Applying pathfinding techniques on the development of an Android game

Paulo Vitor Freitas da Silva* Saulo Moraes Villela

Universidade Federal de Juiz de Fora, Departamento de Ciência da Computação, Brasil

Abstract

Mobile devices are present, more and more, in our daily life. As the most of these devices have low memory and low processing power, the developers need to be smart and creative to develop applications that can be performed quickly and efficiently. This work aims to face these limitations and apply artificial intelligence techniques, which usually require long time processing and large memory usage, on the development of an Android game. We chose the Android platform because it is the most popular nowadays for mobile devices. We developed a maze game, where the player must follow a path to reach a goal, escaping from enemies. The enemies use pathfinding techniques, trying to reach the player. Each enemy has been implemented with a different pathfinding technique, and the behaviors and performances were evaluated and compared. We implemented the uninformed (blind) search strategies breadth-first search, depth-first search and uniform-cost (ordered) search, and the informed (heuristic) search strategies greedy search and A* search. We observed, as expected, that techniques using information of the problem domain are more promising, because they can find good paths and also run in a satisfactory time. Thus, we showed that it is possible to create an Android game, applying these methods, ensuring its fluidity and the user's enjoyment.

Keywords: Android, Artificial Intelligence, Pathfinding, Mobile, Games

1 Introduction

It is well known the intense growth of the mobile market. With it, arises in the same proportion the demand for development of technologies and software that enhance the functionality of these devices. And one of these demands is, especially, the game development. The limitation of processing power and memory capacity of this type of device makes developers for mobile devices have to exercise their creativity and knowledge to create games that are able to run with these restrictions and bring entertainment to the user at the same time.

This paper describes the development of a game for Android devices where we apply artificial intelligence (AI) techniques, trying to face these limitations.The basic idea of the game is that the user guides a character through a maze to reach a goal point, running from enemies who, endowed with some AI methods, will follow paths to the player's position or to the point goal, in order to arrive before the user or catch him. In this context, some AI techniques that we can use are pathfinding methods, among them breadth-first search, depth-first search, uniform cost (ordered) search, greedy search and A star (A*) search.

Mobile devices usually have limited processing power and memory. When developing for this type of device, the developer should be aware of these limitations. On the other hand, develop an application that uses AI techniques can require a lot of memory and machine processing, depending on the treaty issue. Therefore, to develop a game using such techniques is necessary to establish a balance between limiting the mobile device, the complexity of the technique to be implemented and the user enjoyment. That is, the game developed may not require much AI processing, to be able to play against the user, and should also bring some entertainment to the player. In this sense, this work raises the question: is it possible to develop an intelligent game that is executable on a mobile device which has considerable difficulty and that is funny for the player?

Video games make up a solid market, and it is not different on mobile devices. Every day there are new games for these devices, of different styles. Moreover, a special attention is given in the artificial intelligence field, including techniques applied in games. There is a public that enjoys cooperative and competitive games, where there is one or more opponents to play against or in groups. In this context, it is reasonable to develop a game with some AI techniques, able to play against a player, and it is more interesting implement this idea on a mobile device, given its market size, and also given the challenges that arise when trying to apply AI techniques in such device. Creating this game can prove that it is possible for a mobile device entertain its users with games that can carry intelligence to assume the role of the opponent. Furthermore, it is very useful to obtain performance results of mobile devices when we use AI techniques, considering the response time, the memory usage, and other factors related to the device performance.

The main goal of this work is to implement and evaluate pathfinding algorithms in a game for Android. Thus, we can enumerate the following subgoals: implement pathfinding algorithms in a generic way for any graph; implement a game where an agent can follow a route implemented by a pathfinding algorithm; adjust some parameters (map size, number of AI agents etc.) in order to obtain a fluid game; evaluate the game performance and its pathfinding algorithms in a mobile device, comparing the results in different scenarios.

The remainder of this paper is organized as follows. Section 2 gives an introduction to Android and a summary of how to develop to this operating system (OS). Then, section 3 shows the pathfinding methods, since theirs basic structure to the particularities of each method used in this work. After, section 4 presents the implementation details of the game, including the class modeling, the search methods implementation, the code structure, the graph construction, the map editing etc. In section 5 we have the experiments and their results. Finally, in section 6, we present some conclusions of the work and some suggestions for further works.

2 Android

Android is an OS for mobile devices, open source, based on Linux kernel. Its development has been started by Android Inc., which was acquired by Google in 2005 [6]. In 2007, Android become developed and maintained by Open Handset Alliance (OHA), which is an alliance of 84 companies and led by Google [7]. Nowadays it is the most used OS in mobile devices. In the second quarter of 2015 it held 82.8% of the mobile market [5].

Android applications are mostly developed in Java, one of the most used programming languages in the world. Moreover, it is free, and open source. It is object-oriented and has a wide availability of libraries that help implementation. Other languages may also be used to program Android, such as C or C++, using a set of tools called NDK (Native Developer's Kit) [1]. Compiling an appli-

*e-mail: pvfreitas@ice.ufjf.br

cation for Android generates an APK package, and it is considered an Android application.

Google provides an official tool to develop for Android, called Android Studio, where we can program natively in Java [2]. There is a great community to support programming for Android, and Google provides an extensive material containing examples, step-by-step and others to support the creation of applications [4].

Firstly, to program for Android, the developer must choose which will be the minimum API (Application Programming Interface) level, or, in other words, the minimum Android version compatible to the app. The API may differ from each other, with respect to the support methods and Android libraries. To develop the proposed game and test the search algorithms, we chose the 4.1 version, since it is present in devices that exhibit good configuration, and moreover, the latest versions are capable to run an application of that version. Nowadays, 4.1 or higher versions are present in 95.7% of Android devices [3].

Secondly, for programming we chose the Java language because it is natively supported by Android Studio and has the necessary and sufficient resources for the development. Moreover, the game does not use advanced rendering functions that require an optimized language. The chosen API provides the necessary resources to run application with a good performance. When a Java code is compiled for the chosen API, it is converted to a bytecode that runs by the Dalvik virtual machine, which is especially built for Android, rather than Java virtual machine supplied by Oracle [6].

3 Pathfinding techniques

The purpose of using pathfinding techniques is to find a path between the starting point and the goal point on a map, maze or graph for example. Some of these techniques generate states exhaustively, covering all possible paths until find a solution, and others use heuristics or evaluation functions that guide the search, in order to find the solution more quickly and reduce the consumption of computing resources, generating less states, without losing the solution quality compared to exhaustive methods.

3.1 Structure of a pathfinding algorithm

A pathfinding algorithm uses a tree as the main structure. The search is performed during the tree construction, and the addition and expansion of its nodes follow a criterion established by the algorithm. Each tree node represents a state of the search and has the information of what was the parent node who generated it, the action applied to create it, and the path cost from the initial node (root) to it [8].

The edges that connect the nodes can have a value of local cost, which is the transition cost between the nodes connected by the edge. Usually this local cost is associated with the action that created such edge.

In addition to the tree structure, there is usually the use of two lists: the list of open nodes, that contains the generated nodes that have not been visited yet, and the list of closed nodes, that contains the expanded nodes, which have already been visited by the search. These lists can be consulted during the search to avoid the generation of repeated nodes, avoiding cycles (loops). Beside that, the list of closed nodes is used to retrieve the path found by the search if necessary. The order of expanding and inserting nodes in such lists may vary according to the pathfinding algorithm used.

When a node is selected for expansion, all reachable nodes from it are generated and inserted into the list of open nodes, and the expanded node is then inserted into the list of closed nodes and removed from the open list. In the proposed game the insertion of an expanded node in the closed list is carried with the information of who is your parent, so we can retrieve the path obtained by the algorithm when it found a state that is the goal point.

3.2 Uninformed methods

When the pathfinding algorithm is an exhaustive method, it is classified as uninformed method [8]. The solution is obtained by the exhaustive expansion of the nodes. That is, given a node, the method will generate all the next possible nodes, and the next node to be expanded will be selected according to the order in which it was generated, not considering any information of the problem domain. The visit and the expansion of the nodes are performed systematically through a graph. These methods are also known as weak methods. We implemented three uninformed methods: breadth-first, depth-first, and ordered searches.

The breadth-first search generates and expands nodes laterally in the tree. Thus, the tree is generated from level to level. That is, all the nodes of a level d are expanded before any node of the level $d+1$. The criterion for selecting a node to expand is to select the one that is the most time on the list of open nodes, that is, this list has the behavior of a queue. The next node to be expanded will be one on the same level of the newly expanded, or the first generated at the next level when there are no more nodes on the same level.

The depth-first search expands and generates vertically states, firstly performing the search to the 'deepest' node on the tree, then the next deepest, and so on. Given a node at a level d, all of the deeper nodes are visited and expanded before returning to the parent at the level d-1. To expand a node is selected from the open list one that is less time on the list, so the list has the behavior of a stack. In certain cases, it is not guaranteed that the method stops generating nodes. Depending on the problem the algorithm could generate nodes infinitely while there are computational resources. When this kind of problem can occur we can use an extension of the depth-first search, where the depth of the search tree is limited, i.e. the method generates nodes up to a given depth, and if a solution is not found the maximum depth is incremented and the process is repeated. This also allows to establish a maximum size for the path, since the tree depth indicates the size of the found path. In the case of our game there is no need to limit the depth, because, as shown in section 4.5, the application checks if it is not generating repeated states by consulting the open and closed lists, and there is no possibility of generating states infinitely, as the maze shows a limited number of nodes.

In the case of breadth-first and depth-first searches, a sequence of actions that will have priority to expand the nodes must be defined. For our game, we decided that the methods should try to go first to the north, west, south and then east.

Finally, the ordered search (also known as uniform-cost search) expands nodes selecting the one that has the lowest accumulated cost on the open list. This cost is the sum of the local costs of each edge on the path taken since the root to the current state. It is usually used in graphs that have costs on their edges (or weighted graphs). This method is not considered an informed method (see section 3.3) because it does not use data relating to the state as an expansion criterion, but only the path cost from the root to the state. Using this criterion, the ordered search can find the shortest path, i.e., the path with the lowest possible cost on its edges, not the path with the least number of edges as the breadth-first search. In the case of the developed game, the transition cost of the states is constant (as shown in section 4). Therefore, this search has a similar behavior to the breadth-first search, since the cost of a node is equal to its depth on the search tree, doing the method select a node for expansion on the lowest level of the search tree, as the breadth-first search. Nevertheless, for a comparison purpose, the ordered search was also implemented for the proposed game.

3.3 Informed methods

When a search uses some heuristic function, it is considered an informed search [8], where the selection of a node to expand considers an evaluation function, denoted by $f(n)$, which is the main factor that defines the method, where n is the current node. An informed method uses inherent information of the current instance of the problem and its states, in order to select those nodes that are more promising to reach the result faster.

An evaluation function can be defined in various ways, especially using cost information, which is usually denoted by $g(n)$, and heuristic functions, which is denoted by $h(n)$. As seen before on the ordered search, the cost is the value assigned to the transition from one state to another (the cost of the edge).

A heuristic function tries to predict an approximate value of the real cost from the current state to the goal. The heuristic used in the game of this work is the Manhattan distance, where are considered the current state and the destination point. The heuristic is obtained directly from the coordinates of the points on the Euclidean space. Considering a 2D space where the points are positioned with coordinates X and Y, and considering the current state $P_1 = (x_1, y_1)$, and the destination point $P_2 = (x_2, y_2)$, the Manhattan distance of these two points is given by:

$$h_{manhattan}(P_1, P_2) = |x_1 - x_2| + |y_1 - y_2| \quad (1)$$

This heuristic is interesting when the problem works with a pattern of coordinates, where they are discretized, as on a matrix for example. The calculation is simple and can be done quickly.

The heuristic is admissible, since it never overestimates the real cost to reach the goal, i.e. it always estimates a lower or equal cost than the real cost. As in our game there is movement restrictions in the north, south, east and west coordinated poles, with no movements in diagonal, the minimum size of the path to be followed is calculated by this heuristic.

The heuristic is also consistent (or monotonic), since for each node n and each successor n' generated from n with an action a, the estimated cost $h(n)$ to achieve the goal from n is not greater than the cost of going from n to n' plus the estimated cost $h(n')$, that is:

$$h(n) \leq c(n, a, n') + h(n') \quad (2)$$

This definition is a generalization of a triangle inequality, which says that each side of a triangle cannot be greater than the sum of the other two sides. Here, the vertices of this triangle are the nodes n, n', and the goal.

We implemented the informed methods greedy search and the A star (or A*) search. In both searches, we select to expand the node with the lowest evaluation function on the list of open nodes, independently of its depth on the search tree. The evaluation function of the greedy search considers only the heuristic function and the A* search has the evaluation function defined as the sum of the cost and the heuristic values, as follows:

$$f_{greedy}(n) = h(n) \quad (3)$$
$$f_{A^*}(n) = g(n) + h(n) \quad (4)$$

In the proposed game, where the cost is uniform, the A* evaluation function always has a constant value during the search when it generates states that approach the goal, because given a node n_1 with an heuristic value h_1 and cost g_1, walking a step to an adjacent node n_2 approaching the goal, the heuristic value h_2 is h_1 subtracted from one, but the cost will be increased by one. That is, with cancellations of constants, the evaluation function of n_2 is equal to n_1, thus:

$$f_{A^*}(n_2) = g(n_2) + h(n_2) = g(n_1) + 1 + h(n_1) - 1 = f_{A^*}(n_1) \quad (5)$$

That is, we must define the algorithm's behavior in case of a tie when choosing a node to expand. We chose arbitrarily select to expand that node that is less time on the open list. On the other hand, when A* generates a node that is getting away from the player, the cost and heuristics values are both incremented by one, i.e. the generated node has an evaluation higher than its parent. So this node will only be selected to expand if there is no path through the other nodes with lesser value.

4 The game

In this section we describe the modeling and implementation of the game and the pathfinding techniques.

4.1 Class modeling

The Figure 1 shows the class diagram projected for our game. As the main map representation we have the *Board* class, which stores the graph, the enemies and the player positions, the matrix to draw the map (maze), and the game states: victory (PLAYERWIN), defeat (GAMEOVER), game resuming (RESUME) and map construction (BUILDING_BOARD). This class is responsible for run the pathfinding algorithms for each enemy according to the user's movements, and also provides methods to help the map editing.

The matrix (*grid*) is populated with predefined values of barrier (BARRIER), empty space (EMPTY) and the objective position of the player (GOAL). From this matrix the graph with its adjacency list is populated, as we show in section 4.3.

An instance of *Board* maintains an enemy list populated dynamically at runtime. Each enemy has an instance of the *Search* class, which keeps stored its origin point of the search (*start*), its objective point (*end*), its pathfinding method (*searchType*) and its graph under which the search is performed. The origin point is a direct reference to the enemy point, i.e., the enemy movement is mirrored with this point. The same happens with the destination point: as the goal of the enemy is to get in the player's position, an user movement directly updates the end point. Thus, the project maintenance is facilitated, allowing the use of the pathfinding algorithms in different ways: the end point could be another convenient point in the map, as the goal point or other item that in future may be implemented, for example. In addition, an instance of *Search* keeps a list of points (*path*) indicating what was the path found by the search at the last run of the algorithm.

The *Enemy* and *Player* classes were implemented as inheritances of the *Entity* class, allowing the generalization of these objects for use of the searches.

The graph is represented by the *Graph* class with an adjacency list, and keeps in a separate instance the player objective point (GOAL). As a representation of a graph node we have the *GraphNode* class, which indicates the *point* node, its type (empty space or player goal), and its respective adjacency list. We cannot perform the search only with this class, since there is no storage for the cost and heuristic values used in some methods. For this, we have created the *SearchNode* anonymous class belonging to *Search* and as an inheritance of *GraphNode*. This class keeps the node cost, its evaluating value, and who is its parent node (*dad*).

4.2 Code structure

The classes in the project were divided in three packages: data structure (*br.edu.ufjf.core*), artificial intelligence (*br.edu.ufjf.ai*), and main game execution (*br.edu.ufjf.main*), which includes activities and visions of Android. In *core* package are the *Board*, *Entity*, *Enemy*, *Player*, *Graph* and *GraphNode* classes, whose details were discussed in section 4.1. In *ai* package we have the *Search* class, which is responsible for the pathfinding algorithms. Then, in *main* package, are the following classes:

- *Game*: activity that controls and displays the screen where the user can play;
- *GameView*: class that inherits from the *View* Android class, and implements the draw functions for play, and also handle the touchscreen events;

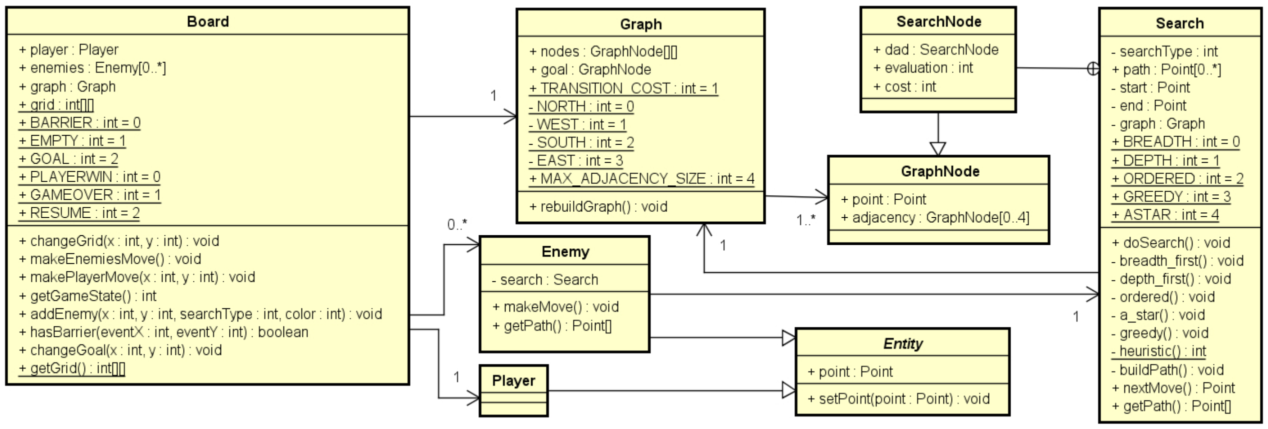

Figure 1: Class diagram

- *GameEditor*: activity that displays the screen for map editing;
- *GameEditorView*: this view has a similar behavior to the *GameView* class, but its functions are geared for map editing and not to play;
- *GameInstanceManager*: this class provides static methods that allow save and load map files, and it allows the crossing of *Board* instances between the Android activities;
- *Menu*: activity that displays the main game menu;
- *NewMap*: activity that displays options for set the width and height to create a new blank maze;
- *Splash*: the initial screen of the game which is displayed for three seconds before the main menu.

4.3 Graph building

When the map is changed, the graph is rebuilt by the *rebuildGraph()* function, of the *Graph* class. To facilitate the nodes retrieve in the graph, since the map is represented by a matrix, the nodes are also stored in a matrix too, called *nodes*, where given a position of the map the corresponding node is retrieved from *nodes* using the same coordinates. This matrix keeps objects of *GraphNode* type.

Firstly, the *rebuildGraph()* function passes through the grid of *Board*. Each position in *grid* generates a graph node in *nodes*, with their coordinates and type (barrier, blank space or player goal). After fill the nodes matrix, the adjacency lists are populated, thus completing the graph representation. The pathfinding algorithms use the adjacency lists to know what are the next possible states from a given node. In other words, these lists indicate what are the possible moves from a given position. It is possible up to four movements: north, south, east and west. In terms of coordinates *x* and *y*, these possible movements are (*x*-1, *y*), (*x*+1, *y*), (*x*, *y*-1), and (*x*, *y*+1).

4.4 Storage of the mazes

The storage for the mazes is made using serialization in file, where an entire object in memory is saved in a file directly [6]. We have reserved ten files (slots) to store the maps. The *GameInstanceManager* class handles these files. To store the mazes in file we have implemented two anonymous classes: *Instance* and *EnemyInstance*. These classes convert the lists in static sized vectors, since the serialization on Android do not allows the storage of lists. Thus, the saved files are from *Instance* type. When reading one of these files, its information are mapped back to a *Board* instance.

4.5 Implementation of the pathfinding methods

The execution of the pathfinding functions is a responsibility of the *Search* class. The pathfinding methods implemented in this class are breadth-first (*breadth_first*), depth-first (*depth_first*), ordered, greedy (*greedy*), and A* (*a_star*) searches. The general functioning of these methods was discussed in section 3. We show the Java code for a generic search on the following listing, and further we show the specifics of each search.

```
 1  private void generic_search(){
 2      List<GraphNode> adjacency;
 3      List<SearchNode> openList = new ArrayList<>();
 4      List<SearchNode> closedList = new ArrayList<>();
 5      SearchNode dadNode = new SearchNode(graph.
           getNode(start));
 6      SearchNode sonNode;
 7      openList.add(dadNode);
 8      while(!openList.isEmpty()&&!reachedObjective()){
 9          openList.remove(dadNode);
10          closedList.add(dadNode);
11          adjacency = dadNode.adjacency;
12          for(GraphNode adjNode : adjacency) {
13              sonNode = new SearchNode(adjNode);
14              sonNode.dad = dadNode;
15              evaluateNode(sonNode);
16              boolean contains = false;
17              if(!openList.contains(sonNode) &&
18                 !closedList.contains(sonNode)) {
19                  openList.add(sonNode);
20              }
21          }
22          dadNode = selectNextNode(openList);
23      }
24      buildPath(dadNode);
25  }
```

In this code, the list of open nodes is the *openList* object, and the list of closed nodes is the *closedList* object. While the open list is not empty (otherwise the search would not have found a path) or the current state of the loop does not match the position of the objective point (calculated by the *reachedObjective()* function), the function continues to expand states, inserting they into the closed list and inserting the new generated nodes into the open list. But, before the insertion of the generated nodes, the function verifies if they have not already been created by consulting the lists (between the lines 17 and 20). This step is important to prevent the search from loop. In this code the state that is being expanded is the *dadNode* object, and the new generated node is the *sonNode*. At the end of the search, the *buildPath()* function is called to retrieve the calculated path. It receives as parameter the last selected node for expansion, which corresponds to the objective point if a path has been found. From this node, this function retrieves the path using the parent node of each node, thus the path construction is made from the end to the beginning. The code of *buildPath()* is listed following:

```
private void buildPath(SearchNode endOfPath){
    path.clear();
    if(reachedObjective()) {
        SearchNode answer = endOfPath;
        while(answer.dad.point.x != start.x ||
              answer.dad.point.y != start.y) {
            path.addFirst(answer.point);
            answer = answer.dad;
        }
        path.addFirst(answer.point);
    }
}
```

Given a node of the path, it is inserted in *path*, then its parent node is selected, and so on. The *while* loop is only interrupted when the parent of the current node is equal to the starting point, indicating that the path was completely recovered.

The implemented pathfinding methods differ at two points in the generic search code: at node evaluation (line 11) and at node selection (line 22). On breadth-first and depth-first searches, the evaluation function have no effect. The difference between these methods is on the selection function: on the breadth-first search, we select the first node on the list of open nodes, and on the depth-first search we select the last one. The new generated nodes are always inserted at the end of the open list. Thus, in breadth-first search this list is a queue and in depth-first search this list is a stack. Thus, the techniques fill the search tree as described in section 3.2.

On the ordered, greedy and A* searches, a new generated node is evaluated by the *evaluateNode()* function. On the ordered search, this evaluation is the sum of the local costs of each edge on the path taken from the root to the point of the current state, as shown on the following code:

```
//evaluation of the ordered search
private void evaluateNode(SearchNode sonNode){
    sonNode.cost=dadNode.cost+Graph.TRANSITION_COST;
    sonNode.evaluation = sonNode.cost;
}
```

Since the cost is uniform, we have that the transition cost (TRANSITION_COST) is one. The evaluation of a new node is the sum of its parent cost and the transition cost.

On greedy search this evaluation is only the heuristic value of the node, as implemented on the following code:

```
//evaluation of the greedy search
private void evaluateNode(SearchNode sonNode){
    sonNode.evaluation=heuristic(sonNode.point,end);
}
```

Finally, on A* search the node evaluation is the sum of the heuristic value and the accumulated cost, as shown on the following code:

```
//evaluation of the A* search
private void evaluateNode(SearchNode sonNode){
    sonNode.cost=dadNode.cost+Graph.TRANSITION_COST;
    sonNode.evaluation=heuristic(sonNode.point, end)
          + sonNode.cost;
    openList.add(sonNode);
}
```

The heuristic function is a direct implementation of the Manhattan distance as defined in equation 1. The code of this function is:

```
private static int heuristic(Point p1, Point p2){
   return Math.abs(p2.x-p1.x) + Math.abs(p2.y-p1.y);
}
```

The node selection step is the same for the ordered, greedy and A* searches: they select the node with the lowest evaluation on the open list.

5 Experiments and Results

5.1 The test mazes

For test the game and the pathfinding algorithms, we have created three mazes, storing them in files. In these mazes the enemies were arranged on the bottom right corner of the mazes, and the player was placed on the top left. Each enemy was colored with a different color: red for breadth-first search, blue for depth-first search, pink for ordered search, grey for greedy search and black for A* search. The player is the green point. The Figure 2 shows the test mazes, and the Figure 3 shows the disposition of the player and of the enemies with the indication of each pathfinding method. The first maze has 30 units wide and 40 units high and has no barriers, thus its graph has 1200 nodes. That is, a pathfinding algorithm can generate 1200 states. This maze was created with the purpose of showing the general functioning of each algorithm on an open maze (without barriers). Each player movement implies on a new execution of each search algorithm to find a new path for each enemy. It is interesting to note that this maze has many optimal paths, so that some optimal methods found different paths.

It can be observed that the greedy and A* searches calculated the same path. The depth-first search gave a lot of laps to find a solution, as shown on Table 1. The breadth-first and ordered searches calculated different paths, although they found paths with the lowest possible cost as A* does. The ordered search tended to follow a diagonal path, given the way it select nodes to expand by choosing one with the lesser accumulated cost, and the breadth-first search followed paths with fewer curves, given its node selection that follows the order of precedence of the actions (north, west, south and east). The Greedy and A* searches displayed paths with fewer curves too, but they followed another direction due to the use of heuristics to expand nodes.

The second maze has 20 units wide and 30 units high. This maze was generated randomly and allows to show the behaviour of the pathfinding algorithms in a map with barriers. The player tried to reach the goal, indicated by the yellow square, but some enemies reached the player before.

We observed a similar behaviour of the algorithms in both mazes 1 and 2: the depth-first search calculated long paths, taking many laps. The A* and ordered searches always calculated the same path, and the greedy search followed different paths.

Lastly, the third maze was made with greater dimensions: 40 units wide and 50 units high. This maze was also generated randomly. In this case, the player reached the goal.

5.2 Results

We grouped the execution results of the pathfinding algorithms in the Tables 1 and 2. We collected the number of generated (Gen.) and expanded (Exp.) states, the size of found path (Size), and the runtime (Time) in milliseconds for each pathfinding algorithm. We show in Table 1 these numbers for each player movement in the first maze, and then the sum of these values for all 3 mazes in Table 2. The tests was executed in a Sony E1 Dual with 512 MB RAM and 1.2 GHz Dual-Core processor.

In the first maze the player has done 34 movements in total. In general we observed that the uninformed methods (breadth-first, depth-first and ordered searches) generated and expanded much more states than the informed methods (greedy and A* searches).

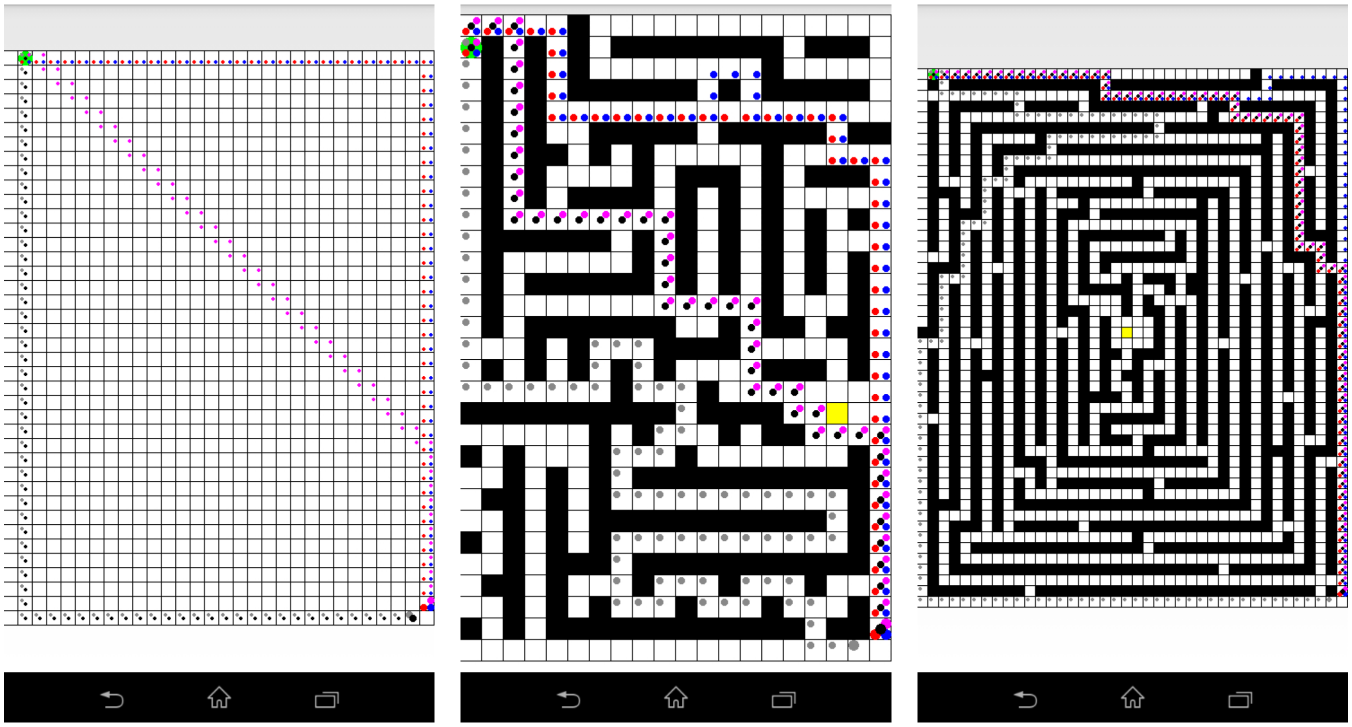

Figure 2: The test mazes. The smaller points indicate the path calculated for each enemy after the first movement of the player. The player begins on the upper left corner and the enemies on the bottom right corner. The first map has 30 units wide and 40 units high, the second map has 20 units wide and 30 units high and the third one has 40 units wide and 50 units high.

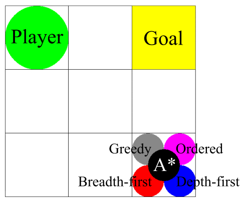

Figure 3: Identification of the game elements: player, goal and the enemies with their pathfinding methods.

The difference of runtimes was also large: the uniformed methods took much longer to respond. The depth-first search deserves a highlight for its abrupt change between the movements of the player: in some cases, this search has generated all possible states, in other cases it generated fewer states than the breadth-first search and took less time, but never found a shortest path and the path sizes varied widely. Moreover, in general, the sums of its values were very high. These variations can be observed at the game runtime: the path indicated by the depth-first search varies all the time. This indicates how bad the depth-first search can be for a pathfinding problem.

Although we said that A*, greedy, ordered and breadth-first searches always found the shortest path, note that the table shows different path sizes to the same movement of such searches. This is because, as these searches followed different paths, at times a player movement favored an enemy to approach it and detracted another to get away, which does not change the fact that these algorithms find the shortest path considering the state of the game in every move, where the enemies are at different positions.

In the second maze the player has done 29 movements. In this maze the depth-first search showed much variation between each movement as in the first maze, despite their summations have not been larger. The breadth-first search also encountered paths with the lowest cost, but presented runtimes and high numbers of generated and expanded states. The same has occurred with the ordered search. In general, the greedy search presented the lowest times, but for this maze it does not found the shortest path, despite using heuristics. This occurs because it follows a path attempting to approach the player but ignoring the path cost, i.e., it can approach the player but follow a large path, as it only uses the heuristic function, as shown in equation 4. The A* search presented good values: the number of expanded and generated states and the runtime were not so high, and the path had the lowest possible cost, as expected.

In the third maze the player has done many movements (97 in total). In this maze the pathfinding algorithms showed similar behaviors to the other mazes, but the difference in results was higher due to the larger size of the map.

These results show that the uninformed methods did not perform well, which makes them strong candidates to be dropped from a final version of the game. The informed methods have better results, both in generating and expanding the states and in runtimes. However, the greedy search does not always find the shortest path, while responds slightly faster than the A* search. That is, the use of these two methods is interesting, and the greedy search can be used when we want a faster response and without concern for its quality, since this search is not optimal, and the A* can be used when we want to get the shortest path with a good runtime.

6 Conclusion

This paper presented a study of the application of pathfinding techniques in developing an Android game. As the results show, techniques that use heuristics are the most recommended, since they have better performance compared to other searches, which is al-

Table 1: Results of the execution of the pathfinding algorithms for the first maze. For each algorithm we show the number of generated (Gen.) and expanded (Exp.) states, the size of the found path (Size), and the runtime (Time) in milliseconds. We show these numbers for each player movement (Mov.).

	Breadth-first				Depth-first				Ordered				Greedy				A*			
Mov.	Gen.	Exp.	Size	Time	Gen.	Exp.	Size	Time	Gen.	Exp.	Size	Time	Gen.	Exp.	Size	Time	Gen.	Exp.	Size	Time
1	1199	1198	66	2565	133	67	66	51	1200	1198	66	2113	172	67	66	31	172	67	66	29
2	1198	1196	64	2102	1200	1134	66	3094	1198	1195	64	1921	168	65	64	16	168	65	64	20
3	1195	1192	62	2017	1200	1136	64	3033	1196	1192	62	2059	164	63	62	17	164	63	62	16
4	1192	1188	60	2059	282	143	142	87	1192	1187	60	2010	159	61	60	13	159	61	60	13
5	1187	1182	58	2190	281	143	142	45	1188	1182	58	2187	155	59	58	13	155	59	58	14
6	1182	1176	56	1995	1200	1064	140	2837	1182	1175	56	2138	150	57	56	13	150	57	56	26
7	1175	1168	54	2104	280	143	142	46	1176	1168	54	2070	146	55	54	12	146	55	54	12
8	1168	1160	52	2242	280	143	142	49	1168	1159	52	2023	141	53	52	10	141	53	52	11
9	1159	1150	50	2022	1200	1064	142	2940	1160	1150	50	1944	137	51	50	17	137	51	50	12
10	1150	1140	48	1923	1200	1064	142	2861	1150	1139	48	1893	132	49	48	9	132	49	48	10
11	1139	1128	46	1790	414	211	210	141	1140	1128	46	1926	128	47	46	8	128	47	46	9
12	1128	1116	44	1902	410	209	208	127	1128	1115	44	2078	123	45	44	49	123	45	44	9
13	1115	1102	42	1838	1200	1000	208	2644	1116	1102	42	1890	119	43	42	20	119	43	42	19
14	1101	1085	40	1795	1200	1002	206	2467	1102	1087	40	1757	114	41	40	6	114	41	40	8
15	1077	1057	38	1744	424	217	216	114	1088	1072	38	1683	110	39	38	7	110	39	38	10
16	1044	1020	36	1518	424	217	216	124	1069	1049	36	1835	105	37	36	6	105	37	36	7
17	1000	972	34	1999	1200	994	216	2793	1050	1028	34	1828	101	35	34	6	101	35	34	5
18	947	915	32	1269	1200	994	216	2453	1017	991	32	1496	96	33	32	32	96	33	32	6
19	883	847	30	1064	542	277	276	211	992	964	30	1406	92	31	30	5	92	31	30	4
20	810	771	28	971	538	275	274	235	945	913	28	1379	88	29	28	5	88	29	28	7
21	730	691	26	814	1200	938	274	2230	911	874	26	1158	88	29	28	16	88	29	28	17
22	649	610	24	638	1200	940	272	2204	847	805	24	1004	83	27	26	4	83	27	26	3
23	565	526	22	615	568	291	290	336	795	749	22	1153	82	27	26	5	82	27	26	3
24	481	444	20	328	568	291	290	246	714	666	20	675	77	25	24	9	77	25	24	4
25	402	368	18	224	1200	924	290	2343	646	596	18	699	76	25	24	3	76	25	24	3
26	331	300	16	160	1200	924	290	2313	547	496	16	406	71	23	22	3	71	23	22	3
27	266	238	14	99	670	343	342	390	467	418	14	303	70	23	22	10	70	23	22	9
28	296	266	14	167	666	341	340	364	371	326	12	176	65	21	20	3	65	21	20	2
29	261	230	12	105	1200	876	340	2024	294	251	10	105	64	21	20	2	64	21	20	3
30	284	251	12	102	1200	878	338	2061	210	171	8	84	59	19	18	2	59	19	18	2
31	235	203	10	68	712	365	364	383	139	107	6	22	58	19	18	2	58	19	18	2
32	250	216	10	81	712	365	364	380	79	55	4	7	53	17	16	3	53	17	16	2
33	192	160	8	60	1200	854	364	1998	39	23	2	2	52	17	16	2	52	17	16	2
34	199	165	8	61	1200	854	364	2202	10	3	0	0	47	15	14	1	47	15	14	2

Table 2: Summations of the execution data of the pathfinding algorithms for each maze. For each algorithm we show the number of generated (Gen.) and expanded (Exp.) states, the size of the found path (Size), and the runtime (Time) in milliseconds.

	Breadth-first				Depth-first				Ordered				Greedy				A*			
Maze	Gen.	Exp.	Size	Time	Gen.	Exp.	Size	Time	Gen.	Exp.	Size	Time	Gen.	Exp.	Size	Time	Gen.	Exp.	Size	Time
1	27190	26431	1154	40631	28304	20681	7956	45826	28526	27734	1122	43430	3545	1268	1234	360	3545	1268	1234	314
2	6334	6139	692	1971	3381	2996	1670	1056	6048	5840	670	1959	2138	1972	1458	266	2865	2552	670	635
3	46406	44835	4520	19152	37663	35050	24326	14073	46809	45138	4520	17079	13550	12249	7534	1760	12496	11347	4520	1327

ready expected according to the literature. In the case of the greedy search, despite using a heuristic function, it did not always calculate the shortest path, but had the lowest runtimes. The A* search always calculated the shortest path, and its runtimes were not much larger than the runtimes of the greedy search.

When compared to the uninformed methods (especially the depth-first search), the greedy and A* searches performed better. A game needs to be fluid, quick to respond to external stimuli (player's actions), and must also present a challenge that make it interesting, without being impossible to beat. So, even if we are seeking for a search method with the best possible performance, it may not be so effective as to render impossible the user's victory. Thus, depending on the situation, the greedy search can be more interesting, since it has low response time and does not always calculate the best path, which would give an advantage to the player.

The type of the proposed game requires almost an immediate response of the pathfinding algorithm, given the rapid dynamic of switching between a player movement and the enemies movements. It would not be interesting if the game had to stop its execution to calculate the movements, otherwise the whole dynamic would be broken. As shown in the results, some algorithms have answered only after two seconds, which, in this context, is a high time. It is better to give preference to algorithms that respond more quickly.

6.1 Future works

This game features an mechanics of alternation between plays, which requires a fast response time of the used pathfinding method. However, we propose another dynamic where the movements of the enemies are calculated regardless of the player's movement, according to some time interval. That is, the enemies will move all the time and it would require for the player to have a quick wit to trace a route to escape from them. In this case, the difficulty would be set by the time interval between the path calculations and the enemies speed. If the pathfinding function does not respond in time, an enemy could continue following a path previously calculated until a new one is obtained. In this context, it would be necessary the implementation of threads, thus bringing the concepts of parallel execution. Issues like synchronization of functions would have to be observed.

Another proposal is a maze in constant motion like a side-scrolling game. The maze would be generated randomly all the time, and the player would have to run away from enemies that are chasing it. These enemies would be always choosing the shortest path to the player, forcing it to choose the shortest path as well. If the player follows a larger path, the enemies would approach it, until one of them catches the player. In this game mode, it would not have a goal point. Instead, the goal would be just to break records: at each time, the player would try to go further. Nowadays many games use ranking mechanic, often synchronizing the players data on the cloud, in order to create a competition between them.

As shown, depending on the map and the positions of the entities, there may be multiple optimal paths. The pathfinding methods have been implemented in order to use the first optimal path found (in the case of optimal methods). However, it may be interesting to allow these methods to find all the optimal paths and then use some function to estimate where the player will follow to choose among the optimal paths which is closest to it. From this, we can also program the enemies to work together to try to hinder the player to follow different paths.

Finally, we also propose the inclusion of items in the game, which can be used by both the player and the enemies, making the pathfinding methods have new information to work: when choosing an item or not . This would require a new layer of artificial intelligence, which would work with probabilities, making the game more dynamic and making enemies have closer to a human player behavior. Examples of items are:

- Mobile barriers: blocks that can be moved, changing the map and requiring the graph update;

- Speed items: in cases that the game has constant movements of the enemies, it could have items that change the speeds of the player or of the enemies;

- Portals: items that allow the transport of an entity from a point to another instantly;

- Lives: the player could have a number of lives, thus, when it was hit by an enemy, it would lose a life instead of lose the entire game, while it has lives;

- Immunity: temporary effect that allows the player be hit by an enemy without anything happening.

The development of other elements, as a collaborative online mode between mobile devices (via Bluetooth or network wireless for example) and the sharing of custom mazes by cloud, is also encouraged.

Acknowledgments

The authors would like to thank Universidade Federal de Juiz de Fora for the financial support, through scientific initiation scholarships (BIC/UFJF).

References

[1] Google. Android ndk. Available at: http://developer.android.com/intl/pt-br/tools/sdk/ndk/index.html, 2016. Accessed February, 2016.

[2] Google. Como baixar o android studio e o sdk tools. Available at: http://developer.android.com/intl/pt-br/sdk/index.html, 2016. Accessed February, 2016.

[3] Google. Dashboards. Available at: http://developer.android.com/intl/pt-br/about/dashboards/index.html, 2016. Accessed June, 2016.

[4] Google. Develop apps. Available at: http://developer.android.com/intl/pt-br/develop/index.html, 2016c. Accessed February, 2016.

[5] IDC. Idc: Smartphone os market share 2015, 2014, 2013 and 2012. Available at: http://www.idc.com/prodserv/smartphone-os-market-share.jsp, 2016. Accessed June, 2016.

[6] Z. Mednieks, L. Dornin, G. B. Meike, and M. Nakamura. *Programando o Android*. Novatec Editora Ltda., 2012.

[7] OHA. Overview. Available at: http://www.openhandsetalliance.com/android_overview.html, 2016. Accessed February, 2016.

[8] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 3 edition, 2009.