Appl Intell (2012) 36:348–368
DOI 10.1007/s10489-010-0265-9

Formation preserving path finding in 3-D terrains

Ali Galip Bayrak · Faruk Polat

Published online: 27 November 2010

Abstract Navigation of a group of autonomous agents that are required to maintain a formation is a challenging task which has not been studied much especially in 3-D terrains. This paper presents a novel approach to collision free path finding of multiple agents preserving a predefined formation in 3-D terrains. The proposed method could be used in many areas like navigation of semi-automated forces (SAF) at unit level in military simulations and non-player characters (NPC) in computer games. The proposed path finding algorithm first computes an optimal path from an initial point to a target point after analyzing the 3-D terrain data from which it constructs a weighted graph. Then, it employs a real-time path finding algorithm specifically designed to realize the navigation of the group from one waypoint to the successive one on the optimal path generated at the previous stage, preserving the formation and avoiding collision. Software was developed to test the methods discussed here.

Keywords Path finding · Formation · Autonomous agents

1 Introduction

1.1 The subject

Navigation of a group of mobile agents in coordination is a popular problem studied in different areas such as robotics, computer games and military simulations. In this paper we consider the problem of moving a team of agents from an initial location to a final location while preserving a predefined formation on a 3-D terrain. Formation is defined to be the tactical arrangement of agents as a team, like column, line and wedge (see Fig. 1), used especially in military forces.

Several path finding algorithms were proposed for single and multiple robots [2, 7, 9, 13, 16, 19, 34]. These algorithms were generally developed for navigation in 2-D environments and can not be used or efficiently adopted to 3-D terrains. Besides, they mainly focus on the physical capabilities of the robots, which is not required in computer simulations, which is our primary interest in this paper. In computer games, especially in strategy games, algorithms are designed for moving a group of soldiers or vehicles in the simulated battlefields [8, 29, 33]. In military simulations, there has been growing interest in modeling behaviors at both individual and unit level to simulate decision making and tactics used in real life for simulation based training, analysis and acquisition (OneSAF [21], TeBAT [32], ModSAF [10], SWARMM [18]). In that respect, navigation of individual soldiers/vehicles and that of a unit require efficient algorithms to be embedded in Semi-Automated Forces (SAF) [23]. There are only a few published works for the

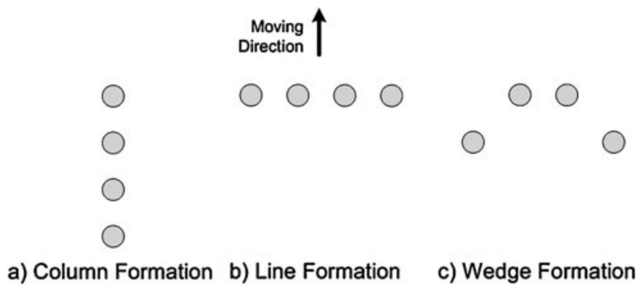


Fig. 1 Some common formations

A.G. Bayrak
EPFL-IC-LAP, INF136, Ecole Polytechnique Fédérale de Lausanne, Station 14, 1015, Lausanne, Switzerland
e-mail: aligalip.bayrak@epfl.ch

F. Polat (✉)
Department of Computer Engineering, Middle East Technical University, 06531 Ankara, Turkey
e-mail: polat@ceng.metu.edu.tr

so-called path finding task in the areas such as military simulations and computer games. Motivated by this, we developed algorithms for formation preserving navigation task of multiple autonomous agents in 3-D terrains and tested them on real 3-D maps.

1.2 Scope and objective

In this paper, we developed a novel algorithm for finding an efficient path between two points for a group of agents that are also required to maintain a predefined formation and avoid collisions with each other and with obstacles in 3-D terrains. We divided the problem into three main parts; constructing a search graph from 3-D terrain data, finding the path and maintaining the formation while moving [3].

An overview of the proposed method is given in Fig. 2. We first analyze and hence identify important terrain features to construct a directed weighted search graph. In this graph, *nodes* correspond to the waypoints some of which will define the route, *edges* correspond to the accessibility of connected waypoints, finally *weights* to the cost between the connected waypoints. This graph is constructed once at the beginning before the team actually starts to move.

Then, with the help of an off-line planner, we determine a path for the team on the constructed search graph. The planner first uses an informed search technique, namely A^*, and finds an optimal path (a sequence of waypoints in the order they will be visited). Then it uses a smoothing algorithm in order to smooth the found path, which may be jagged. Finally, the planner determines each agent's goal position on every waypoint on the path. Note that, the off-line planner is also executed before the agents start to move.

The last step is to move the agents in real-time using the on-line planner. The agents, in parallel, follows the path found by the high-level planner, by using a real-time algorithm that plans and navigates agents between two successive waypoints on the solution path, avoiding collisions and maintaining the formation. With the help of our on-line planner, the team is able to rearrange the formation when an agent loses its mobility, which may be faced in games and military simulations.

Concerning the measurement of quality we focus on reducing the search space, namely the weighted graph derived through the terrain analysis, which is the most critical issue. Then we employed the classical A^* algorithm with an admissible heuristics function and enjoy using a complete and optimal algorithm. Then, the on-line planner navigates each agent with a constant cost per time step. Our method aims at finding the shortest path and at the same time preserve the formation. In case it is not possible to find a path keeping the initial formation, we employ a method that tolerates to some degree, specified by the user, scaling down the formation necessary enough to find a path. Note that semi-automated

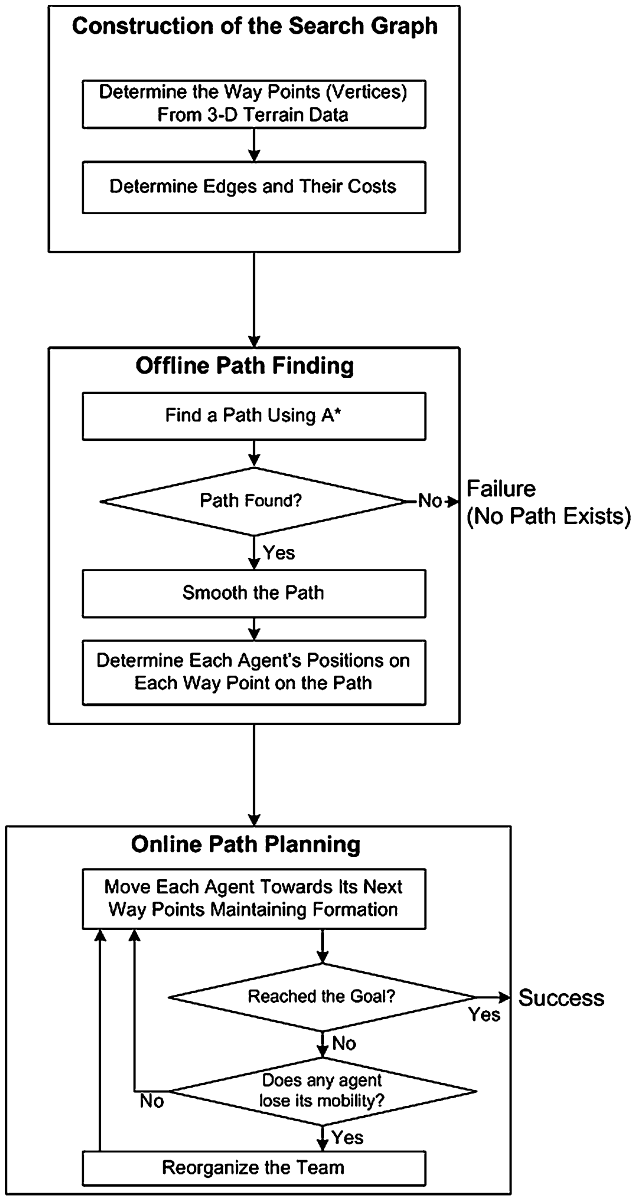

Construction of the Search Graph
Determine the Way Points (Vertices) From 3-D Terrain Data
Determine Edges and Their Costs
Offline Path Finding
Find a Path Using A*
Path Found?
No
Failure (No Path Exists)
Yes
Smooth the Path
Determine Each Agent's Positions on Each Way Point on the Path
Online Path Planning
Move Each Agent Towards Its Next Way Points Maintaining Formation
Reached the Goal?
Yes
Success
No
Does any agent lose its mobility?
No
Yes
Reorganize the Team

Fig. 2 Steps of the proposed formation preserving path finding method

forces (SAF in short) or non-player characters that must navigate and also maintain formation as much as possible without human intervention may experience worst case scenarios that require changing formation radically. The way the method handles this sort of situations is described in Algorithm 6 during the off-line planning phase, simply scale the formation width and if necessary formation depth. At the extreme case (i.e., scaling down the formation to the end), the next waypoint for the whole group will be determined as the next waypoint for each team member and hence the team will move in Line formation.

We developed the path planning algorithms based on the assumption that the terrain is known, which is most of the time a reasonable assumption for military simulation applications and computer games. For example, in military operations, commanders work on maps with different resolutions on which they mark and plan the waypoints and corridors for all troops.

Our method can easily be extended for partially unknown environments as follows. Off-line planner is run for the known part of the environment, then the on-line planner is started. In case some unknown part becomes accessible (observable) during the navigation, on-line planner is paused and off-line planner is restarted to generate the new path, then on-line planner resumes its operation taking into account the new path. This way, it is possible to use the method for partially unknown or dynamic environments.

1.3 Outline

In Sect. 2, related work is given. In Sect. 3, we define the environment and describe how terrain features are extracted from the 3-D terrain data and how the search graph is constructed. In Sect. 4, we first give an overview of the proposed approach to formation preserving path finding and then describe our off-line and on-line planning algorithms. Experimental results and sample runs are given in Sect. 5. Finally, conclusions and future research directions are given in Sect. 6.

2 Related work

There is a growing interest in autonomous agents and multi-agent systems in computer simulations [26]. In most of the games, there are non-player characters that act autonomously to overcome some real-life problems, either in cooperation with or against the player. While the single-agent decision making mechanism is more popular in games, in some games, especially in real-time strategy games, multi-agent systems are used, e.g. agents can work on a cooperative task in order to beat the player. Also, in military simulations, soldiers and vehicles, either individually or at unit level can simulate the decision making and tactics used in real-life.

Since mostly the agents are mobile, the most frequently faced problem in such simulations is the path finding problem. Many algorithms have been developed for single-agent path planning, however there are less publications for multi-agent systems [31]. Multi-agent formation preserving path finding is used especially in military simulations and games. In military simulations like OneSAF [21] (One Semi-Automated Forces) and ModSAF [10] (Modular Semi-Automated Forces) and games like Force21 [33] units are needed to move on 3-D terrain, while preserving the formation.

The path finding problem mainly consists of three parts: how to represent the environment and construct a search graph, how to search and find a path on the constructed graph and how to realize the found path in real-time.

For representation of environment, there are many approaches. However, most of them are designed for 2-D and not very efficient in 3-D domains. The representations generally focus on two important factors. One is the number of vertices it generates for the search graph and the other is the reliability of the generated graph. The number of vertices is very important, such that generally the higher the number of vertices, the more time it takes while searching on the graph. Reliability is a key factor, such that the constructed graph should preserve the connectedness of vertices. For example, if one point is reachable from the other, then on the search graph it should be so, and vice versa.

The most popular environment representation is the grid representation. The environment is divided into equal sized square cells, each of which is either reachable or not, and these cells are considered as the vertices of graph. The edges of the graph are between the reachable neighboring cells. This representation can be implemented in such a way that it preserves reliability, but efficiency is not the main concern of the representation.

Another popular representation is the visibility graph [4]. It considers the environment as a huge polygon, in which there are small polygons representing the unreachable areas. Then, each corner of these polygons are considered as vertices of the graph and the visibility of a pair to each other determines the edges. This representation is efficient in 2-D environments.

Delaunay triangulation can also be used for environment representation [4]. It divides the environment into triangles, using the corners of edges. Centers of these triangles can be used as vertices and edges are inserted between neighboring ones if they are accessible. Voronoi diagram is another method, which is dual to Delaunay triangulation in graph theoretical approach.

Probabilistic roadmap algorithms (PRM) use a similar three phased approach [11, 15, 27]: generate a connected graph (road-maps) in obstacle-free space randomly, try to connect initial and target points to the roadmap and finally search a path on the roadmap between initial and target point. There is no guarantee that an existing path on the original map will be found using a PRM. PRMs are classified as probabilistically complete. If the sampling coverage of the terrain approaches 100 percent then PRM can be used to find any valid path present in the original environment. Despite this property, PRMs have been shown to work quite well in practice, solving a large number of problems in very complex environments. The key to building a good PRM is the sampling strategy; a sufficient number of points have to be selected to reasonably approximate the original environment.

The second important phase in path finding is the search on the constructed graph. Since time efficiency is one of the most important factors in real-time applications, informed search techniques, especially A^*, are generally used. A^* is the most widely-known form of best-first search [25]. A^*, at each step during the search, expands the node having the lowest $f(n)$ value, where $f(n) = g(n) + h(n)$, $g(n)$ is the path cost from the start node to node n and $h(n)$ is the heuristic function which is the estimated cost of the cheapest path from node n to goal node. A^* is optimal if $h(n)$ is an *admissible heuristic*, that is, provided that $h(n)$ never overestimates the cost to reach the goal.

The last step in path finding is the realization of the found path. In the single agent case, this problem is easy to handle. The agent can move on the line between successive waypoints along the path and simple algorithms can be used in order to avoid collisions. However, the multi-agent path finding problem brings many additional constraints compared to the single-agent case, especially if run on 3-D terrains. Agents should consider the mobility, position and velocity of other agents. Representation of environment is an important factor at this point. The *Virtual structure*, introduced in [16] by Lewis and Tan, considers the agent group as a rigid body and planning is done for this body. The most common method is to let each agent decide its own path according to its relative position with respect to the other agents. Desai et al. presented a graph theoretical approach where each agent determines its location according to the locations of other agents and there is a leader agent who does not follow any other agent but leads the group [9]. The relation between two agents consists of the relative distance and orientation between them. There are several methods that make use of priorities among agents to describe a formation in a team [19].

In [5], Bourgeot et al. presented a method for planning path for a biped robot in 3D terrains. Similar to our approach, the method first determines a reference path and then navigates the biped robot through reference track. 3D environment made up of triangles is classified into three types of grounds, namely flat ground, tilt ground and stair ground. A sequence of suitable footholds keeping robot stability and motion continuity is generated. The path-planning algorithm navigates the robot over a path which avoids tilt ground and which prefers pitching than rolling when slope is not avoidable.

There are several studies on crowd simulation in social sciences, psychology, civil and traffic engineering, etc. Most of them use the discretization of the environment. Some common approaches include agent-based methods [24], cellular-automata methods [17], potential fields [22] and particle dynamics [12]. Recently, a novel approach for crowd simulation based on continuum dynamics has been proposed by Treuille et al. [30]. This work computes a dynamic potential field that simultaneously integrates global navigation with local obstacle avoidance. The work on crowd simulation is concentrated on the problem of handling large groups of individuals, inspired from dynamics of flock behaviour and fish schooling. Due to the number and diversity of individuals involved in crowds, the activity of formation maintenance and navigation is not task oriented.

3 Description and representation of the environment

3.1 Properties of the environment

The environment is a 3-D virtual world made up of a terrain and objects placed on it. The terrain is represented as a height map. The natural and man-made objects such as trees and buildings are represented with polygons and they are treated as obstacles. The environment is considered to be fully observable from the point of view of the autonomous agents, but our method can easily be extended for partially observable environments with a little improvement. The percepts and actions of agents are handled at discrete time steps, in other words, the environment changes in discrete-time points. However, agents can move to a range of continuous points on the terrain at each time step, i.e., movements take place in continuous-space.

Agents are mobile and holonomic[1] and organized as a team having a specified formation. The team aims to reach to a specified target point with a predefined average speed. The team has to maintain formation and avoid collisions, and to achieve this agents in the team may tune their speeds. Agents in the team are considered to be physically identical and furthermore they are restricted by their physical properties, such as maximum/minimum slope they can ascend/descend, maximum positive/negative acceleration and maximum speed they can reach. We assume that agents in the team can communication with each other in order to preserve formation while moving.

3.2 Representation of the environment and construction of the search graph

Concerning the representation of the environment, the most common approach is the *grid* representation where the environment is divided into a number of square cells of the same size, where each cell is considered either as obstacle, or not. Search algorithms for finding a path from one cell to another, may consider the set of midpoints of these cells. The drawback of this method is that cell size should be determined carefully. Figure 3 shows some examples of the grid representation with different cell sizes. If the size is small, the number of points in the search space will be huge, as shown

[1] If the controllable degrees of freedom is equal to the total degrees of freedom then the robot is said to be holonomic.

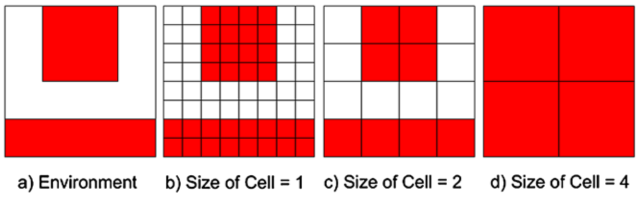


Fig. 3 Grid representation

in Fig. 3(b). If the size is big, the homogeneity (obstacle/not) of the cells will decrease since each cell is considered either an obstacle or not, and even it may result in failure of the search as shown in Fig. 3(d). The ideal cell size for this case is shown in Fig. 3(c).

There are some other algorithms based on triangulation, Voronoi diagrams and visibility graphs [4]. However, in 3-D terrains having lots of contour lines, there are too many points that these algorithms generate and need to work on. So, we prefer to modify the grid representation, that minimizes the number of points and preserves homogeneity. In [14], Kambhampati and Davis propose a multi-resolution representation with quad trees. In this paper, we employ a simpler but successful method that analyzes the terrain data, i.e., height map, to extract useful features from which it identifies waypoints to be used by the path planner.

Concerning the representation, we consider the terrain as a 3-D mathematical surface and find its critical and singular points, and construct a search graph using these points as vertices of the graph. Because, by using only the singular and critical points we can determine the most important features of a surface.

First, we divide the map into small square cells and take the center points of these cells, as in the grid method. The size of a cell should be small enough such that it could be considered homogeneous. Size of a cell should be determined according to the requirements of the application domain, specifically storage requirements of the terrain data and computational cost of the application problem. These may include the size of the environment, number of agents, computational cost of the line-of-sight algorithm, expected response time and many other application specific requirements. Let *sizeOfCell* denote the size of a cell. Then, determine the critical points of the terrain as described in Algorithm 1. The extremum points of the terrain are considered critical points of it. The basic idea of the algorithm is as follows. First, *mark* the points that are *local maxima* and *local minima* according to their height. All marked points are added to the set of vertices of the search graph. A point can be considered *local maximum* if its height is greater than all of its neighbors (4 or 8 neighborhoods can be assumed for simplicity, we take 4 neighborhood for this study but the method is easily extended to 8 neighborhood) and *local minimum* if its height is less than all of its neighbors. Also, mark

Algorithm 1 DetermineCriticalPoints(terrainGridData): SetOfPoints

```
1: // neighbor($x$, $dir$): The point that is sizeOfCell unit far
   from $x$ (i.e. neighbor of $x$) in the direction $dir$. $dir$ is one
   of the following: NORTH, SOUTH, WEST, EAST
2: // height($x$): The height of point $x$
3: $S \leftarrow \emptyset$ // Let $S$ denote the set of points to be returned
4: for all point $p$ in terrainGridData do
5:     if $\forall z\ (z \in \{NORTH, SOUTH, EAST, WEST\} \Longrightarrow height(p) > height(neighbor(p, z)))$ then
6:         $S \leftarrow S \cup \{p\}$ // $p$ is local maximum, add it to the set $S$
7:     else if $\forall z\ (z \in \{NORTH, SOUTH, EAST, WEST\} \Longrightarrow height(p) < height(neighbor(p, z)))$ then
8:         $S \leftarrow S \cup \{p\}$ // $p$ is local minimum, add it to the set $S$
9:     else if $\forall z\ (z \in \{NORTH, SOUTH\} \Longrightarrow height(p) > height(neighbor(p, z))) \wedge \forall z\ (z \in \{EAST, WEST\} \Longrightarrow height(p) < height(neighbor(p, z)))$ then
10:        $S \leftarrow S \cup \{p\}$ // $p$ is partial extremum, add it to the set $S$
11:    else if $\forall z\ (z \in \{EAST, WEST\} \Longrightarrow height(p) > height(neighbor(p, z))) \wedge \forall z\ (z \in \{NORTH, SOUTH\} \Longrightarrow height(p) < height(neighbor(p, z)))$ then
12:        $S \leftarrow S \cup \{p\}$ // $p$ is partial extremum, add it to the set $S$
13:    end if
14: end for
15: return $S$
```

the points that are *partial extrema*. A point can be considered as *partial extremum* if it is either maximum or minimum on every opposing pair and it is not local maximum or local minimum. For example, if the height of a point is greater than the height of its south and north neighbors and less than the height of its west and east neighbors it is *partial extremum*. The points that are marked (local maxima, local minima and partial extrema) will form the set of extremum points.

We mark and use the extremum points in search graphs because of two main reasons. The first one is the mathematical property of such points. For example, if we take two consecutive extremum points on a curve and if there is no other singular/critical points between them, curve is smooth in the region between them. In our case, instead of using all the points as vertices of the search graph, we use only the two consecutive extremum points and omit the other points in between, if none of them are *marked* points (all types of *marked* points will be discussed later). The second reason is the physical property of such points. For example, military teams may sometimes prefer to move in valleys (sequence of

Algorithm 2 DeterminesingularPoints(terrainGridData): SetOfPoints

```
1: $S \leftarrow \emptyset$ // Let $S$ denote the set of points to be returned
2: for all point $p$ in terrainGridData do
3:     if $\exists z$ ($z \in$ {NORTH, SOUTH, EAST, WEST} $\wedge$
       inaccessible($p$, neighbor($p, z$))) then
4:         $S \leftarrow S \cup \{p\}$ // $p$ is singular, add it to the set $S$
5:     end if
6: end for
7: return $S$
```

local/partial minimum points) in order to minimize the fuel consumption and hide from the enemy.

The terrain may be rough and hence there may be some very small hills or cavities. To eliminate such extremum points, the condition in (1) is checked against every marked point p. Note that, nearestMarked(p, z) denotes the marked point that is nearest to the point p in direction z. The condition checks four nearest marked points in four neighboring directions and returns whether the height differences between p and each of these four points are within a threshold. If the condition evaluates to true for p, then we unmark p. k can be considered as the height that an agent can pass over.

$$\forall z(z \in \{\text{NORTH, SOUTH, EAST, WEST}\} \implies |\text{height}(p) - \text{height}(\text{nearestMarked}(p, z))| > k) \tag{1}$$

The next step is to determine and *mark* the *singular points*. In mathematics, a point p of a curve c is considered as singular if c is not *well-behaved* in some manner, such as differentiability, at p. Generally, the term is used for the points where the curve is not differentiable. In our work, we classified a point as singular, if it is not accessible from at least one of its neighbors. A point is inaccessible from its neighbor if the slope between them prevents the agent movement (because of agent's physical capability). Algorithm 2 returns the set of singular points.

Figure 4 contains a sample terrain where local maxima and minima are shown with small blue circles and singular points are shown with small red squares.

Singular points are marked in order to indicate inaccessibilities because of terrain features. If we do not mark these points, two vertices in the constructed search graph may be connected even if there is no reachability between them in the terrain. Marking these points guaranties the disconnectedness in such a case.

The next step is to mark the borders of the obstacles. If a point is on the border of at least one obstacle, then it is marked. The borders of obstacles can be determined by using *Bresenham's line drawing algorithm* [6] or any other line drawing algorithm. Figure 5 contains an example where dark squares represent the marked points.

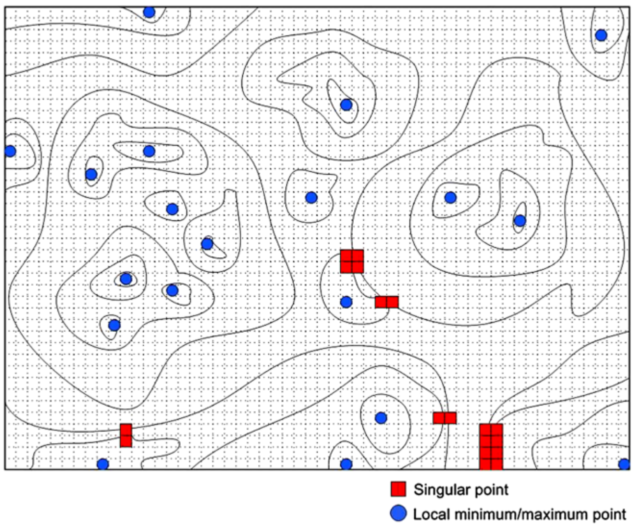


Fig. 4 Sample environment with singular and extremum points

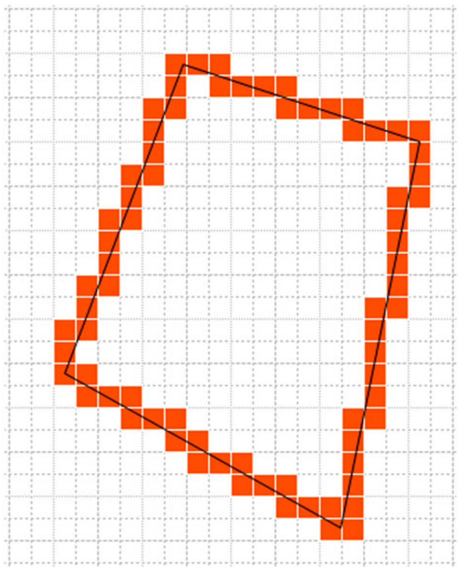


Fig. 5 Obstacle borders shown with orange

Then, we mark the initial and goal locations of agents. Note that if the terrain is flat then there will be no critical/singular points and our algorithm will not mark any point. In such a situation, since we mark the initial and the goal locations, a path is guaranteed to be found.

The points marked up to this point can express the characteristics of the terrain. However, since some of these marked points might be the only marked point in its row/column, the constructed search graph will be disconnected in such a case and the search will fail if the initial and the goal locations are not in the same connected component. To prevent this, we mark some additional points using Algorithm 3. This algorithm is just used to make the constructed graph connected if the terrain is actually connected.

Algorithm 3 AlonePoints(terrainGridData)

```
1: $lastAloneRow \leftarrow 0$
2: for $i = 1$ to $n$ /*$n$ denotes the number of rows*/ do
3:   $numberOfMarkedPoints \leftarrow 0$, $lastMarked \leftarrow 0$
4:   for $j = 1$ to $m$ /*$m$ denotes the number of columns*/ do
5:     if isMarked($i$, $j$) then
6:       $numberOfMarkedPoints{+}{+}$
7:       if $numberOfMarkedPoints > 1$ then
8:         break
9:       else
10:        $lastMarked \leftarrow j$
11:      end if
12:    end if
13:  end for
14:  if $numberOfMarkedPoints == 1 \wedge lastAloneRow \neq 0$ then
15:    mark($lastAloneRow$, $lastMarked$)
16:    $lastAloneRow \leftarrow i$
17:  end if
18: end for
19: $lastAloneColumn \leftarrow 0$
20: for $i = 1$ to $m$ do
21:   $numberOfMarkedPoints \leftarrow 0$, $lastMarked \leftarrow 0$
22:   for $j = 1$ to $n$ do
23:     if isMarked($j$, $i$) then
24:       $numberOfMarkedPoints{+}{+}$
25:       if $numberOfMarkedPoints > 1$ then
26:         break
27:       else
28:         $lastMarked \leftarrow j$
29:       end if
30:     end if
31:   end for
32:   if $numberOfMarkedPoints == 1 \wedge lastAloneColumn \neq 0$ then
33:     mark($lastAloneColumn$, $lastMarked$)
34:     $lastAloneColumn \leftarrow i$
35:   end if
36: end for
```

Note that if the terrain is not connected, the graph will naturally be disconnected. Let's call a point *alone*, if it is the only marked point in its row. Let p_{ij} denote the point on row i and column j. The algorithm, starting from the first row, traverses all the rows and skips the first *alone* point and if it finds an *alone* point p_{kl} on row k, it marks the point p_{kn} where n denotes the column of the previous alone point p_{mn}. Figure 6a shows an example of this step. A similar process is repeated for the *alone* points columnwise as shown in Fig. 6b. After this step, each of the alone points is connected to at least one other alone point.

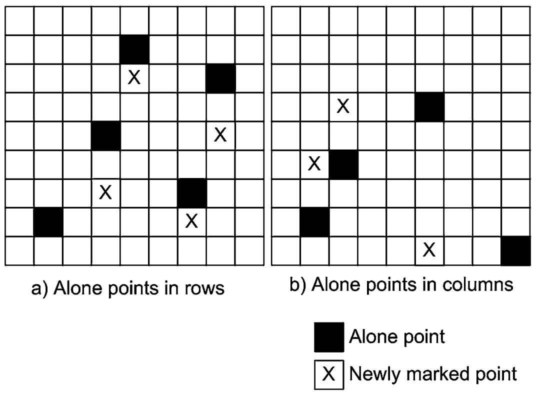


Fig. 6 Preventing *alone* points

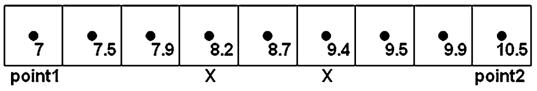


Fig. 7 Marking additional points using *height difference factor*

If the agent only considers the points marked so far in determining the path to its destination, the search will guarantee to find a path if there is one, but the path realized may not be natural since it only considers critical and singular points. In order to prevent this and find more natural paths not only considering extremum and singular points, we mark some additional points as follows. Let's call two marked points *adjacent* if they are on the same row or column and there are no other marked points or obstacles between them. For each adjacent point pairs u and v, we draw a line connecting u to v. Starting from u, find and mark the first point w such that the difference between heights of w and u is greater than or equal to a predefined value called *height difference factor*. We repeat this by considering the most recently marked point in place of the current point (w in place of u in the second iteration) until reaching v. Note that the *height difference factor* can be set to any value as desired, depending on the requirements of the application problem. Two consecutive recently marked points on the line are considered *adjacent* from now on. Figure 7 contains an example where *height difference factor* is 1. The points shown with 'X' are recently marked.

The marked points so far will form the vertices of the graph on which the search will be performed. Since each of the marking algorithm traverses each point of the grid at most constant times, the total complexity of the algorithms used so far is $O(N)$, where N denotes the number of points in the grid. The algorithm decreases the number of vertices significantly compared to grid method, especially if the terrain is not very rough.

The edges of the graph are determined as follows. For each marked point (i.e., vertex) u, if any adjacent point v is accessible from u, an edge incident from u to v is created.

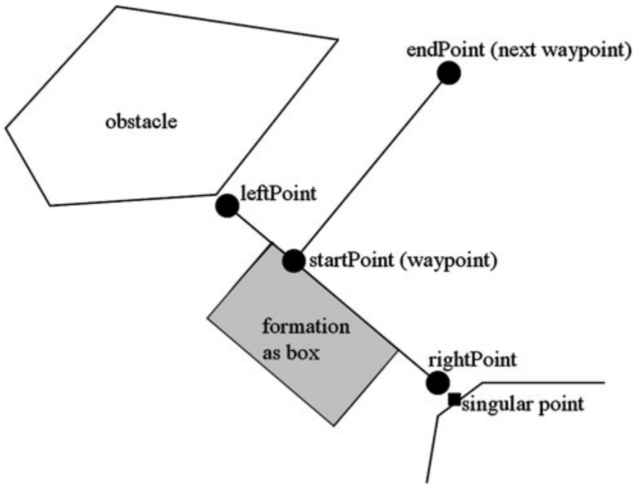


Fig. 8 Checking whether the formation can pass through a point

A point is accessible from another if the slope between them is in the range that is determined by the physical capability of the agent and there is no obstacle in between them.

The cost of an edge is calculated as follows. We divide the edge into small pieces (line segments) each of which has the length *sizeOfCell*, and then sum the costs of all these small pieces. The cost of a small piece is initially assumed to be the Euclidean distance between the two corners (end points) of the piece. Then, we multiply this cost with a function of slope such that the higher is the slope the higher is the cost. The last factor that affects the cost of an edge is whether the desired formation can be maintained along it. Note that this is just a heuristic for calculating the costs of edges of the search graph and the calculation is done before the team actually starts to move. Considering the formation as a rectangular box and checking whether this box can move along the edge can be a solution. However, moving along the whole edge for all edges will be computationally costly. So, we used a computationally cheap method given in Algorithm 4, which checks whether this box can pass through the two vertices incident with the edge. This does not always give the desired result but is a good approximation. Figure 8 shows an example for Algorithm 4 to determine whether the formation can pass through a waypoint, namely *startPoint*. The algorithm finds two points, *leftPoint* and *rightPoint*, on the line perpendicular to the edge, denoting the left and right boundaries of the line accessible from *startPoint* and checks whether the formation can fit into the particular area between them.

If this algorithm returns false, we multiply the cost of the edge with a function of the distance between *leftPoint* and *rightPoint*, such that, the smaller is the distance the higher is the cost. Repeat the same procedure for the end point of the edge, namely *endPoint*. After this step, construction of the search graph is completed. The algorithmic complexity of edge cost detection using above algorithm is $O(E * formationWidth + N)$, if we traverse all grid points once in north-south direction and once in east-west direction and consider at most $2 * formationWidth$ points for each edge, where E is the number of edges and N is the number of grid points. Adding up this complexity with the complexity of vertex determination, total algorithmic complexity of the graph construction is found as $O(E * formationWidth + N)$.

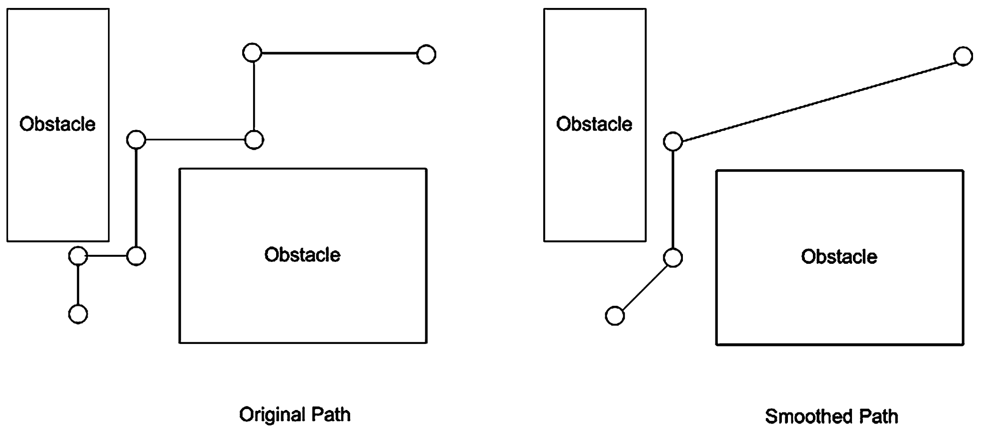
Algorithm 4 CanFit(formationWidth, edge): Boolean

```
1: $startPoint \leftarrow edge.start$
2: $lineP \leftarrow$ a line perpendicular to the $edge$ and passing at $startPoint$
3: $leftPoint \leftarrow startPoint$
4: for $i = 1$ to $formationWidth$ do
5:     if inaccessible($startPoint$, $leftPoint$) then
6:         break
7:     end if
8:     $leftPoint \leftarrow$ point that is $i * sizeOfCell$ far from $startPoint$ on the line $lineP$ and to the left of the $startPoint$
9: end for
10: $rightPoint \leftarrow startPoint$
11: for $i = 1$ to $formationWidth$ do
12:     if inaccessible($startPoint$, $rightPoint$) then
13:         break
14:     end if
15:     $rightPoint \leftarrow$ point that is $i * sizeOfCell$ far from $startPoint$ on the line $lineP$ and to the right of the $startPoint$
16: end for
17: if EuclideanDistance($leftPoint$, $rightPoint$) $\geq formationWidth$ then
18:     return true
19: end if
20: return false
```

4 The proposed formation preserving path planning method

In this section, we will discuss our formation preserving path planning method (Fig. 2). We first construct the search graph as described in previous section. Then, we use an informed search technique (i.e., A^*) to find an optimal path in this search graph. If there exists a path, we use a smoothing algorithm in order to make the found path smooth, which may be jagged due to the number of neighborhoods used in the graph construction. After that, for each waypoint along the path, we determine each agent's position at that waypoint, assuming that the team is able to arrive at the waypoint. All these steps are executed off-line before the team actually starts to move. At this point, we have a path (sequence of waypoints) for each individual agent. Then, in real

Fig. 9 Path smoothing process



time, each agent moves along its own path in coordination with its teammates maintaining the formation and avoiding collision. During on-line path finding, the group is able to reorganize itself in case some agent loses its mobility. When all the team members reach their goal points, the task is accomplished. In the following subsections, we will describe each of the steps of off-line and on-line path planning in detail.

4.1 Off-line path finding

The number of points to be considered in the search in a realistic application can be very large because of the terrain size. That's why there is a need for using an informed search technique to plan the path. We employ the A^* algorithm on the constructed weighted graph to produce an optimal path. Since 4 neighborhood is considered during graph construction phase, the Manhattan distance is used as a heuristic function (namely $h(.)$). Manhattan distance between two points is the sum of the absolute values of differences in each coordinate axis (i.e., in 2-D, $|x_1 - x_2| + |y_1 - y_2|$). If an 8 neighborhood was considered, Euclidean distance might have been used. For the actual distance traveled from source (namely $g(.)$), we use the cost function mentioned in the previous section to determine the edge costs. The heuristic function is admissible, since the Manhattan distance between any two points will never overestimate the actual cost between them.

The path generated by A^* may be jagged because of the number of neighborhoods used. In order for the path to be more realistic, it should be smoothed. For path smoothing, we used the method described in [19] given in Algorithm 5, with some small changes. The basic idea is to remove a point from the path if its predecessor is visible to its successor. For visibility of two points to each other, we can use any Line-of-Sight algorithms. Figure 9 shows an example of the smoothing process.

This algorithm is effective in 2-D. Because of the triangle inequality, the cost of smoothed path can not be larger than the cost of original path. Since the terrain is 3-D and the cost function does not only consider the Euclidean distance, the cost may sometimes grow during this smoothing process. In order to make the path smoother but prevent the cost from growing too much, step 5 of the algorithm is revised as follows:

Algorithm 5 SmoothPath(listOfWayPoints): SequenceOfWayPoints

```
 1: $start \leftarrow 1$
 2: $returnList.add(listOfWayPoints[1])$
 3: $length \leftarrow listOfWayPoints.length()$
 4: for $end = 2$ to $length - 1$ do
 5:     if invisible($listOfWayPoints[start]$,
        $listOfWayPoints[end + 1]$) then
 6:         $returnList.add(listOfWayPoints[end])$
 7:         $start \leftarrow end$
 8:     end if
 9: end for
10: $returnList.add(listOfWayPoints[length])$
11: return $returnList$
```

$$\begin{aligned}
&\textbf{if } \text{invisible}(listOfWayPoints[start],\\
&\qquad listOfWayPoints[end+1])\\
&\quad \vee \text{cost}(listOfWayPoints[start],\\
&\qquad listOfWayPoints[end+1])\\
&\quad > \text{cost}(listOfWayPoints[start], listOfWayPoints[end])\\
&\quad + \text{cost}(listOfWayPoints[end],\\
&\qquad listOfWayPoints[end+1])\\
&\quad + \epsilon \textbf{ then}
\end{aligned}$$

4.1.1 Formation representation

The spatial structure of a formation should be represented precisely. In [16], Lewis and Tan introduced the concept of *virtual structure*, considering the agent group as a rigid body and all the planning is done for this body. Another popular method is to let each agent decide its own path according to its relative position with respect to other agents. In [9], Desai et al. presented a graph theoretical approach where each agent determines its location according to the locations of other agents and there is a leader agent who does not follow any other agent but leads the group. The relation between two agents consists of relative distance and orientation between them. There are several methods that make use of priorities among agents to describe a formation in a team [13, 19].

In our work, we defined a formation by specifying the relative positions of agents and their priorities. Agents are given ID's from 1 to n where n denotes the number of agents. The agent with the lowest ID (i.e., 1) has the highest priority and is called the *leader*. The priority is mainly used to determine the order of movement within each discrete time step in on-line path planning. An agent's relative position is given with respect to that one of the other agents having smaller ID. The relative position of agent a with respect to agent b is defined with two variables, $\Delta depth(a,b)$ and $\Delta width(a,b)$. $\Delta depth(a,b)$ is the distance between a and b in the movement direction of the group. $\Delta width(a,b)$ is the distance between a and b in the axis perpendicular to the forward facing direction of the group. In addition, the numbering of agents brings about another constraint that if agent a is closer to the front of the group in *depth* compared to agent b, a has lower ID than b's. Note that, one can design any formation using our representation and Fig. 10 shows how some commonly used formations are represented. In the figure, circles are the agents and numbers in the circles are ID's of the agents. A directed edge from agent a to agent b means that position of a is described with respect to position of b in the formation. a is called predecessor of b and b is called successor of a. The example formations in Fig. 10 are chosen to be the most common formations that are used in military domain. However, our formation representation scheme can be used with any formation defined by the user. For example, *wedge formation* given in the experiments contains branch. And circular formation can be defined by the user as a polygon, since the relations are given as distances in two dimensions, namely $\Delta depth(.,.)$ and $\Delta width(.,.)$.

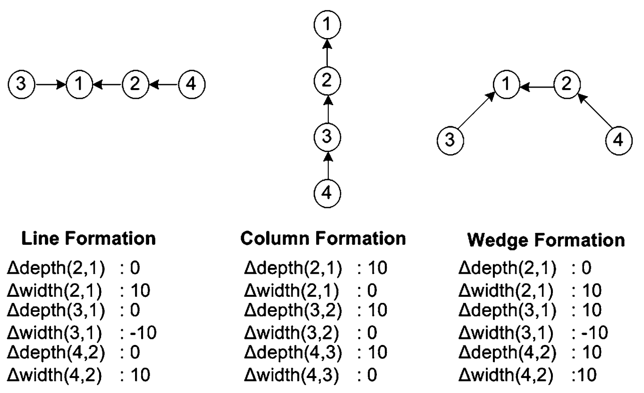


Fig. 10 Formation representation

Algorithm 6 PlaceAgents(formation, wayPoint, nextWayPoint)

```
1: lineM ← the line passing through wayPoint and nextWayPoint
2: lineP ← the line perpendicular to the lineM and passing at wayPoint
3: Find leftPoint and rightPoint as in Algorithm 4, with these arguments: CanFit (formation.width, edge(wayPoint, nextWayPoint))
4: backPoint ← wayPoint
5: for $i = 1$ to formation.depth do
6:    if inaccessible (wayPoint, backPoint) then
7:       break
8:    end if
9:    backPoint ← the point that is $i * sizeOfCell$ far from wayPoint on the line lineM and to the back of the wayPoint considering facing direction as front (see Fig. 11)
10: end for
11: midPointLR ← midpoint of leftPoint and rightPoint (see Fig. 12.a)
12: midPointBR ← front midpoint of bounding rectangle of formation (see Fig. 12b)
13: Put the bounding rectangle of formation in such a way that midPointLR and midPointBR are superposed (see Fig. 12c)
14: Scale bounding rectangle in left-right direction such that it can fit between leftPoint and rightPoint (see Fig. 13a)
15: Scale bounding rectangle in front-back direction such that it can fit between midPointBR and backPoint (see Fig. 13b)
16: for $i = 1$ to $n$ /*$n$ denotes the number of agents*/ do
17:    pointOfAgent ← the place of agent in the bounding rectangle found above (see Fig. 14)
18:    lineOfAgent ← line parallel to lineP and passing through pointAgent
19:    pointOfReference ← the point that is intersection of lineOfAgent and lineM
20:    if inaccessible(pointOfReference, pointOfAgent) then
21:       update the place of agent in the bounding rectangle as pointOfReference
22:    end if
23: end for
```

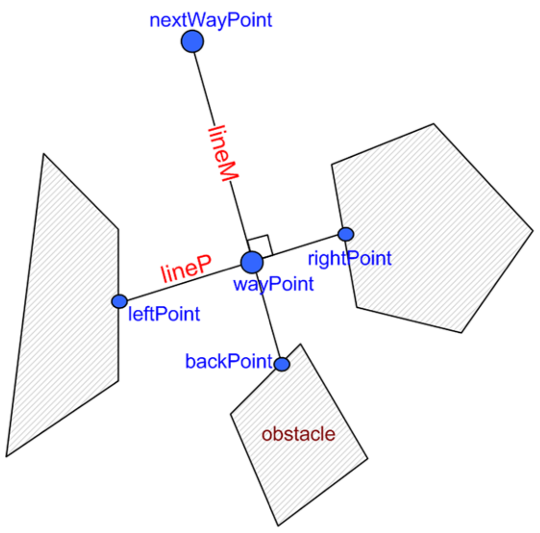


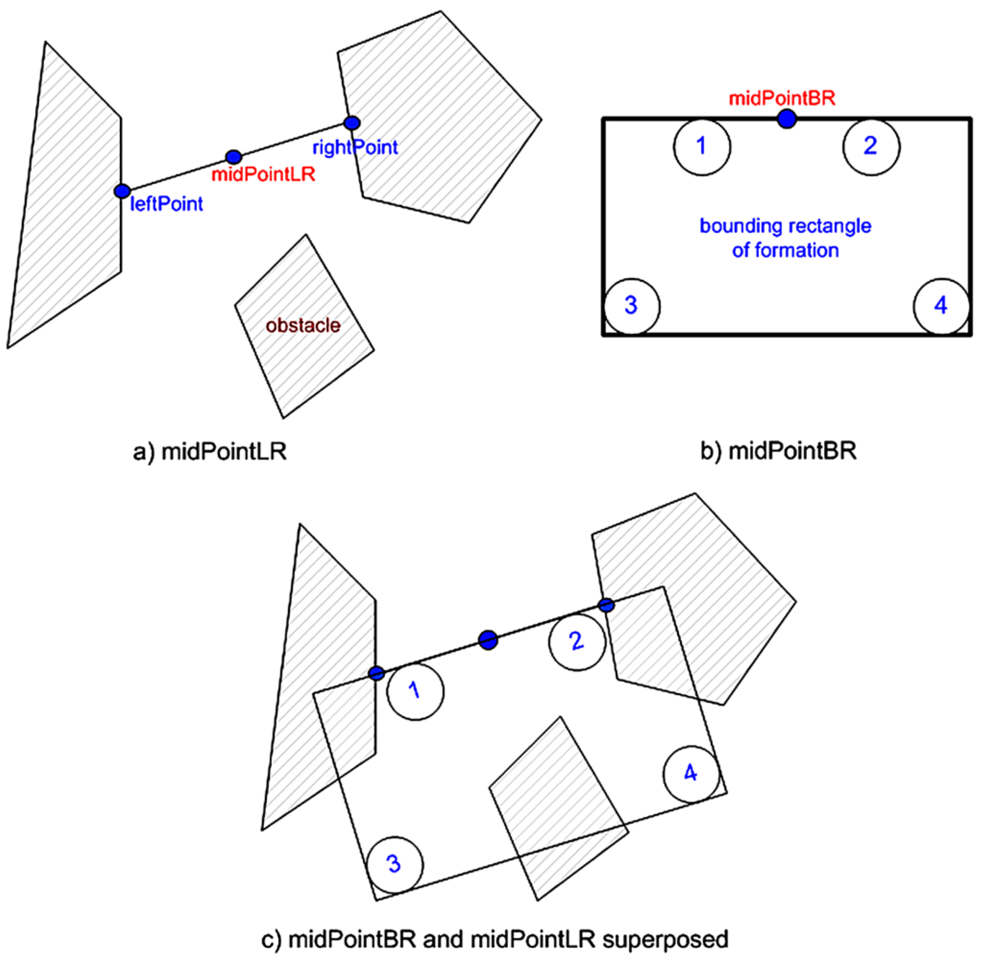
Fig. 11 Determination of *leftPoint*, *rightPoint* and *backPoint*

4.1.2 *Determining agent positions at every waypoint of the path*

At any waypoint of the path found by off-line planner, we use Algorithm 6 to determine each agent's position in accordance with the predefined formation. The method described in the algorithm is as follows. First, by using the same method as in Algorithm 4, for a waypoint, we determine the two points *leftPoint* and *rightPoint*, considering the edge between the waypoint and its successor. Then, determine the farthest accessible point to the waypoint, called *backPoint*, that is on the line passing through the waypoint and its successor and is to the back of the *wayPoint* considering facing direction of group as front. Figure 11 shows an example of *leftPoint*, *rightPoint* and *backPoint*.

Then, we determine the midpoint of *leftPoint* and *rightPoint* and superpose the front midpoint of *bounding rectangle* of the formation on this midpoint. *Bounding rectangle* of the formation can be defined as follows. Draw a line passing through the leader agent which is perpendicular to facing direction of the formation. Then, draw a parallel line passing through the agent that is farthest from the first line. Finally, draw two perpendicular lines to these lines passing

Fig. 12 Superposing *midPointLR* and *midPointBR*



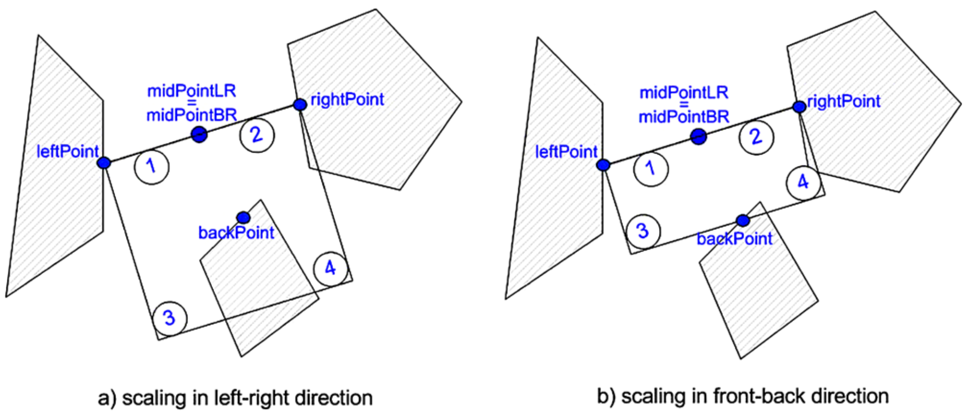

midPointLR
=
midPointBR
rightPoint
leftPoint
1
2
3
4
backPoint
a) scaling in left-right direction
b) scaling in front-back direction

Fig. 13 Scaling bounding rectangle to fit into an area

through the leftmost and the rightmost agents with respect to the moving direction. These 4 lines will form the bounding rectangle. This superposing is illustrated by an example in Fig. 12.

After that, the *bounding rectangle* is scaled in both left-right and front-back directions in order to fit to the specific area near the waypoint, if there are inaccessible points near it. This scaling processing is done to make sure that the bounding box is not occupied by the unreachable regions (e.g. obstacles). If the distance between *leftPoint* and *rightPoint* is not smaller than the width of the formation, then there is no need to scale in left-right direction. Likewise, if the distance from *backPoint* to the front of *bounding rectangle* is not smaller than depth of formation, then there is no need to scale in front-back direction, too. Figure 13 shows an example of scaling process.

Finally, for each of the agents' positions in the *bounding rectangle* formed so far, a line passing through the position and parallel to the line passing through *leftPoint* and *rightPoint* is drawn and the intersection of this line with the line passing through the waypoint and successor of the waypoint is determined. If this intersection point is inaccessible from the position, the position of agent is shifted to the intersection point. This is because this intersection point is necessarily accessible from the waypoint and it is important for each agent's position to be accessible from the waypoint, in order to move in coordination. An example of shifting process is given in Fig. 14.

We use bounding rectangle for approximating the shape of the group. Since the size of the rectangle is not fixed and scaled down if necessary, the team has the ability to pass through any waypoint. The agents will get closer when they are passing through a narrow passage, and will return back to the normal distances in other cases. Hence, we don't think that it will cause any problems to use a bounding rectangle instead of bounding polygon, for example. We think that bounding polygon, or similar approach, will be algorithmically much more complicated than ours and will not have a significant benefit compared to rectangle approach.

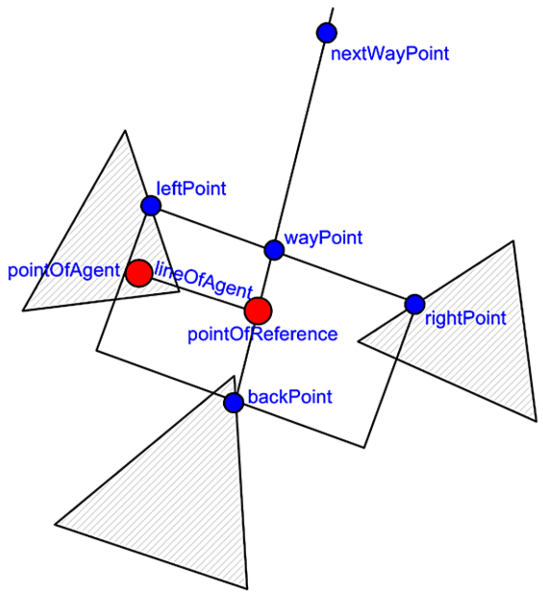


Fig. 14 Shifting inaccessible points

In addition, similar approach, i.e., bounding box, is used in many different areas, like physics engines in games etc., and shown to be effective.

Using this algorithm, agent positions at any waypoint are determined and off-line path finding step is completed. Next step is to move the agents in real time by using on-line path planning method given in next subsection.

4.2 On-line path finding

Having the positions calculated with the help of Algorithm 6 for each agent in the team at any waypoint, each agent navigates between its own waypoints in real time, avoiding the collisions. Agents plan their next moves and execute them at discrete time steps where the time between two successive time steps is very small. At any time step, agents plan and execute their moves in the order of their priorities (i.e., ID's). Note that the on-line planning algorithm is complete since the priority of the agents introduced in Sect. 4.1.1 *Formation representation* are designed using *follower* scheme, where the agents closer to the front of the formation are followed by the others that are on the back and higher priority agents move before the others. Similar distributed approaches are used in other similar works as well, e.g. [19].

We introduce an online path finder algorithm, low level planner, given in Algorithm 7, that navigates an agent from one waypoint to another which may take more than one time step. Note that, at the beginning of each time step, all the agents tune their speeds in order to maintain formation using Algorithm 8.

Basic idea behind the Algorithm 7 is to move the agent towards the next waypoint at each time step and if this can not be accomplished, get closer to the line from last visited waypoint to the next waypoint since it is guaranteed that the next waypoint is accessible from the last visited one. The reason why all agents do not move along this line is to move in formation.

In order to move more smoothly, the agents can change their direction a few steps before facing obstacles. This can be done by changing the definition of canMove(*dir*) in the first line of the algorithm as follows:

'... with its current speed in the direction ...' → ... with *t* times (where *t* determines the lookahead value, if it is huge the computational cost increases, if it is 1 it is same as main algorithm) its current speed in the direction ...

Agents tune their speeds at each time step in order to preserve the formation while moving by using Algorithm 8. According to the distance between an agent and its predecessor, it tunes its speed and tells the predecessor to do so when the distance exceeds a threshold.

If an agent gets more than one contradictory request, the decelerate request has higher priority over accelerate request. However, an accelerate request has higher priority over a normalize (tuning to average speed) request.

4.2.1 Rearrange formation

In the case of mobility loss of an agent, the remaining agents should rearrange their role according to Algorithm 9. The team re-organizes by making each agent get the role (relative position and ID) of its predecessor. This is only a state change for the team, the formation will be restored in time as the team moves. Figure 15 shows an example, where the team is moving in column formation and the agent with ID 2 has lost its mobility.

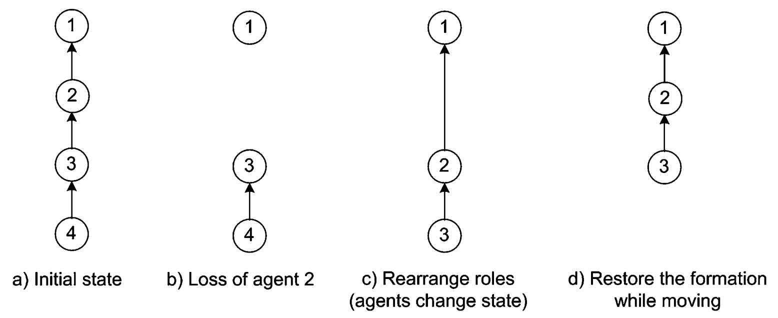
5 Experimental results and sample run

The experimental results of the proposed algorithms are given in this section. First, the testing platform and experimental setup are given. Then, the performance of off-line

Algorithm 7 OnlinePathFinder(lastVisitedWayPoint, nextWayPoint)

```
1: /* Definitions:
   canMove(dir): agent checks whether it is possible to
   move without colliding with any other agent/obstacle
   with its current speed in the direction dir, and if so,
   moves in that direction and returns true; otherwise, does
   nothing and returns false.
   direction(dir, LEFT/RIGHT, angle): returns the direc-
   tion that is angle degrees on the LEFT/RIGHT of dir.
   e.g. direction(dir, LEFT, 90) returns the direction that is
   perpendicular to and on the left of dir. */
2: while distance(agent.location, nextWayPoint) > $\epsilon$ do
3:   targetDir ← vector(agent.location, nextWayPoint)
4:   if canMove(targetDir) then
5:     continue
6:   else if canMove(direction(targetDir, LEFT, 45)) then
7:     continue
8:   else if canMove(direction(targetDir, RIGHT, 45)) then
9:     continue
10:  else if canMove(direction(targetDir, LEFT, 90)) then
11:    continue
12:  else if canMove(direction(targetDir, RIGHT, 90)) then
13:    continue
14:  end if
15:  secondPhase:
16:  lastTargetDir ← vector(lastVisitedWayPoint,
     nextWayPoint)
17:  if agent is on the right of lastTargetDir then
18:    side ← RIGHT
19:  else
20:    side ← LEFT
21:  end if
22:  if canMove(targetDir) then
23:    continue
24:  else if canMove(direction(targetDir, side, 45)) then
25:    continue
26:  else if canMove(direction(targetDir, side, 90)) then
27:    goto secondPhase
28:  else if canMove(direction(targetDir, side, 135)) then
29:    goto secondPhase
30:  else
31:    Wait for a predefined amount of time (because the
       path may have been unavailable for an amount of
       time), and then goto secondPhase
32:  end if
33: end while
```

Fig. 15 Rearranging formation


a) Initial state
b) Loss of agent 2
c) Rearrange roles (agents change state)
d) Restore the formation while moving

Algorithm 8 TuneSpeeds(positionsOfAgents, directionOfAgents)

```
1: /* $p$ is a predefined threshold (a real number between 0 and 1) that specifies to what degree the relative position of any agent wrt its predecessor can be altered. */
2: for $i = 2$ to $numberOfAgents$ do
3:   $dist \leftarrow$ distance in $depth$ between the agent $i$ and agent $predecessor(i)$ according to the $direction(predecessor(i))$
4:   if $dist < \Delta depth(i, predecessor(i)) * (1 - p)$ then
5:     $decelerate(i)$
6:   else if $dist > \Delta depth(i, predecessor(i)) * (1 + 2 * p)$ then
7:     $decelerate(predecessor(i))$
8:   else if $dist > \Delta depth(i, predecessor(i)) * (1 + p)$ then
9:     $decelerate(i)$
10:  else
11:    $normalizespeed(i)$
12:    $normalizespeed(predecessor(i))$
13:  end if
14: end for
```

Algorithm 9 RearrangeTeam(agent, remainingFormation)

```
1: $current \leftarrow agent$
2: while $current$ has successor in $remainingFormation$ do
3:   $child \leftarrow$ the lowest numbered successor of the $current$
4:   add $child$ to list $L$
5:   $current \leftarrow child$
6: end while
7: $current \leftarrow agent$
8: for $i = 1$ to $lengthOfList(L)$ do
9:   replace $current$ with $L[i]$
10:  $current \leftarrow L[i]$
11: end for
```

planner is discussed. Our method was developed for primarily simulation of military operations. Therefore, the path to be generated must guarantee that the team is able to move in formation to the desired destination with minimum cost. Size of the search graph is critical for the running time of off-line algorithm, which guarantees to find optimal paths. Success of our method is due to the nice reduction on the size of the search graph as can be seen in the experimental study.

Then, experimental results of on-line planner are given. On-line planner is quite fast as it requires constant time per iteration. The on-line planner tolerates deviation from the desired formation to some extend, controlled by a user-specified threshold p as specified in Algorithm 8. Experimental results justify that formation maintenance is quite successful measured in terms of pairwise isolation to be explained next.

To sum up, our experimental study justify that our method is efficient and optimizes multiple criteria, *length of the solution path* and *degree of formation maintenance*. Finally, some sample screenshots from software are given.

5.1 Experimental setup

All the given algorithms are implemented using C++ programming language. To visualize the 3-D environment, the OGRE (Object-Oriented Graphics Rendering Engine) API [20] is used. The code was written platform independently and the software was both tested in Linux and Windows platforms. Tests were run on a PC, which has Intel Core2 1.80 GHz CPU and 1 GB memory.

We randomly generated 15 terrain data and one real world terrain data to test the algorithms. Randomly generated data are of sizes 2500×2500, 2000×2000, 1500×1500, 1000×1000 and 500×500, and we generated three instances from each size. The real world data is 2000×2000. Algorithm 10 is used to generate random *heightmaps*,[2] which is converted to 3-D mesh to represent terrain. The algorithm first creates a heightmap whose pixels are initialized to 0. Then, it randomly creates *oblate semi-spheroids*[3] at randomly selected points and having random but bounded major axis and minor axis lengths. Finally, it adds up the elevations of all these spheroids at each pixel.

[2]*Heightmap* is an image used for storing terrain elevation data.

[3]*Oblate spheroid* is a special type of ellipsoid, formed by rotating an ellipse around its minor axis.

Algorithm 10 GenerateHeightMap(terrainSize, numberOfSpheroids, maxMajorAxis, maxMinorAxis): HeightMap

```
1: Create a height map $H$ with size $terrainSize \times terrainSize$ and initialize all pixels as 0
2: for $i = 1$ to $numberOfSpheroids$ do
3:     $x \leftarrow$ random($terrainSize$) // Assume that $random(i)$ generates an integer between 1 and $i$
4:     $y \leftarrow$ random($terrainSize$)
5:     $r \leftarrow$ random($maxMajorAxis$)
6:     $h \leftarrow$ random(min($maxMinorAxis, r$))
7:     Create an oblate semi-spheroid which is centered at $(x, y)$, with major axis of length $r$ and minor axis of length $h$ and add the elevation of this spheroid at each pixel to the corresponding pixels of $H$
8: end for
9: return $H$
```

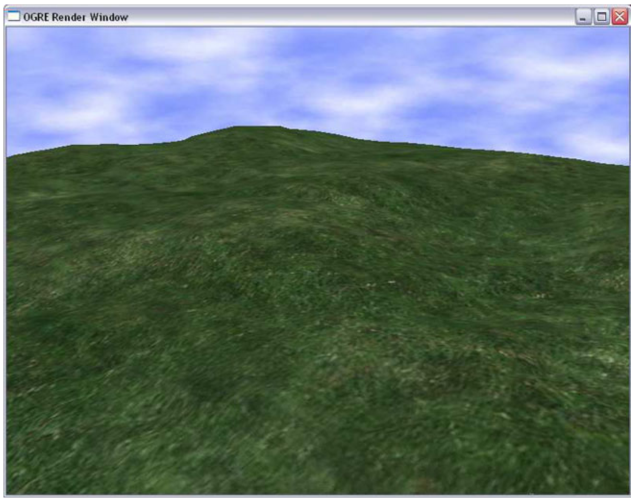


Fig. 16 Sample smooth terrain

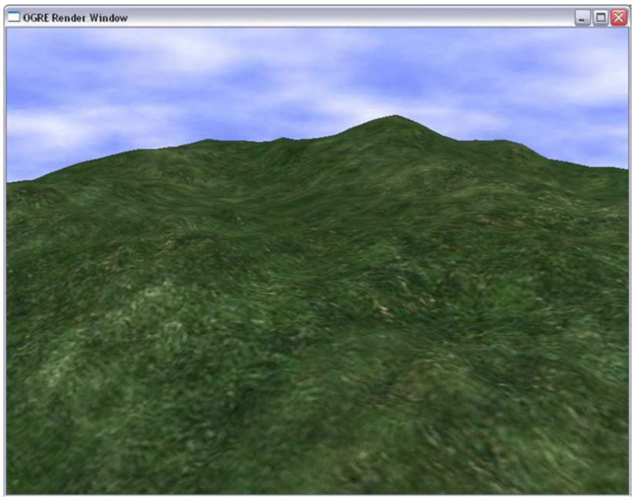


Fig. 17 Sample rough terrain

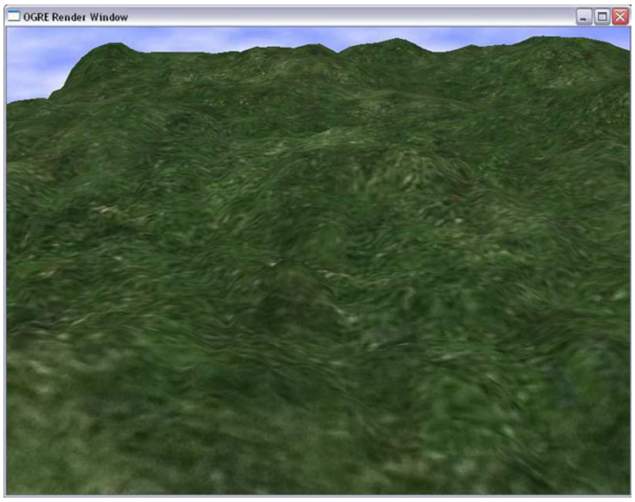


Fig. 18 Sample mountainous terrain

By increasing the number of spheroids or *minor axis/major axis* ratio, more rough terrains can be generated. One of the three randomly created terrains of the same size is relatively smooth, one is relatively rough and the other is mountainous. Tested terrains of size 1000×1000 are given in Figs. 16, 17 and 18.

5.2 Performance evaluation of off-line planner

In this part, experimental results of the off-line planner will be discussed. Results are given in Tables 1, 2, 3, 4, 5 and 6. For each terrain, 4 methods are tested. These are grid method, our proposed method with *height difference threshold* 2 (HDT = 2), 5 (HDT = 5) and not using *height difference threshold* (i.e., HDT = ∞ or No HDT). Details of the *height difference threshold* is given in Sect. 3. For each method, the number of points (i.e, vertices) in the constructed search graph, time consumed for graph construction in seconds, time used for finding a path on constructed graph (time for A^* + time for path smoothing), cost of the path found after running A^* and cost of the path after smoothing are given. Note that the cost of the path is taken as the length of the path (sum of the edge weights of the path). The calculation of the edge weights is described in Sect. 3.2. In each test data, team's mission is to find a path from one corner of terrain to the opposing corner.

From these results, first, we can conclude that our method is very effective in time, especially the method named *NoHDT*. As can be seen in all terrains, the method decreases the number of points steeply and as a result, the search can finish in a very short time period compared to grid method. Even if the problem is to find a path once in a given terrain, the method is very effective since *time constructing graph* + *time running* A^* is small compared to grid method. But in real life applications, e.g. games and military simulations,

Table 1 Experiment on 2500 × 2500 terrains

Height difference threshold	Number of points (vertices)	Time constructing graph (sec)	Time running A^* (sec)	Cost of path found by A^*	Cost of smoothed path
First terrain					
HDT = 2	410837	29.85	8.96	4950	3859
HDT = 5	158312	29.05	6.44	4964	3831
No HDT	3652	10.76	0.24	5431	4188
Grid	6250000	20.98	18.52	5099	3853
Second terrain					
HDT = 2	857802	37.77	11.94	5228	4182
HDT = 5	358043	26.46	6.76	5248	4196
No HDT	38609	22.13	0.85	5453	4376
Grid	6250000	19.85	9.75	5476	4503
Third terrain					
HDT = 2	887673	35.58	18.13	5279	4287
HDT = 5	403464	36.21	11.96	5317	4687
No HDT	120731	24.66	1.35	5516	4913
Grid	6250000	19.28	16.41	5481	4280

Table 2 Experiment on 2000 × 2000 terrains

Height difference threshold	Number of points (vertices)	Time constructing graph (sec)	Time running A^* (sec)	Cost of path found by A^*	Cost of smoothed path
First terrain					
HDT = 2	308047	19.63	5.68	3902	2937
HDT = 5	119164	19.4	4.87	3906	2907
No HDT	2837	7.24	0.2	5783	2907
Grid	4000000	16.32	16.5	4035	3309
Second terrain					
HDT = 2	608702	29.4	8.36	4091	3353
HDT = 5	255044	19.93	4.73	4085	3259
No HDT	30768	19.27	0.95	4172	3500
Grid	4000000	16.23	8.19	4290	3646
Third terrain					
HDT = 2	667764	19.05	16.58	4236	3636
HDT = 5	306255	19.89	8.54	4259	3607
No HDT	99102	17.15	1.04	4343	4146
Grid	4000000	16.64	16.24	4367	3564

Table 3 Experiment on 1500 × 1500 terrains

Height difference threshold	Number of points (vertices)	Time constructing graph (sec)	Time running A^* (sec)	Cost of path found by A^*	Cost of smoothed path
First terrain					
HDT = 2	145285	12.74	2.54	2822	2083
HDT = 5	51417	12.75	0.92	2829	2094
No HDT	2441	5.40	0.09	5141	2098
Grid	2250000	7.08	6.94	2918	2343
Second terrain					
HDT = 2	269544	12.77	3.66	2964	2379
HDT = 5	109337	12.94	1.67	2978	2436
No HDT	19035	9.97	0.21	3227	2327
Grid	2250000	7.20	6.33	3060	2336
Third terrain					
HDT = 2	361108	12.55	5.79	3043	2480
HDT = 5	172921	12.79	2.46	3047	2622
No HDT	73490	11.93	0.61	3163	2714
Grid	2250000	9.6	15.46	3154	2460

Table 4 Experiment on 1000 × 1000 terrains

Height difference threshold	Number of points (vertices)	Time constructing graph (sec)	Time running A^* (sec)	Cost of path found by A^*	Cost of smoothed path
First terrain					
HDT = 2	77661	6.89	2.78	1822.38	1336.71
HDT = 5	27774	6.73	1.09	1823.71	1336.71
No HDT	1192	1.66	0.03	1878.03	1336.71
Grid	1000000	3.75	3.17	1864.63	1476
Second terrain					
HDT = 2	161407	6.98	4.17	1901.5	1513.29
HDT = 5	66539	4.74	1.3	1914.43	1504.8
No HDT	7119	3.02	0.07	2157.72	1854.16
Grid	1000000	4.19	3.58	1965.35	1567.92
Third terrain					
HDT = 2	172522	4.59	3.43	1987	1590.36
HDT = 5	78881	6.92	2.08	2010.55	1600.45
No HDT	31574	4.24	0.29	2087.38	2005.66
Grid	1000000	3.82	18.6	2054.35	1642

the search graph is only constructed at the beginning of the simulation once, which can be used for many navigation tasks throughout the simulation. So, efficiency of the *time running A^** is the key criteria in such applications. We can gain up to 100 times better results in this criteria if we use the method *NoHDT*.

Second criteria for the search is the cost of path. We can conclude from the results that, the lengths of the paths found

Table 5 Experiment on 500 × 500 terrains

Height difference threshold	Number of points (vertices)	Time constructing graph (sec)	Time running A^* (sec)	Cost of path found by A^*	Cost of smoothed path
First terrain					
HDT = 2	16348	1.11	0.22	741.554	540.507
HDT = 5	5378	1.05	0.07	746.535	540.507
No HDT	373	0.35	0.02	1093.96	540.507
Grid	250000	0.69	1.24	760.892	540.507
Second terrain					
HDT = 2	43812	1.12	0.52	787.862	632.598
HDT = 5	18053	1.57	0.32	782.584	608.401
No HDT	2621	0.82	0.02	852.89	703.412
Grid	250000	0.94	1.11	825.305	632.04
Third terrain					
HDT = 2	44818	1.11	0.54	796.319	630.541
HDT = 5	20370	1.1	0.25	812.369	718.666
No HDT	9804	0.98	0.06	903.353	840.836
Grid	250000	0.7	0.86	842.652	625.592

Table 6 Experiment on real world terrain

Height difference threshold	Number of points (vertices)	Time constructing graph (sec)	Time running A^* (sec)	Cost of path found by A^*	Cost of smoothed path
HDT = 2	199765	19.5	8.3	3942.27	3125.81
HDT = 5	95703	19.65	2.73	3953.93	3145.27
No HDT	62933	19.52	1.08	3977.06	3188.32
Grid	4000000	16.3	7.86	4075.39	3147.34

using our proposed methods are approximately equal to and generally better than the one found using grid method. Note that in Table 5, the cost of smoothed path for the first terrain seems to be all the same because that terrain is small and the smoothest one and all the methods end up with the same result after smoothing. In Table 1–5, the cost of the path found by A^*, which is expected to be minimal, is not the lowest among other methods. The reason for this is the fact that heuristic estimation used by A^* is not admissible. As explained in Sect. 3.2. *Representation of the environment and construction of the search graph*, cost of an edge is dependend on whether the desired formation can be maintained between two nodes incident with that edge, or not, in addition to parameters such as distance, slope, etc. Because of this parameter, the cost function is not linear in terms of the distance between two nodes, which might result in lower costs for our method than the grid based method. Finally, we would like to state that our algorithm is complete. Furthermore, there is not any case where the grid-base method can find a solution but ours cannot, because we reflect all of the accessibilities and inaccessibilities of the grid method to our search graph.

Table 7 Results of on-line planner of 4 agents

Terrain	Formation	Average error in depth				Average error in width			
		A1	A2	A3	A4	A1	A2	A3	A4
1	Column	–	1.78	3.04	3.04	–	0.25	0.21	0.21
	Line	–	1.79	1.76	1.70	–	1.10	1.01	1.06
	Wedge	–	2.46	1.10	2.71	–	1.21	0.95	1.37
2	Column	–	3.99	3.99	3.93	–	0.28	0.33	0.23
	Line	–	3.21	2.96	2.94	–	1.02	1.22	1.55
	Wedge	–	2.59	1.72	3.04	–	1.33	1.09	1.46
3	Column	–	3.54	3.26	3.68	–	0.32	0.28	0.45
	Line	–	3.26	3.12	3.29	–	2.30	1.32	2.41
	Wedge	–	3.58	3.08	3.63	–	2.09	1.39	2.59
Real	Column	–	2.52	2.60	2.65	–	0.18	0.20	0.22
	Line	–	1.44	1.45	1.41	–	1.11	1.22	1.39
	Wedge	–	2.54	2.48	2.60	–	1.21	1.28	1.38

Table 8 Results of on-line planner of 8 agents

Terrain	Formation	Average error in depth	Average error in width
1	Column	2.16	0.34
	Line	1.62	1.54
	Wedge	1.72	1.51
2	Column	3.12	0.41
	Line	3.10	1.82
	Wedge	2.52	1.75
3	Column	3.31	0.37
	Line	3.14	2.20
	Wedge	3.50	2.01
Real	Column	1.42	0.23
	Line	1.45	1.63
	Wedge	1.48	1.74

5.3 Performance evaluation of on-line planner

Experimental results of on-line planner will be discussed in this part. These results are obtained from the on-line path planner during the team is moving on the path found by the off-line planner using our method (No HDT) on 2000 × 2000 terrains and the real world terrain used above. Results can be seen in Tables 7, 8 and 9. On each of the four terrains, three

Table 9 Results of on-line planner of 16 agents

Terrain	Formation	Average error in depth	Average error in width
1	Column	1.98	0.45
	Line	1.86	2.23
	Wedge	1.73	2.15
2	Column	2.94	0.43
	Line	3.21	3.26
	Wedge	2.71	3.21
3	Column	3.79	0.62
	Line	3.74	3.51
	Wedge	3.37	3.48
Real	Column	1.88	0.45
	Line	1.75	2.41
	Wedge	1.80	2.11

common formations (column, line, wedge) are tested with teams of four, eight and sixteen agents. Formations are defined as in Fig. 10 in Sect. 4 for four agents. Formations of eight and sixteen agents are defined similarly. As previously mentioned, the formation is specified by each agent's relative position with respect to its predecessor. Remember that the relative position of a with respect to its predecessor b is defined with two variables $\Delta depth(a,b)$ and $\Delta width(a,b)$. $\Delta depth(a,b)$ is the distance between a and b in the movement direction of the group and $\Delta width(a,b)$ is the distance between a and b in the axis perpendicular to the movement direction of the group. In the on-line planner, fluctuation of these $\Delta depth$ and $\Delta width$ values up to 20% are considered in range and up to 40% are tolerable. These values are selected due to our domain analysis in the development of an agent-based military simulation framework, consulting to military domain experts. The results in Table 7 show the average absolute values of distance errors in depth and in width of each agent. In Tables 8 and 9, the average $\Delta depth(a,b)$ and $\Delta width(a,b)$ of all agents are given. Note that the relative distances in formations are set to 10 units, so values up to 2 are considered in range and up to 4 are considered tolerable.

Results show that, agents are successful in preserving their relative position with respect to their predecessor (especially in width), since the errors in both are in tolerable range. Because of the terrain features (e.g. roughness), it becomes hard, sometimes impossible, to preserve the distances and that's why the errors increase in 2nd and 3rd terrains.

5.4 Sample runs

Screenshots from sample runs of three common formations of 4 agents, line, column and wedge, are given in Figs. 19, 20 and 21, respectively. In figures, the red lines show the paths travelled by agents. Each agent forms a red line by putting small red spheres to its instant position at every time step during simulation.

Screenshots for teams of 8 agents in column formation and 16 agents in line formation are given in Figs. 22 and 23, respectively.

Also, a sample run in which the team in line formation passing through a passage is given. In Fig. 24, four screenshots from the sample run is given. The paths realized by each agent during the run can be seen in Fig. 25.

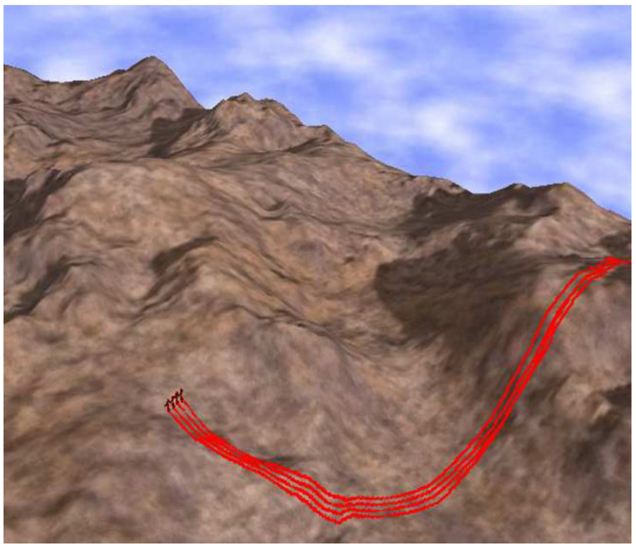


Fig. 19 Team in line formation

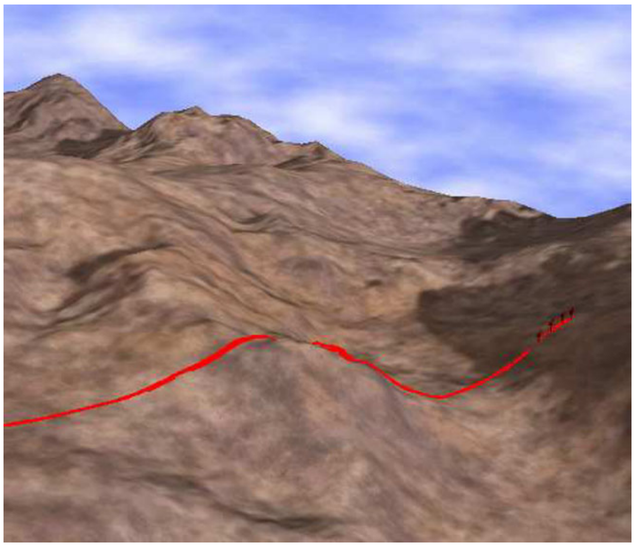


Fig. 20 Team in column formation

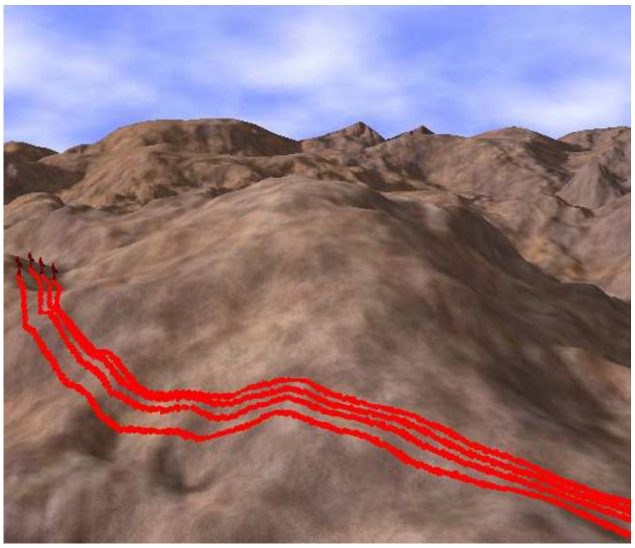


Fig. 21 Team in wedge formation

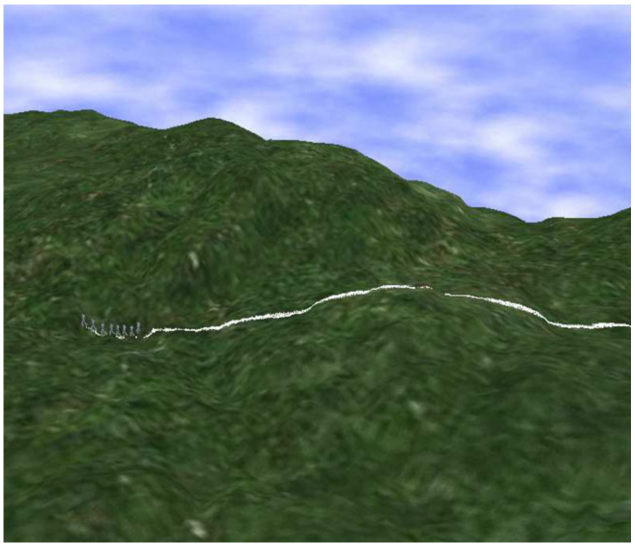


Fig. 22 Team of 8 agents in column formation

6 Conclusions and future work

In this work, we developed an algorithm for planning a path for a group of autonomous agents that need to move in a specified formation in 3-D terrains. We developed software in order to test the proposed methods and visualize the environment and behavior of agents in 3-D. Our method can be used especially in computer games and military simulations.

The proposed method, considering the group as a rigid body, first constructs a search graph by analyzing and identifying important terrain features from height map. We use grid representation and the number of moving directions is selected to be 4. The number of moving directions can be at most 8 because of grid representation of the terrain, i.e., there are 8 neighbors of any grid cell. The number of moving directions can be increased if the grid cell is made an *n-gonal* ($n \geq 8$) instead of squares. But increasing the number of neighborhood of cell, the size of the search graph will increase, making both the terrain analysis and the path search costly.

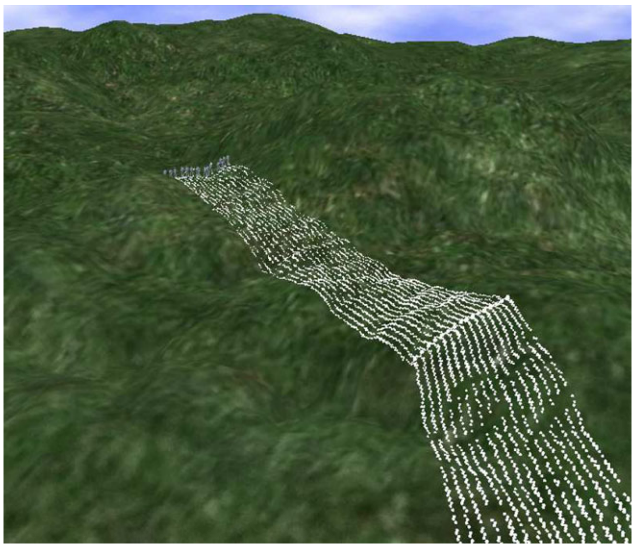


Fig. 23 Team of 16 agents in line formation

Then a high level planner that uses A^* algorithm determines an optimal path from an initial location to a target location. Then with the help of a low level on-line planner, each agent in the team navigates between waypoints on the solution path, avoiding collisions with each other and with objects in the environment. We defined a representation for group formation on which algorithms were developed to maintain formation while moving from one waypoint to another. The proposed method also has the ability to repair the formation when an agent or some agents in the team loses mobility, which is quite possible in real-life applications. Concerning the on-line planning, the possibility of agents getting stuck is not possible. Off-line planner guarantees to find a sequence of waypoints for the overall team from which individual waypoints are generated. The only thing left is to maintain formation between two successive waypoints. For this we have defined priorities by totally ordering the agents, and any agent will only be trying to keep its relative position with respect to another agent (the next high priority agent). The highest priority agent will surely be able to reach its designated next waypoint, at the same time the next highest priority agent following the first one, and so on.

We tested our proposed algorithms both on randomly generated and real-world terrains. The off-line planner has brought a significant gain in time performance, which might be the most crucial factor in real-life applications especially in games, over grid based terrain representation. Results obtained from the on-line planner were also very satisfactory.

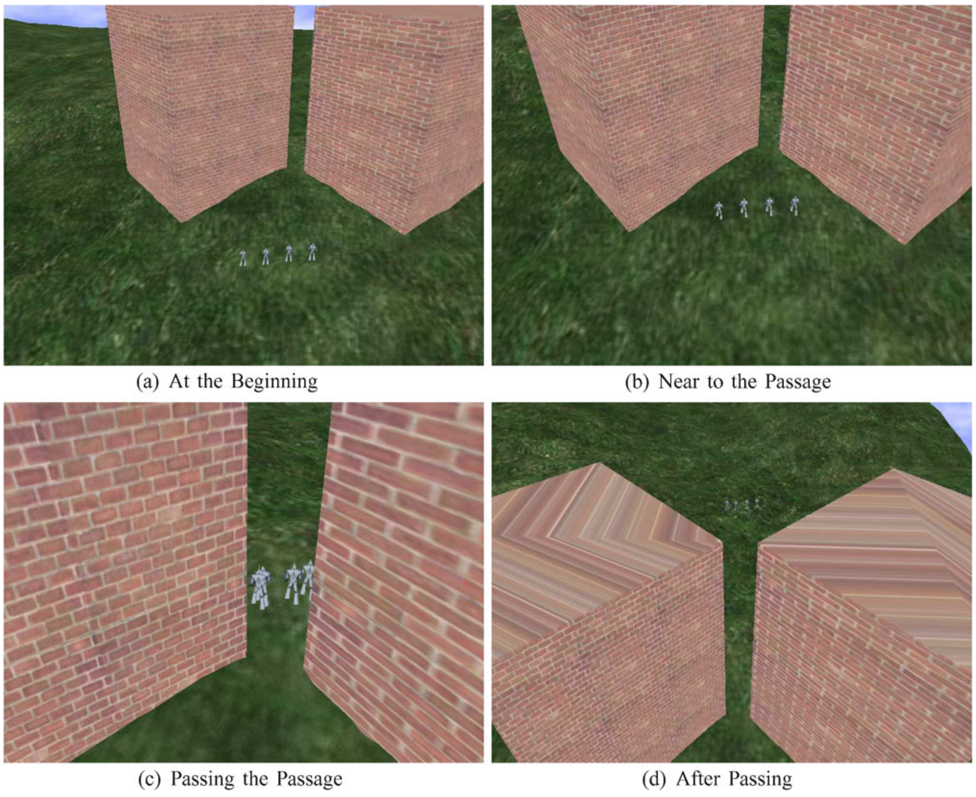


Fig. 24 Team passing a passage—screenshots

The team maintained the formation through most of the path, and when it is necessary to break the formation, e.g. a team in line formation passing through a narrow passage, the team recovered the formation in a very short time period.

The algorithms were implemented and tested on static environments but they are easily adaptable to dynamic environments. Reconstructing only the changed part of the search graph and using D^* [28] like algorithms would be a solution. D^* is a A^* like search algorithm especially used in dynamic environments. Also, it is considered that the environment is known at the beginning of the simulation. In the same way, by considering the observability of a region as dynamism of the environment, this problem can also be handled. As future work, methods will be adapted to dynamic and partially observable environments.

Additional research will be performed on the formation preserving navigation of hierarchical agent teams. For example, each of the fireteams (Fireteam is the smallest unit in military) will move in wedge formation, while the squad (Squad is one level higher unit of fireteam) will move in column formation. By making some changes in formation representation, our method may also be used for this task.

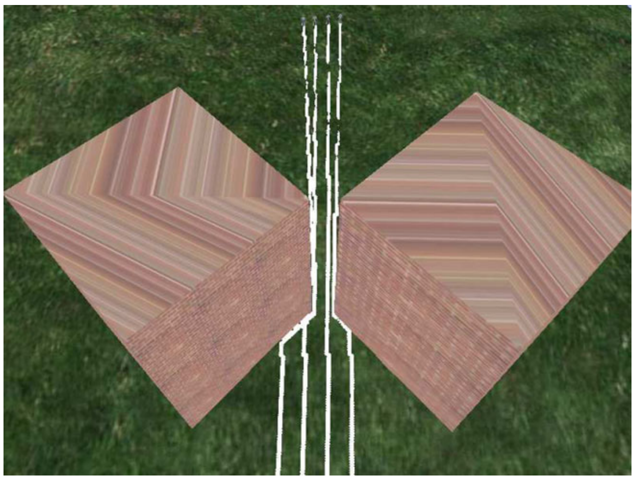


Fig. 25 Team passing a passage—paths

References

1. Bahceci E, Soysal O, Sahin E (2003) A review: pattern formation and adaptation in multi-robot systems. Technical report, Robotics Institute, Carnegie Mellon University
2. Balch T, Arkin RC (1998) Behavior-based formation control for multirobot teams. IEEE Trans Robot Autom 14(6):926–939
3. Bayrak AG, Polat F (2008) Formation preserving navigation of agent teams in 3-d terrains. In: Proceedings of industrial simulation conference (ISC), pp 148–155
4. Berg MD, Krefeld MV, Overmars M, Schwarzkopf O (2000) Computational geometry: algorithms and applications. Springer, Berlin
5. Bourgeot J-M, Cislo N, Espiau B (2002) Path-planning and tracking in a 3d comlex environment for an anthropomorphic biped robot. In: Proceedings of the IEEE/RSJ international conference on intelligent robots and systems IROS, Lausanne, Switzerland, pp 2509–2514

6. Bresenham JE (1965) Algorithm for computer control of a digital plotter. IBM Syst J 4(1):25–30
7. Dain RA (1998) Robot wall-following algorithms using genetic programming. Appl Intell 8(1):33–41
8. Dawson C (2002) Formations. AI Game Programming Wisdom. AI Wisdom, pp 272–281. Charles River Media, Boston
9. Desai JP, Kumar V, Ostrowski JP (1999) Control of changes in formation for a team of mobile robots. In: Proceedings of IEEE international conference on robotics and automation, vol 2, pp 1556–1561
10. Fugere J, LaBoissonniere F, Liang Y (1999) An approach to design autonomous agents within modsaf. In: Proceedings of IEEE SMC'99 conference on systems, man, and cybernetics, Tokyo, Japan, vol 2, pp 534–539
11. Geraerts R, Overmars MH (2002) A comparative study of probabilistic roadmap planners. In: Workshop on the algorithmic foundations of robotics (WAFR'02). Springer, Berlin, pp 43–57
12. Helbing D, Buzna L, Johansson A, Werner T (2005) Self-organized pedestrian crowd dynamics: experiments, simulations, and design solutions. Transp Sci 39(1):1–24
13. Hsu C.-H., Liu A (2005) Multiagent-based multi-team formation control for mobile robots. J Intell Robot Syst 42(4):337–360
14. Kambhampati S, Davis L (1986) Multiresolution path planning for mobile robots. IEEE J Robot Autom 2(3):135–145
15. Kavraki L, Latombe JC (1998) Probabilistic roadmaps for robot path planning. In: Practical motion planning in robotics: current and future directions. Addison-Wesley, Reading, pp 33–53
16. Lewis MA, Tan KH (1997) High precision formation control of mobile robots using virtual structures. Auton Robots 4(4):387–403
17. Loscos C, Marchal D, Meyer A (2003) Intuitive crowd behavior in dense urban environments using local law. In: Proceedings theory practice of computer graphics (TPCG 03), pp 122–129
18. McIlroy D, Smith B, Heinze C, Turner M (1997) Air defence operational analysis using the SWARMM model. In: Asia pacific operations research symposium
19. Ngo VT, Nguyen AD, Ha QP (2005) Toward a generic architecture for robotic formations: planning and control. In: Proceedings of the sixth international conference on intelligent technologies, pp 89–96
20. OGRE—Open Source 3D Graphics Engine. http://www.ogre3d.org
21. Parsons D, Surdu J, Jordan B (2005) Onesaf: a next generation simulation modeling the contemporary operating environment. In: Proceedings of Euro-simulation interoperability workshop
22. Pradhan SK, Parhi DR, Panda AK, Behera RK (2006) Potential field method to navigate several mobile robots. Appl Intell 25(3):321–333
23. Reece DA (2003) Movement behavior for soldier agents on a virtual battlefield. Presence 12(4):387–410
24. Reynolds CW (1987) Flocks, herds, and schools: a distributed behavioral model. Comput Graph 21(4):25–34
25. Russell S, Norvig P (2003) Artificial intelligence a modern approach. Prentice Hall, New York
26. Sahli N, Moulin B (2009) Ekemas, an agent-based geo-simulation framework to support continual planning in the real-word. Appl Intell 31(2):188–209
27. Sanchez G, Ramos F, Frausto J (1999) Locally-optimal path planning by using probabilistic roadmaps and simulated annealing. In: Proceedings IASTED robotics and applications international conference
28. Stentz A (1994) Optimal and efficient path planning for partially-known environments. In: Proceedings of the IEEE international conference on robotics and automation, vol 4, pp 3310–3317
29. Tomlinson SL (2004) The long and short of steering in computer games. Int J Simul 1–2(5):33–46
30. Treuille A, Cooper S, Popović Z (2006) Continuum crowds. ACM Trans Graph 25(3):1160–1168
31. Undeger C, Polat F (2007) Rttes: real-time search in dynamic environments. Appl Intell 27(2):113–129
32. Vaughan J, Connell R, Lucas A, Ronnquist R (2003) Towards complex team behavior in multi-agent systems using a commercial agent platform. In: Lecture notes in computer science, vol 2564. Springer, Berlin, pp 175–185
33. Verth JV, Brueggemann V, Owen J, McMurry P (2000) Formation-based pathfinding with real-world vehicles. In: Proceedings of the game developers conference
34. Voorbrk F, Massion N (2001) Decision-theoretic planning for autonomous robotic surveillance. Appl Intell 14(3):252–262

Ali Galip Bayrak received his BS and MSc degrees from Middle East Technical University, Computer Engineering department in 2005 and 2008. He worked on automated agents and multi-agent systems during his MSc studies. He is currently a PhD student in EDIC (Doctoral program in computer and communication sciences) in EPFL. He is working in Processor Architecture Laboratory on hardware cryptography, particularly side-channel attacks.

Faruk Polat is a professor in the Department of Computer Engineering of Middle East Technical University, Ankara, Turkey. He received his BSc in computer engineering from the Middle East Technical University, Ankara, in 1987 and his MS and PhD degrees in computer engineering from Bilkent University, Ankara, in 1989 and 1993, respectively. He conducted research as a visiting NATO science scholar at Computer Science Department of University of Minnesota, Minneapolis in 1992–1993. His research interests include artificial intelligence, multi-agent systems and object oriented data models.