AI_Lab_01

PB21111686_赵卓

实验内容

- 用A*算法解决文档1.1的情景
- 用α-β剪枝算法对中国象棋进行决策

算法实现过程

- A*算法
  - 启发式函数
    - 对于这种只能水平或者竖直移动的情景，我们采取曼哈顿距离作为启发式函数，即水平距离和竖直距离的和。
    - admissible证明：由于只能水平或者竖直移动，因此最短的路程就是水平直线移动到目的地上方或者下方，然后竖直移动到目的地，即最短路程就是曼哈顿距离，永远不会高估消耗值。因此用曼哈顿距离作为启发式函数是admissible的。
    - consistent证明：假设当前点为s(x,y)，目的地为p(a,b)，当前进行一次移动到点q(x',y')，则c(s,q)=1，h(s)=|x-a|+|y-b|，h(q)=|x'-a|+|y'-b|，而q为(x+1,y)或(x-1,y)或(x,y+1)或(x,y-1)。则c(s,q)+h(q)=1+|x-a±1|+|y-b|或1+|x-a|+|y-b±1|≥|x-a|+|y-b|=h(s)，即该启发函数满足consistent性质。
    - 据此编写启发式函数代码如下：

```
int Heuristic_Funtion(pair<int, int> &end_point, Search_Cell *current)
{
 return abs(current->x - end_point.first) + abs(current->y - end_point.second);
 }
```

  - A*算法过程
    - 有了上述启发式函数，我们就可以方便地实现A*算法。具体来说，用一个open_list维护未访问结点，close_list维护已经访问的结点。open_list是以f+g，即当前路径耗散和启发式函数值优先的队列。这样我们用一个循环，每次从open_list中取出优先值访问，然后将这个结点的附近可行结点加入open_list，再将这个结点加入close_list。接着进行下一次循环，直到当前取出的结点是目的地即可。再注意对体力和路径的记录，对每个结点记录下到达它的路径，以及到达时的体力，每访问下一个结点体力减一，若到达补给站则体力加满。如果最后体力为0了，那就需要返回重新搜索，直到找到目的地。
    - 据此可编写代码如下：

```
while (!open_list.empty())
{
  Search_Cell *current = open_list.top();
  open_list.pop();
  visit[current->x][current->y] = 1;
  if (current->x == end_point.first && current->y == end_point.second)
  {
      step_nums = current->g;
      way = current->way;
      break;
  }
  if (current->energy <= 0)
      continue;
  vector<pair<int, int>> location =
  {{current->x + 1, current->y},
   {current->x - 1, current->y},
  {current->x, current->y - 1},
   {current->x, current->y + 1}};
  for (auto &loc : location)
  {
      int i = loc.first, j = loc.second;
      if (i >= 0 && i < M
      && j >= 0 && j < N
      && Map[i][j].type != 1
       && visit[i][j] == 0)
      {
          Search_Cell *next = new Search_Cell;
          next->x = i;
          next->y = j;
          next->g = current->g + 1;
          next->h = Heuristic_Funtion(end_point, next);
          if (Map[i][j].type == 2)
              next->energy = T;
          else
              next->energy = current->energy - 1;
          if (i == current->x + 1)
              next->way = current->way + "D";
          else if (i == current->x - 1)
              next->way = current->way + "U";
          else if (j == current->y + 1)
              next->way = current->way + "R";
          else if (j == current->y - 1)
              next->way = current->way + "L";
          open_list.push(next);
      }
  }
```

```
            close_list.push_back(current);
        }
```

- 效果分析
  如果将启发式函数设为0，那么就退化为一致代价搜索，这种情况下也能找到解，但是效率要低得多。A*算法会剪枝掉很多无效的搜素过程。

- α-β剪枝算法
  - 预备工作
    在进行α-β剪枝之前，我们首先要将预备工作完成，包括棋子合法动作的获取，棋盘分数评估，以及博弈树结点的完善。
    - 棋子合法动作
      以“相”棋的走法为例，“相”不能过河，而且只能走2×2的正方形对角线，并且当正方形中间有棋子时不能走，这和“马”棋的蹩马腿类似，我们也补全了框架中蹩马腿的缺失部分。知悉规则后，我们将当前“相”棋的四个到达位置列出，并且对应四个阻挡位。然后对四个位置遍历，如果阻挡位有棋子则不能走，没有则看到达位，如果到达位没有棋子或者是敌方棋子，则可行，如果是我方棋子则也不可行。其他棋子类似。据此可编写代码如下：

```
std::vector<Move> XiangMoves;
int dx[] = {2, 2, -2, -2};
int dy[] = {2, -2, 2, -2};
int wrong_x[] = {1, 1, -1, -1};
int wrong_y[] = {1, -1, 1, -1};

for (int i = 0; i < 4; i++)
{
  int nx = x + dx[i];
  int ny = y + dy[i];
  int wx = x + wrong_x[i];
  int wy = y + wrong_y[i];
  Move cur_move;
  cur_move.init_x = x;
  cur_move.init_y = y;
  cur_move.next_x = nx;
  cur_move.next_y = ny;
  if (color)
  {
      if (nx >= 0 && nx < 9 && ny >= 5 && ny <= 9 && board[wy][wx] == '.')
      {
          if (board[ny][nx] != '.')
          {
               bool cur_color = (board[ny][nx] >= 'A' && board[ny][nx] <= 'Z');
              if (cur_color != color)
                  XiangMoves.push_back(cur_move);
          }
          else
              XiangMoves.push_back(cur_move);
      }
  }
  else
  {
      if (nx >= 0 && nx < 9 && ny >= 0 && ny <= 4 && board[wy][wx] == '.')
      {
          if (board[ny][nx] != '.')
          {
              bool cur_color = (board[ny][nx] >= 'A' && board[ny][nx] <= 'Z');
              if (cur_color != color)
                  XiangMoves.push_back(cur_move);
          }
          else
              XiangMoves.push_back(cur_move);
      }
```

```
    }
  }
```

- 棋盘分数评估
  当前棋盘分数主要由两部分组成：棋力评估（即每个棋子在当前位置的棋力）和棋子价值评估（即每个棋子具有的价值，车的价值肯定比卒要高很多）。这部分比较简单，只需要将棋盘的棋子统计然后将不同棋子价值加起来，然后将棋盘的每个位置遍历一次得到棋力评估，相加得到MAX和MIN两方的价值。然后MAX-MIN即是分数评估值。
- 博弈树完善
  为了方便得到最终的move操作，在博弈树中添加了move记录得到这个结点进行的move操作。同时完善了updateBoard，实现子结点的更新。

```
GameTreeNode *updateBoard(std::vector<std::vector<char>> cur_board,
 Move move, bool color)
 {
     cur_board[move.next_y][move.next_x] = cur_board[move.init_y][move.init_x];
     cur_board[move.init_y][move.init_x] = '.';
     GameTreeNode *child_node = new GameTreeNode(color, cur_board, move);
     return child_node;
 }
```

- α-β减枝算法过程
  完成上述准备工作之后开始α-β减枝。算法很简单：从根节点开始，对子结点开始遍历，若当前结点是MAX结点，则更新α=max(α,α-β(child_node))；若当前结点是MIN结点，则更新β=min(β,α-β(child_node))。在α≥β时剪枝直接返回。据此可编写代码如下：

```
int alphaBeta(GameTreeNode *node, int alpha, int beta, int depth)
{
  if (depth == 0)
  {
      return node->getEvaluationScore();
  }
  else
  {
    ChessBoard board = node->getBoardClass();
    std::vector<Move> moves = board.getMoves(node->getcolor());
    std::vector<std::vector<char>> cur_board = board.getBoard();
    if (node->getcolor())
    {
        for (auto &move : moves)
        {
            GameTreeNode *child_node = node->updateBoard(cur_board,
            move, !node->getcolor());
            node->children.push_back(child_node);
            alpha = max(alpha, alphaBeta(child_node, alpha, beta, depth - 1));
            if (alpha >= beta)
                break;
        }
        node->setEvaluationScore(alpha);
        return alpha;
     }
      else
      {
        for (auto &move : moves)
        {
            GameTreeNode *child_node = node->updateBoard(cur_board,
            move, !node->getcolor());
            node->children.push_back(child_node);
            beta = min(beta, alphaBeta(child_node, alpha, beta, depth - 1));
            if (alpha >= beta)
                break;
        }
        node->setEvaluationScore(beta);
        return beta;
      }
  }
return 0;
}
```

实验结果

- A*算法结果符合预期，可见于output中。如果将启发式函数设为0，可以找到结果，但是效率会低很多。
- α-β剪枝算法我们设置搜索深度为4，根据棋局对比，输出结果效果很好，可以在当前棋局找到比较明显的优秀选择走法，具体内容见output。α-β剪枝减去了很多无效的搜索，大大提高了效率。