人工知能 09.29 課題

5CS47 藤岡碧志

1　課題概要

今回の課題は A*アルゴリズムを用いて 8 パズル問題を解くことである。与えられたパズルの初期状態およびゴール状態を以下に示す (空欄を-1 として表現している)。

$$Init = \begin{bmatrix} 8 & 1 & 5 \\ 2 & -1 & 4 \\ 6 & 3 & 7 \end{bmatrix}, Goal = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & -1 \end{bmatrix}$$

A*アルゴリズムにおける発見的関数 $f(p)$ は以下のように定義される。

$$f(p) = g(p) + h(p)$$

$$\begin{cases} g(p) : \text{初期状態から現状態 } p \text{ までのタイルを動かした回数} \\ h(p) : \text{現状態 } p \text{ において個々のタイルの目標位置までのマンハッタン距離の総和} \end{cases}$$

パズルを解いていく流れを以下に示す。

1. 現在の状態 (初期状態) から次状態候補を生成する
2. 次状態候補の各々について発見的関数値を計算して求める
3. 発見的関数値が最小のものを選択し次の状態として決定する

この一連の操作を繰り返し、探索を行うことで解を求める。

2　アルゴリズムの説明

作成したプログラムのアルゴリズム及び処理フローについて説明を行う。

2.1　フローチャート

フローチャートを図１に示す。

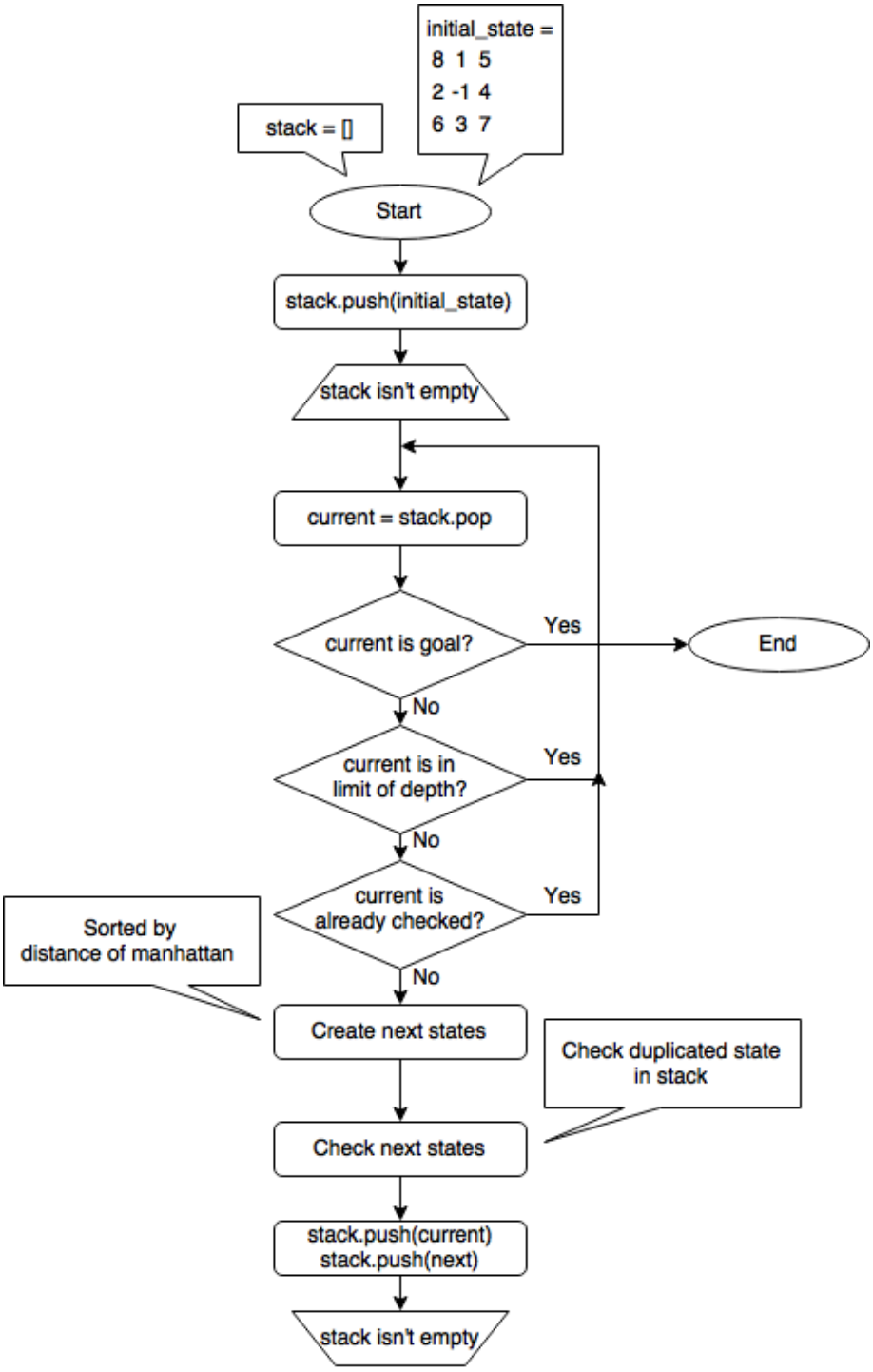

図１　８パズル問題 フローチャート

2.2 説明

作成したプログラムでは、スタックを用いた深さ優先探索を行う。フローチャート (図 1) を参考に、実装したアルゴリズムの処理手順を以下に示す。() 内の要素がフローチャートと対応している。

1. 初期状態と空のスタックを用意して開始する (Start)
2. 初期状態をスタックへ格納する (stack.push)
3. スタックから状態を 1 つ取り出し、現在の状態 (以下現状態) とする (stack.pop)
4. 現状態がゴール状態と等しければ終了、そうでなければ続ける (goal?)
5. 現状態が探索木の下限まで達してしまったなら 3. へ戻る (limit depth?)
6. 現状態がすでに探索済みであれば 3. へ戻る (already checked?)
7. 現状態から取りうる次状態候補群を生成する (Create next states)
    - 次状態候補群はマンハッタン距離についてソートしておく (Sorted by manhattan)
8. 次状態候補群がスタックに含まれているものと重複していないか確認する (Check next states)
    - 重複していた場合、探索済みとする
9. 現状態、次状態候補群の順でスタックに格納する (stack.push × 2)
    - 次状態候補群はマンハッタン距離の大きいものから順に格納する
10. 3. から 9. までの手順をスタックが空でない限り繰り返す (stack isn't empty)

各処理の詳細はソースコードにあるコメントに記載することとする。

3　ソースコード

以下に作成したプログラムのソースコードを示す。探索処理の本体は 171 行目 `operate_stack(root, lim_depth)` であり、図 1 のフローチャートはこの部分の詳細説明となっている。

ソースコード 1　eight_puzzle.rb

```ruby
# 3x3の8パズル問題を解くためのモジュール
module EightPuzzle
  # 各状態を保持するためのノードとなるクラス
  #
  # 各ノードは単一の親ノードの情報を持ち、子ノードの情報は持たない
  class Node
    attr_accessor :board, :parent, :visited

    # ノードの生成
    def initialize(board)
      @board    = Board.new(board)
      @parent   = nil
      @visited  = false
    end

    # 親とするノードの設定
    def add_parent(parent)
      self.parent = parent
    end

    # 現在のノードの位置からルートノードまでの距離(深さ)を求める
    def depth
      return 1 if root?
      1 + parent.depth
    end

    # ルートノードかどうか
    def root?
      parent.nil?
    end

    # ノードを探索したことのフラグ付けを行う
    def visit
      self.visited = true
    end

    # ファイル出力用表示の定義
    def outs
      <<~OUTS
      Current : #{board.state[0]}
                #{board.state[1]}
                #{board.state[2]}
      Evaluated value : #{board.dist + depth}
      ========================
      OUTS
    end

    # デバッグ向け簡易表示
    def to_s
      "Board : #{board.state}, Dist : #{board.dist}, Depth : #{depth}"
    end
  end

  # パズルの状態を保持するためのクラス
  class Board
    attr_accessor :state, :dist

    # パズル盤の生成
    def initialize(state)
      @state = state
      @dist  = 0

      manhattan
    end

    # 空欄となっている場所のインデックスを(x, y)で返す
    def index_blank
      state.each_with_index do |row, y|
        row.each_with_index do |fig, x|
          return [x, y] if fig == -1
        end
      end
```

```ruby
    end

    # 現状態とゴール状態との間のマンハッタン距離を求める
    #
    # ゴール状態をあらかじめ設定してあるため、一般解としては運用できない
    def manhattan
      self.dist = 0
      state.each_with_index do |row, y|
        row.each_with_index do |fig, x|
          case fig
          when 1 then self.dist += (y + x)
          when 2 then self.dist += (y + (x - 1).abs)
          when 3 then self.dist += (y + (x - 2).abs)
          when 4 then self.dist += ((y - 1).abs + x)
          when 5 then self.dist += ((y - 1).abs + (x - 1).abs)
          when 6 then self.dist += ((y - 1).abs + (x - 2).abs)
          when 7 then self.dist += ((y - 2).abs + x)
          when 8 then self.dist += ((y - 2).abs + (x - 1).abs)
          when -1 then next
          end
        end
      end
    end

    # 状態の複製をする
    #
    # 参照渡しである配列を2次元で表現しているためコピーに工夫が必要
    def duplicate
      dup_state = []
      state.each do |row|
        dup_state << row.dup
      end
      dup_state
    end

    # 指定したインデックスの要素を空白と入れ替えた状態を返す
    def move(x, y)
      new_state = duplicate
      blank_x, blank_y = index_blank
      new_state[y][x], new_state[blank_y][blank_x] = new_state[blank_y][blank_x],
          new_state[y][x]
      new_state
    end

    # ゴール状態となっているか
    #
    # マンハッタン距離が0であればパズルが解けている
    def goal?
      dist.zero?
    end
  end

  # パズルを解くためのソルバ
  #
  # 処理をまとめたもの
  module Solver
    # 次状態候補を生成する
    #
    # 現状態のノードに探索したことのフラグ付けも行う
    def self.make_next_states(node)
      node.visit
      next_states = []
      b_x, b_y = node.board.index_blank
      next_states << Node.new(node.board.move(b_x, b_y - 1)) unless (b_y - 1) < 0
      next_states << Node.new(node.board.move(b_x - 1, b_y)) unless (b_x - 1) < 0
      next_states << Node.new(node.board.move(b_x + 1, b_y)) unless (b_x + 1) > 2
      next_states << Node.new(node.board.move(b_x, b_y + 1)) unless (b_y + 1) > 2
      next_states
    end

    # 次状態候補をマンハッタン距離の降順となるようにソートして返す
    def self.next_states_ordered_desc(node)
      next_states = make_next_states(node)
      next_states.sort_by { |n| n.board.dist }.reverse
    end

    # 次状態候補に親ノードの設定を含めたものを返す
    def self.next_states_parent_as(parent)
      next_states = next_states_ordered_desc(parent)
      next_states.each { |n| n.add_parent(parent) }
      next_states
    end

```

```ruby
    # 探索点を移動する前に、選択候補の状態が重複した動きになっていないか確認する
    #
    # スタックの中に同じ盤面のものが含まれていれば探索済みとする
    def self.next_states_check_moving(next_states, stack)
      next_states.each do |ns|
        ns.visit if stack.any? { |st| st.board.state == ns.board.state }
      end
    end

    # 次状態候補をスタックへ入れる
    def self.push_next_states(next_states, stack)
      next_states.each { |n| stack.push(n) }
    end

    # 深さ優先探索をスタックを用いて行う
    #
    # 実行した時間をファイル名とした結果ファイルの出力も行う
    def self.operate_stack(root, lim_depth)
      stack = []
      stack.push(root)
      timestamp = Time.now.to_i.to_s
      until stack.empty?
        current = stack.pop
        outs_current_to(timestamp, current)
        if current.board.goal?
          stack.push(current)
          break
        end
        next if current.depth >= lim_depth
        next if current.visited
        next_states = next_states_parent_as(current)
        next_states_check_moving(next_states, stack)
        stack.push(current)
        push_next_states(next_states, stack)
      end
      stack.last
    end

    # 得た解手順の表示
    def self.puts_trace(node_solved)
      if node_solved.nil?
        puts 'Unsolved'
        return
      end
      puts_trace(node_solved.parent) unless node_solved.root?
      puts node_solved.outs
    end

    # 探索内容をファイルへ出力させる
    def self.outs_current_to(filename, current)
      `echo "#{current.outs}" >> #{filename + '.result.txt'}`
    end

    # 以下デバッグ用表示処理 #

    # スタックの中身の表示
    def self.puts_stack(stack)
      print "Stack : \n"
      stack.each { |n| print "    #{n}\n" }
    end

    # 現在の状態から選択されうる次状態の表示
    def self.puts_next_states(next_states)
      print "Nexts : \n"
      next_states.each { |n| print "    #{n}\n" }
    end

    # デバッグ用表示処理終わり #
  end

  # 8パズルを解く
  #
  # 全体の処理をmainから1行で呼ぶためのエイリアス
  # 探索木の深さをlim_depthとする
  def self.from_init_to_solve(puzzle, lim_depth)
    root = Node.new(puzzle)
    last = Solver.operate_stack(root, lim_depth)
    Solver.puts_trace(last)
  end
end

########## main ##########
EightPuzzle.from_init_to_solve([[8, 1, 5], [2, -1, 4], [6, 3, 7]], 100)
```

4　実験結果

4.1　実行環境

プログラムの作成と実行に利用した環境を以下に示す。

- macOS Sierra 10.12.6
- ruby 2.4.0
- zsh 5.4.1

4.2　パズルの状態推移

上述したソースコードを実行することで `(timestamp).result.txt` というテキストファイルが得られる。このファイルには以下の内容が出力される。

- 現在のパズルの状態 (盤面)
- 現在の盤面における発見的関数の値 (探索点の深さ ＋ マンハッタン距離)

表示形式は 38 行目の `outs` に、ファイルへの出力処理は 203 行目の `outs_current_to(filename, currrent)` に定義されている。

4.3　発見的関数の値とその推移

探索回数に対する発見的関数の値をグラフで表したものを図 2 に示す。

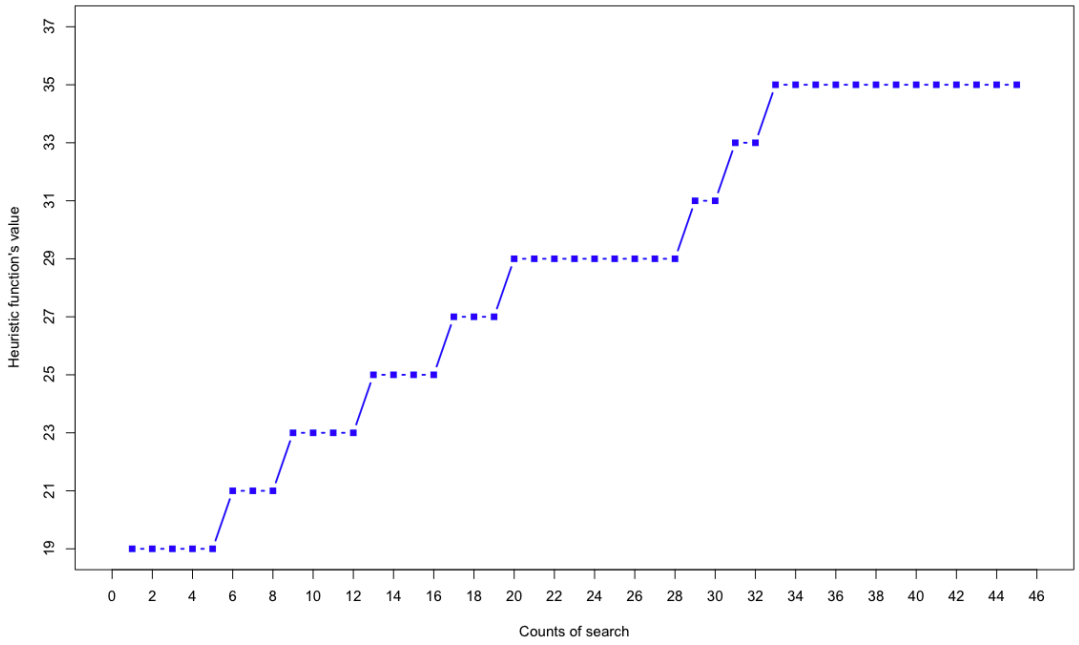

図 2　発見的関数値とその推移