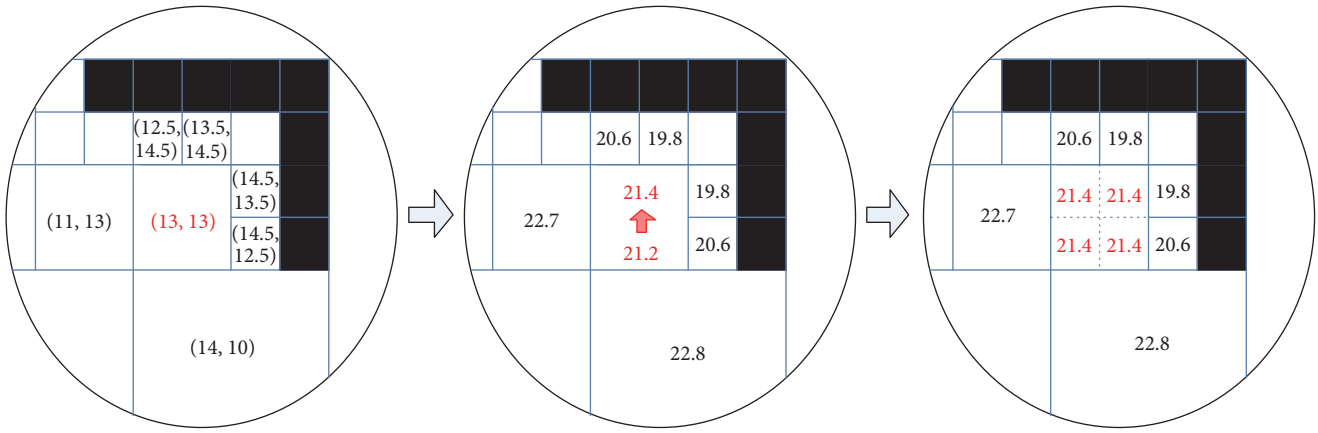


Figure 5: A numerical example of planning and heuristic generalization based on quad-tree modeling.

QueryANode function takes charge of the ancestral querying. Based on the convenient encoding method, the process of backtracking to the very upper layer can also be finished in constant time. Therefore the total time used for completing the querying depends on the depth of the tree and the layer gap between this target node and its nearest ancestor. The worst case is obviously $\mathcal{O}(N)$.

Furthermore, expressing the same quad-trees in distinct data structures means disparate memory usage. For traditional pointer-based quad-trees, six entries are required to obtain complete representation, which include 5 pointers referring to the one parent and 4 child nodes and one integer labeling its state. For maps of large ranges or containing dense obstacles, the required memory is prohibitive.

By contrast, pointer-less quad-trees save substantial memory space via well-designed encoding principles. For instance, the method in [30] needs only four integers to express a tree node (i.e., one for layer, two for coordinates, and one for value), which saves more than 33% of memory. Nonetheless, our codified quad-trees have much more remarkable memory-efficient effect. There are no integer entries needed but a bit sequence. For a map sized of 256×256, a bitset instance at the length of 24 in C++ is adequate to represent a tree node. Considering that there are several bits left without use, we can exploit 2 bits of them to specify the state of corresponding node. Therefore, only 3 bytes in memory are needed. Supposing each entry in the two methods above requires 4 bytes, our encoding method can reduce the memory usage to 12.5% and 18.75%, respectively. Certainly, an integer to indicate the state is a more natural option but consumes a little more memory.

6.2. Analysis of State Abstractions Applied in LRTS. Sturtevant in [32] proposed several characteristics that are expected to be considered when comparing methods of space representations: memory usage, the ease of localization, planning, smoothing, path following, and dynamic modification of the representation. Here based on some of them, we make a comparison between EQ LRTS and PR LRTS which typically applies hierarchical abstraction in real-time search.

(1) Memory Usage. We have illuminated in the above sections that our encoding principles are complete for representing the entire quad-trees and simultaneously far more memory-efficient than their counterparts. For the clique hierarchy, this needs a data structure used for states mapping upwards and relationship of neighborhood on every level. Besides, for intermediate states are always of irregular shapes, methods of simply codifying them are not feasible. Thereby this will result in evidently more memory space used.

(2) Ease of Localization. Given the spatial coordinates of an underlying state, quad-tree abstraction localizes it by upwards computing the codes of its ancestors via bit operations and finds them in already set hash table. Regarding the cliques, it is necessary to find the relevant abstract states in all higher levels through convenient pointers. As a consequence, the two take no evident advantage over each other.

(3) Planning. To the virtue of multiresolution property, EQ LRTS has a reduced search space which is comprised of quad-tree leaf nodes. Searching and planning can be directly implemented on these nodes, just as the original LRTS is operated on the grids. As a contrast, PR LRTS requires finishing the planning at every time on each level. Though its search space is reduced hierarchically, it still seems somewhat trivial.

(4) Smoothing and Path Following. To the best of our knowledge, there is no efficacious online smoothing method to support real-time search on quad-trees. Paths generated by quad-trees are suboptimal because they are constrained to segments between the centers of the cells. As for its counterpart, PR LRTS has little suboptimality which is a noticeable quality.

(5) Dynamic Modification. Partial environment changes only require the quad-trees to conduct local modification which merely involves node insertion and removal. This is not the case for clique abstraction which always has to repair and maintain the connectivity between states on the entire hierarchy. Thereby EQ LRTS possesses more efficient dynamic modification.

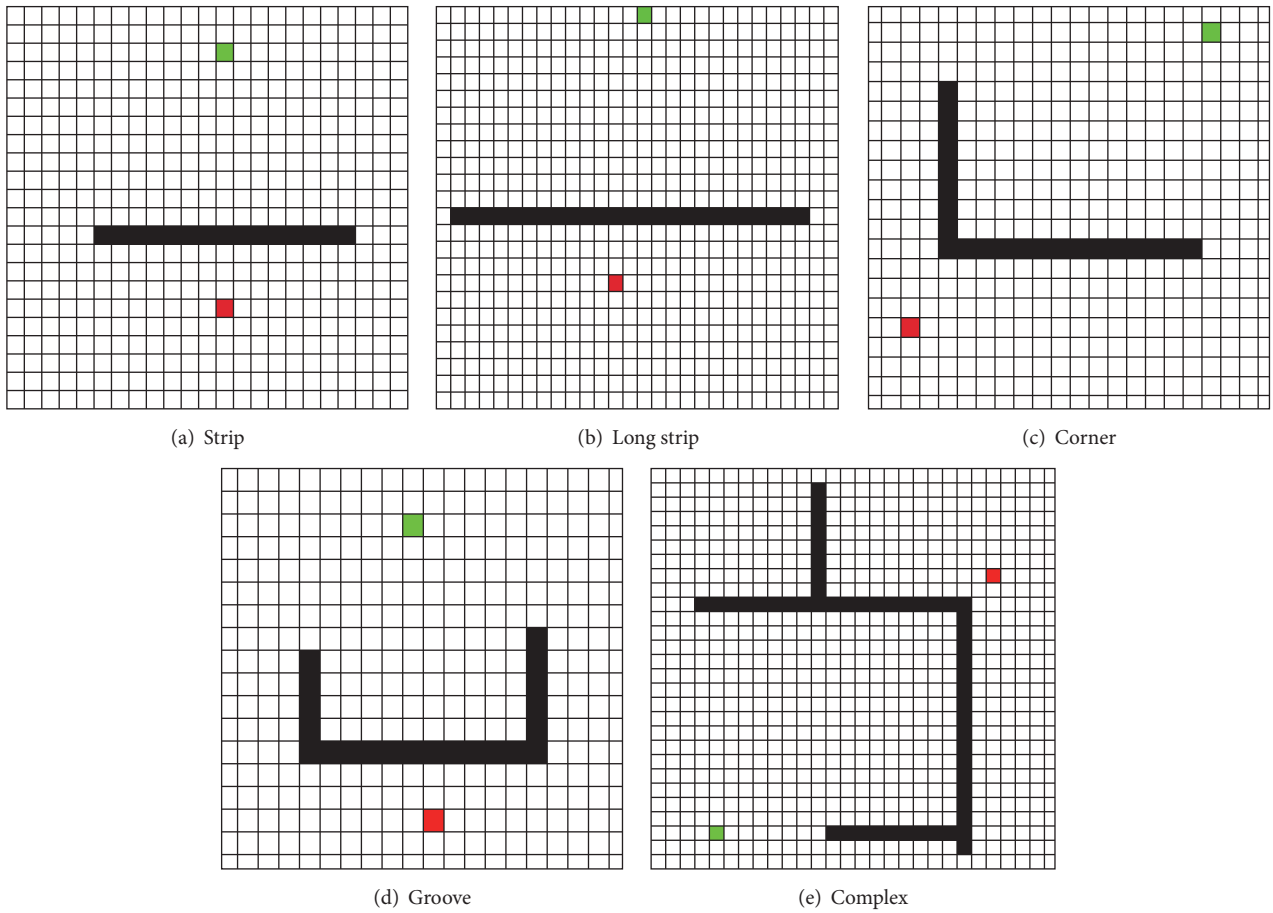


Figure 6: Five representative types of topographies which engender heuristic depressions.

Seen form these comparisons, the two methods possess significant disparities. EQ LRTS mainly takes advantages over PR LRTS on memory usage and ease of localization, while PR LRTS is able to generate more smooth paths.

6.3. Experiments and Evaluation. In experiments, particular terrains are selected to test the algorithm and several metrics are proposed to record the number of traveled states and repeated visits which can indicate the "scrubbing" behaviors of agents. All the results of LRTS on quad-trees will be compared with LRTS on uniform grid maps. And we conducted the study via Visual Studio 2010 and on Intel Core i7-4790 CPU at 3.60 GHz.

6.3.1. Terrain Selection. We select terrains of five typical formations (*ordinary strips, strips of extreme length, corners, grooves,* and *complexes*, shown in Figure 6) to test agents' behaviors where heuristic depressions exist. And these obstacle distributions are imitations of long walls, rooms, malls, and so on in reality. As for soldiers in RTS games, their enemy can form fortification in such formations.

In such maps, heuristic depressions in front of the "walls" drive agents to undertake heuristic updates at a more considerable frequency compared to agents in usual maps. From an intuitive perspective, the "blind" agents are eager to escape form the trenches, long walls, or semiclosed rooms quickly but scrubbing behavior makes them look "depressed" and even "crazy."

6.3.2. Evaluation Metrics. Several indicators can scale the algorithm performance at heuristic local minima from different perspectives, shown in Table 1.

6.3.3. Results and Analysis. Under every terrain mentioned in Section 6.3.1, we design 15 maps artificially to acquire results statistically, which share similar sizes to eliminate differences within the same terrain. Maps of different groups could engender evident discrepancy, owing to their distinct obstacle distribution and agent behaviors. Nonetheless, generally speaking, the combination of quad-tree abstraction and LRTS takes obvious advantage over its counterpart.

Seen from the chart in Figure 7, all of the three criteria witness a significant improvement of the algorithm performance on accelerating the escape. Specifically, worthy of notice are the following:

(1) Real-time search on the quad-tree abstraction just visits about 10% or even less nodes compared with

Table 1: Performance measures of the algorithms to specify their amount of planning, state revisitations, heuristic updates, and runtime. These indicate whether EQ LRTS effectively speeds up heuristic learning.

Metric name	Semantics
NRVN	The number ratio of visited nodes of LRTS on quad-trees and uniform cells. Smaller NRVN means that EQ LRTS possesses a more effective planning process and visits fewer nodes.
TRRV	The times ratio of repeated visits of LRTS on quad-trees and uniform cells, which is the core index to illuminate how many times the agents visit the nodes they have already traversed. Smaller TRRV means less dramatic "thrashing" and vice versa.
RUT	The ratio of updates times, which is an inner and microcosmic index to point out how many times the program has done the heuristic updates. This index has a similar effect to TRRV but is dependent on the heuristic function to some extent.
RR	The runtime ratio, which can be seen as a bottom line. Although more calculation and operations are involved in quad-tree modeling than uniform cells, running EQ LRTS is not expected to consume a lot more time than its counterpart.

Table 2: The contrast between EQ LRTS and original LRTS on the runtime.

Methods	Terrains				
	Strips	Long strips	Corners	Grooves	Complexes
	Average runtime (ms)				
LRTS on Quad-tree	1.06	2.91	10.28	12.11	16.30
LRTS on uniform cells	0.74	4.40	6.02	6.80	10.87
Average RR	1.84	0.70	2.20	2.28	1.60

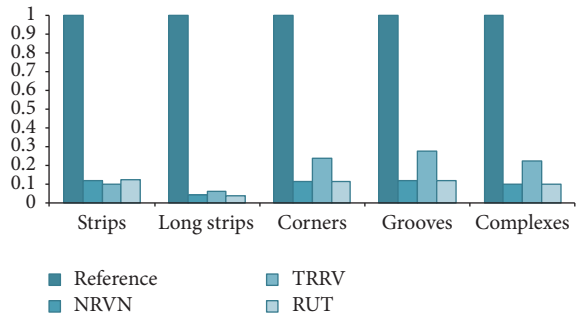


Figure 7: Experimental results of NRVN, TRRV, and RUT which reflect the performance of the algorithms.

the common method, substantially curtailing the planning amount. As Section 1 states, small planning amount is an important pursuit of real-time search.

(2) The quantity of repeated visits plunges to just around 20% (not that approaching in different formations), rendering the agents not so crazy that they can stop thrashing in an acceptable amount of time.

(3) The values of NRVN and RUT are pretty closed to each other no matter what the terrains look like. This fact educes that the indicator of how many times updates happen is tied up to the total planning amount.

(4) Each of the three indicators shows its best in the terrain of long strips, far better than those in other map types.

What is more, we have to specially be concerned with RR. In the whole, the proposed method runs for a little bit more time to complete the planning given that it has to dispose of necessary operations to determine the neighbor nodes. The result is shown in Table 2.

The table tells that though EQ LRTS consumes more time in the mass, there does not exist an evident gap between the average RR in the two methods, and even the new method takes an advantage in the terrain of long strips. Furthermore, both of the methods are time-saving, just spending several milliseconds to finish the searches.

Figure 7 and Table 2 interpret our experiments based on average values, to show improved effects on the whole. However, we have to pay some concern on the differences among samples.

Figure 8 shows the box plots of NRVN, TRRV, and RUT of all the 75 maps. The boxes tell us the value ranges, medians, quartiles, and singular points. From the figure, we can see that the algorithm improves the original one to different extent when different maps are used. Specifically,

(1) at local minima generated by strips, long strips, and complexes, the proposed method improves the three criteria hovering at a relatively steady level, shown by their narrow boxes. This means that local change of maps of the same terrains has a little influence on the algorithm performance,

(2) by contrast, the maps of corners and grooves have rather wide boxes, indicating that corresponding values of metrics are distributed incompactly. For instance in Figure 9, the NRVN of Figure 9(a) is

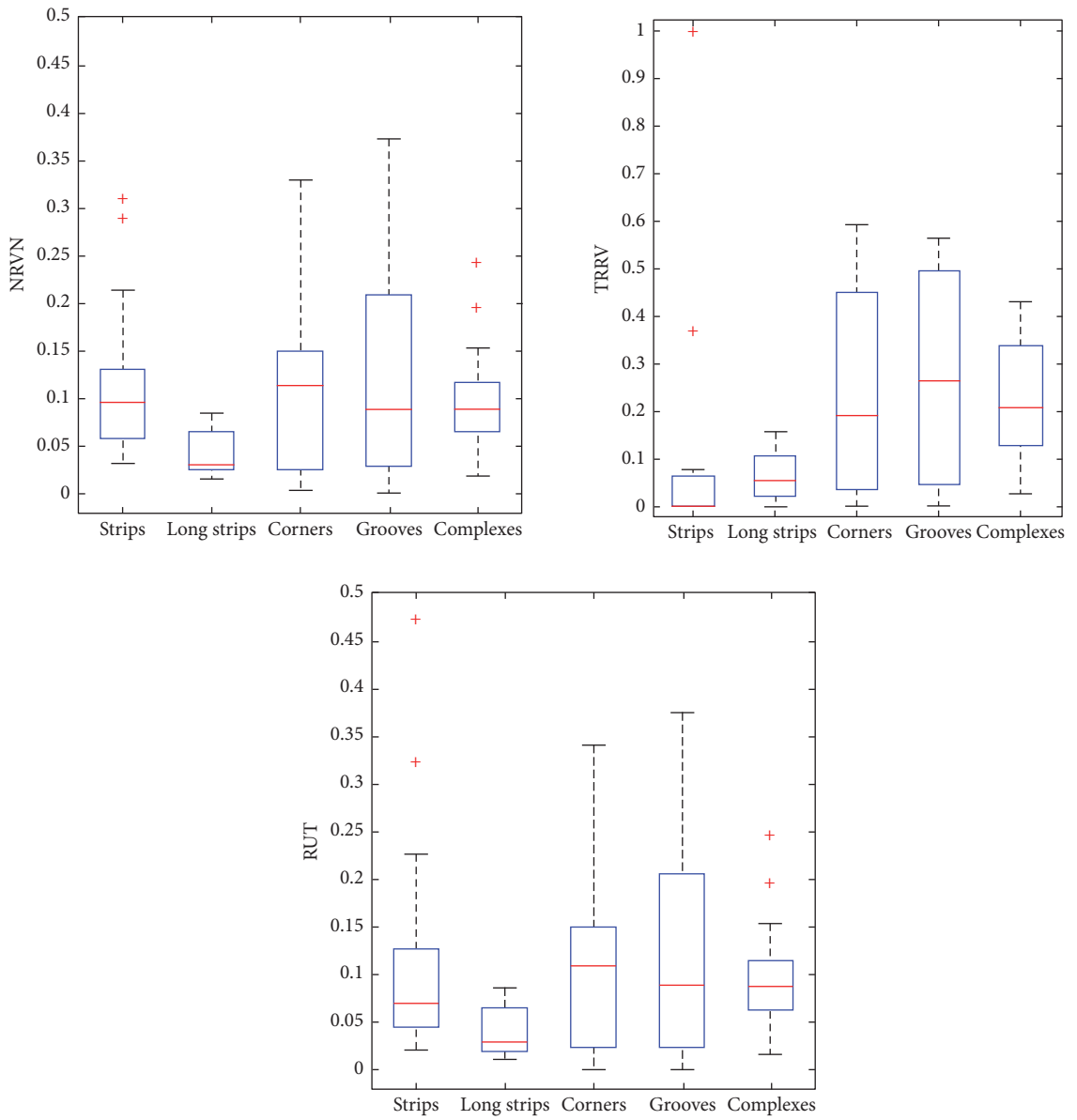


Figure 8: Box plots of NRVN, TRRV, and RUT. The "+" symbols stand for outliers, also called singular points, whose quantities are far smaller than those of lower quartiles or much greater than those of the higher quartiles.

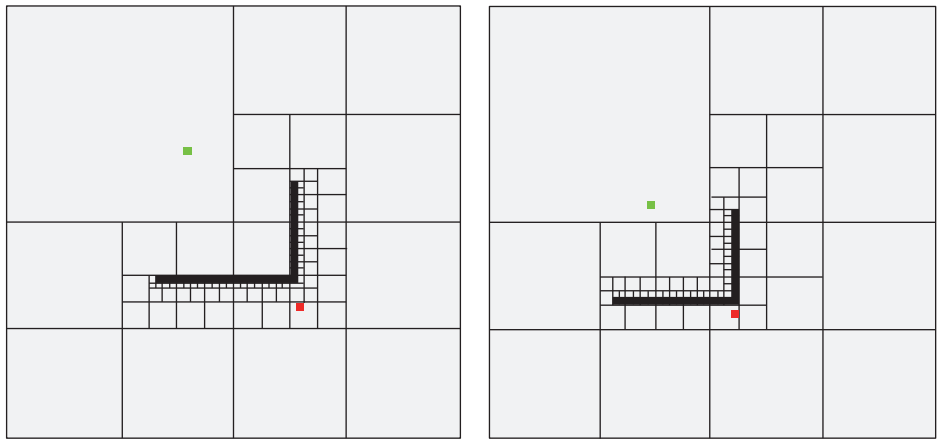
just 1.4%, while Figure 9(b) stands over 22%. Back to the problem, within the same terrain formations and approaching scales, maps which differ a little from each other may induce results which are numerically far apart. In other words, local structures of terrains (especially corners and grooves) impact significantly the quad-tree modeling, although its performance still overrides the traditional one.

7. Conclusions and Future Work

In this paper we present an algorithm named EQ LRTS to incrementally update quad-tree representation for agents' search space and accelerate learning in real-time search. Our encoding principles are proved to be a lot more memory-efficient than other methods expressing quad-trees. And with local repair mechanism, agents can refine their quad-tree models during exploration. More importantly, EQ LRTS applies the idea of state aggregation and heuristic generalization to refine heuristics and help agents escape from heuristic depressions quickly. Simultaneously, the algorithm also opens future work of smoothing the quad-tree generated paths at online stage, which will improve the quality of paths.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding of the publication of this article.



(a) (b)

Figure 9: A little change of terrain bringing about significant difference in quad-tree modeling.

Acknowledgments

The work described in this paper is sponsored by the National Natural Science Foundation of China under Grant no. 61473330 and Grant no. 61573369. The authors appreciate the fruitful discussion with the Sim812 Group: Yong Peng, Shiguang Yue, and Qi Zhang.

References

[1] 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*, vol. 4, no. 2, pp. 100–107, 1968.

[2] R. E. Korf, "Real-time heuristic search," *Artificial Intelligence*, vol. 42, no. 2-3, pp. 189–211, 1990.

[3] M. Shimbo and T. Ishida, "Controlling the learning process of real-time heuristic search," *Artificial Intelligence*, vol. 146, no. 1, pp. 1–41, 2003.

[4] D. Furcy and S. Koenig, "Speeding up the convergence of real-time search," in *Proceedings of the 17th National Conference on Artificial Intelligence and 12th Conference on Innovative Applications of Artificial Intelligence*, pp. 891–897, Austin, Tex, USA, August 2000.

[5] L. Y. Shue and R. Zamani, "An admissible heuristic search algorithm," in *Proceedings of the International Symposium on Methodologies for Intelligent Systems (ISMIS '93)*, pp. 69–75, Trondheim, Norway, June 1993.

[6] V. Bulitko, "Learning for adaptive real-time search," Tech. Rep., Computer Science Research Repository (CoRR), 2004.

[7] S. Russell and E. Wefald, *Do the Right Thing: Studies in Limited Rationality*, MIT Press Series in Artificial Intelligence, MIT Press, Cambridge, Mass, USA, 1991.

[8] T. Ishida, "Moving target search with intelligence," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 525–532, San Jose, Calif, USA, July 1992.

[9] V. Bulitko and G. Lee, "Learning in real-time search: a unifying framework," *Journal of Artificial Intelligence Research*, vol. 25, pp. 119–157, 2006.

[10] S. Koenig and X. Sun, "Comparing real-time and incremental heuristic search for real-time situated agents," *Autonomous Agents and Multi-Agent Systems*, vol. 18, no. 3, pp. 313–341, 2009.

[11] S. Koenig and M. Likhachev, "A new principle for incremental heuristic search: theoretical results," in *Proceedings of the 16th International Conference on Automated Planning and Scheduling (ICAPS '06)*, pp. 402–405, Cumbria, UK, June 2006.

[12] C. Hernańdez and P. Meseguer, "Improving LRTA∗(k)," in *Proceedings of the 20th International Joint Conference on Artificial Intelligence (IJCAI '07)*, pp. 2312–2317, Hyderabad, India, January 2007.

[13] V. Bulitko, M. Luštrek, J. Schaeffer, Y. Björnsson, and S. Sigmundarson, "Dynamic control in real-time heuristic search," *Journal of Artificial Intelligence Research*, vol. 32, pp. 419–452, 2008.

[14] V. Bulitko, Y. Björnsson, and R. Lawrence, "Case-based subgoaling in real-time heuristic search for video game pathfinding," *Journal of Artificial Intelligence Research*, vol. 39, pp. 269–300, 2010.

[15] C. Hernández and J. A. Baier, "Fast subgoaling for pathfinding via real-time search," in *Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS '11)*, pp. 327–330, Freiburg, Germany, June 2011.

[16] B. Lau, C. Sprunk, and W. Burgard, "Efficient grid-based spatial representations for robot navigation in dynamic environments," *Robotics and Autonomous Systems*, vol. 61, no. 10, pp. 1116–1130, 2013.

[17] N. Sturtevant and M. Buro, "Partial pathfinding using map abstraction and refinement," in *Proceedings of the National Conference on Artificial Intelligence*, pp. 1392–1397, 1992.

[18] R. C. Holte, M. B. Perez, R. M. Zimmer, and A. J. MacDonald, "Hierarchical A*: searching abstraction hierarchies efficiently," in *Proceedings of the 1996 13th National Conference on Artificial Intelligence (AAAI '96)*, pp. 530–535, Portland, Oregon, August 1996.

[19] R. C. Holte, T. Mkadmi, R. M. Zimmer, and A. J. MacDonald, "Speeding up problem solving by abstraction: a graph oriented approach," *Artificial Intelligence*, vol. 85, no. 1-2, pp. 321–361, 1996.

[20] V. Bulitko, N. Sturtevant, J. Lu, and T. Yau, "Graph abstraction in real-time heuristic search," *Journal of Artificial Intelligence Research*, vol. 30, pp. 51–100, 2007.

[21] V. Bulitko, N. Sturtevant, and M. Kazakevich, "Speeding up learning in real-time search via automatic state abstraction," in *Proceedings of the 20th National Conference on Artificial Intelligence and the 17th Innovative Applications of Artificial Intelligence Conference (AAAI '05/IAAI '05)*, pp. 1349–1354, Pittsburgh, Pa, USA, July 2005.

[22] C. Hernndez and P. Meseguer, "LRTA*(k)," in *Proceedings of the International Joint Conference on Artificial Intelligence*, July 2005.

[23] N. R. Sturtevant, V. Bulitko, and Y. Björnsson, "On learning in agent-centered search," in *Proceedings of the 9th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS '10)*, pp. 333–340, Toronto, Canada, May 2010.

[24] N. R. Sturtevant and V. Bulitko, "Learning where you are going and from whence you came: h- and g-cost learning in real-time heuristic search," in *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI '11)*, pp. 365–370, Barcelona, Spain, July 2011.

[25] C. Hernandez and J. A. Baier, "Avoiding and escaping depressions in real-time heuristic search," *Journal of Artificial Intelligence Research*, vol. 43, pp. 523–570, 2012.

[26] Y. Quanjun, Q. Long, P. Yong, and D. Wei, "Learning in real-time search on C-space GVDs," *Frontiers of Computer Science*, In press.

[27] N. Sturtevant and R. Jansen, "An analysis of map-based abstraction and refinement," in *Proceedings of the International Symposium on Abstraction, Reformulation, and Approximation (SARA '07)*, pp. 344–358, Whistler, Canada, July 2007.

[28] A. Botea, M. Müller, and J. Schaeffer, "Near optimal hierarchical path-finding," *Journal of Game Development*, vol. 1, pp. 7–28, 2004.

[29] R. Lawrence and V. Bulitko, "Database-driven real-time heuristic search in video-game pathfinding," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 5, no. 3, pp. 227–241, 2013.

[30] Q. Yin, L. Qin, X. Liu, and Y. Zha, "Incremental construction of generalized Voronoi diagrams on pointerless quadtrees," *Mathematical Problems in Engineering*, vol. 2014, Article ID 456739, 14 pages, 2014.

[31] Y. Li, M. Q.-H. Meng, S. Li et al., "A quadtree based neural network approach to real-time path planning," in *Proceedings of the IEEE International Conference on Robotics and Biomimetics (ROBIO '07)*, pp. 1350–1354, IEEE, Sanya, China, December 2007.

[32] N. R. Sturtevant, "Choosing a search space representation," in *Game AI Pro: Collected Wisdom of Game AI Professionals*, pp. 253–258, CRC Press, 2013.