Aplicação do Algoritmo A* ao problema de busca do fantasma do *Pacman*

Alexsander Andrade de Melo

10 de dezembro de 2014

Resumo

Este trabalho consiste na implementação do algoritmo de busca A* para o problema de busca do fantasma do *Pacman*, o qual compõem parte dos requisitos necessários para obtenção de êxito na Atividade Acadêmica de Inteligência Artificial 2014-02 do curso de Ciência da Computação da Universidade Federal Rural do Rio de Janeiro.

1 Introdução

1.1 Algoritmo de busca A*

O algoritmo A* se baseia no algoritmo *Best First* utilizando a heurística

$$f(n) = g(n) + h(n),$$

onde $g(n)$ é o custo real do nó inicial s até o nó corrente n, e $h(n)$ é uma estimativa heurística da distância entre o nó corrente n até o nó destino t. Neste nosso problema de busca do fantasma do *Pacman*, o nó inicial s refere-se à posição atual do fantasma, e o nó destino t à posição corrente do *Pacman*.

Para tanto, o algotimo utiliza uma lista de prioridades, denominada *lista aberta* (*open-set*), ordenada de forma não-decrescente de acordo com os valores da heurística f de cada nó. A cada iteração do algoritmo retira-se o primeiro nó desta lista (ou seja, um nó de

menor custo) e o adiciona a uma outra lista, denominada denominada *lista fechada* (*closedset*). (Esta lista é a que define o caminho a ser percorrido da posição atual até se chegar ao objetivo.) Caso o nó atual coincida com o objetivo (isto é, o nó t), o algoritmo encerra a sua execução retornando o caminho percorrido (determinado pela lista fechada); caso contrário, são visitados todos os nós vizinhos do nó atual, verificando se tais pertencem a lista aberta, caso não pertençam, são adicionados de acordo com a prioridade do menor custo de f. Além disso, também são definidas as relações de precedência dos nós vizinhos com relação ao nó atual no caminho a ser percorrido (ou seja, definindo que o nó atual é o nó pai desses nó no caminho), e caso seja observado que o caminho em verificação é menos custoso do que um outro caminho definido anteriormente, são recalculados e atualizados os custos g, h e f do nó vizinho, bem como a sua relação de precedência. Assim, o algoritmo A* sempre encontra o caminho ótimo, caso este exista.

Uma observação relevante a se fazer é que, o algoritmo A* durante a contrução do caminho localiza diversas ramificações (e isso é importante, para que seja determinado sempre o caminho ótimo), no entanto quando o fantasma for percorrer o caminho para se chegar no *Pacman* é desejado que não exista tais ramificações durante o percurso. Assim, após a execução do algoritmo A* (ou instantes antes do mesmo terminar sua execução) é feita uma reconstrução do caminho removendo-se todos esses caminhos alternativos, deixando apenas o caminho ótimo para ser percorrido pelo fantasma.

Segue abaixo um pseudo-código do algoritmo A* obtido em [3].

```
function A*(start,goal)
  closedset := the empty set
  openset := {start}
  came_from := the empty map

  g_score[start] := 0

  f_score[start] := g_score[start] + heuristic_cost_estimate(start, goal)

  while openset is not empty
      current := the node in openset having the lowest f_score[] value
      if current = goal
          return reconstruct_path(came_from, goal)

      remove current from openset
      add current to closedset
      for each neighbor in neighbor_nodes(current)
          if neighbor in closedset
```

```
                continue
            tentative_g_score := g_score[current] + dist_between(current,neighbor)

            if neighbor not in openset or tentative_g_score < g_score[neighbor]
                came_from[neighbor] := current
                g_score[neighbor] := tentative_g_score
                f_score[neighbor] := g_score[neighbor] + heuristic_cost_estimate(neighbor, goal)
                if neighbor not in openset
                    add neighbor to openset

    return failure

function reconstruct_path(came_from,current)
    total_path := [current]
    while current in came_from:
        current := came_from[current]
        total_path.append(current)
    return total_path
```

1.2 Distancia de Manhattan

Como vimos anteriormente, para aplicarmos o algoritmo A* em um problema de busca em grafos é necessário definir uma distância heurística h a ser utilizada. Desta forma, neste nosso trabalho, adotamos a *distância de Manhattan* como tal heurística, que é dada como segue:

$$d = |x_1 - x_0| + |y_1 - y_0|,$$

onde $p = (x_0, y_0)$, $q = (x_1, y_1) \in \mathbb{R}^2$ são pontos do percurso.

Utilizamos a distância de Manhattan neste trabalho pois, além de sua simplicidade, a mesma coincide bem com o problema de busca do fantasma do *Pacman* que não anda em diagonal, apenas para frente, para trás, para direita ou para esquerda, que é exatamente o que a distância de Manhattan considera.

2 Implementação

A implementação foi feita baseada no *Template* disponibilizado pelo professor no *quiosque do aluno*, utilizando a linguagem C/C++ juntamente com a biblioteca OpenGL para parte gráfica. Neste *template* se encontrava disponível toda immplementação referente à visualização através do OpenGL, bem como a parte de manipulação de listas, fornecidas

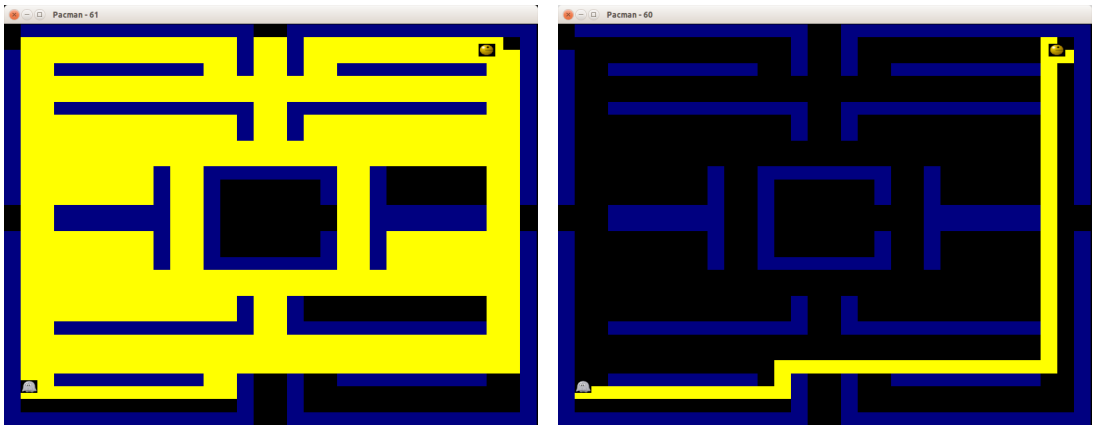
pela implementação de uma lista duplamente encadeada. Sendo necessário, basicamente, a implementação da estratégia de busca, que neste caso foi o algoritmo A*. No Apêndice se encontram as principais implementações que tiveram que ser realizadas para resolução do problema de busca do fantasma do *Pacman*.

2.1 Dificuldades

A principal dificuldade na realização deste trabalho foi a manipulação das listas encadeadas para armazenamento das listas de nós abertos e de nós fechados do algoritmo A*, não por falta de conhecimento no uso de estruturas baseadas em *ponteiros*, mas sim ao tentar alinhar os objetivos do problema com esse tipo de estrutura.

3 Resultados

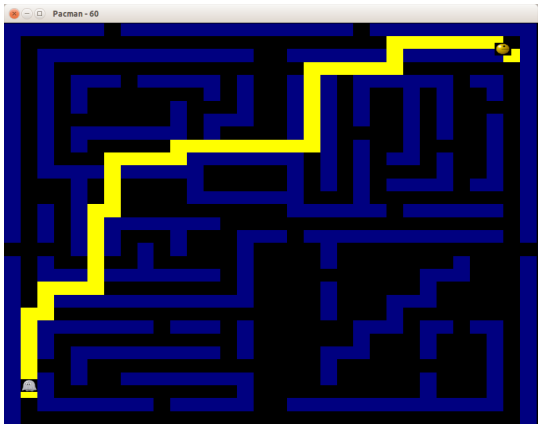
Nesta seção apresentamos imagens que descrevem o caminho encontrado pelo fantasma após a execução do algoritmo A*. Para cada cenário, apresentaremos sempre duas imagens: uma antes de reconstruir o caminho (removendo as ramificações) e uma outra apenas com o caminho ótimo, sem as ramificações.



(a) Caminho com ramificações (b) Caminho sem ramificações

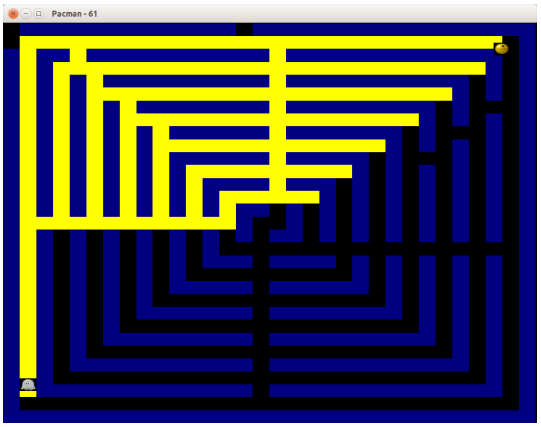
Figura 1: Cenário `board.txt`

(a) Caminho com ramificações

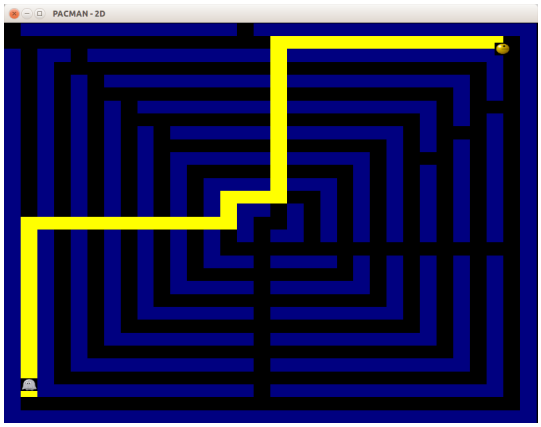


(b) Caminho sem ramificações

Figura 2: Cenário `board02.txt`

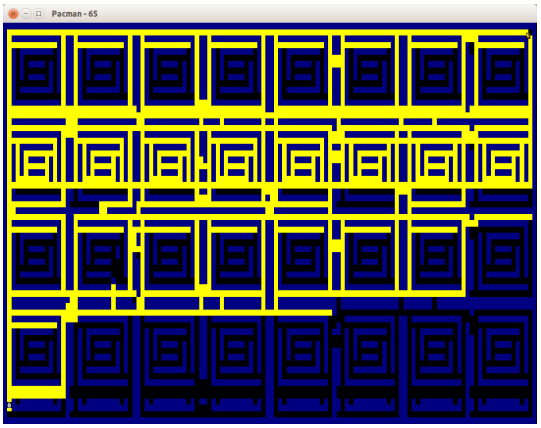


(a) Caminho com ramificações

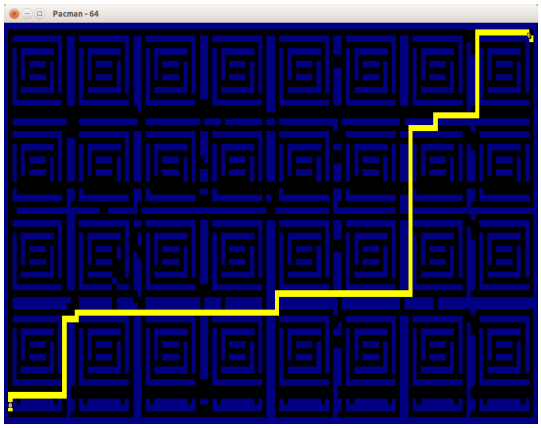


(b) Caminho sem ramificações

Figura 3: Cenário `board03.txt`

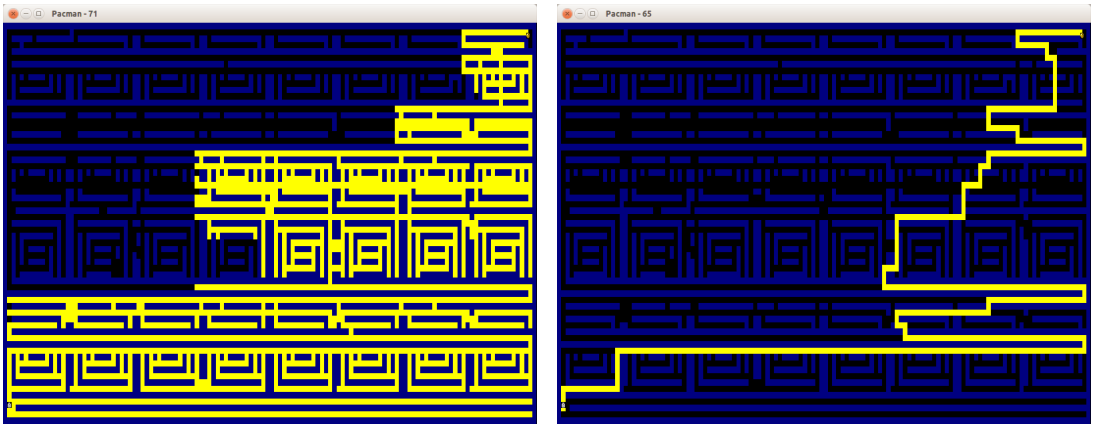


(a) Caminho com ramificações



(b) Caminho sem ramificações

Figura 4: Cenário `board03_a.txt`



(a) Caminho com ramificações (b) Caminho sem ramificações

Figura 5: Cenário `board03_b.txt`

4 Conclusão

Com base nas imagens mostradas na seção anterior, podemos perceber que o uso do algoritmo A* para o problema de busca do fantasma do *Pacman* utilizando como heurística a distância de Manhattan, produziu resultados bons quanto à exatidão, levando sempre ao fantasma localizar o menor caminhor para o *Pacman*.

Anexo

Implementação 1: `Enemy.cpp`

```
1  void Enemy::createOpenLists(stNode* start){
2
3          mTopOpen = (stNode*) malloc (sizeof(stNode));
4
5          mTopOpen->ptrFather = NULL;
6          mTopOpen->ptrChild = NULL;
7
8          mTopOpen->x = floor(start->x);
9          mTopOpen->y = floor(start->y);
10
11         mTopOpen->G = 0;
12         mTopOpen->H = fh(mTopOpen->x, mTopOpen->y);
13         mTopOpen->F = mTopOpen->G + mTopOpen->H;
14 }
15
16
17 bool Enemy::learningPath(stNode* start, stNode* goal)
18 {
19         mTopClose = NULL;
20         createOpenLists(start);
21
22         bool inOpenSet = false;
23         bool inClosedSet = false;
24         int score = 0;
25
26         int i, j;
27
28         stNode* current = NULL;
29         stNode** mNeighbors = NULL;
30         stNode* came_from[mScene->getWidth()][mScene->getHeight()];
31
32
33         for(i = 0; i < mScene->getWidth(); i++)
34                 for(j = 0; j < mScene->getHeight(); j++)
35                         came_from[i][j] = NULL;
36
37
38         while(mTopOpen != NULL)
39         {
40
```

```
41                  //get node in openset having the lowest f score
42                  current = createAndCopy(mTopOpen);
43
44                  if(isEqual(current, goal))
45                  {
46                          current->ptrFather = came_from[current->x][current->y];
47                          goal->ptrFather = current->ptrFather;
48
49                          return SUCCESS;
50                  }
51
52                  //remove current from openset
53                  removeList(&mTopOpen);
54
55                  /*came_from[current->x][current->y] is necessarilly in closed list and
                        not in open list.*/
56                  current->ptrFather = came_from[current->x][current->y];
57
58                  insert(&mTopClose, current);
59
60                  //for each neighbor of current
61                  mNeighbors = getNeighbors(current);
62
63
64                  for(i = 0; i < 4; i++)
65                  {
66                          if(mNeighbors[i]->x < 0 || mNeighbors[i]->y < 0 || mNeighbors[i
                                ]->x >= (mScene->getWidth()) || mNeighbors[i]->y >= (mScene->
                                getHeight()))
67                          {
68                                  free(mNeighbors[i]);
69                                  continue;
70                          }
71
72                          //if neighbor in closed set
73                          inClosedSet = inTheList(mTopClose, mNeighbors[i]->x, mNeighbors[i
                                ]->y) != NULL;
74                          if(inClosedSet)
75                                  continue;
76
77                          score = current->G + mScene->getPosition(mNeighbors[i]->x,
                                mNeighbors[i]->y) + 1;
78
79                          stNode* node = inTheList(mTopOpen, mNeighbors[i]->x, mNeighbors[i
                                ]->y);
80                          inOpenSet = node != NULL;
81
82                          if (!inOpenSet)
83                                  node = mNeighbors[i];
```

```
84
85
86                          if((!inOpenSet) || (score < node->G))
87                          {
88                                  came_from[node->x][node->y] = current;
89
90                                  node->G = score;
91                                  node->H = fh(node->x, node->y);
92                                  node->F = node->G + node->H;
93
94                                  if(!inOpenSet)
95                                  {
96                                          //add neighbor to open set
97                                          insertSorted(&mTopOpen, node);
98                                  }
99                          }
100                 }
101         }
102
103         return FAIL;
104 }
105
106
107 stNode** Enemy::getNeighbors(stNode* father)
108 {
109         stNode** mNeighbors = (stNode**) malloc (sizeof(stNode*) * 4);
110
111         short int i = 0;
112
113         for (i = 0; i < 2; i++)
114         {
115                 mNeighbors[i] = (stNode*) malloc (sizeof(stNode));
116                 mNeighbors[i]->ptrChild = NULL;
117                 mNeighbors[i]->ptrFather = NULL;
118                 mNeighbors[i]->G = (father->G) + 1;
119
120                 mNeighbors[i]->x = (father->x) + (i - 1);
121                 mNeighbors[i]->y = (father->y) + i;
122
123                 mNeighbors[i+2] = (stNode*) malloc (sizeof(stNode));
124                 mNeighbors[i+2]->ptrChild = NULL;
125                 mNeighbors[i+2]->ptrFather = NULL;
126                 mNeighbors[i+2]->G = (father->G) + 1;
127
128                 mNeighbors[i+2]->x = (father->x) + i;
129                 mNeighbors[i+2]->y = (father->y) + (i - 1);
130         }
131
132         return mNeighbors;
```

```
133 }
134
135
136 //Mahatan distance - Heurist function
137 int Enemy::fh(int x, int y){
138     return    (int) fabs(mAgent->X - (float)x) + fabs(mAgent->Y - (float) y);
139 }
140
141 /*Update position of Enemy*/
142 void Enemy::updated(float elapsedTime)
143 {
144         bool ll = false;
145
146         if((mTopClose == NULL) && (!mLearning))
147         {
148                 stNode* start = (stNode*) malloc (sizeof(stNode));
149                 stNode* goal = (stNode*) malloc (sizeof(stNode));
150
151                 //define start
152                 start->x = floor(this->X);
153                 start->y = floor(this->Y);
154                 start->G = start->H = start->F = 0;
155                 start->ptrChild = NULL;
156                 start->ptrFather = NULL;
157
158                 //define goal
159                 goal->x = floor(mAgent->X);
160                 goal->y = floor(mAgent->Y);
161                 goal->G = goal->H = goal->F = 0;
162                 goal->ptrChild = NULL;
163                 goal->ptrFather = NULL;
164
165                 if(mTopOpen != NULL)
166                 {
167                         destroyList(&mTopOpen);
168                         mTopOpen = NULL;
169                 }
170
171                 ll = learningPath(start, goal);
172
173                 if(ll)
174                 {
175                         printf("\nLearning with success!");
176
177                         //remove bifurcations
178                         buildPath(goal->ptrFather);
179                         //mScene->setCloseList(mTopClose);
180                 }
181                 else
```

```
182                         printf("\n:(");
183         }
184
185         //update postion of enemy
186         stNode* nextNode = mTopClose;
187
188         if(nextNode != NULL)
189         {
190                 /*Update position of enemy*/
191                 /*CODE HERE*/
192
193                         if(nextNode != NULL)
194                                 removeList(&mTopClose);
195
196                 check(X, Y);
197         }
198 }
```

Referências Bibliográficas

[1] A* search algorithm. Disponível em https://www.princeton.edu/~achaney/tmve/wiki100k/docs/A*_search_algorithm.html.

[2] Introduction to a*. Disponível em http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html.

[3] A* search algorithm. Disponível em http://en.wikipedia.org/wiki/A*_search_algorithm, Novembro 2014.

[4] Best-first search. Disponível em http://en.wikipedia.org/wiki/Best-first_search, Agosto 2014.

[5] Rajiv Eranki. Pathfinding using a* (a-star). Disponível em http://web.mit.edu/eranki/www/tutorials/search/, 2002.

[6] Sandro Ferreira Fernando Osório, Gustavo Pessin and Vinícius Nonnenmacher. Inteligência artificial para jogos: Agentes especiais com permissão para matar... e raciocinar! Technical report, PPG de Computação Aplicada, Universidade do Vale do Rios dos Sinos, São Leopoldo, RS - Brasil.