Universidade Federal de Juiz de Fora
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

Aplicação de Técnicas de Busca de Caminhos no Desenvolvimento de um Jogo para Android

Paulo Vitor Freitas da Silva

JUIZ DE FORA
MARÇO, 2016

Aplicação de Técnicas de Busca de Caminhos no Desenvolvimento de um Jogo para Android

Paulo Vitor Freitas da Silva

Universidade Federal de Juiz de Fora

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Bacharelado em Ciência da Computação

Orientador: Saulo Moraes Villela

JUIZ DE FORA

MARÇO, 2016

Aplicação de Técnicas de Busca de Caminhos no Desenvolvimento de um Jogo para Android

Paulo Vitor Freitas da Silva

MONOGRAFIA SUBMETIDA AO CORPO DOCENTE DO INSTITUTO DE CIÊNCIAS EXATAS DA UNIVERSIDADE FEDERAL DE JUIZ DE FORA, COMO PARTE INTEGRANTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE BACHAREL EM CIÊNCIA DA COMPUTAÇÃO.

Aprovada por:

Saulo Moraes Villela
D.Sc. em Engenharia de Sistemas e Computação

Luciana Conceição Dias Campos
D.Sc. em Engenharia Elétrica

Romualdo Monteiro de Resende Costa
D.Sc. em Informática

JUIZ DE FORA
03 DE MARÇO, 2016

Resumo

Atualmente a vida sem dispositivos móveis é impensável. A demanda para esse tipo de aparelho é crescente, e consequentemente a demanda por aplicativos para esses dispositivos também. Em contrapartida, esses aparelhos geralmente possuem pouco poder de processamento e memória limitada, o que demanda cuidado ao se implementar métodos que requerem desempenho do aparelho. Este trabalho propõe o estudo da aplicação de técnicas de busca de caminhos na criação de um jogo para dispositivos com Android.

Palavras-chave: Android, Inteligência Artificial, Busca de Caminhos, *Path Finding*, Dispositivos Móveis, Jogos

Abstract

Nowaday, life without mobile devices is unthinkable. The demand for this type of device is increasing, and consequently the demand for applications for these devices as well. In contrast, these devices generally have low processing power and limited memory available, which makes it a challenge to implement methods that require device performance. This work proposes the development of a game for Android devices with path finding algorithms, facing such constraints.

Keywords: Android, Artificial Inteligence, Path Finging, Mobile, Games

Agradecimentos

Agradeço aos meus pais e irmã, por todo o apoio e motivação. Ao meu orientador, Saulo, pelo suporte, ajuda, e toda luz que deu para me guiar nesse trabalho. Aos professores, mestres, que me acompanharam durante o curso, sempre dispostos a ensinar e ajudar, transmitindo todo seu conhecimento e nos motivando a produzir mais. Aos colegas e fiéis amigos que fiz durante o curso, pelo companheirismo e experiências compartilhadas. E aos colegas do projeto que ajudaram testando e revendo códigos, e dando ideias novas para a continuação do trabalho.

Sumário

Lista de Abreviações

IA	Inteligência Artificial
SO	Sistema Operacional
A*	A estrela
API	Application Programming Interface
NDK	Native Developer's Kit
JDK	Java Development Kit

1 Introdução

1.1 Contexto e motivação

É notório o intenso crescimento do mercado de dispositivos móveis. Com ele surge, na mesma proporção, a demanda por desenvolvimento de tecnologias e softwares que incrementem as funcionalidades desses dispositivos. E uma dessas demandas diz respeito especialmente ao desenvolvimento de jogos. A limitação de processamento e memória desse tipo de dispositivo faz com que os desenvolvedores para dispositivos móveis tenham que exercitar sua criatividade e conhecimento para criar jogos que sejam capazes de serem executados dadas essas limitações e, ao mesmo tempo, tragam entretenimento ao usuário.

Este trabalho propõe estudar a aplicação de técnicas de inteligência artificial (IA) no desenvolvimento de um jogo para dispositivos com Android, enfrentando, assim, essas limitações. A ideia básica do jogo é que o usuário guie um personagem através de um labirinto até atingir um ponto objetivo, correndo de inimigos que, dotados com algum método de inteligência artificial, percorrerão um caminho até a posição do jogador ou até o ponto objetivo, a fim de chegar antes do usuário ou atrapalhá-lo. Nesse contexto, as técnicas de IA a serem usadas são as de busca de caminhos, dentre elas a buscas em largura, em profundidade, ordenada, gulosa, e A* (a estrela).

A IA é uma grande área da ciência da computação, e também uma área em franco e constante desenvolvimento. Nesse sentido, é tratado neste trabalho os conceitos básicos de IA e estudado como aplicá-los no jogo, além de estabelecer, para tanto, regras que ajudem a determinar uma heurística que será usada na implementação do jogo.

1.2 Justificativa

Dispositivos móveis possuem poder de processamento e memória limitados. Ao desenvolver para esse tipo de dispositivo, o desenvolvedor deve ter consciência dessas limitações. Em contrapartida, desenvolver uma aplicação que utilize técnicas de IA pode demandar

muita memória e processamento da máquina, dependendo do problema tratado. Por isso, para desenvolver um jogo que utilize tais técnicas é necessário estabelecer um ponto de equilíbrio entre a limitação do dispositivo móvel, a complexidade da técnica a ser implementada e a diversão do usuário. Ou seja, o jogo desenvolvido tem que, ao mesmo tempo, não exigir muito processamento de IA, para ser capaz de jogar contra o usuário, e também deve trazer algum entretenimento ao jogador. Nesse sentido, esse trabalho levanta a questão: é possível desenvolver um jogo inteligente que seja executável em um dispositivo móvel, que possua dificuldade considerável e que seja divertido para o jogador?

Jogos eletrônicos compõem um mercado sólido, e não é diferente nos dispositivos móveis. A cada dia surgem novos jogos para esses aparelhos, de diferentes estilos, para todos os gostos. Além disso, muito se estuda na área de inteligência artificial, incluindo as técnicas que são aplicadas em jogos. Há um público que gosta de jogos cooperativos e competitivos, em que há um ou mais oponentes para se jogar contra ou em grupos. Nesse contexto, é plausível estudar e desenvolver um jogo com alguma técnica de IA, capaz de jogar contra uma pessoa, e mais interessante ainda é a implementação dessa ideia em um dispositivo móvel, dado o tamanho do mercado em que tanto os dispositivos móveis quanto os jogos eletrônicos se mostram fortalecidos, e dado também os desafios que surgem ao tentar aplicar IA em um aparelho desse tipo. Criar esse jogo permite provar que é possível um dispositivo móvel entreter seu usuário com jogos que portam consigo inteligência para assumir o papel do adversário. Além disso, é muito útil se obter resultados sobre o desempenho de aparelhos móveis quando usados para processar IA, levando em conta o tempo de resposta, uso de memória, entre outros fatores ligados ao desempenho do aparelho.

1.3 Objetivos

O objetivo desse trabalho é implementar e avaliar algoritmos de busca de caminhos num jogo para Android. Sendo assim, pode-se dividir o objetivo desse trabalho nos seguintes tópicos:

- Implementar algoritmos de busca de caminho de forma genérica para um grafo

qualquer;

- Implementar um jogo com percurso de caminhos com elementos que utilizem os algoritmos de busca implementados;
- Ajustar parâmetros (tamanho do mapa, número de agentes inteligentes, etc.) de forma a obter um jogo fluido;
- Avaliar o desempenho do jogo e de seus algoritmos de IA no dispositivo móvel, comparando os resultados de diferentes cenários.

1.4 Organização do texto

O restante do texto está organizado da seguinte maneira: no capítulo 2 é dada uma introdução ao Android e um resumo de como é desenvolver para esse sistema operacional (SO). Em seguida, no capítulo 3 os métodos de busca de caminho são abordados, desde a estrutura básica de uma busca desse tipo até as particularidades de cada um dos métodos usados nesse trabalho. Após, o capítulo 4 traz os detalhes da implementação do jogo, incluindo a modelagem das classes, a implementação dos métodos de busca, a estrutura do código, a construção do grafo, o desenho das telas, a edição dos mapas, etc. No capítulo 5 tem-se os experimentos realizados e seus resultados com suas respectivas análises. Por fim, no capítulo 6, é apresentada a conclusão do trabalho, juntamente com uma listagem de sugestões de trabalhos futuros a partir deste.

2 Android

O Android é um sistema operacional para dispositivos móveis, de código aberto, baseado no *kernel* do Linux. Foi desenvolvido pela Android, Inc., a qual foi adquirida pelo Google em 2005 (Mednieks et al, 2012). Em 2007, o Android passou a ser desenvolvido e mantido pela *Open Handset Alliance*, que é uma aliança de 84 empresas e liderada pelo Google (OHA, 2016). É o SO mais utilizado em dispositivos móveis atualmente. Só em 2014, ocupou 84,70% do mercado, com aproximadamente 1 bilhão de usuários ativos, e 1,3 milhões de aplicativos disponíveis em sua loja. A versão mais usada naquele ano foi a Jelly Bean (4.1) (Tecmundo, 2014). A distribuição das versões do Android atualmente nos dispositivos são mostradas na Tabela 2.1.

Tabela 2.1: Distribuição do uso das versões Android em 2016 (Google, 2016d)

Versão	Nome	API	Distribuição
2.2	Froyo	8	0.1%
2.3.3 - 2.3.7	Gingerbread	10	2.7%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	2.5%
4.1.x	Jelly Bean	16	8.8%
4.2.x	Jelly Bean	17	11.7%
4.3	Jelly Bean	18	3.4%
4.4	KitKat	19	35.5%
5.0	Lollipop	21	17.0%
5.1	Lollipop	22	17.1%
6.0	Marshmallow	23	1.2%

2.1 Desenvolvendo para Android

Os aplicativos Android são desenvolvidos principalmente em linguagem Java, uma das linguagens mais usadas do mundo. Além disso, é gratuita, e de código-fonte aberto. Ela é orientada a objetos, e possui uma ampla disponibilidade de bibliotecas que auxiliam a implementação. Outras linguagens também podem ser utilizadas para programar para Android, como o C ou o C++, utilizando-se um conjunto de ferramentas denominado NDK (*Native Developer's Kit*) (Google, 2016a). Compilar um aplicativo para Android

gera um pacote *.apk*, considerado por si só uma aplicação Android.

O Google fornece uma ferramente oficial para desenvolvimento para Android, chamada Android Studio, onde pode-se programar nativamente em linguagem Java (Google, 2016b). Existe uma grande comunidade para suporte a programação para Android, e a própria Google fornece um extenso material que contém exemplos, passo-a-passo, entre outros para suportar a criação de aplicativos (Google, 2016c).

Primeiramente, para programar para Android, o desenvolvedor deve escolher qual o nível mínimo da API (*Application Programming Interface*) da aplicação. O número da API para cada versão do Android é mostrado na Tabela 2.1. Escolher essa API mínima significa definir para qual versão mínima do Android a aplicação será compatível. As API podem apresentar diferenças entre si, no que diz respeito ao suporte de métodos e bibliotecas do Android. Para desenvolver o jogo aqui proposto, escolheu-se a versão 4.1, por estar presente em aparelhos que apresentam uma boa configuração, possuir uma parcela de mercado considerável, e, além disso, as versões mais recentes são capazes de executar um aplicativo dessa versão.

Em segundo, escolheu-se a linguagem Java para a programação, por ser nativamente suportada pelo Android Studio, e por possuir os recursos necessários e suficientes para apoiar o desenvolvimento. Além disso, o jogo não usa funções de renderização[1] avançadas que exijam o uso de uma linguagem otimizada para tal. A API fornece os recursos necessários para que a aplicação seja executada com um bom desempenho. Ao compilar um código em Java para a API escolhida, ele é convertido num *bytecode* lido pela máquina virtual *Dalvik*, construída especialmente para o Android, em vez da máquina virtual JDK (Java Development Kit) comum do Java, fornecido pela Oracle (Mednieks et al, 2012).

2.2 Estrutura de uma aplicação Android

O Android trabalha com o conceito de atividades. Uma atividade do Android é um elemento da aplicação que interage diretamente com o usuário, preenchendo toda a tela

[1] Quando usada em processamento gráfico, é o processo que fornece o produto (imagem) final a partir de uma representação mais simples, aplicando a essa representação técnicas de textura, iluminação, sombra, entre outras técnicas de computação gráfica.

do dispositivo, e que permanece em execução enquanto estiver sendo mostrada ao usuário. Na programação, criar uma atividade Android significa criar uma classe que estende *Activity*, que fornece métodos prontos para o desenvolvedor, e o obriga a implementar alguns métodos, principalmente o *onCreate()*, que é executado tão logo a atividade é iniciada. É no *onCreate()* que é definido como a tela será desenhada: se a partir de uma tela pré-definida em arquivo XML, ou executando uma classe que estende a classe *View* (Mednieks et al, 2012).

Uma *View* é uma visão do Android, onde, via linguagem de programação, é definido como o desenho da tela ocorre, através da função *onDraw()*. Outras funções de uma visão também têm que ser implementadas, como a *onTouch()*, que trata os eventos de toque de tela (Mednieks et al, 2012).

Além das visões e atividades, uma aplicação possui um arquivo importante denominado *AndroidManifest.xml*, ou arquivo de manifesto. Esse arquivo define os parâmetros básicos e de inicialização da aplicação. Ele indica qual a versão mínima da API, quais as permissões que o Android precisa conceder sobre o dispositivo (como o uso de câmera, de conexão com internet, etc.), lista as atividades da aplicação e indica qual delas é a primeira a ser executada, entre outras informações (Mednieks et al, 2012). Como mostrado no capítulo 4, o jogo aqui proposto precisa gravar e ler mapas de arquivo, portanto foi necessário inserir essa permissão no arquivo de manifesto no projeto da aplicação.

Quando as telas são definidas via arquivo XML, esses arquivos são armazenados numa pasta especial do projeto, denominada *layout*. Esses arquivos são editáveis tanto via texto, quanto via interface gráfica fornecida pelo Android Studio, onde é possível dispor botões, textos, imagens, entre outros elementos, e definir para cada um deles qual é o evento a ser disparado. Um botão pode, por exemplo, iniciar uma outra atividade. Por fim, arquivos de mídia, como imagens, são armazenados na pasta *drawable* (Mednieks et al, 2012).

3 Buscas de Caminhos

O objetivo de usar técnicas de busca de caminhos (*path finding*) é de encontrar um caminho entre o ponto de partida e o ponto objetivo, seja num mapa ou num grafo por exemplo. Algumas dessas técnicas geram estados exaustivamente, podendo percorrer todos os caminhos possíveis até encontrar uma solução, e outras usam heurísticas ou funções de avaliação que permitem direcionar a busca e gerar menos estados, na intenção de encontrar a solução mais rapidamente e economizar recursos computacionais, sem perder qualidade da solução em relação aos métodos exaustivos.

3.1 Estrutura de uma busca de caminho

Uma busca de caminho usa uma árvore como estrutura principal. É durante a construção da árvore que a busca é realizada, e a adição e expansão de seus nós obedece a um critério estabelecido pelo algoritmo usado (as particularidades dessa construção para cada algoritmo são abordadas nas seções 3.2 e 3.3). Como descreve Russell (2010), cada nó da árvore representa um estado da busca, e possui as seguintes informações:

- ESTADO: o estado que o nó representa;
- PAI: o nó que gerou o estado;
- AÇÃO: a ação que foi executada que levou a geração do estado;
- CUSTO: custo do caminho a partir do nó inicial até o nó atual.

As arestas que ligam os nós podem possuir um valor de custo local, que é o custo da transição entre os nós ligados pela aresta. Geralmente esse custo local está associado à ação que criou tal aresta.

Além da estrutura de árvore, geralmente há o uso de duas listas: a lista de nós abertos e a lista de nós fechados. Na primeira são armazenados os nós gerados e que ainda não foram visitados, e na segunda se encontram os nós expandidos, que já foram visitados

pela busca. Essas listas podem ser consultadas durante toda a busca para evitar a geração de nós repetidos, evitando assim ciclos (*loops*). Além disso, a lista de nós fechados é usada para recuperar o caminho encontrado pela busca, quando necessário. A ordem de leitura e de inserção de nós nessas listas pode variar de acordo com o algoritmo de busca utilizado.

3.2 Métodos não-informados

Segundo Russell (2010), quando a busca utiliza um método exaustivo, ela é classificada como método não-informado de busca. Nele, a solução é obtida a partir da expansão exaustiva de todos os nós. Ou seja, dado um nó, o método irá gerar todos os próximos nós possíveis a partir dele, e o próximo nó a ser expandido será escolhido unicamente pela ordem em que ele foi gerado, não levando em consideração nenhuma informação que o nó possui. A visita e a expansão dos nós são feitas de forma sistemática, através de um grafo. Esses métodos também são conhecidos como métodos fracos.

3.2.1 Busca em largura

A busca em largura gera e expande nós lateralmente na árvore de busca. Sendo assim, a árvore é gerada nível a nível, ou seja, todos os nós de um determinado nível d são expandidos antes de qualquer nó do nível $d+1$. O critério de seleção de nó para expansão é selecionar aquele que está a mais tempo na lista de nós abertos, fazendo com que essa lista tenha o comportamento de uma fila. Selecionado um nó para expansão, todos os nós possíveis a partir dele são gerados e inseridos na lista de abertos, e então o nó expandido é inserido na lista de fechados, e o próximo a ser expandido será um nó do mesmo nível do recém-expandido, ou quando não houver mais nós no mesmo nível, o primeiro gerado no próximo nível. No caso do jogo aqui proposto, a inserção do nó expandido na lista de fechados é feita junto com a informação de quem é o seu nó "pai" (aquele que a partir dele obteu-se o nó expandido). Posteriormente será mostrado porque essa informação é necessária. Ao expandir um nó, deve ser definido uma sequência de ações que terão prioridades. Aqui, foi definido que, primeiro, deve tentar-se ir para o norte, depois oeste, depois sul, e por fim leste.

3.2.2 Busca em profundidade

A busca em profundidade gera e expande estados verticalmente, efetuando primeiro a busca até o nó mais "profundo" na árvore, depois o próximo mais profundo, e assim por diante. Dado um nó em um nível d, todos os próximos nós mais profundos são visitados e expandidos antes de retornar para o "pai" no nível $d - 1$ na árvore. Para seleção de nó a ser expandido, é extraído da lista de abertos aquele que está a menos tempo na lista, fazendo com que ela tenha o comportamento de uma pilha. Selecionado um nó para expansão, todos os nós possíveis a partir dele são gerados e inseridos na lista de abertos, e o próximo a ser expandido é um desses estados recém-gerados. Assim como na busca em largura, o nó expandido também é inserido na lista de fechados juntamente com a indicação de quem é seu "pai". A expansão de um nó para esse método também segue uma regra de prioridades assim como na busca em largura, e também foi definido como ir primeiro para o norte, depois oeste, depois sul, e por fim leste.

Em determinados casos, não é garantida a parada de geração de nós num determinado nível na busca em profundidade. Dependendo do problema, o algoritmo poderia gerar nós "infinitamente", enquanto houvesse recursos computacionais. Quando esse tipo de problema pode ocorrer, pode-se usar uma extensão da busca em profundidade, onde a profundidade da árvore de busca é limitada, ou seja, o método gera nós até uma determinada altura da árvore, e caso não seja encontrada uma solução para o problema, a profundidade máxima é incrementada, e o processo se repete. Isso também permite estabelecer um tamanho máximo para o caminho máximo, uma vez que a altura da árvore indica o tamanho do caminho encontrado. No caso do jogo desse trabalho, não há necessidade de se limitar a profundidade, pois, como é mostrado na seção 4.6, a aplicação verifica se não está gerando estados repetidos ao consultar as listas de nós abertos e fechados, e como o mapa apresenta um número limitado de nós, não há a possibilidade de geração de estados infinitamente.

3.2.3 Busca ordenada

A busca ordenada gera e expande nós selecionando na lista de abertos aquele de menor custo acumulado. Esse custo é dado pela soma dos custos locais de cada aresta do caminho

percorrido da raiz até o ponto do estado atual da busca. Geralmente é usada em grafos que possuem custos em arestas (ou grafos ponderados). Esse método não é considerado informado (vide seção 3.3) pois não utiliza dados inerentes ao estado como critério para expansão, mas sim o custo do caminho já percorrido da raiz até o estado. Usando esse critério, a busca ordenada consegue encontrar o caminho de custo mínimo, ou seja, o caminho que soma o menor custo possível de suas arestas, e não o caminho que possui o menor número possível de nós. No caso do jogo aqui desenvolvido, os custos das transições (arestas) dos estados é constante (como é mostrado no capítulo 4). Portanto, essa busca apresenta comportando semelhante a uma busca em largura, pois o custo do estado é igual à sua altura na árvore de busca, fazendo com que o método selecione um nó para expansão do menor nível que possuir estados abertos na árvore de busca, assim como a busca em largura faz. Mesmo assim, para efeitos de comparação, essa busca é implementada no jogo.

3.3 Métodos informados

Também segundo Russell (2010), quando a busca utiliza alguma função heurística, ela é considerada um método informado de busca, onde a seleção de um nó para expansão leva em consideração uma função de avaliação do estado, denotada por $f(n)$, que é o principal fator que define o método, onde n é o nó atualmente selecionado para expansão na busca. Em outras palavras, um método informado faz uso de informações adicionais, inerentes à instância atual do problema e aos estados dela, com o objetivo de selecionar para expansão aqueles nós que são mais promissores para fornecer o resultado mais rapidamente.

Uma função de avaliação pode ser definida de diversas formas, em especial usando informações de custo, geralmente denotada por $g(n)$, e funções heurísticas, denotadas por $h(n)$. O custo, como visto na busca ordenada (seção 3.2.3), é o valor atribuído à transição de um estado a outro (custo da aresta). Ou seja, é levado em consideração o custo para partir de um estado e ir para o próximo. Esse custo é sempre o valor real da transição entre os estados. Por exemplo, na busca de um caminho entre duas cidades, onde podem haver várias outras cidades durante o percurso, o custo é representado pela distância real a ser percorrida de uma cidade até a cidade subsequente.

Já a função heurística tenta prever um valor aproximado do custo real do estado atual da busca até o objetivo. No caso da busca de caminhos, uma das heurísticas mais utilizadas é a distância Euclidiana, definida pela distância em linha reta do estado atual ao ponto objetivo. Considerando $P_1 = (x_1, y_1)$ o ponto do estado atual da busca e $P_2 = (x_2, y_2)$ o ponto objetivo, sua equação é dada por:

$$h_{euclidiana}(P_1, P_2) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{3.1}$$

Essa heurística leva em consideração o fato de que o menor caminho entre dois pontos é a linha reta que os une. No entanto, esse é um caso ideal e não necessariamente o real, já que, na prática, é mais provável que seja necessário percorrer outros caminhos ligados a outros pontos que não seguem a linha reta imaginária. Sendo assim, toma-se a distância Euclidiana como valor estimado, e não como valor real. Embora não seja o valor real do custo, essa heurística permite que sejam expandidos muito menos estados que nos métodos não-informados, uma vez que ela permite priorizar para expansão os nós cujos estados estão mais próximos do destino, sem aumentar o custo assintótico da busca.

Outra função heurística importante para problemas de busca de caminhos é a distância de Manhattan. Nesse caso, também são considerados o estado atual da busca e o ponto de destino, mas a heurística é obtida diretamente das coordenadas dos pontos no espaço euclidiano. Considerando um espaço 2D, onde os pontos são posicionados com coordenadas X e Y, e considerando o estado atual $P_1 = (x_1, y_1)$ e o ponto de destino $P_2 = (x_2, y_2)$, a distância de Manhattan desses dois pontos é dada por:

$$h_{manhattan}(P_1, P_2) = |x_1 - x_2| + |y_1 - y_2| \tag{3.2}$$

Essa heurística é interessante quando o problema trabalha com um padrão de coordenadas, em que elas são discretizadas, como, por exemplo, numa matriz. O cálculo é simples e tão eficiente quanto a distância Euclidiana nesse caso (Russell, 2010).

Tanto a distância de Manhattan quanto a distância Euclidiana são ditas heurísticas otimizadas pois são admissíveis e consistentes. Admissíveis pois elas nunca superestimam o custo real para alcançar o objetivo, ou seja, elas sempre estimam um custo menor do

que o custo real. No caso da distância Euclidiana, tem-se que a menor distância possível entre dois pontos é uma linha reta, portanto essa heurística é naturalmente admissível. Já a distância de Manhattan é admissível aqui pois há a restrição de movimentos nos pólos coordenados norte, sul, leste e oeste, não havendo movimentos nas diagonais.

Essas heurísticas também são consistentes (ou monotônicas) pois, para cada nó n e cada sucessor n' gerado a partir de n com uma ação a, o custo estimado $h(n)$ para alcançar o objetivo a partir de n não é maior que o custo de ir de n para n' somado do custo estimado $h(n')$:

$$h(n) \leq c(n, a, n') + h(n') \tag{3.3}$$

Essa definição é a generalização da desigualdade triangular, que diz que cada lado de um triangulo não pode ser maior que a soma dos outros dois lados. Aqui, os vértices desse triangulo são os nós n, n', e o objetivo.

3.3.1 Busca gulosa

A busca gulosa (ou busca heurística) gera e expande estados considerando somente a função de heurística. Ela seleciona na lista de abertos o nó que possui o menor valor de heurística, independentemente do estado e da altura dele na árvore de busca. Sendo assim, sua função de avaliação é definida simplesmente como segue:

$$f_{gulosa}(n) = h(n) \tag{3.4}$$

É importante observar que na busca gulosa a consistência da heurística não faz sentido ser considerada pois, por definição, essa busca não leva em consideração a função de custo $g(n)$.

3.3.2 Busca A estrela

A busca A estrela (ou A*) tem sua função de avaliação definida como a soma do custo com a heurística:

$$f_{A*}(n) = g(n) + h(n) \tag{3.5}$$

Assim como na busca gulosa, a A* seleciona para expansão na lista de nós abertos aquele com menor valor de avaliação. É o método mais usado em jogos que utilizam busca de caminhos com custo nas arestas, e considerado o mais eficiente.

No caso do jogo aqui proposto em que o custo é uniforme, a função de avaliação da busca A* sempre apresenta um valor constante durante a busca quando gera estados que se aproximam do jogador, pois dado um nó n_1 com valor de heurística h_1 e custo g_1, ao andar um passo para um nó adjacente n_2 se aproximando do jogador, o valor da heurística h_2 é h_1 subtraído de um, mas o custo será incrementado, ficando $g_2 = g_1 + 1$. Ou seja, com as constantes se anulando, a função de avaliação de n_2 será igual ao de n_1, como mostra a equação 3.6.

$$f_{A*}(n_2) = g(n_2) + h(n_2) = g(n_1) + 1 + h(n_1) - 1 = g(n_1) + h(n_1) = f_{A*}(n_1) \tag{3.6}$$

Ou seja, no passo de escolha de um nó para expansão, deve-se definir o comportamento do algoritmo para casos de empate. Na implementação do jogo, escolheu-se, arbitrariamente, retirar para expansão aquele nó que está a menos tempo na lista de abertos. Já nos casos em que a A* gera um nó que se afasta do jogador, a heurística e o custo são ambos incrementados em um, fazendo o nó ter um valor de avaliação maior que o do pai. Assim esse nó só será selecionado para expansão caso não haja caminho através dos outros nós de menor valor.

Como dito anteriormente, a heurística aqui usada é otimizada, e isso implica diretamente que A* é uma busca ótima, pois sua definição obedece a definição de consistência (equação 3.3), além, claro, da heurística ser admissível.

4 O jogo

Nesse capítulo é descrita a implementação do jogo e das técnicas de busca de caminho.

4.1 Modelagem de classes

A Figura 4.1 mostra o diagrama de classes projetado para o jogo. Como representação principal do mapa tem-se a classe *Board*, que armazena o grafo (*graph*), as posições dos inimigos (*enemies*) e do personagem do jogador (*player*), a matriz (*grid*) para desenho do mapa, e o estado do jogo: vitória do jogador (*PLAYERWIN*), perda do jogador (*GAME-OVER*), jogo em aberto (*RESUME*), e construção de mapa (*BUILDING_BOARD*). Essa classe também é responsável por disparar os algoritmos de busca para cada inimigo de acordo com os movimentos do usuário, e também fornece métodos para auxiliar a edição do mapa pelo usuário.

A matriz (*grid*) é preenchida com valores pré-definidos de barreira (*BARRIER*), espaço vazio (*EMPTY*) e posição do objetivo (*GOAL*). A partir dessa matriz o grafo com lista de adjacências é montado, como é mostrado na seção 4.3.

Uma instância de *Board* mantém uma lista de inimigos preenchida dinamicamente em tempo de execução. Cada inimigo contém uma instância da classe de busca (*Search*), que por sua vez mantém armazenada qual é o seu ponto de origem da busca (*start*), o ponto objetivo da busca (*end*), qual tipo de busca deve ser executada (*searchType*) e o grafo (*graph*) sob o qual a busca é executada. O ponto de origem é referência direta ao ponto do inimigo, ou seja, o movimento do inimigo é espelhado nesse ponto. O mesmo ocorre com o ponto de destino: sendo o objetivo do inimigo chegar na posição do jogador, o movimento do usuário atualiza diretamente o ponto de destino. A implementação dos pontos dessa forma facilita a manutenção do projeto, permitindo o uso das buscas de diversas formas: o ponto objetivo pode ser outro ponto conveniente no mapa que não seja o jogador, como, por exemplo, o ponto de *GOAL* ou algum item que futuramente pode ser implementado. Adicionalmente, uma instância de *Search* mantém alimentada uma

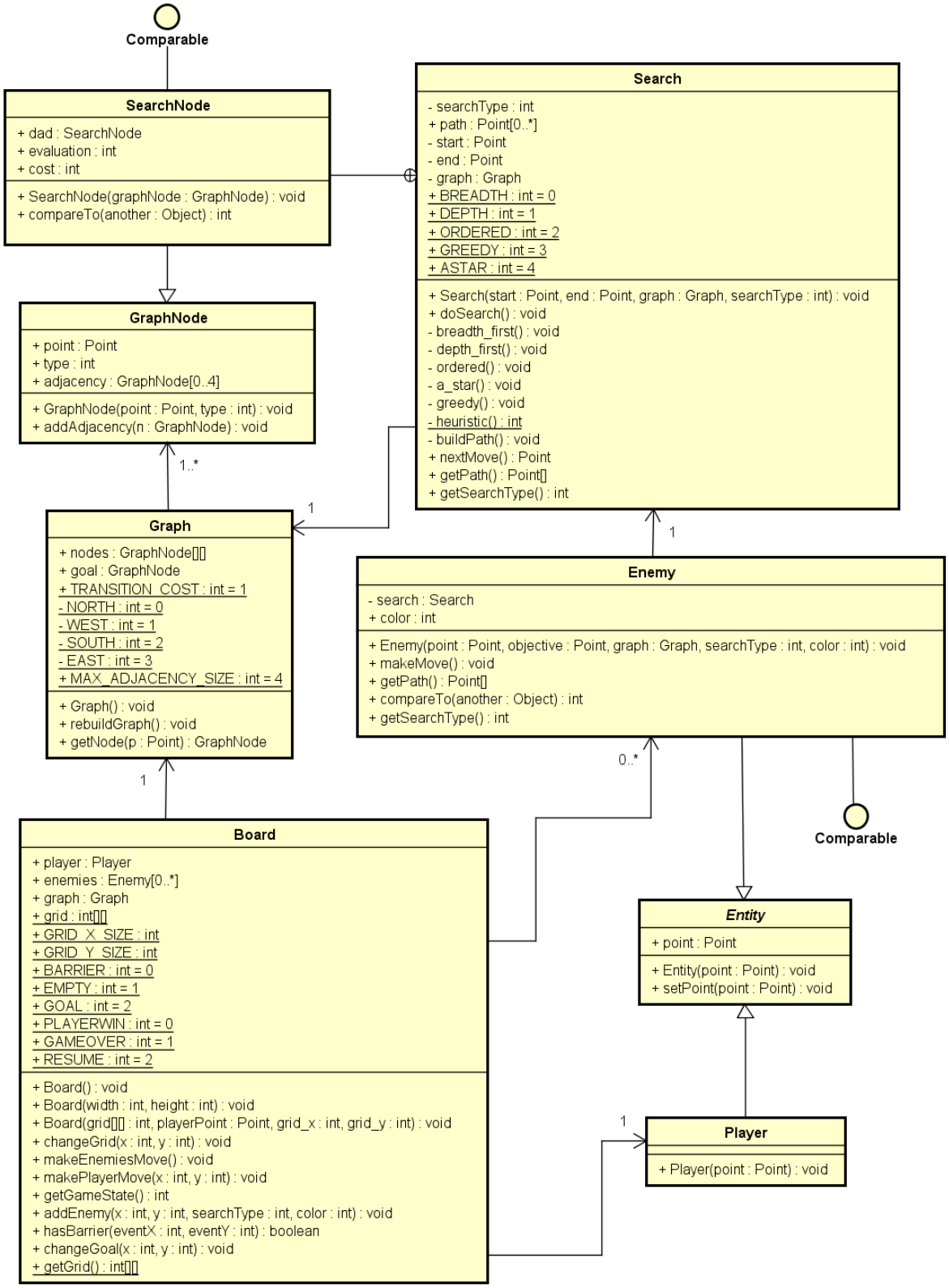


Figura 4.1: Diagrama de classes

lista de pontos (*path*) que indica qual foi o caminho que a busca encontrou no grafo.

As classes *Enemy* e *Player* foram implementadas herdando da classe abstrata *Entity*, permitindo a generalização desses objetos para uso das buscas.

O grafo é representado através da classe *Graph* com lista de adjacências, e adicionalmente mantém numa instância separada o ponto objetivo do jogador (*GOAL*). Como representação de um nó no grafo, tem-se a classe *GraphNode*, que indica o ponto do nó, o tipo (espaço vazio comum ou objetivo do jogador), e sua respectiva lista de adjacências. Essa classe sozinha não é suficiente para executar as buscas, uma vez que não armazena as informações de custo e heurística necessárias em alguns métodos. Para isso, existe a classe anônima *SearchNode* que pertence à *Search* e herda de *GraphNode*. Essa classe armazena o custo do nó (*cost*), o valor de avaliação dele (*evaluation*), e quem é o nó pai que o gerou (*dad*).

4.2 Estrutura do código

As classes no projeto foram divididas em três pacotes: de estrutura de dados (*br.edu.ufjf.core*), de inteligência artificial (*br.edu.ufjf.ai*) e de execução principal do jogo (*br.edu.ufjf.main*), o que inclui atividades e visões do Android. No pacote de *core* estão as classes *Board*, *Entity*, *Enemy*, *Player*, *Graph*, e *GraphNode*, cujos funcionamentos são detalhados na seção 4.1. No pacote *ai*, está a classe *Search*, responsável pela execução dos algoritmos de IA. Por fim, no pacote *main*, estão as classes listadas a seguir:

- *Game*: atividade que controla e exibe a tela em que o usuário pode jogar;
- *GameView*: é uma classe que estende *View*, e contém as funções de desenho da tela para jogar, e também trata os eventos de *touch* na tela;
- *GameEditor*: atividade que controla e exibe a tela para edição do mapa;
- *GameEditorView*: essa *View* possui comportamento semelhante a *GameView*, no entanto, suas funções são voltadas para a edição do mapa, e não para jogar/
- *GameInstanceManager*: essa classe fornece métodos estáticos que permitem salvar e carregar um mapa de arquivo, além de permitir a passagem das instâncias de *Board*

entre as atividades do Android;

- *Menu*: atividade que exibe o menu inicial do jogo;
- *NewMap*: atividade que exibe as opções para ajuste de largura e altura para criar um novo mapa em branco;
- *Splash*: tela inicial do jogo, que é mostrada por três segundos antes de exibir a tela de menu.

4.3 Formação do grafo

A construção ou atualização do grafo quando há alguma alteração no mapa é realizada pela função *rebuildGraph()* da classe *Graph*. Para facilitar a consulta de nós no grafo, considerando que o mapa é representado por uma matriz, os nós são armazenados também numa matriz denominada *nodes*, onde dada uma posição x e y do mapa, o nó correspondente é recuperado de *nodes* usando as mesmas coordenadas x e y. Essa matriz armazena objetos do tipo *GraphNode* (ou nó de grafo).

Primeiramente, a função *rebuildGraph()* percorre o *grid* de *Board*. Cada posição na matriz *grid* gera um nó de grafo na matriz *nodes*, com suas respectivas coordenadas e tipo (barreira, espaço livre ou objetivo do jogador). Após o preenchimento da matriz de nós, a lista de adjacências de cada um é preenchida, completando assim a representação do grafo. Os algoritmos de busca usam as listas de adjacências para saber quais são os próximos estados possíveis a partir do nó de um dado estado. Em outras palavras, essas listas indicam quais são os movimentos possíveis a partir de uma dada posição. São possíveis no máximo quatro movimentos: norte, sul, leste e oeste. Em termos de coordenadas x e y no mapa, esse movimentos possíveis são $(x-1, y)$, $(x+1, y)$, $(x, y-1)$, $(x, y+1)$, como ilustra a Figura 4.3b.

O código da função de formação do grafo é listado a seguir.

```
1 public void rebuildGraph(){
2   nodes = new GraphNode[Board.GRID_X_SIZE][Board.GRID_Y_SIZE];
3   for(int i = 0; i < Board.GRID_X_SIZE; i++){
4     for(int j = 0; j< Board.GRID_Y_SIZE; j++){
5       nodes[i][j]=new GraphNode(new Point(i,j),Board.grid[i][j]);
```

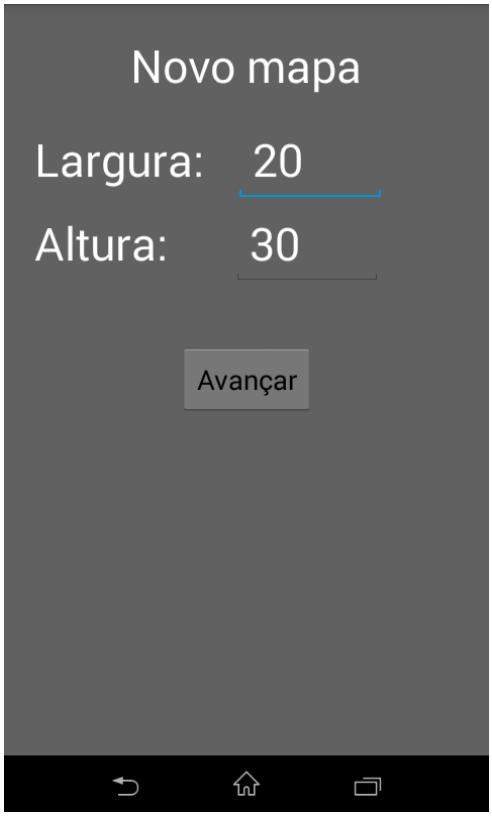
```
6         if(Board.grid[i][j]==Board.GOAL)
7           goal = nodes[i][j];
8       }
9     }
10    for(int i = 0; i < Board.GRID_X_SIZE; i++){
11      for(int j = 0; j< Board.GRID_Y_SIZE; j++){
12        if(nodes[i][j].type!=Board.BARRIER) {
13          if (i < Board.GRID_X_SIZE - 1) {
14            if (Board.grid[i + 1][j] != Board.BARRIER) {
15              nodes[i][j].addAdjacency(nodes[i + 1][j]);
16            }
17          }
18          if (j < Board.GRID_Y_SIZE - 1) {
19            if (Board.grid[i][j + 1] != Board.BARRIER) {
20              nodes[i][j].addAdjacency(nodes[i][j + 1]);
21            }
22          }
23          if (i > 0) {
24            if (Board.grid[i - 1][j] != Board.BARRIER) {
25              nodes[i][j].addAdjacency(nodes[i - 1][j]);
26            }
27          }
28          if (j > 0) {
29            if (Board.grid[i][j - 1] != Board.BARRIER) {
30              nodes[i][j].addAdjacency(nodes[i][j - 1]);
31            }
32          }
33        }
34      }
35    }
36  }
```

4.4 Telas

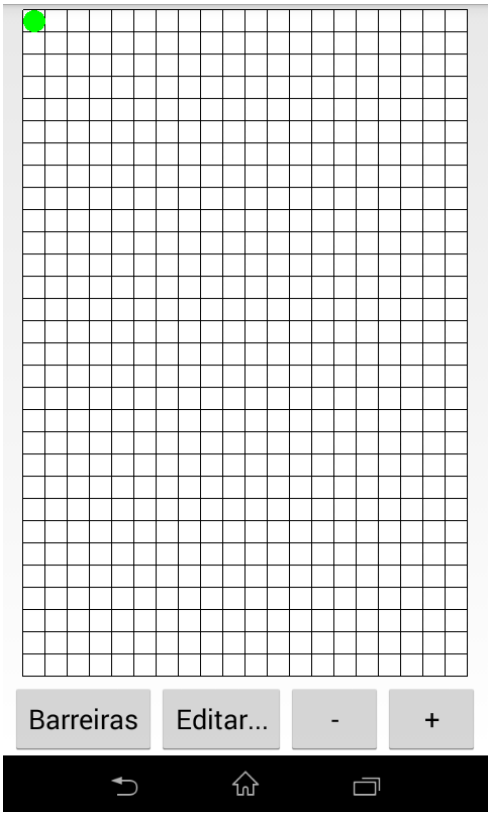
A seguir são mostradas as telas do jogo. A Figura 4.2a é a tela inicial do aplicativo. Depois de três segundos, é mostrada a tela de menu (Figura 4.2b). Nessa tela, é possível escolher criar um novo mapa, editar um mapa já salvo anteriormente, excluir um mapa, ou jogar algum dos mapas salvos em arquivo. Ao escolher criar um novo mapa, o usuário pode definir a largura e altura dele na tela da Figura 4.2c. A largura e a altura possuem um valor mínimo de três unidades, e valor máximo cem cada uma, para que o mapa não fique denso a ponto de não ser possível identificar os elementos na tela com clareza. Na tela de edição de mapa (Figura 4.2d, o usuário pode aproximar ou afastar o mapa com *zoom*. As barreiras são editáveis clicando no botão "Barreiras", e a edição delas é concluída no

(a)

(b)

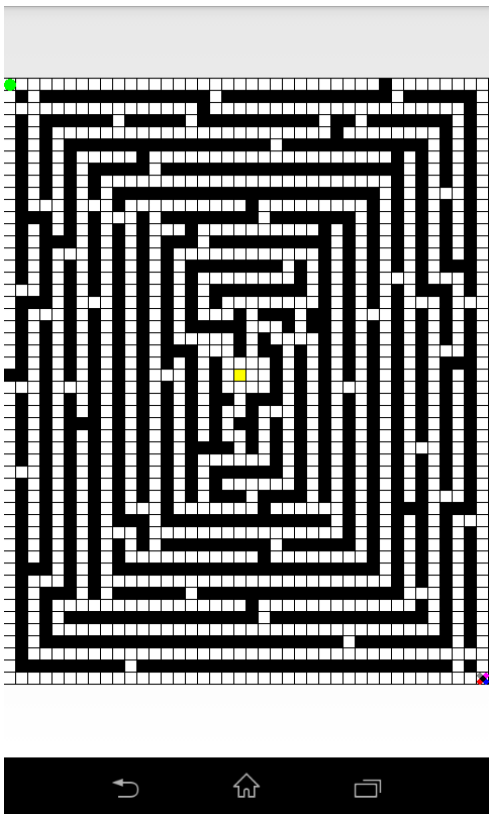


(c)

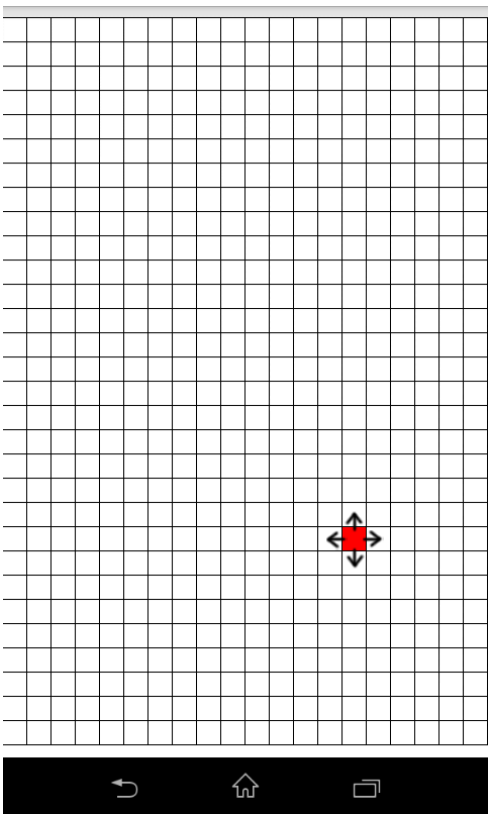


(d)

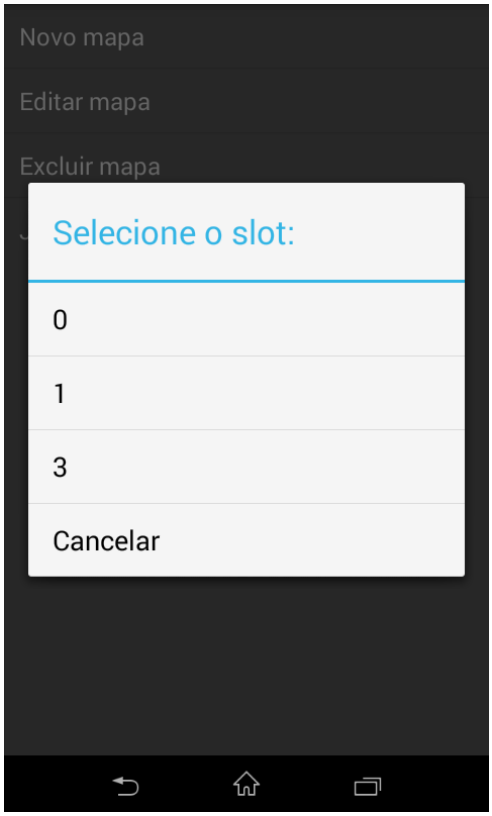
Figura 4.2: (a) Tela inicial do jogo. (b) Tela de menu. (c) Tela para criar novo mapa em branco. (d) Tela de edição de mapas.



(a)



(b)



(c)

(d)

Figura 4.3: (a) Tela para jogar um mapa salvo. (b) Movimentos possíveis a partir de um nó. (c) Tela de exemplo de seleção de um mapa salvo em arquivo. (d) Opções mostradas ao jogador quando o jogo termina.

mesmo botão. O botão "Editar" dá ao usário as opções de inserir um novo inimigo, excluir os inimigos, alterar a posição inicial do jogador, alterar a posição do objetivo, limpar o mapa, e concluir a edição. Ao terminar a edição, o usuário pode escolher um *slot* para salvar o mapa. São ao todo dez *slots* disponíveis, numerados de zero a nove. A tela para jogar é mostrada na Figura 4.3a, onde para mover seu personagem, o jogador deve clicar nas extremidades norte (para cima), sul (para baixo), leste (para direita) e oeste (para esquerda) da tela. A Figura 4.3b ilustra os movimentos possíveis para um inimigo ou para o jogador. A tela da Figura 4.3c dá um exemplo de *slots* como mapas salvos para serem carregados, e a tela da Figura 4.3d mostra as opções que o jogo apresenta quando o jogador perde ou ganha.

4.5 Armazenamento de mapas

O armazenamento de mapas em arquivo é feita através de serialização, onde um objeto inteiro em memória em salvo diretamente num arquivo (Mednieks et al, 2012). Foram reservados 10 arquivos para salvar os mapas, um para cada *slot*. A classe *GameInstanceManager* manipula esses arquivos. Para armazenar os dados dos mapas em arquivo, foram criadas classes anônimas: *Instance* e *EnemyInstance*. Essas classes convertem as listas em vetores de tamanho fixo, uma vez que a serialização no Android não permite o armazenamento de listas. Sendo assim, os arquivo salvos são do tipo *Instance*, e não *Board*. Ao ler um desses arquivos, as informações são mapeadas de volta para uma instância de *Board*.

4.6 Implementação das buscas de caminho

A execução das buscas de caminho é responsabilidade da classe *Search*. As buscas implementadas nessa classe são a busca em largura (*breadth_first*), em profundidade (*depth_first*), ordenada (*uniform*), gulosa (*greedy*), e A* (*a_star*). O funcionamento geral dessas buscas é tratado no capítulo 3. A seguir é listado o código da busca em largura:

```
1 private void breadth_first(){
2   List<GraphNode> adjacency;
```

```
3   List<SearchNode> openedList = new ArrayList<SearchNode>();
4   List<SearchNode> closedList = new ArrayList<SearchNode>();
5   SearchNode dadNode = new SearchNode(graph.getNode(start));
6   SearchNode sonNode;
7   openedList.add(dadNode);
8   while(!openedList.isEmpty()
9         && (dadNode.point.x!=end.x || dadNode.point.y!=end.y)){
10    openedList.remove(0);
11    closedList.add(dadNode);
12    adjacency = dadNode.adjacency;
13    for(GraphNode adjNode : adjacency) {
14      sonNode = new SearchNode(adjNode);
15      sonNode.dad = dadNode;
16      boolean contains = false;
17      for (SearchNode n : openedList) {
18        if (sonNode.compareTo(n) == 0) {
19          contains = true;
20        }
21      }
22      for (SearchNode n : closedList) {
23        if (sonNode.compareTo(n) == 0) {
24          contains = true;
25        }
26      }
27      if (!contains) {
28        openedList.add(sonNode);
29      }
30    }
31    dadNode = openedList.get(0);
32  }
33  buildPath(dadNode);
34 }
```

Nesse código, a lista de estados abertos é o objeto *openedList*, e a lista de fechados é o *closedList*. Enquanto a lista de abertos não estiver vazia (caso contrário a busca não teria encontrado um caminho) ou o estado atual do laço não corresponder a posição do ponto objetivo (ponto *end*), a função continua expandindo estados. Aqui a lista de abertos tem o comportamento de uma fila: na linha 28 o estado recém gerado é inserido no final da lista, e na linha 31 o próximo estado a ser expandido é o primeiro da lista. A medida que os estados vão sendo expandidos, eles vão sendo inseridos na lista de fechados. Vale notar que entre as linhas 16 e 26 é verificado se o estado recém criado já não existe na lista de fechados ou de abertos, a fim de evitar *loops* na busca. Nesse código o estado que está sendo expandido é representado pela variável *dadNode* e o nó recém criado é a variável *sonNode*. No final da busca, a função *buildPath* é chamada, recebendo como parâmetro o

último nó selecionado para expansão. Se um caminho foi encontrado, esse nó corresponde a posição do nó objetivo. A partir dele, *buildPath* irá recuperar o caminho, subindo na árvore de busca utilizando o nó pai da classe anônima *SearchNode*, portanto a construção é feita do fim ao começo. A seguir é listado o código dessa função.

```
1  private void buildPath(SearchNode endOfPath){
2    path.clear();
3    if((endOfPath.point.x==end.x && endOfPath.point.y==end.y)) {
4      SearchNode answer = endOfPath;
5      while(answer.dad.point.x != start.x
6            || answer.dad.point.y != start.y) {
7        path.addFirst(answer.point);
8        answer = answer.dad;
9      }
10     path.addFirst(answer.point);
11   }
12 }
```

Dado um nó do caminho, ele é inserido em *path*, e então o nó pai dele é selecionado, e assim por diante. O laço *while* só é interrompido quando o nó pai do nó atual é igual ao ponto de partida, indicando que o caminho foi completamente recuperado.

As outras funções de busca possuem implementação semelhante à da busca em largura listada acima, diferindo apenas na forma como os estados são selecionados para expansão. Na busca em profundidade, a diferença está na linha 31. Em vez de selecionar para expansão o primeiro nó da lista de abertos, é selecionado o último, assim a lista tem o comportamento de uma pilha.

No caso da busca ordenada, os nós criados são inseridos na lista de abertos junto com os seus valores de custo e de avaliação. A inserção é feita da seguinte forma:

```
1 sonNode.cost = dadNode.cost + Graph.TRANSITION_COST;
2 sonNode.evaluation = sonNode.cost;
3 openedList.add(sonNode);
```

Como o custo é uniforme, temos que o custo da transição de estado (*TRANSITION_COST*) é igual a um. O valor de avaliação do nó recém criado é igual ao custo do nó pai somado do custo da transição. A seleção de nós para expansão é feita no trecho de código que segue, onde é selecionado o nó de menor valor de avaliação:

```
1 SearchNode next = openedList.get(0);
```

```
2 for (SearchNode n : openedList) {
3      if (n.evaluation <= next.evaluation)
4           next = n;
5 }
6 adjacency = next.adjacency;
7 dadNode = next;
```

A busca gulosa e a busca A* possuem a seleção de nó para expansão igual ao implementado na busca ordenada. O que difere nelas é a função de avaliação. Na busca gulosa, o valor de avaliação é apenas o valor da heurística, conforme a equação 3.4:

```
1 sonNode.evaluation = heuristic(sonNode.point, end);
2 openedList.add(sonNode)
```

A função *heuristic* é a implementação direta da distância de Manhattan definida na equação 3.2. O código da função é o seguinte:

```
1 private static int heuristic(Point p1, Point p2){
2      return Math.abs(p2.x - p1.x) + Math.abs(p2.y - p1.y);
3 }
```

Por último, a busca A* calcula o valor de avaliação do nó da seguinte forma, conforme definido na equação 3.5:

```
1 sonNode.cost = dadNode.cost + Graph.TRANSITION_COST;
2 sonNode.evaluation = heuristic(sonNode.point, end)+sonNode.cost;
3 openedList.add(sonNode);
```

5 Experimentos e resultados

5.1 Os mapas de teste

Para testar o jogo e os algoritmos de busca, foram criados três mapas em arquivo. Nesses mapas, os inimigos foram dispostos na mesma posição, numa extremidade do mapa, e o jogador na extremidade oposta, representado por um ponto verde. Na casa do canto inferior direito, estão posicionados os inimigos dotados com as buscas em largura, em profundidade, ordenada, gulosa e A*, representados pelos pontos vermelho (canto inferior esquerdo), azul (canto inferior direito), rosa (superior direito), cinza (superior esquerdo) e preto (centro da casa), respectivamente, como indica a Figura 5.1. Os mapas são mostrados nas Figuras 5.2, 5.3 e 5.4.

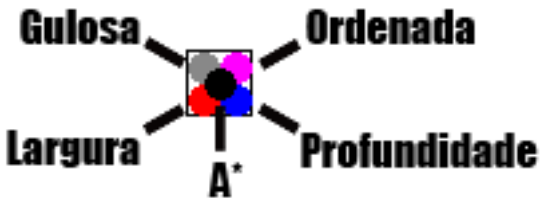
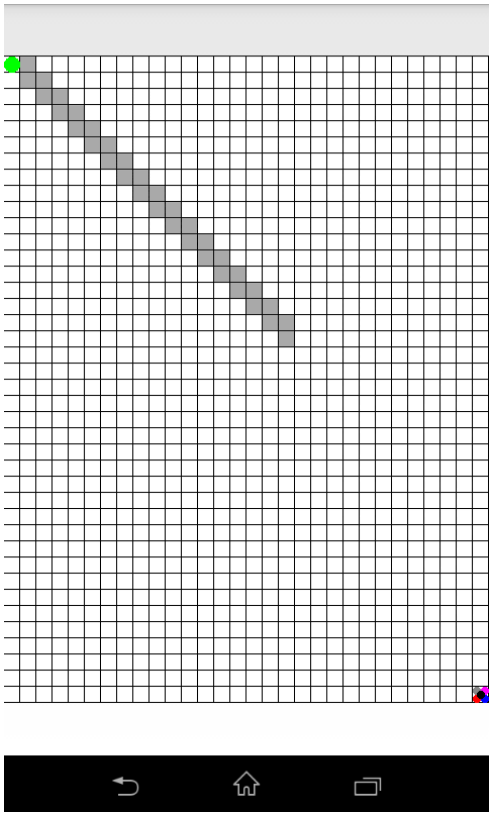


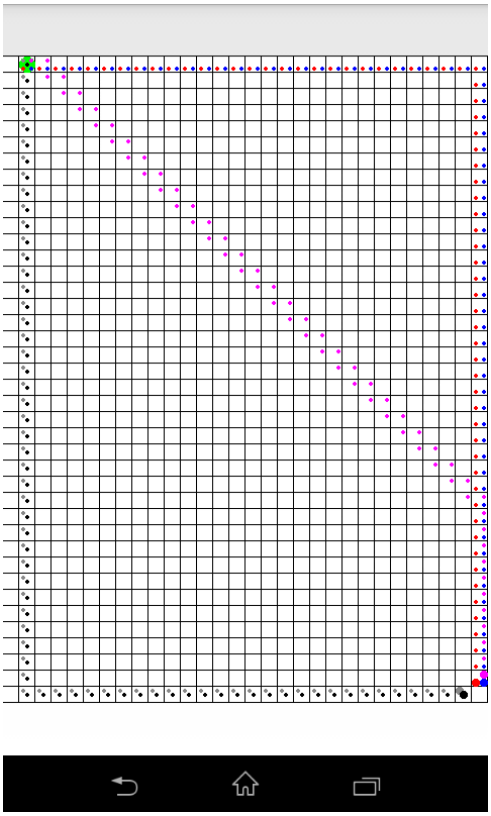
Figura 5.1: Disposição dos inimigos no início do jogo

O primeiro mapa (Figura 5.2) tem 30 unidades de largura e 40 de altura, e não possui barreiras, portanto possui 1200 nós no grafo. Ou seja, um algoritmo de busca nesse mapa pode gerar até 1200 estados. Esse mapa foi criado com a finalidade de mostrar o funcionamento geral de cada algoritmo, ao considerar um mapa aberto, sem barreiras. O percurso que o jogador segue é indicado pelas casas em cinza na Figura 5.2a. O primeiro movimento do jogador é indicado na Figura 5.2b e o último movimento é mostrado na Figura 5.2c. Cada movimento do jogador implica numa nova execução de cada algoritmo de busca para encontrar um novo caminho para cada inimigo. É interessante notar que nesse mapa existem muitos caminhos ótimos possíveis, tanto que alguns métodos ótimos encontraram caminhos diferentes.

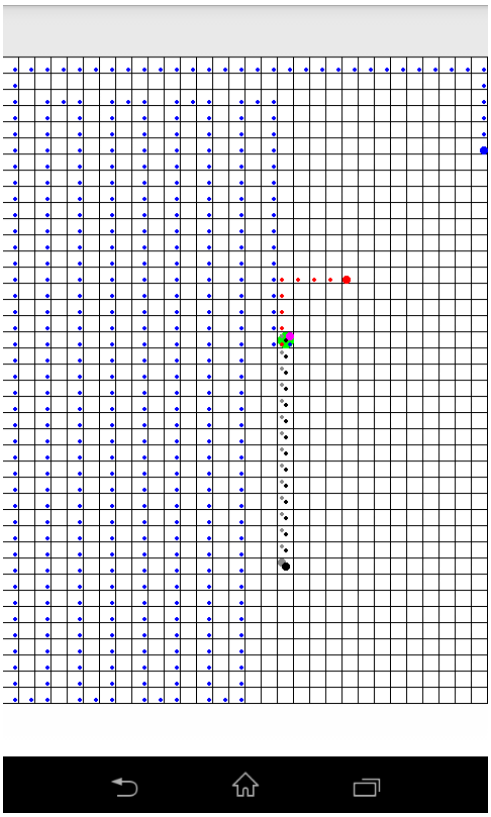
Pode-se observar que as buscas gulosa e A* calcularam os mesmos caminhos. A



(a)

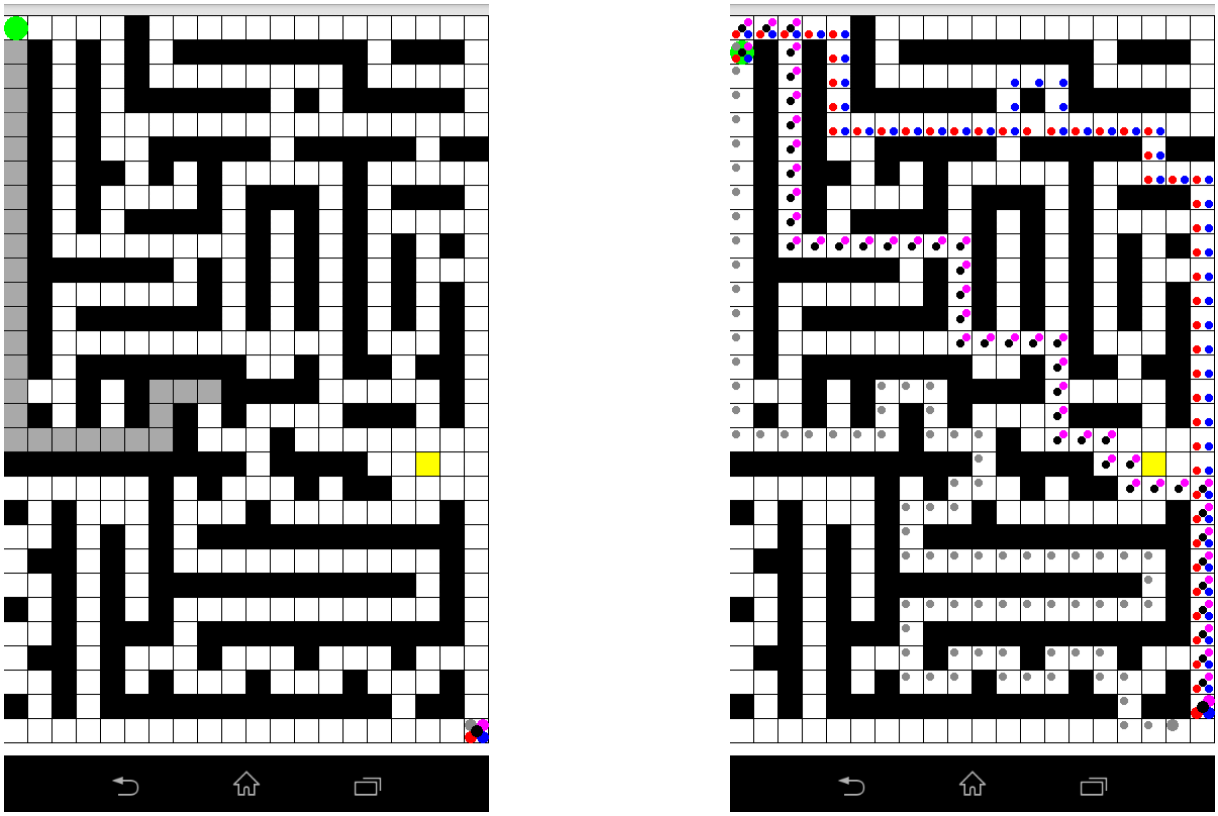


(b)

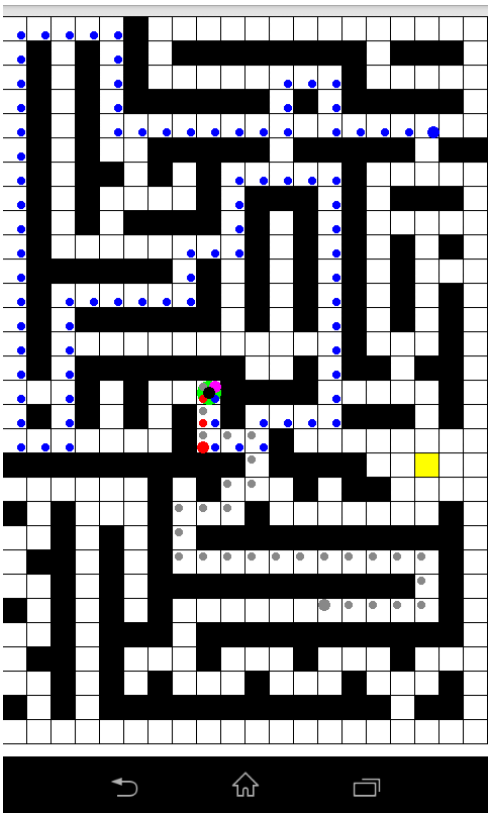


(c)

Figura 5.2: Mapa 1 com 30 unidades de largura e 40 de altura. Os pontos menores indicam o caminho calculado por cada um dos respectivos inimigos. (a) O jogador começa no canto superior esquerdo e percorre os quadrados em cinza, e os inimigos no canto inferior direito. (b) mostra o primeiro movimento do jogador. (c) mostra o último movimento.

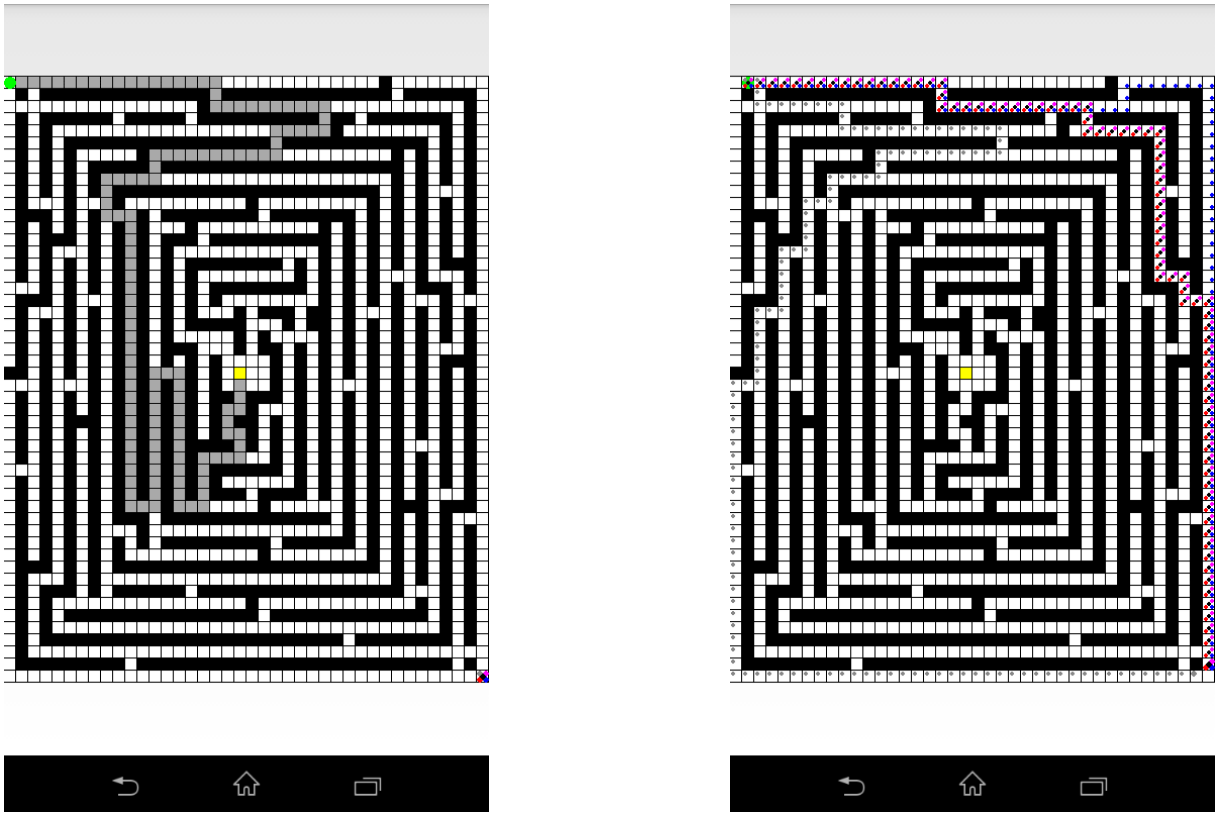


(a) (b)

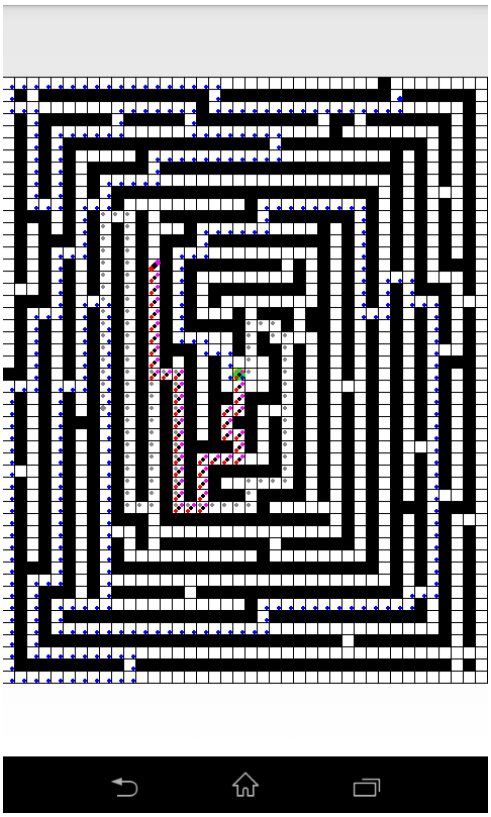


(c)

Figura 5.3: Mapa 2 com 20 unidades de largura e 30 de altura. Os pontos menores indicam o caminho calculado por cada um dos respectivos inimigos. (a) O jogador começa no canto superior esquerdo e percorre os quadrados em cinza, e os inimigos no canto inferior direito. (b) mostra o primeiro movimento do jogador. (c) mostra o último movimento.



(a) (b)



(c)

Figura 5.4: Mapa 3 com 40 unidades de largura e 50 de altura. Os pontos menores indicam o caminho calculado por cada um dos respectivos inimigos. (a) O jogador começa no canto superior esquerdo e percorre os quadrados em cinza, e os inimigos no canto inferior direito. (b) mostra o primeiro movimento do jogador. (c) mostra o último movimento.

busca em profundidade deu muitas voltas até encontrar uma solução. A busca em largura e a busca ordenada, apesar de acharem um caminho com o menor custo possível assim como a busca a A* faz, calcularam caminhos diferentes. A busca ordenada tendeu a seguir um caminho diagonal, dada a sua forma de selecionar para expansão um estado que apresenta o menor custo do caminho gerado até então, e a busca em largura seguiu caminhos com menos curvas e mais retas, dada a sua forma de seleção de nó que segue a ordem de precedência das ações, que é, como dito na seção 3.2, norte, oeste, sul e leste. Já as buscas A* e gulosa também apresentaram caminhos com poucas curvas, mas seguiram outra direção devido ao uso da heurística para expansão de nós.

O mapa 2 é o mostrado na Figura 5.3, e possui 20 unidades de largura 30 de altura. Esse mapa foi gerado de forma aleatória pelo usuário e permite mostrar o comportamento dos algoritmos num mapa com barreiras. O caminho percorrido pelo jogador, indicado pelas casas em cinza, tenta chegar no ponto objetivo, mas alguns inimigos chegam no jogador antes disso. O movimento inicial é indicado na Figura 5.3b e a jogada final com situação de perda do jogador é indicada na Figura 5.3c;

Pode-ser observar que, assim como no mapa 1, a busca em profundidade no mapa 2 encontrou longos caminhos, dando muitas voltas. A busca A* e a ordenada, nesse caso, seguiram sempre o mesmo caminho, e a gulosa seguiu por caminhos diferentes.

Por fim, o mapa 3, mostrado na Figura 5.4, foi montado com dimensões maiores: 40 unidades de largura e 50 de altura. Esse mapa também foi gerado de maneira aleatória pelo usuário. No caso desse mapa, o jogador conseguiu chegar no objetivo, seguindo o percurso indicado pelas casas em cinza. O primeiro movimento é mostrado na Figura 5.4b, e o último na Figura 5.4c.

5.2 Resultados

Os números de estados gerados, expandidos, os tamanhos dos caminhos calculados e o tempo de execução foram mostrados no console do Android Studio, com a aplicação rodando em modo *debug* num dispositivo conectado via cabo USB. O dispositivo foi um *smartphone* Sony E1 Dual, com 512 MB de memória RAM e processador *Dual-Core* 1,2 GHz.

Tabela 5.1: Dados de desempenho dos métodos de busca para o mapa 1

	Largura				**Profundidade**				**Ordenada**				**Gulosa**				**A***			
Movimento	Gerados	Expandidos	Tamanho	Tempo (ms)	Gerados	Expandidos	Tamanho	Tempo (ms)	Gerados	Expandidos	Tamanho	Tempo (ms)	Gerados	Expandidos	Tamanho	Tempo (ms)	Gerados	Expandidos	Tamanho	Tempo (ms)
1	1199	1198	66	2565	133	67	66	51	1199	1198	66	2113	172	67	66	31	172	67	66	29
2	1198	1196	64	2102	1199	1134	66	3094	1198	1195	64	1921	168	65	64	16	168	65	64	20
3	1195	1192	62	2017	1199	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	1199	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	1199	1064	142	2940	1160	1150	50	1944	137	51	50	17	137	51	50	12
10	1150	1140	48	1923	1199	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	1199	1000	208	2644	1116	1102	42	1890	119	43	42	20	119	43	42	19
14	1101	1085	40	1795	1199	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	1199	994	216	2793	1050	1028	34	1828	101	35	34	6	101	35	34	5
18	947	915	32	1269	1199	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	1199	938	274	2230	911	874	26	1158	88	29	28	16	88	29	28	17
22	649	610	24	638	1199	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	1199	924	290	2343	646	596	18	699	76	25	24	3	76	25	24	3
26	331	300	16	160	1199	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	1199	876	340	2024	294	251	10	105	64	21	20	2	64	21	20	3
30	284	251	12	102	1199	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	1199	854	364	1998	39	23	2	2	52	17	16	2	52	17	16	2
34	199	165	8	61	1199	854	364	2202	10	3	0	0	47	15	14	1	47	15	14	12
$\sum$	27190	26431	1154	40631	28304	20681	7956	45826	28526	27734	1122	43430	3545	1268	1234	360	3545	1268	1234	314

Os dados de execução para cada algoritmo no mapa 1 são mostrados na Tabela 5.1. Cada linha da tabela indica um movimento do jogador, em sequência, conforme o caminho indicado na Figura 5.2a. A última linha apresenta um somatório dos valores de cada coluna. Os tempos são dados em milissegundos (ms).

É possível observar que, no geral, os métodos não informados (largura, profundidade e ordenada) geraram e expandiram muitos mais estados que os métodos informados (gulosa e A*). As diferenças entre os tempos de execução também foram grandes: as buscas não informadas demoraram muito mais. A busca em profundidade merece um destaque por sua variação brusca entre os movimentos do jogador: em alguns, essa busca gerou todos os estados possíveis, em outros casos gerou menos estados que a busca em largura e demorou menos tempo, mas nunca achou um caminho menor, e os tamanhos dos caminhos também variam bastante. Além disso, os somatórios de seus valores, no geral, foram bem altos. Essas variações podem ser observadas em tempo de execução do jogo: o caminho indicado pela busca em profundidade varia o tempo todo. Isso indica o quão ruim a busca em profundidade é para o tipo de jogo aqui proposto.

É importante observar que, apesar de ter sido dito que as bucas A*, gulosa, ordenada e em largura sempre encontraram o menor caminho nesse mapa, a tabela apresenta tamanhos diferentes de caminhos para um mesmo movimento entre essas buscas. Isso ocorre porque, como essas buscas seguiram caminhos diferentes, em certos momentos um movimento do jogador favoreceu um inimigo ao se aproximar dele e prejudicou outro ao se afastar, o que não muda o fato deses algoritmos encontrarem o menor caminho considerando o estado do mapa em cada movimento, onde os inimigos estão em posições diferentes.

Os resultados da execução das buscas de caminho no mapa 2 são mostrados na Tabela 5.2. Assim como no mapa 1, a busca em profundidade apresentou muita variação entre cada movimento, apesar dos seus somatórios não terem sido os maiores. A busca em largura, novamente, encontra caminhos com os menores custos, mas trás tempos de execução e número de estados gerados e expandidos relativamente altos. Com a busca ordenada ocorreu o mesmo. A busca gulosa apresentou, no geral, os menores tempos, mas nesse tipo de mapa não calculou o menor caminho, apesar de usar a heurística. Isso ocorre

Tabela 5.2: Dados de desempenho dos métodos de busca para o mapa 2

	Largura				**Profundidade**				**Ordenada**				**Gulosa**				**A***			
Movimento	Gerados	Expandidos	Tamanho	Tempo (ms)	Gerados	Expandidos	Tamanho	Tempo (ms)	Gerados	Expandidos	Tamanho	Tempo (ms)	Gerados	Expandidos	Tamanho	Tempo (ms)	Gerados	Expandidos	Tamanho	Tempo (ms)
1	350	349	48	222	97	82	52	60	351	349	48	232	143	135	80	69	240	219	48	128
2	351	351	48	192	97	82	52	63	351	350	48	110	141	133	78	17	294	276	48	103
3	351	350	46	140	97	82	52	9	350	348	46	118	139	131	76	21	280	261	46	70
4	349	348	44	109	97	82	52	7	349	347	44	109	137	129	74	14	230	212	44	50
5	347	346	42	93	96	82	52	7	346	344	42	103	85	77	72	7	215	197	42	56
6	345	344	40	96	96	82	52	10	345	343	40	138	83	75	70	15	187	172	40	44
7	343	342	38	134	96	82	52	136	342	339	38	214	80	73	68	5	176	162	38	27
8	340	336	36	89	96	82	52	8	341	337	36	108	78	71	66	5	163	148	36	22
9	327	320	34	100	96	82	52	9	328	319	34	103	76	69	64	5	153	138	34	20
10	314	307	32	96	96	82	52	9	316	308	32	123	75	68	62	9	143	126	32	21
11	300	292	30	93	96	82	52	11	300	289	30	81	74	67	60	5	124	110	30	19
12	282	271	28	67	95	82	52	13	287	275	28	96	74	67	58	5	110	97	28	12
13	261	252	26	61	94	82	52	7	260	249	26	83	72	65	56	19	99	85	26	34
14	244	233	24	121	93	82	52	8	247	236	24	65	72	65	54	10	81	68	24	11
15	226	216	22	43	93	82	52	7	224	211	22	36	69	63	52	6	72	59	22	5
16	232	220	22	43	94	82	52	7	234	222	22	47	68	62	50	4	61	46	22	3
17	240	229	22	67	94	82	52	7	209	196	20	39	67	61	48	6	41	30	20	1
18	225	213	20	61	94	82	52	9	198	184	18	48	67	61	46	5	36	27	18	1
19	201	190	18	40	94	82	52	7	166	153	16	33	64	59	44	5	32	24	16	2
20	185	170	16	28	351	339	52	114	140	129	14	24	64	59	42	3	31	23	14	1
21	163	148	14	32	181	163	96	42	126	113	12	23	61	57	40	3	27	21	12	0
22	131	122	12	27	179	161	94	25	97	87	10	9	61	57	38	4	23	18	10	1
23	94	85	10	9	176	159	92	35	68	55	8	14	60	56	36	12	20	15	8	4
24	65	51	8	4	174	157	90	369	37	31	6	2	58	54	34	3	12	9	6	0
25	35	28	6	1	172	155	88	31	22	17	4	1	58	54	32	3	7	5	4	0
26	24	19	4	3	170	153	86	24	10	7	2	0	56	52	30	3	5	3	2	0
27	9	7	2	0	167	151	84	22	4	2	0	0	56	52	28	3	3	1	0	0
$\sum$	6334	6139	692	1971	3381	2996	1670	1056	6048	5840	670	1959	2138	1972	1458	266	2865	2552	670	635

porque, como ela usa apenas a função heurística, como indica a equação 3.4, ela segue um caminho tentando se aproximar do jogador mas sem considerar o custo do caminho, ou seja, ela pode, num dado momento, estar se aproximando do jogador mas seguindo um caminho grande para isso. A busca A* apresentou valores medianos: número de estados expandidos, gerados, e tempo de execução não foram tão altos, e o caminho, como sempre, apresentou o menor custo possível.

O resultados da execução dos métodos de busca no mapa 3 são mostrados na Tabela 5.3. Por ser um mapa grande, o jogador teve que efetuar muitas jogadas (97 no total). Portanto, para uma melhor visualização, a cada 3 movimentos, dois foram suprimidos da tabela. Mesmo assim, os somatórios apresentados na última linha consideram todos os movimentos, até os que foram suprimidos. A omissão desses movimentos não altera a análise geral dos resultados, que apresentaram comportamentos semelhante aos do mapa 2.

Esses resultados mostram que os métodos não informados não apresentaram bom desempenho, o que os torna fortes candidatos para serem descartados da versão final do jogo. Os métodos informados foram melhores, tanto na geração e expansão de estados quanto no tempo de resposta. No entanto a busca gulosa nem sempre apresenta o menor caminho, ao passo que responde ligeiramente mais rápido que a busca A*. Ou seja, o uso desses dois métodos é interessante, podendo a gulosa ser usada quando se deseja uma resposta mais rápida e sem a necessidade de ser a melhor resposta, já que essa busca não é ótima, e a A* usada quando se deseja obter o menor caminho sem prejudicar muito o tempo de execução.

Tabela 5.3: Dados de desempenho dos métodos de busca para o mapa 3

	Largura				**Profundidade**				**Ordenada**				**Gulosa**				**A***			
Movimento	Gerados	Expandidos	Tamanho	Tempo (ms)	Gerados	Expandidos	Tamanho	Tempo (ms)	Gerados	Expandidos	Tamanho	Tempo (ms)	Gerados	Expandidos	Tamanho	Tempo (ms)	Gerados	Expandidos	Tamanho	Tempo (ms)
1	920	911	86	752	101	91	90	57	926	916	86	572	228	207	130	47	229	213	86	48
4	925	915	84	631	130	121	92	11	933	923	84	584	217	199	124	31	247	230	84	39
7	864	852	78	513	158	145	116	18	867	854	78	485	212	195	114	29	147	135	78	14
10	780	765	72	399	152	139	110	16	793	777	72	408	190	173	108	25	138	126	72	13
13	669	654	66	311	154	142	108	18	674	657	66	295	184	166	106	21	133	123	66	13
16	573	554	60	235	153	142	108	16	589	568	60	221	160	145	96	17	122	112	60	11
19	488	474	54	230	153	142	108	50	491	476	54	205	158	144	90	16	91	82	54	6
22	427	414	48	219	157	146	108	21	436	423	48	122	242	224	130	37	85	76	48	5
25	363	349	42	116	157	146	108	17	367	352	42	97	179	162	122	21	79	70	42	5
28	358	346	40	152	158	146	108	17	368	355	40	87	173	158	116	20	125	113	40	11
31	385	371	40	96	159	146	108	22	385	371	40	98	154	140	110	16	129	116	40	11
34	427	410	40	184	158	146	108	19	440	421	40	168	205	189	106	75	131	118	40	12
37	409	393	40	124	158	146	108	17	417	400	40	112	124	111	62	11	65	56	40	3
40	393	379	40	168	158	146	108	17	403	388	40	107	112	100	56	8	68	59	40	4
43	378	361	40	114	159	146	108	17	386	369	40	99	83	73	50	5	68	59	40	4
46	359	343	40	185	159	146	108	18	371	353	40	94	71	62	44	4	67	58	40	4
49	354	339	40	93	563	520	398	255	359	341	40	86	50	43	42	2	105	93	40	7
52	359	341	40	90	556	514	392	199	368	349	40	85	51	43	42	2	136	121	40	13
55	363	345	40	84	549	508	386	218	371	352	40	87	58	48	42	3	140	124	40	13
58	362	347	40	81	594	550	428	229	369	353	40	95	71	60	42	15	135	120	40	31
61	367	356	40	103	541	500	378	193	371	357	40	91	84	73	42	5	138	123	40	13
64	381	371	40	93	588	544	422	229	383	371	40	99	121	109	42	10	126	113	40	12
67	399	387	40	190	534	494	372	219	398	385	40	102	116	103	56	9	135	121	40	12
70	413	401	40	126	580	537	416	213	416	403	40	115	91	79	56	6	124	111	40	22
73	438	421	40	122	527	487	366	189	445	426	40	142	95	84	56	7	120	107	40	10
76	469	454	40	244	574	531	410	214	471	455	40	148	120	109	56	10	113	103	40	9
79	506	485	40	177	521	481	360	175	513	493	40	168	217	201	56	31	123	112	40	11
82	520	493	40	256	567	525	404	205	535	508	40	203	167	154	60	18	148	136	40	20
85	497	472	40	230	514	475	354	239	510	484	40	164	98	87	60	8	186	176	40	24
88	496	473	40	164	560	519	398	220	481	457	40	183	154	143	60	50	176	165	40	20
91	453	431	40	170	507	469	348	180	434	412	40	119	94	84	60	6	193	182	40	25
94	418	399	40	137	554	513	392	220	404	383	40	104	128	117	76	12	161	148	40	17
97	406	386	40	126	777	741	342	526	393	372	40	101	199	180	100	37	166	151	40	18
$\sum$	46406	44835	4520	19152	37663	35050	24326	14073	46809	45138	4520	17079	13550	12249	7534	1760	12496	11347	4520	1327

6 Conclusões

Esse trabalho apresentou um estudo de aplicação de técnicas de busca de caminhos no desenvolvimento de um jogo para a plataforma Android. Como mostram os resultados, as técnicas que usam heurísticas são as mais indicadas, por terem apresentado desempenho melhor em relação às outras buscas, o que já é esperado de acordo com a literatura. No caso da busca gulosa, apesar de usar função heurística, nem sempre calcula o menor caminho, mas apresentou os menores tempos de execução. Já a busca A* sempre calculou o menor caminho, e seus tempos de execução não foram muito maiores que os da busca gulosa.

Quando comparadas às buscas de métodos não informados, principalmente à busca em profundidade, a busca gulosa e a busca A* apresentaram desempenho muito melhor. Um jogo, para obter sucesso, precisa ser fluido, rápido para responder a estímulos externos (ações do jogador), e também deve apresentar algum desafio que o faça ser interessante, sem no entanto ser impossível de ser vencido. Portanto, mesmo que se esteja buscando um método de busca com o melhor desempenho possível, ele não pode ser tão eficaz a ponto de impossibilitar a vitória do usuário. Sendo assim, dependendo da situação, a busca gulosa pode ser mais interessante, por apresentar baixo tempo de resposta e não calcular sempre o melhor caminho, o que daria alguma vantagem ao jogador.

O tipo de jogo proposto aqui exige uma resposta praticamente imediata do algoritmo de busca, dada a dinâmica rápida da alternância entre um movimento do jogador e os movimentos dos inimigos. Não seria interessante que o jogo tivesse que parar para calcular os movimentos, caso contrário toda a dinâmica seria quebrada. Como foi mostrado nos resultados, alguns algoritmos chegaram a responder somente depois de dois segundos, o que, nesse contexto, é um tempo alto. É melhor dar preferência aos algoritmos que responderam mais rapidamente.

6.1 Trabalhos futuros

A mecânica do jogo apresenta uma alternância entre jogadas, o que exige um tempo de resposta rápido do método de busca utilizado. No entanto, propõe-se aqui uma outra dinâmica, onde o movimento dos inimigos são calculados a todo momento, independentemente do movimento do jogador, de acordo com intervalos de tempo. Ou seja, os inimigos estariam o tempo todo em movimento e isso exigiria que o jogador tivesse um raciocínio rápido para traçar uma rota para fugir dos inimigos. A dificuldade nesse caso seria definida pelo intervalo de tempo entre os cálculos e a velocidade dos inimigos. Caso não fosse possível concluir um cálculo de caminho a tempo, um inimigo poderia continuar seguindo um caminho calculado previamente até que um novo seja obtido. Nesse contexto, seria necessária a implementação de métodos em *threads*, trazendo assim os conceitos de execução paralela. Questões como a de sincronização das funções teriam que ser observadas.

Outra proposta é a de um mapa em movimento constante no estilo de jogos de plataforma. O mapa seria gerado aleatoriamente a todo instante, e o jogador teria que fugir de inimigos que estão correndo atrás dele. Esses inimigos estariam sempre escolhendo o menor caminho até o jogador, forçando-o a escolher o menor caminho também. Toda vez que o jogador errasse o caminho menor, os inimigos se aproximariam cada vez mais, até um deles alcançar o jogador. Nesse modo de jogo não teria um ponto objetivo. O objetivo seria apenas quebrar recordes: a cada tentativa, tentar chegar mais longe. Muitos jogos atualmente usam esse mecanismo de *ranking*, muitas vezes sincronizando e disponibilizando os dados dos jogadores na nuvem, a fim de criar competição entre eles.

Como é mostrado, dependendo do mapa e da posição das entidades, podem existir vários caminhos ótimos. Os métodos foram implementados de forma a utilizar o primeiro caminho ótimo encontrado (no casos dos métodos ótimos). Porém, pode ser interessantes permitir que esses métodos encontrem todos os caminhos ótimos e então utilizem alguma função que estime por onde o jogador irá passar, para escolher, dentre os caminhos ótimos, o que mais se aproxima do usuário. Partindo disso, também é possível programar os inimigos para trabalharem em conjunto para tentarem encurralar o jogador ao seguirem caminhos diferentes.

Por último, propõe-se também a inclusão de itens no jogo (em qualquer modo),

que possam ser usados tanto pelo jogador quanto pelos inimigos, fazendo com que os métodos de busca tenham uma nova informação para trabalhar: quando escolher um item ou não. Isso exigiria uma nova camada de inteligência artificial, que trabalharia com probabilidades, tornando o jogo mais dinâmico e fazendo os inimigos terem um comportamento mais próximo a de um jogador humano. Exemplos de itens são:

- Barreiras móveis: blocos que podem ser movidos mudando assim o mapa e exigindo a atualização do grafo;
- Itens de velocidade: nos casos em que o jogo funcionaria com uma dinâmica de movimentação constante (sem alternância), poderiam haver itens que alteram a velocidade tanto do jogador quanto dos inimigos, tanto positivamente quanto negativamente para ambos;
- Portais: itens que permitiriam o transporte da entidade de um ponto a outro do mapa instantaneamente;
- Vida: o jogador poderia ter "vida", permitindo que quando ele fosse atingido por algum inimigo, em vez de perder o jogo, ele perderia uma "vida" e continuaria até não ter mais "vidas" disponíveis;
- Imunidade: efeito temporário que permite ao jogador passar por um inimigo sem que nada aconteça.

O desenvolvimento de outros elementos, como um modo *online* colaborativo entre celulares (via *bluetooth* ou uma rede *wireless* por exemplo) e o compartilhamento de mapas personalizados pela nuvem, também é encorajado.

Referências Bibliográficas

Google. **Android ndk**. Disponivel em: http://developer.android.com/intl/pt-br/tools/sdk/ndk/index.html, 2016a. Acessado em: fevereiro de 2016.

Google. **Como baixar o android studio e o sdk tools**. Disponivel em: http://developer.android.com/intl/pt-br/sdk/index.html, 2016b. Acessado em: fevereiro de 2016.

Google. **Develop apps**. Disponivel em: http://developer.android.com/intl/pt-br/develop/index.html, 2016c. Acessado em: fevereiro de 2016.

Google. **Dashboards**. Disponivel em: http://developer.android.com/intl/pt-br/about/dashboards/index.html, 2016d. Acessado em: fevereiro de 2016.

Mednieks, Z.; Dornin, L.; Meike, G. B. ; Nakamura, M. **Programando o Android**. Novatec Editora Ltda., 2012.

OHA. **Overview**. Disponivel em: http://http://www.openhandsetalliance.com/android_overview.html, 2016. Acessado em: fevereiro de 2016.

Russell, S.; Norvig, P. **Artificial Intelligence: A Modern Approach**. 3. ed., Campus, 2010.

Tecmundo. **ios, android e windows phone: números dos gigantes comparados**. Disponivel em: http://www.tecmundo.com.br/sistema-operacional/60596-ios-android-windows-phone-numeros-gigantes-comparados-infografico.htm, 2014. Acessado em: fevereiro de 2016.