人工知能 課題番号 12「パズルの探索プログラム」

工学部電子情報工学科 03-175001 浅井明里

2017 年 11 月 16 日

1　今回実装したパズルの探索プログラムについて

今回は数独及び迷路の解法を幅優先探索、深さ優先探索、A*探索で求めるパズルの実装を行なった。

数独は 3 × 3 の小さいマスが一つもしくは縦横同数に複数個連結された正方形の枠内に 1 から 9 までの数字を入れるパズルの一つであり、縦列、横列、また同じ 3 × 3 のマス目にある数字が複数回登場してはならないという制約条件を満たさなくてはいけない。また、開始時点にあらかじめいくつかの数字は既に記入されていることとする。図 1 に問題例を示す。

迷路については障害物を回避しながら、左上のスタート地点から右下のゴール地点に到達するための適切な経路を探索する。図 2 に問題例を示す。

```
Testing on 9x9 board...
Problem:
[0, 0, 0, 8, 4, 0, 6, 5, 0]
[0, 8, 0, 0, 0, 0, 0, 0, 9]
[0, 0, 0, 0, 0, 5, 2, 0, 1]
[0, 3, 4, 0, 7, 0, 5, 0, 6]
[0, 6, 0, 2, 5, 1, 0, 3, 0]
[5, 0, 9, 0, 6, 0, 7, 2, 0]
[1, 0, 8, 5, 0, 0, 0, 0, 0]
[6, 0, 0, 0, 0, 0, 0, 4, 0]
[0, 5, 2, 0, 8, 6, 0, 0, 0]
```

図 1　数独問題の例

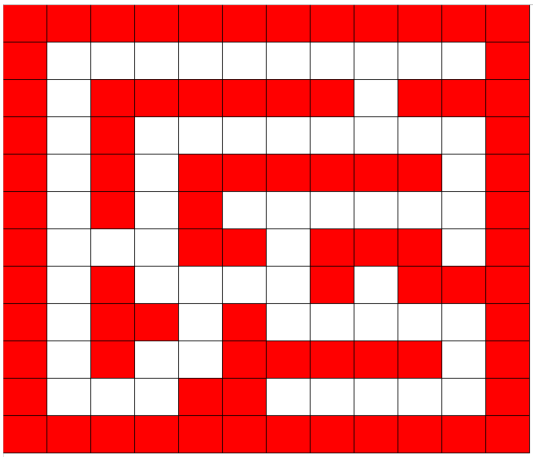

図 2　迷路問題の例

1.1　数独の探索アルゴリズム

数独の探索プログラムについては sudoku ディレクトリ以下にまとめて配置されており、以下のコマンドをターミナルで実行することにより、幅優先探索、深さ優先探索 A*探索の実行及び結果の確認ができる。

```
$ python3 sudoku.py
```

1.1.1　幅優先探索アルゴリズムの実装について

幅優先探索では解を求めるために木もしくはグラフの全てのノードを網羅的に探索、走査するするアルゴリズムであり、以下の手順で探索を行う。

1. 初めのノードを空の Queue に追加する。
2. ノードを Queue の最初から取り出し、ノードが得たい結果であれば探索をやめて結果を返すが、そうでなければノードの全ての未探索の子を Queue に追加する。
3. Queue が空になればグラフの全てのノードを探索したことを意味するため、探索をやめる。

以下に数独問題における幅優先探索のプログラムを示す。node.expand(problem) により、あるノードから数独問題の制約条件を満たし、実現可能なノードを得ることができ、仮にこれらの子ノードの一つが目標状態を実現していれば、すなわち全てのマスが 1 から 9 までの数字で埋められて入れば探索をやめ、そうでなければ全ての子ノードを Queue である frontier に追加してる。

```
def BFS(problem):
    # Create initial node of problem tree holding original board
    node = Node(problem.initial)
    # Check if original board is correct and immediately return if valid
    if problem.goal_test(node.state):
        return node

    frontier = Queue()
    frontier.put(node)

    visited_node_num = 0
    while (frontier.qsize() != 0):
        node = frontier.get()
        for child in node.expand(problem):
            visited_node_num += 1
            if problem.goal_test(child.state):
                print ("Found solution")
                print("The number of visited nodes : " + str(visited_node_num))

                return child
            frontier.put(child)
    return None
```

1.1.2 深さ優先探索アルゴリズムの実装について

深さ優先探索は木もしくはグラフのの最初のノードから、目的のノードが見つかるか、葉 (子のないノード) に行く着くまで深く探索をすすめ、その後はバックトラックし、最も近くの探索の終わっていないノードまで戻る。ノードは Stack に格納される。

1. 初めのノードを空の Stack に積む。
2. ノードを Stack の最初から取り出し (一番最近追加したノードが取り出される)、Node が得たい結果であれば探索をやめて結果を返すが、そうでなければノードに接続した子ノードをスタックに積み、これ以上子ノードを持たなくなるまで深く探索を進める。
3. 子ノードを持たないノードに到達した際には、未訪問のノードまで戻る
4. Stack 空になればグラフの全てのノードを探索したことを意味するため、探索をやめる。

以下に数独問題における深さ優先探索のプログラムを示す。最初のノードをスタックに追加し、スタックの一番上のノードを取り出してそれが目標状態であれば探索をやめ、そうでなければそのノードから到達可能な小ノードをスタックについてし、目標状態に到達するもしくはスタックが空になるまで探索を続ける。

```
def DFS(problem):
      initial_node = Node(problem.initial)
      if problem.goal_test(initial_node.state):
              return initial_node.state

      stack = []
```

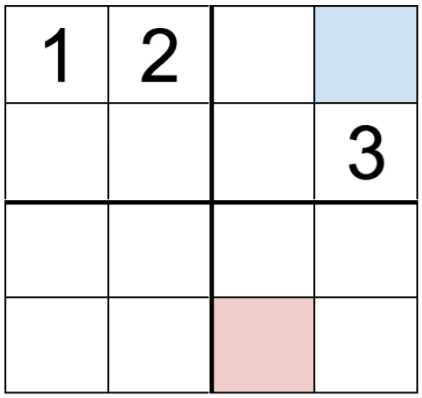

図 3　Minimum Remaining Values の例

```
        stack.append(initial_node)

        visited_node_num = 0
        while stack:
                node = stack.pop()
                visited_node_num+=1
                if problem.goal_test(node.state):
                        print ("Found solution")
                        print("The number of visited nodes : " + str(visited_node_num))
                        return node.state
                stack.extend(node.expand(problem))

        return None
```

1.1.3　A*探索アルゴリズムの実装について

A*探索アルゴリズムとは、グラフ探索アルゴリズムの一つであり、グラフ上でスタートからゴールまでの道を見つけるというグラフ探索問題において、ヒューリスティック関数を用いて探索を行うアルゴリズムである。ヒューリスティック関数をどう設定するかは問題によって異なる。数独問題を解く上でヒューリスティックとなりうるとされるものに Minimum Remaining Values, Constraint Propagation 等があるが、今回は Minimum Remaining Values をヒュースティックと用いた。

Minimum Remaining Values とは、ある状態から次に埋めるべきマス目を選ぶ際、ランダムもしくはインデックス順で行うのではなく、「制約上、選択可能な数字が最も小さいものから順に、すなわち制約の厳しいものから順に選択する」というものである。図 3 の例を用いて Minimum Remaining Values について説明する。図において、青いセルは行で見たときに既に 1, 2 が使用されており、またブロックで見たときに既に 3 が使用されており、制約を満たす選択肢は 4 のみである。一方、赤色のセルは行、列、ブロックいずれで見ても他の数字がまだ使用されておらず、1,2,3,4 の全ての数字が利用可能であり、制約を満たす選択肢は 4 つ存在する。よって青のセルの方が remaining values が小さくなるために、次に選択されるのは青のセルとなる。

A*探索法については、Minimum remaining values をヒューリスティックとし、次に埋めるセルを選択する際、これが最小となるものを次のセルとして選択、有効な数字のうち最小のものでそのセルを埋めるという方式をとった。

```
# Calcumate minimum remaining values for each cell.
def heuristic(self, board, state):
    min_remainings = [
        [self.type + 1 for i in range(1, self.type + 1)] for i in range(1, self.type + 1)]
    for row in range(board):
        for column in range(board):
            if state[row][column] == 0:
                valid_nums = self.filter_quad(self.filter_column(
                    self.filter_row(state, row), state, column), state, row, column)
                min_remainings[row][column] = len(valid_nums)
    return min_remainings

# Get an empty spot with a heuristic.
def get_spot(self, board, state):
    min_remainings = self.heuristic(board, state)

    minimum = np.array(min_remainings).min()
    for row in range(board):
        for column in range(board):
            if min_remainings[row][column] == minimum and state[row][column] == 0:
                return row, column

def AStar(problem):
    initial_node = Node(problem.initial)
    if problem.goal_test(initial_node.state):
        return initial_node.state

    stack = []
    stack.append(initial_node)

    visited_node_num = 0
    while stack:
        node = stack.pop()
        visited_node_num += 1
        if problem.goal_test(node.state):
            return node.state
        stack.extend(node.expand(problem))

    return None
```

1.2 迷路の探索アルゴリズム

迷路の探索プログラムについては、maze ディレクトリ以下、主に maze_solver.py というファイルに幅優先探索、深さ優先探索、A*探索のプログラムが記述されており、以下のコマンドで実行及び結果の確認ができる。

```
$ python3 maze.py
```

この迷路においてグラフは辞書型で表されており、i 行 j 列のセルに隣接するセルの情報がこのグラフ辞書の (i, j) というキーに対応した値となる。またこの値は隣接するセルのインデックスと接続している邦楽のタプルで与えられる。例えば図 4 のような迷路をグラフに落とし込むとする。1 行 0 列のマスは南北のセルからは行き来が可能だが、東西側は両方とも行き止まりないしは壁になるため、隣接しているとは言えない。よっ

図 4　迷路のグラフ表現の例

てこのセルのグラフ辞書での値は以下の通りになる。

```
  graph[1][0] = [("S", (2, 0)), ("N", (0, 0))]
```

このように構成されたグラフ辞書に基づいて、探索を行う。

1.2.1　幅優先探索アルゴリズムの実装について

幅優先探索については、sudoku の場合と同様、Queue を使用し、あらかじめ設定された goal 地点に到達すれば終了、そうでなければ Queue から新たに取り出し、取り出されたノードから移動可能な全てのノードを新たに Queue に追加しこれを Queue が空になるまで続ける。

```
def solve_bfs(maze, graph):
    start, goal = (1, 1), (len(maze) - 2, len(maze[0]) - 2)
    queue = deque([("", start)])
    visited = set()
    while queue:
        path, current = queue.popleft()
        if current == goal:
            return path
        if current in visited:
            continue
        visited.add(current)
        for direction, neighbour in graph[current]:
            queue.append((path + direction, neighbour))
    return "Cannot find any way."
```

1.2.2　深さ優先探索アルゴリズムの実装について

深さ優先探索プログラムも数独問題と同様、Stack を利用し、Stack が空になるもしくはゴールに到達するまで探索を続ける。

```
def solve_dfs(maze, graph):
    start, goal = (1, 1), (len(maze) - 2, len(maze[0]) - 2)
    stack = deque([("", start)])
    visited = set()
    while stack:
        path, current = stack.pop()
```

```
        if current == goal:
            return path
        if current in visited:
            continue
        visited.add(current)
        for direction, neighbour in graph[current]:
            stack.append((path + direction, neighbour))
    return "Cannot find any way."
```

1.2.3 A*探索アルゴリズムの実装について

迷路問題における A*探索アルゴリズムは、ダイクストラ法を用いてスタート地点からゴール地点へ最も少ないコストで到達可能な経路を見つける音を目指す。実相においてはあるセルからゴール地点までの距離をヒューリスティックとして用い、Priority Queue に可能な経路の情報を追加、削除していく。

```
def heuristic(cell, goal):
    return abs(cell[0] - goal[0]) + abs(cell[1] - goal[1])

def solve_astar(maze, graph):
    start, goal = (1, 1), (len(maze) - 2, len(maze[0]) - 2)
    pr_queue = []
    heappush(pr_queue, (0 + heuristic(start, goal), 0, "", start))
    visited = set()
    while pr_queue:
        _, cost, path, current = heappop(pr_queue)
        if current == goal:
            return path
        if current in visited:
            continue
        visited.add(current)
        for direction, neighbour in graph[current]:
            heappush(pr_queue, (cost + heuristic(neighbour, goal), cost + 1,
                                path + direction, neighbour))
    return "Cannot find any way."
```

2 探索の効率性に関する結果と考察

2.1 数独問題についての結果の考察

4 つの数独問題について、探索法ごとに正解に到達するまでに探索したノードの数を比較したものが次の表 1 である。 表から数独問題のサイズが大きくなるほど、より多くの手が想定できるために、幅優先探索、深さ優先探索では指数的に探索するノードの数が増加していることがわかり、特に幅優先探索についてはこの傾向が顕著であることがわかる。一方 A*探索は問題のサイズが大きくなったとしても探索するノードの数は他の二つと比較してそこまで増加はしておらず、ヒューリスティックを用いいることにより、より効率的な探索・走査ができていることがわかる。

2.2 迷路問題についての結果の考察

迷路問題については、次の二つの問題について幅優先探索、深さ優先探索、A*探索により解法を求めた。どちらの問題についても、深さ優先探索と A*探索問題で得られた経路は一致し、また幅優先探索の導いた経路

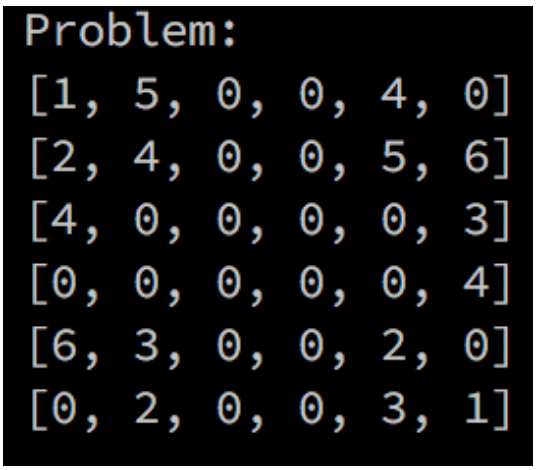

図 5　数独 問題 1

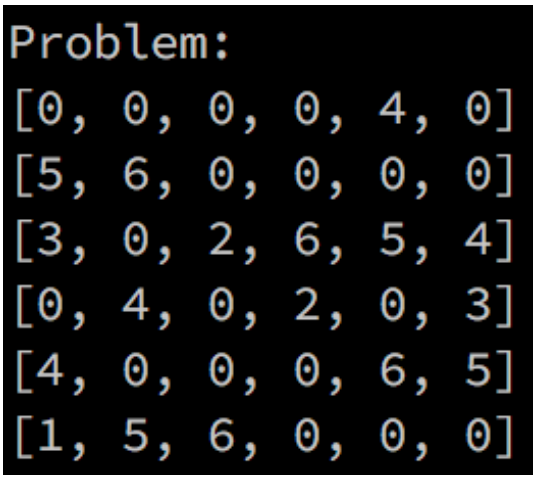

図 6　数独 問題 2

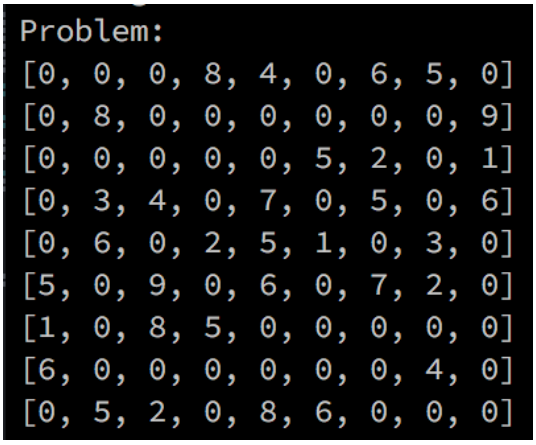

図 7　数独 問題 3

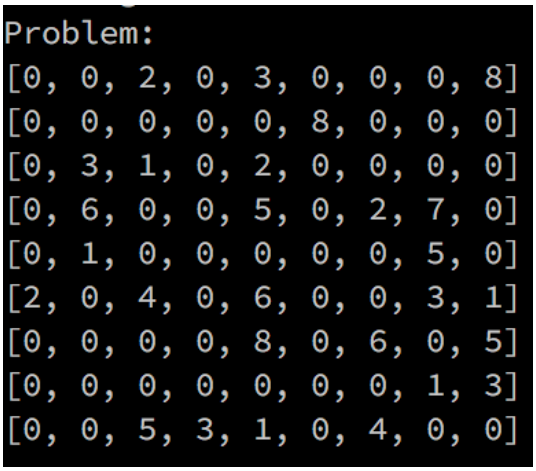

図 8　数独 問題 4

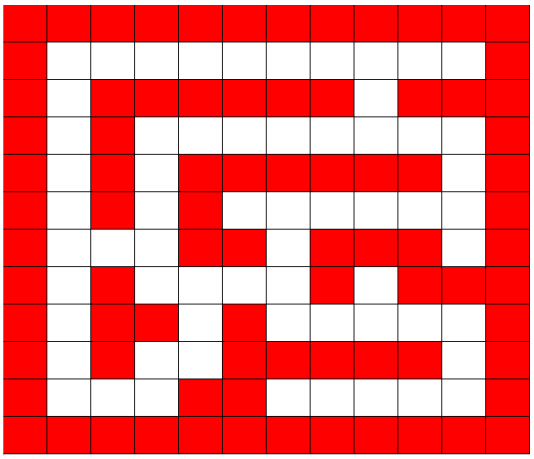

図 9　迷路問題 1

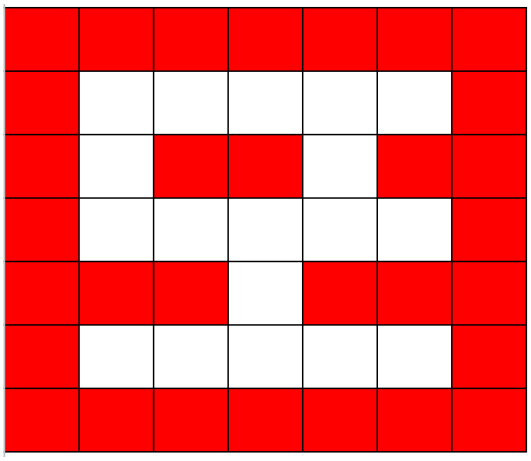

図 10　迷路問題 2

表 1　DFS, BFS, A*により探索したノード数比較

問題番号	幅優先探索	深さ優先探索	A*探索
1	78	23	21
2	48	20	20
3	1932	939	49
4	285352	16427	410

と比較し、問題 1 は距離が 7、問題 2 は距離が 2、幅優先探索の結果得られた経路より短くなった。問題 1 の解法は図 11 及び図 12、問題 2 の解放は図 13 及び 14 で確認できる。

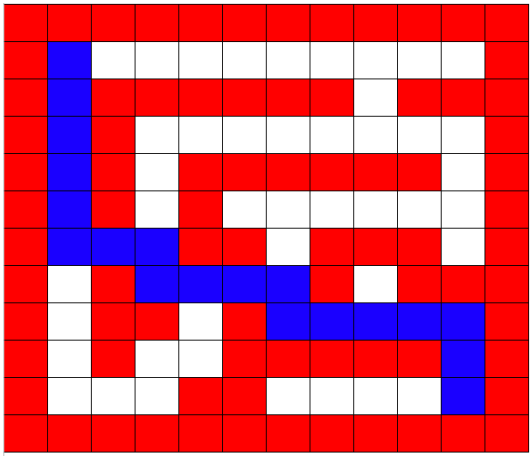

図 11　迷路問題 1 深さ優先探索、A*探索の結果得られた経路

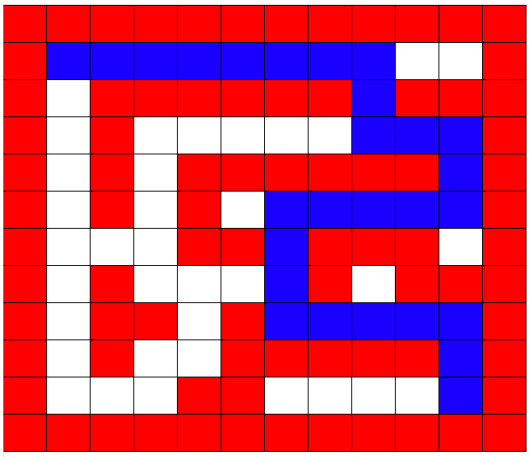

図 12　迷路問題 1 幅優先探索の結果得られた経路

この経路を得るために探索したノードの数は以下の通りである。表 2 より結果として深さ優先探索は最短経路を求められてはいる一方で、探索するノードの数が迷路のサイズの上昇とともに他の二つの探索法を比較して、大きく増加していることがわかる。一方 A*探索は深さ優先探索と同様の解を導きつつも、迷路のサイズが増加しても探索数はさほど増加せず、大きく、かつある程度分岐可能性の存在する迷路においては、ヒューリスティックを活用した A*探索法がより効率的により短い経路を探索するのに有効であることがわかる。

表 2　迷路問題において DFS, BFS, A*により探索したノード数比較

問題番号	幅優先探索	深さ優先探索	A*探索
1	38	120	43
2	17	35	17

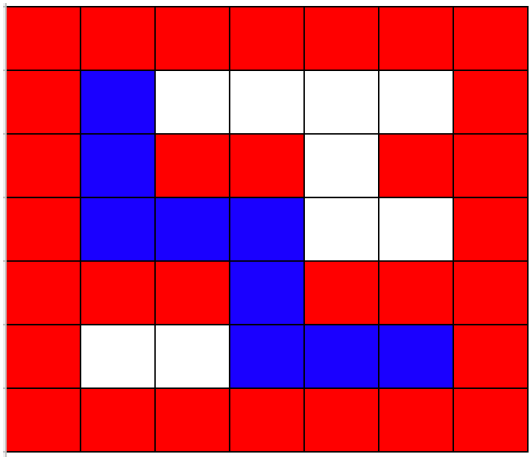

図 13　迷路問題 2 深さ優先探索、A*探索の結果得られた経路

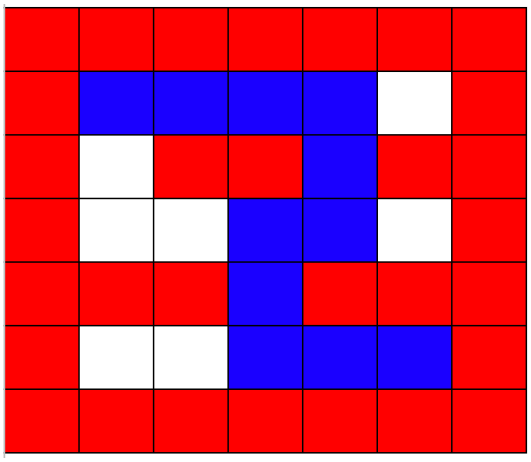

図 14　迷路問題 2 幅優先探索の結果得られた経路

参考文献

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein. ”Introduction to Algorithms”, MIT Press, 2009.

[2] 伊庭斉志, 『人工知能と人工生命の基礎』, オーム社 , 2013.