Resolução do jogo Roll the Block utilizando Métodos de Pesquisa em Linguagem C++ (Tema 5/ Grupo 37)

Bruno Miguel da Silva Barbosa de Sousa
(201604145)
MIEIC
FEUP
Porto, Portugal
up201604145@fe.up.pt

Francisco Manuel Canelas Filipe
(201604601)
MIEIC
FEUP
Porto, Portugal
up201604601@fe.up.pt

O seguinte artigo está inserido na Unidade Curricular de Inteligência Artifical e aborda o tema de Resolução de Problemas através de Métodos de Pesquisa.

Nos próximos parágrafos iremos introduzir um pouco este tipo de problemas, aprofundando com mais detalhe o tema escolhido (Roll the Block), que consiste num jogo onde o utilizador tem como objetivo movimentar um bloco dentro de um mapa até chegar a um ponto objetivo dentro do mesmo.

Desta forma, este artigo não só apresenta e descreve o problema, como também o formula enquanto problema resolúvel através de métodos de pesquisa e aborda, ainda que numa fase inicial, o seu desenvolvimento.

Inteligência Artificial, Pesquisa, Algoritmo BFS, Algoritmo DFS, Algoritmo Guloso, Algoritmo A*, Algoritmo de Aprofundamento Progressivo, Algoritmo de Custo Uniforme

I. Introdução

Este artigo aborda o tema de Resolução de Problemas através de Métodos de Pesquisa, um tema já abordado nesta Unidade Curricular. Como tal, e para demonstrar os conhecimentos adquiridos sobre este área, iremos desenvolver um programa que tem por base este mesmo tópico.

A Resolução de Problemas através de Métodos de Pesquisa é uma área bastante importante da Inteligência Artificial pois foi um ponto de partida para a resolução de problemas onde existem múltiplos caminhos a seguir num dado momento. Um dos exemplos mais presente atualmente onde podem ser aplicados este tipo de métodos são os vídeojogos.

Este artigo é uma breve síntese daquilo que será a nossa abordagem em relação ao tema escolhido (Tema 5 – Roll the Block), onde iremos descrever e formular o problema como sendo um problema resolúvel através de Métodos de Pesquisa.

II. Descrição do Problema

O problema Roll the Block tem como objetivo principal movimentar um paralelepípedo através de um mapa desde um ponto inicial até um ponto objetivo.

Este problema é caraterizado por três componentes. A primeira delas é um mapa de jogo (1) composto por várias casas quadradas de tamanho unitário. A segunda componente é um paralelepípedo de dimensões 1x1x2 (2) que o utilizador pode controlar movendo pelo mapa de jogo. A última componente é uma casa objetivo (3) para onde o utilizador deve movimentar o bloco.

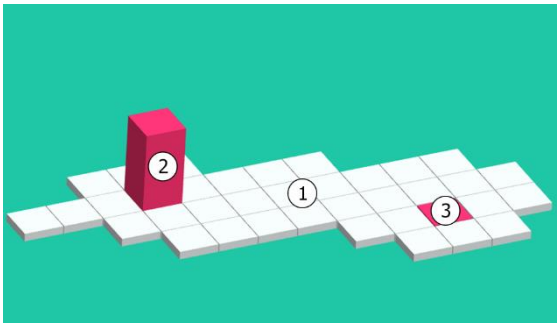

Fig1. Exemplo das componentes do problema

O movimento do bloco é feito de tal maneira que, quando este se movimenta numa direção, o bloco roda 90º na direção do movimento. Dando como exemplo a figura acima apresentada, se, caso o utilizador decidisse jogar para baixo, o bloco iria tomar uma orientação horizontal onde a face de dimensão 2 ocuparia as duas casas livres diretamente abaixo da posição atual do bloco.

Caso o utilizador mova o seu bloco de tal forma que este fique parcial ou totalmente fora do mapa, este perde o jogo.

III. Formulação do Problema

Representação do estado

Lista com posições onde o bloco pode estar (suportes).

Posição -> Coordenada x e y.

Bloco divido em duas posições, quando o bloco está em pé são iguais/uma delas é nula.

A 1ª posição será a posição de cima (quando deitado na vertical) ou a posição da esquerda (quando deitado na horizontal).

Exemplo:

(1,1), (1,1) : Em pé na posição (1,1)
(1,2), (1,3) : Deitado na vertical nas posições (1,2) e (1,3)
(1,2), (2,2) : Deitado na horizontal (1,2) e (2,2)

Estado Inicial

Posições inicias do bloco obtidas a partir do mapa do nível.

Suportes (posições onde o bloco pode estar) obtidas a partir do mapa do nível.

Posição objetivo do bloco obtidas a partir do mapa do nível.

Teste Objetivo

O bloco está em pé e está na posição final (obtida a partir do mapa do nível).

Operadores

Mover bloco para cima:

Pré condições

Se está em pé: Duas posições acima da posição do bloco têm de ser suportes.

Se está deitado na vertical: A posição acima da posição mais acima do bloco (1ª posição) tem de ser suporte.

Se está deitado na horizontal: As posições acima de cada uma das posições do bloco têm de ser suportes.

Efeito

Se está em pé: As posições do bloco passarão a ser as duas posições acima da sua posição anterior.

Se está deitado na vertical: A posição do bloco passará a ser a posição acima da sua posição mais acima (1ª posição) anterior.

Se está deitado na horizontal: As posições do bloco passarão a ser as duas posições acima de cada uma das posições anteriores do bloco.

Custo: 1

Mover bloco para baixo:

Pré condições

Se está em pé: Duas posições abaixo da posição do bloco têm de ser suportes.

Se está deitado na vertical: A posição abaixo da posição mais abaixo do bloco (2ª posição) tem de ser suporte.

Se está deitado na horizontal: As posições abaixo de cada uma das posições do bloco têm de ser suportes.

Efeito

Se está em pé: As posições do bloco passarão a ser as duas posições abaixo da sua posição anterior.

Se está deitado na vertical: A posição do bloco passará a ser a posição abaixo da sua posição mais abaixo (2ª posição) anterior.

Se está deitado na horizontal: As posições do bloco passarão a ser as duas posições abaixo de cada uma das posições anteriores do bloco.

Custo: 1

Mover bloco para esquerda:

Pré condições

Se está em pé: Duas posições à esquerda da posição do bloco têm de ser suportes.

Se está deitado na vertical: As posições à esquerda de cada uma das posições do bloco têm de ser suportes.

Se está deitado na horizontal: A posição à esquerda da posição mais à esquerda do bloco (1ª posição) tem de ser suporte.

Efeito

Se está em pé: As posições do bloco passarão a ser as duas posições à esquerda da sua posição anterior.

Se está deitado na vertical: As posições do bloco passarão a ser as duas posições à esquerda de cada uma das posições anteriores do bloco.

Se está deitado na horizontal: A posição do bloco passará a ser a posição à esquerda da sua posição mais à esquerda (1ª posição) anterior.

Custo: 1

Mover bloco para direita:

Pré condições

Se está em pé: Duas posições à direita da posição do bloco têm de ser suportes.

Se está deitado na vertical: As posições à direita de cada uma das posições do bloco têm de ser suportes.

Se está deitado na horizontal: A posição à direita da posição mais à direita do bloco (2ª posição) tem de ser suporte.

Efeito

Se está em pé: As posições do bloco passarão a ser as duas posições à direita da sua posição anterior.

Se está deitado na vertical: As posições do bloco passarão a ser as duas posições à direita de cada uma das posições anteriores do bloco.

Se está deitado na horizontal: A posição do bloco passará a ser a posição à direita da sua posição mais à direita (2ª posição) anterior.

Custo: 1

IV. Trabalho Relacionado

Em relação a implementações do mesmo jogo encontrámos uma em JavaScript, chamada Bloxy[1] realizada por Terry Oshea.

Quanto a implementações dos algoritmos propostos no enunciado na linguagem C++, encontrámos as seguintes:

- Geeks for Geeks Breadth First Search[2]
- Geeks for Geeks Depth First Search[3]
- Geeks for Geeks Iterative Deepening Search[4]
- Uniform Cost Search por Saleem Jair[5]
- Geeks for Geeks A* Search Algorithm[6]

Apesar de a maioria dos algoritmos ser procura em grafos, será relativamente fácil adapta-los para a procura no nosso problema.

V. Implementação do Jogo

Na nossa implementação da solução do problema incorporámos 3 componentes principais:

- Representação do estado do tabuleiro
- Operadores
- Funções do jogo

Representação do Estado do Tabuleiro

Para a representação do tabuleiro foi utilizada uma matriz de caracteres implementada através de vetores (vector<vector<char>). Esta matriz contém:

- 'g': representa uma casa do mapa para onde o bloco do utilizador pode jogar.
- ' ': representa uma casa que não pertence ao mapa e, caso o bloco do utilizador se coloque nela, perde.
- '|': representa o bloco do utilizador quando este se encontra em pé.
- '-': representa bloco do utilizador quando este se encontra deitado, quer na horizontal ou vertical.
- '_': representa a casa de destino, para a qual o utilizador se deve mover.
- 'X': representa um botão que, quanto ativo, acrescenta as casas 'x' ao mapa. Para ativar este botão o bloco do utilizador tem de estar em pé nesta posição.
- 'C': representa um botão que, quanto ativo, acrescenta as casas 'c' ao mapa. Para ativar este botão o bloco do utilizador pode estar quer em pé ou deitado nesta posição.
- 'x': representa uma casa para a qual o bloco do utilizador se pode mover e que apenas fica ativa quando este se coloca em pé na casa 'X'.
- 'c': representa uma casa para a qual o bloco do utilizador se pode mover e que apenas fica ativa quando este se coloca quer em pé ou deitado na casa 'C'.

Por questões de eficiência temporal e espacial quando é lido o ficheiro com um mapa são criados 3 sets e 3 posições (Position é uma struct com duas coordenadas):

Sets

- Set das posições possíveis (set<Position>).
- Set das posições que vão ser expandidas com o botão 'C' (set<Position>).
- Set das posições que vão ser expandidas com o botão 'X' (set<Position>).

Posições

- Posição da casa objetivo.
- Posição do botão 'C'.
- Posição do botão 'X'.

Para além disso, para cada estado do jogo são guardados:

- As posições do bloco: Em que uma delas é "nula" (coordenadas -1) caso o bloco esteja na vertical.
- Dois booleanos para guardar se os botões 'C' e 'X' estão ativos.
- Um apontador para o estado pai (Estado a partir do qual foi gerado este estado).
- Heurística (estimativa por baixo do custo até à posição final).
- Custo desde a posição inicial (dado que todos os nossos operadores têm custo 1, indica também a profundidade).

Operadores

Na nossa implementação foram desenvolvidos 4 operadores diferentes:

- Up.
- Down.
- Left.
- Right.

Para todos os operadores a pré-condição necessária é: ser possível mover-se na direção do operador em questão, isto é, depois de aplicado o operador, o bloco tem de se situar obrigatoriamente em posições possíveis do mapa.

Em termos de custo todos os operadores têm custo unitário.

Para explicar melhor o raciocínio por detrás dos operadores apenas iremos abordar o operador **Up** visto que para os restantes o processo é semelhante.

Para cada operador é necessário considerar 3 casos possíveis:

Bloco em pé

Neste caso, uma das posições do bloco (pos2) tem coordendas -1. Qualquer movimento numa das direções resulta num bloco deitado em que a pos2 é a mais afastada da posição anterior.

Caso se desloque para uma casa contendo o botão 'C', o booleano que indica o estado ativo ou não deste é alterado. Dado que depois de aplicado este operador o bloco fica deitado, não é necessário verificar se a casa contém o botão 'X' (o bloco precisaria de estar em pé).

```
if (isStanding(state)){
    Position pos1;
    Position pos2;

    pos1.i = state->pos1.i - 2;
    pos1.j = state->pos1.j;

    pos2.i = state->pos1.i - 1;
    pos2.j = state->pos1.j;

    if (find(state, pos1) && find(state, pos2)) {
      if (pos1 == expandCircle || pos2 == expandCircle)
        state->expandedCircle = !state->expandedCircle;

      state->pos1 = pos1;
      state->pos2 = pos2;
      return true;
     }
  return false;
}
```

Bloco deitado na horizontal

Neste caso, um movimento para cima não altera a orientação do bloco, isto é, este continuará deitado na horizontal, apenas se alteram as coordenadas i (linha do mapa em que o bloco se situa) reduzindo o seu valor em 1 unidade.

Tal como no caso anterior é verificado se, após o movimento, cada uma das posições se situa numa casa contendo o botão 'C'. Dado que depois de aplicado este operador o bloco permanece deitado na horizontal, não é necessário verificar se a casa contém o botão 'X' (o bloco precisaria de estar em pé).

```
if (isHorizontal(state)){
    Position pos1;
    Position pos2;

    pos1.i = state->pos1.i - 1;
    pos1.j = state->pos1.j;

    pos2.i = state->pos2.i - 1;
    pos2.j = state->pos2.j;

   if (find(state, pos1) && find(state, pos2))
   {
      if (pos1 == expandCircle || pos2 == expandCircle)
        state->expandedCircle = !state->expandedCircle;

      state->pos1 = pos1;
      state->pos2 = pos2;
      return true;
   }

   return false;
}
```

Bloco deitado na vertical

Neste caso, um movimento para cima vai alterar a orientação do bloco, isto é, o bloco passa a estar em pé. Neste caso em vez de alterar as coordenadas i do bloco alteram-se as coordenadas j.

Sendo assim, não só se verifica se a posição onde o bloco se situa contém um botão 'C', mas também para o caso de conter um botão 'X' visto que estes apenas aceitam blocos que estejam colocados em pé.

```
if (isVertical(state))
{
    Position pos1;
    Position pos2;

    pos1.i = state->pos1.i - 1;
    pos1.j = state->pos1.j;

    pos2.i = -1;
    pos2.j = -1;

if (find(state, pos1))
    {
       if (pos1 == expandCircle)
        state->expandedCircle = !state->expandedCircle;

       if (pos1 == expandCross)
         state->expandedCross = !state->expandedCross;

       state->pos1 = pos1;
       state->pos2 = pos2;
       return true;
     }

     return false;
}
```

Teste Objetivo

Para verificar a condição de terminação apenas é necessário, através do estado atual do jogo, comparar a posição atual do bloco com a posição objetivo. Para além disso é também obrigatório que o bloco esteja em pé, ou seja, a pos2 deste mesmo tem coordenadas -1.

```
bool checkDone(const shared_ptr<State> &state)
{
   return state->pos1 == goal && state->pos2.i == -1 &&
state->pos2.j == -1;
}
```

Funções

Leitura de Níveis através de Ficheiros

Na implementação da leitura de níveis através de ficheiros implementámos dois métodos principais:

- Leitura e armazenamento do nível enquanto matriz.
- Interpretação da matriz e inicialização das estruturas de dados necessárias.

O primeiro método começa por receber como parâmetro o nome do nível inserido pelo utilizador na execução do programa. Com este parâmetro, acede ao ficheiro em questão e lê cada linha do ficheiro guardando-a na variável global puzzle (matriz de caracteres).

```
void readLevel(char *level){
    ifstream mapFile;
    mapFile.open("levels/" + string(level) + ".txt");

    if(!mapFile.is_open()){
        cout << "Map file not found\n";
        exit(1);
    }
    puzzle.clear();
```

```
    string temp;
    while (getline(mapFile, temp)){
        vector<char> line(temp.begin(), temp.end());
        puzzle.push_back(line);
    }

    mapFile.close();
}
```

O Segundo método analisa a variável anteriormente populada (iterando através de um ciclo "for") e, de acordo com os caracteres apresentados na secção de "Representação do Estado do Tabuleiro", inicializa as seguintes variáveis:

- Set de posições possíveis (possible).
- Set das posições que vão ser expandidas com o botão 'C' (expandableCircle).
- Set das posições que vão ser expandidas com o botão 'X' (expandableCross).
- Posição inicial do bloco do utilizador (start).
- Posição objetivo (goal).

```
for (int i = 0; i < puzzle.size(); ++i)
   for (int j = 0; j < puzzle[i].size(); ++j)
       switch (puzzle[i][j]){
         case 'g':
           pos.i = i;
           pos.j = j;
           possible.insert(pos);
           break;
         case '-':
           if (lying){
              start->pos2.i = i;
              start->pos2.j = j;
           }
           else{
              start->pos1.i = i;
              start->pos1.j = j;
              lying = true;
           }
           pos.i = i;
           pos.j = j;
           possible.insert(pos);
           break;
         case '|':
            start->pos1.i = i;
            start->pos1.j = j;
            start->pos2.i = -1;
            start->pos2.j = -1;
            pos.i = i;
            pos.j = j;
            possible.insert(pos);
            break;
         case '_':
           goal.i = i;
           goal.j = j;
           pos.i = i;
           pos.j = j;
           possible.insert(pos);
           break;
         case 'X':
           expandCross.i = i;
           expandCross.j = j;
           pos.i = i;
           pos.j = j;
           possible.insert(pos);
           break;
         case 'x':
            pos.i = i;
            pos.j = j;
            expandableCross.insert(pos);
            break;
         case 'C':
            expandCircle.i = i;
            expandCircle.j = j;
            pos.i = i;
            pos.j = j;
            possible.insert(pos);
            break;
         case 'c':
            pos.i = i;
            pos.j = j;
            expandableCircle.insert(pos);
            break;
         default:
            break;
        }
```

Visualizar em Modo Texto

Para a visualização em modo texto é mostrado o caracter associado a cada posição. Para além disso é mostrado o número de cada coluna e de cada linha e garantido um espaçamento entre células para ajudar à leitura do tabuleiro.

Numa determinada célula, é mostrado:

- '|': caso o bloco esteja em pé nessa posição.
- '-': caso o bloco esteja deitado nessa posição.
- '_': caso seja a posição final.
- 'C': caso seja um botão para expandir (deitado ou em pé).

- ‘X’: caso seja um botão para expandir (apenas em pé).
- ‘c’: caso sejam posições expandíveis pelo botão ‘C’ e estejam expandidas.
- ‘x’: caso sejam posições expandíveis pelo botão ‘X’ e estejam expandidas.
- ‘g’: caso seja uma posição para onde o bloco pode mover.
- ‘ ‘: caso seja uma posição fora do mapa ou posições expandíveis ainda não expandidas.

```
for (i = 0; i <= 2 * puzzle.size(); i++)
{
   if (i % 2 != 0)
     cout << (char)(i / 2 + 'A');

   for (j = 0; j <= 2 * puzzle[0].size(); j++)
   {
     if (i % 2 == 0)
     {
       if (j == 0)
         cout << " ";
       if (j % 2 == 0)
         cout << " ";
       else
         cout << "---";
     }
     else
     {
       if (j % 2 == 0)
         cout << "|";
       else
      {
        if (state->pos1.i==i/2 && state->pos1.j==j/2)
        {
           if(state->pos2.i==-1 && state->pos2.== -1)
              cout << " | ";
           else
              cout << " - ";
         }
        else if (state->pos2.i==i/2 && state->pos2.j==j/2)
           cout << " - ";
         else if (puzzle[i / 2][j / 2] == '|')
           cout << " g ";
         else if (puzzle[i/2][j/2]=='x' && !state-
>expandedCross)
            cout << "   ";
        else if (puzzle[i / 2][j / 2] == 'c' && !state-
>expandedCircle)
           cout << "   ";
        else
           cout << ' ' << puzzle[i / 2][j / 2] << ' ';
      }
     }
   }
   if (i % 2 != 0)
      cout << (char)(i / 2 + 'A');
   cout << endl;
}
```

Validar uma jogada

O nosso mecanismo de validação de jogadas está presente na implementação dos operadores. Tal como já foi visto anteriormente cada operador tem 3 casos possíveis:

- Bloco em pé.
- Bloco deitado na horizontal.
- Bloco deitado na vertical.

As funções implementadas para os operadores têm como valor de retorno um booleano que indica se o operador pode ser aplicado ou não (e aplica-o caso seja possível). Um valor de retorno “false” indica que a jogada não é válida e, como tal, não é efetuada.

Executar uma jogada

Tal como na validação de uma jogada, a execução foi implementada dentro dos operadores. Estes recebem como parâmetro o estado atual do jogo contendo a posição do bloco. Utilizando os valores desta posição é possível verificar a orientação deste e, por consequência, executar a respetiva jogada que lhe compete.

Para assegurar que o bloco é movido para uma casa válida do mapa é necessário:

- Verificar que a posição de destino se encontra no set de posições possíveis.
- Caso a posição de destino seja um ‘x’ ou um ‘c’ verificar que o utilizador tem o botão ‘X’ ou ‘C’ ativos.

Para isso implementou-se o seguinte método:

```
bool find(const shared_ptr<State> &state, const
Position &pos){
    return possible.find(pos) != possible.end() ||
(state->expandedCircle && expandableCircle.find(pos) !=
expandableCircle.end()) || (state->expandedCross &&
expandableCross.find(pos) != expandableCross.end());
}
```

Se tudo isto se verificar a jogada é efetuada com sucesso, a posição do bloco no estado atual é atualizada e, caso a posição do bloco se encontre num dos botões (e dependendo da sua orientação), atualizar o booleano que indica o estado ativo ou não deste botão.

```
if (find(state, pos1) && find(state, pos2))
{
   if (pos1 == expandCircle || pos2 == expandCircle)
     state->expandedCircle = !state->expandedCircle;

   state->pos1 = pos1;
   state->pos2 = pos2;
   return true;
}
```

Avaliar um Estado

Dependendo dos algoritmos utilizados a avaliação de um estado pode ser efetuada analisando o custo, a heurística ou então estas duas componentes combinadas.

Custo

O custo de um determinado estado é a soma do custo de todos os operadores aplicados desde o estado inicial até esse estado. Dado que todos os nossos operadores têm custo unitário, este custo será igual à profundidade do estado na árvore de pesquisa.

Heurística

A nossa heurística baseou-se na distância Manhattan desde a posição atual do bloco até à posição final. Apenas a distância Manhattan não serviria como heurística dado que é possível com os nossos operadores mover 3 posições apenas utilizando 2 operações, o que faria com que a heurística em determinados estados fosse maior que o custo real. Dado que esse é o caso de maior distância percorrida utilizando o menor número de operadores e que apenas acontece numa das direções, para obter uma aproximação por baixo do custo que falta, apenas é necessário dividir a distância até ao final em cada direção por 1,5 (3/2).

Com a adição de botões de expansão (‘X’ e ‘C’) que são necessários para completar o nível, é calculada (utilizando a heurística referida anteriormente) a distância ao botão aparentemente mais próximo, a distância desse botão ao outro (no caso de existirem os dois) e a distância desse outro à posição final.

VI. Algoritmos de Pesquisa

Nesta secção iremos aprofundar melhor a nossa implementação dos algoritmos propostos para o desenvolvimento deste projeto. Apenas é apresentado o código para o Algoritmo BFS pois as implementações dos algoritmos são todas semelhantes, sendo indicados os aspetos em que diferem.

Algoritmo BFS

Este algoritmo centra-se na análise em largura da árvore de pesquisa. Desta forma, para garantir este tipo de análise, foi utilizada uma fila de prioridade que, à medida que são gerados estados através da aplicação dos operadores, os guarda garantindo que quanto mais acima na árvore estejam situados, maior é a sua prioridade.

Neste algoritmo a primeira solução a ser encontrada também é a primeira a ser retornada. No entanto, visto que os operadores têm custo unitário, a primeira solução é aquela que necessita do menor número de operações.

```
queue<shared_ptr<State>> states;
states. push(start);
++nodes;

while (states.size() != 0)
{
   shared_ptr<State> actual = states.front();
   states.pop();

   if (checkDone(actual))
    return actual;
 for (int i = 0; i < operators.size(); ++i)
 {
   shared_ptr<State> new_state(new State);
   actual->clone(*new_state);

   if (operators[i](new_state) &&
!findDuplicate(actual, new_state))
   {
     new_state->parent = actual;
     new_state->cost = actual->cost + 1;
     states.push(new_state);
     ++nodes;
   }
  }
 }
 return nullptr;
```

Algoritmo DFS

Este algoritmo em termos de implementação é muito semelhante ao anterior. A única diferença consiste na estrutura de dados utilizada para guardar os estados. Neste caso é utilizada uma pilha que contém no seu topo os últimos estados gerados. Assim, a cada expansão da árvore, os nós filhos são os nós do topo e os primeiros a ser analisados.

Este algoritmo retorna a primeira solução, não garantindo a solução ótima.

Algoritmo Guloso

A sua implementação é muito semelhante às duas anteriores diferindo apenas na estrutra de dados utilizada para guardar os estados e no facto de ser utilizada a heurística.

Este algoritmo utiliza uma lista de prioridades que tem como operador de ordenação a comparação de heurísticas. Desta forma, a lista dá maior prioridade aos estados com menor heurística, garantindo que os estados mais próximos (utilizando a heurística referida anteriormente) ao destino são analisados primeiro.

```
class greedyCompare{
public:
bool operator() (shared_ptr<State> lhs, shared_ptr<State> rhs){
    return lhs->heuristic > rhs->heuristic;
    }};
```

Algoritmo de Custo Uniforme

A implementação deste algoritmo foi baseada no Algoritmo Guloso descrito anteiormente. A única diferença entre estes dois é o operador de ordenação da fila de prioridade que, no caso do Algoritmo de Custo Uniforme, utiliza o custo em vez da heurística. Desta forma, a fila de prioridade fica ordenada em relação ao número de jogadas efetuadas (quanto menor este valor maior a prioridade).

Apesar disto, esta implementação traduz-se numa implementação do Algoritmo DFS. Isto acontece pois os operadores desenvolvidos têm todos custo unitário não afetando em circunstância alguma as escolhas feitas pelo algoritmo.

```
class uniformCostCompare{
public:
 bool operator() (shared_ptr<State> lhs, shared_ptr<State> rhs)
 {
    return lhs->cost > rhs->cost;
 }
};
```

Algoritmo A*

A implementação deste algoritmo foi baseada na junção dos Algoritmos Guloso e Custo Uniforme descritos anteriormente. Uma vez mais, a única diferença está presente no operador de comparação da fila de prioridade. No caso do Algoritmo A*, combina a heurística e o custo através de uma soma. Esta soma é o valor a ser utilizado na comparação de estados para a ordenação da fila.

```
class aStarCompare{
public:
 bool operator() (shared_ptr<State> lhs, shared_ptr<State> rhs)
 {
   int lhsCost = lhs->cost + lhs->heuristic;
   int rhsCost = rhs->cost + rhs->heuristic;
   return lhsCost > rhsCost;
 }
};
```

Algoritmo de Aprofundamento Progressivo

Esta implementação difere um pouco das anteriores pois vai testando diferentes níveis de profundidade na procura da solução. Desta forma englobámos o Algoritmo DFS descrito anteriormente dentro de um ciclo *while* que vai incrementando a profundidade, unidade a unidade, até encontrar a solução. Caso seja alcançada a profundidade especificada para uma determinada iteração do ciclo *while* sem existir qualquer solução, avança-se para a iteração seguinte, eliminando os nós obtidos da anterior e atualizando o valor da profundidade.

VII. Experiências e Resultados

De forma a analisar os algoritmos implementados, após a execução dos vários algoritmos armazenámos alguns variáveis referentes a: tempo de execução, profundidade da árvore, nós armazenados em memória e nós analisados ao longo da execução. Tendo em conta estas componentes para os vários algoritmos aplicadas aos diferentes níveis desenvolvidos foi possível construir a seguinte análise:

Análise do tempo de execução ao longo dos níveis desenvolvidos:

Numa fase inicial os valores são bastante reduzidos, mas, há medida que os níveis vão tomando uma maior complexidade, algoritmos como Pesquisa em Largura, Pesquisa em Profundidade e Custo Uniforme demonstram uma maior dificuldade em manter o tempo de execução relativamente baixo. Podemos também verificar através da análise dos dados que o algoritmo mais rápido é o Guloso, seguido do A*, algoritmos que usam a heurisitca para mais rapidamente se aproximarem da solução.

Fig2. Análise do Tempo de Execução

Análise da profundidade ao longo dos níveis desenvolvidos:

Em relação à análise da profundidade da árvore é possível visualizar algumas diferenças entre algoritmos.

Analisando apenas o algoritmo de Pesquisa em Profundidade é possível verificar uma grande oscilação nos resultados obtidos. Isto acontece pois neste algoritmo a expansão é feita sempre à esquerda na árvore não olhando a qualquer outro critério. Neste algoritmo é ainda possível verificar uma quebra que indica que este algoritmo não conseguiu terminar a sua execução dentro de um tempo razoável.

Podemos também verificar que no caso dos Algoritmos de Pesquisa em Largura, A*, Custo Uniforme e Aprofundamento Progressivo, a solução encontrada é ótima. O Algoritmo Guloso, apesar de utilizar pesquisa informada não chega à solução ótima devido a só se importar com a heuristica, o que leva a tomar decisões não ótimas a longo prazo.

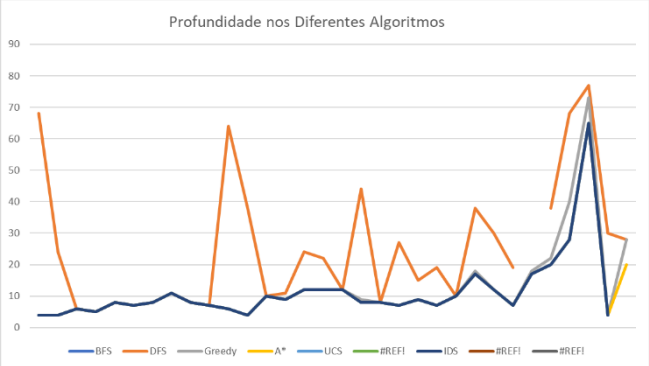

Fig3. Análise da Profundidade

Análise dos nós em memória ao longo dos níveis desenvolvidos:

Relativamente ao número de nós armazenados em memória, o comportamento é semelhante ao do tempo de execução pois estes partilham de uma proporcionalidade direta. Tipicamente a demora a encontrar a solução dá-se devido a uma maior profundidade da árvore e, portanto, um número mais elevado de nós.

Por outro lado, devido a dar "reset" aos nós expandidos a cada iteração, o Algoritmo de Aprofundamento Progressivo apresenta um número pequenos de nós em memória apesar de demorar um tempo considerável.

De notar que há a possibilidade de o programa não conseguir resolver o problema devido a falta de memória (especialmente utilizando o algoritmo de Pesquisa em Largura em mapas de maior dimensão).

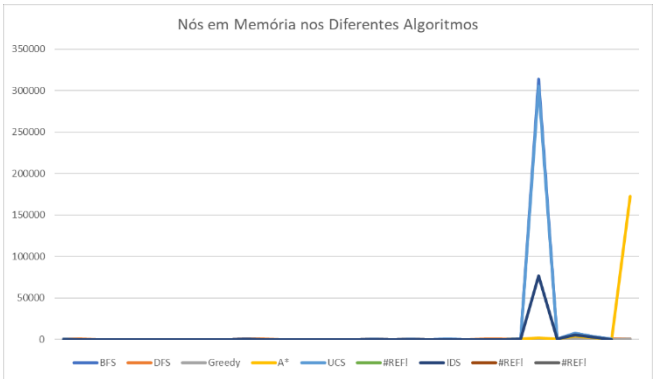

Fig4. Análise dos Nós em Memória

Análise dos nós analisados nos algoritmos ao longo dos níveis desenvolvidos:

Tendo falado do número de nós em memória é necessário também falar do número de nós analisados.

Nem todos os nós expandidos são analisados. Isto acontece pois, num determinado momento é necessário fazer uma escolha acerca do caminho a seguir na árvore. Após ser feita esta escolha é analisado o nó correspondente e, caso seja encontrada uma solução,são ignorados os restantes.

No entanto o algoritmo de Aprofundamento Progressivo é uma exceção a esta regra. Visto que a cada iteração a árvore é reposta, há um número elevado de nós que são calculados várias vezes. Desta forma, o valor dos nós analisados será sempre superior ao valor dos nós em memória.

De notar que os Algoritmos Guloso e Pesquisa em Profundidade (quando encontra solução rapidamente) têm geralmente um menor número de nós analisados, dado a tentarem ir diretamente à solução. O Algoritmo A* também tém geralmente um número menor de nós pois tentar tomar caminhos "bons" em direção à solução, mas analiza mais nós que os algoritmos anteriores pois também tem em conta outros nós que possam ser "piores" localmente mas que levam a uma melhor solução.

Os restante algoritmos, devido a analisarem praticamente todos os nós, levam a um número eleveado de nós analisados.

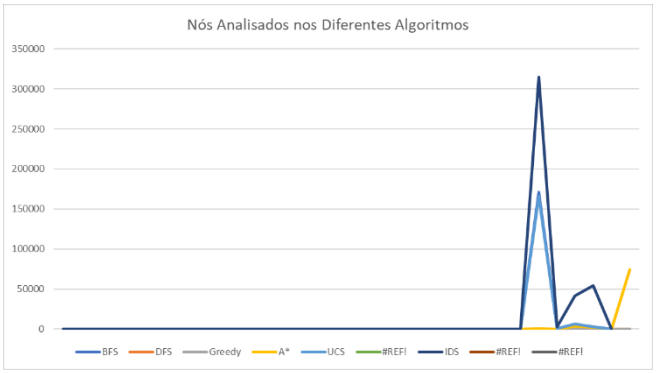

Fig5. Análise dos Nós Analisados durante a Execução

Em anexo a este relatório enviámos um ficheiro pdf contendo as tabelas com os valores específicos de cada algoritmo para cada nível. Os níveis aries e taurus foram retirados da aplicação Bloxorz[8], os níveis 1,2,3,4 e 33 foram retirados do jogo da Miniclip Bloxorz[9] e os níveis ez e new foram inventados por nós.

VIII. Conclusões e Perspetivas de Desenvolvimento

Ao longo do desenvolvimento deste projeto não foi notória qualquer dificuldade acrescida. Fizemos uma abordagem incremental que, na nossa opinião, foi bem implementada. Em geral foi um projeto bem conseguido e realizado com sucesso.

Quanto ao conteúdo abordado pelo mesmo (Resolução de Problemas utilizando Métodos de Pesquisa), considerámos ser algo interessante e bastante importante para um primeiro contacto com a Unidade Curricular, visto que ajudou a consolidar melhor os conteúdos abordados nas aulas teóricas. Para além disso foi possível visualizar melhor as diferenças

nos vários métodos de pesquisa e as implicações de cada um destes.

Após o desenvolvimento deste projeto, sentimo-nos mais aptos para, ao analisar um problema, reconhecer quais os métodos mais adequados a aplicar.

Em suma, este projeto foi um bom primeiro contacto com a área da Inteligência Artificial pois, apesar de já termos implementado agentes de Inteligência Artificial noutras UC's, só agora é que conseguimos perceber melhor o funcionamento deste tipo de mecanismos.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Oshea, Terry. (2017, April 20). Bloxy. Retirado de https://github.com/TerryOShea/bloxy?fbclid=IwAR004lExdra0-Hmqeoy2Q9-fHFFvcLngWNzTuwmXuSvlSgmqIt_esVINCUM

[2] geeksforgeeks. Breadth First Search or BFS for a graph. Retirado de https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/?fbclid=IwAR0C6s1Vx664GLTdeGh3iy5d-cpTDMECmztc9yeSfeEeYUxCX3oFNvCpnKw

[3] geeksforgeeks. Depth First Search or DFS for a graph. Retirado de https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/?fbclid=IwAR3i-APiqmBiOA98Wc6uKbsaYB310nSB1eudU7yCF3XSB7bqi9dKUJfuk-A

[4] geeksforgeeks. Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS). Retirado de https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/?fbclid=IwAR1pJHs6TMqjueoW6CCYs-0TDqnQTgsCMvGYh_Lm8Z14LDcaX6jYM_UzX8k

[5] Jain, Sameer. (2018, September 13). UCF.cpp. Retirado de https://gist.github.com/sameer-j/76c38850f876661daf5d?fbclid=IwAR0TafXBwkFmEA30Dtss0fQNfRfDbsqOBfN9wCowZHiBWSRjqqheUhVT-IY

[6] geeksforgeeks. A* Search Algorithm. Retirado de https://www.geeksforgeeks.org/a-search-algorithm/?fbclid=IwAR1DC9riMXym3U56Ga7glTXI_72VCdcIwjC3-MJqNDMQBy6_11Yu-E4s-YY

[7] Figure 1. Exemplo das componentes do problema. Adaptado de CNET. Retirado a 10 de Março de 2019 de https://download.cnet.com/Bloxorz-Block-And-Hole-for-Windows-10/3000-2111_4-77556180.html

[8] Google play. Bloxorz.Última atualização a 20 de Março de 2019 https://play.google.com/store/apps/details?id=com.bitmango.go.bloxorzpuzzle

[9] miniclip. Bloxorz. https://www.miniclip.com/games/bloxorz/pt/