2023年12月17日

人工知能
中間レポート

所属：工学部電子情報工学科
学籍番号：03230420
氏名：小林莉久

概要と目的

まず、今回は課題(4)の川渡り問題の解の探索に取り組んだ。このレポートでは、解の探索に用いたプログラムの概要と、工夫した点の解説、また、ヒューリスティックの試行錯誤の過程と、探索結果についてまとめる。

実装したプログラムについて

ソースコードと、実行結果のテキストファイルについては添付のファイルに示しているため、ここではあらためて掲載することはせず、今回実装したクラスとメソッドの概要と工夫した点について述べる。

1. **全体の実装方針**

   今回は、pythonのclassを用いて、解を探索するためのメソッドを、各小問ごとに定義して、再利用可能にした。

2. **initメソッドについて**

   夫婦のペア数をNとしたとき、各ノード（盤面）の状態を、N×2 + 4の長さのリストで保持することにした。最初のN要素は、夫の現在位置（左岸：0、真ん中の島：1、右岸：2）、続くN要素は嫁の現在位置とした。各夫婦内でのindexの差は常にNとなっている。続く、N×2+1番目の要素はボートの現在位置とした。以上により、各夫婦及びボートの位置を数値的に一元管理することを可能にした。これはA*サーチにおいてヒューリスティックスを考える時に役立つ。続いて、N×2+2番目の要素はA*サーチのためのf

値、その次には深さを示すd値を保持する。これによって、各ノードのf値と深さを簡単に参照することができる。最後の要素は、直前にたどったノードの、self.closedというリストにおけるindexであり、この値によって、最終的に発見した解においてたどったノードを順番に出力することができる。

3. **forwardメソッドについて**

   このメソッドはopenのリストからある一つのノードを取り出して、そこから次に探索可能なノードを全て見つけopenリストに追加するものである。また、仮に次の一手でゴールに辿り着く場合、そこで探索を終了させる機能も有する。

   このメソッド内では、初めに引数modeによって、３種類の探索方法を選択できるようにすることで、探索の実行を容易にした。次に探索可能な全ノードの確認については、まずボートの位置によって分岐させ、ボートと同じ位置にいる人を１人か２人、全通り取り出して、抽出した人が他の場所に移動することが許されるかどうかをチェックする、というアルゴリズムで実装した。ここで、条件を満たした場合、人とボートの位置の移動に加えて、f値やd値、親ノードのindexも計算して盤面の状態として追加している。また、前述の通りゴールに辿り着けば、その時点でそれ以上の探索を取りやめる。

4. **calc_hメソッドについて**

   このメソッドでは、盤面の状態に基づいて、A*サーチのためのh値を計算する。ここで用いたヒューリスティクスについては、後に詳しく述べる。

5. **searchメソッドについて**

   ここでは、forwardメソッドを繰り返し用いることによって、ゲームの設定、夫婦数、探索方法に基づいて、解を探索する。

6. **他のメソッドについて**

   前述のforwardメソッド、calc_hメソッド、searchメソッドとほとんど同じ構造で、細部だけ、問題の設定に合わせたメソッドを追加で２セット作った。これらは、真ん中の島を経由することが許される(2)の問題、左岸から右岸への直接の移動を少なくとも1回含むことが必要な(3)の問題に対応している。(3)についてはゴールに辿り着いた時に、check_routeメソッドを呼び出すことで、直接移動を含んでいるかどうかを判定し、含んでいなければ探索を続ける、という処理を追加している。

探索結果

1. **(1)について**

   初めにmain関数を以下のように設定した。

```
def main():

   newPuzzle = Puzzle(3)

   newPuzzle.search("deep")

   newPuzzle.search("wide")

   newPuzzle.search("A")
```

   続いて、ターミナル上で以下のコマンドを実行した。

```
python main.py > result_01.txt
```

   その結果から、展開数は少ないが最終的な解の深さが大きくなってしまう深さ優先探索、解の深さは小さいが展開数が大きくなってしまう幅優先探索という、二つの手法のいいとこ取りをした結果がA*サーチとなっており、展開数、解の深さともに、３つの手法の中で最小になっていることが読み取れる。

2. **(2)について**

   初めにmain関数を以下のように設定した。

```
def main():

   newPuzzle = Puzzle(4)

   newPuzzle.search("wide")
```

   続いて、ターミナル上で以下のコマンドを実行した。

```
python main.py > result_without_island.txt
```

その結果から44個のノードを探索した上で、解が見つからなかったことが確認できる。

次に、真ん中に島があれば解が存在することを確認するために、島がある条件下で、深さ優先探索と、幅優先探索を試す。

まず初めに、main関数を以下のように設定した。

```
def main():

   newPuzzle = Puzzle(4)

   newPuzzle.search_island("deep")
```

続いて、ターミナル上で以下のコマンドを実行した。

```
python main.py > result_02_with_island_deep.txt
```

この結果から、241個のノード展開で、深さ187の解が見つかったことがわかる。しかし、この解は深すぎるため、より浅い解を見つけたい。そこで、手始めに幅優先探索によって最短の解を確認する。

幅優先探索におけるmain関数の設定は以下の通りである。

```
def main():

   newPuzzle = Puzzle(4)

   newPuzzle.search_island("wide")
```

この条件下において、ターミナル上で以下のコマンドを実行した。

```
python main.py > result_02_with_island_wide.txt
```

この結果から、最短の解の深さは27であることが確認できる。しかし、この探索方法では2066個のノードを展開しているため、計算量が大きい。そこで、深さ優先探索並みに探索数が小さく、かつ幅優先探索で見つけた最適解並みに浅い解を見つけることを目的としてA*サーチを実行する。

ここからは、最適なパラメータを見つけるため、複数のヒューリスティクスに対して、A*サーチを実行する。ただし、最適解を見つけることを保証する探索方法ではなく、可能な限り最適解に近い解を、少ないノード展開数で見つけることを目的として探索方法の改善をおこなった。

まず、main関数を以下のように設定した。この設定はこれ以降継続して使用する。

```
def main():

   newPuzzle = Puzzle(4)

   newPuzzle.search_island("A")
```

続いて、初めにcalc_h_islandメソッドを以下のように設定した。

```
   def calc_h_island(self, board):

       return 0
```

次に、ターミナル上で以下のコマンドを実行した。

```
python main.py > result_02_1.txt
```

f = d であることから、この結果は当然、幅優先探索の結果に相当し、2066個のノード展開によって、深さ27の解が見つかったことが確認できる。

ここからヒューリスティクスを改善していく。

まず、左岸の夫婦のペア数が少なく、右岸のペア数が多い方がゴールに近いという自然な考え方から、左岸のペア数、右岸のペア数を計算してh値の計算に組み込むことを考

える。

初めに左岸のペア数についてのみ考慮したヒューリスティクスが以下である。

```
def calc_h_island(self, board):

    left_pair = 0

    for i in range(self.N):

        if board[i*2] == 0 and board[i*2+1] == 0:

            left_pair += 1

    h = left_pair * 100

    return h
```

左岸のペア数は少ない方がいいので、h値はペア数に正の値をかけたものとしている。100という定数は実験的に性能が良かったために選んだ。

この設定においてターミナル上で以下のコマンドを実行した。

```
python main.py > result_02_2.txt
```

この時の、ノード展開数が1164、解の深さは29である。深さについては申し分ないが、これではまだ、展開数が大きすぎる。

続いて、右岸のペア数のみを考慮したものが以下である。

```
def calc_h_island(self, board):

    right_pair = 0

    for i in range(self.N):

        if board[i*2] == 2 and board[i*2+1] == 2:

            right_pair += 1
```

```
        h = 100/(right_pair + 1)

        return h
```

右岸のペア数は多い方がゴールに近いため、ペア数が多い時に、h値が小さくなるように、ペア数を分母に入れて計算している。1の加算はゼロ割りを防ぐためである。

この設定においてターミナル上で以下のコマンドを実行した。

```
python main.py > result_02_3.txt
```

この時の、ノード展開数が837、解の深さは27である。深さについては最適となったが、まだ展開数が大きい。

よって次に、左岸の情報と、右岸の情報を組み合わせたヒューリスティクスを考える。

この時のh値の計算式が以下である。

```
    def calc_h_island(self, board):

        left_pair = 0

        right_pair = 0

        for i in range(self.N):

            if board[i*2] == 0 and board[i*2+1] == 0:

                left_pair += 1

            elif board[i*2] == 2 and board[i*2+1] == 2:

                right_pair += 1

        h = 1000/(right_pair + 1) + left_pair * 10

        return h
```

この設定においてターミナル上で以下のコマンドを実行した。

```
python main.py > result_02_4.txt
```

この時のノード展開数は411、解の深さは29である。深さについては十分であり、ノード展開数についても、だいぶ改善したことがわかる。

これをさらに改善するため、ボートの位置情報も追加してみる。

ボートの情報を付与したヒューリスティクスが以下である。

```
def calc_h_island(self, board):
    left_pair = 0
    right_pair = 0
    boat_position = board[self.N*2]
    for i in range(self.N):
        if board[i*2] == 0 and board[i*2+1] == 0:
            left_pair += 1
        elif board[i*2] == 2 and board[i*2+1] == 2:
            right_pair += 1
    h = 1000/(right_pair + 1) + left_pair * 10 + boat_position * 5
    return h
```

この設定においてターミナル上で以下のコマンドを実行した。

```
python main.py > result_02_5.txt
```

この時のノード展開数は354、解の深さは29である。解の深さを維持したまま、ノード展開数が大きく削減されていることがわかる。

しかし、まだ、深さ優先探索の展開数241には辿り着いていない。そこで、発想を転換して無駄なノード探索を減らす方法を考える。

ここで重要となるのは、夫婦間の交換可能性である。つまり、例えば夫婦1、夫婦2、夫婦3、夫婦4の順に右岸に辿り着く解と、論理的に等価な解が、夫婦4、夫婦3、夫婦2、夫婦1の順に辿り着く解のについても存在するということである。このような論理的に等価の解に通ずる探索ルートを並行して探索しているとしたら、それは無駄である。つまり、ここまでに議論したhの計算アルゴリズムに加えて、夫婦ごとの重みづけをすることで、無駄な探索を防ぐことができ、展開ノード数を削減できると考えられる。

ここで、実験的に良い結果が得られたヒューリスティクスを以下に示す。

```
def calc_h_island(self, board):
    h = 0
    left_list = [0 for _ in range(self.N)]
    right_list = [0 for _ in range(self.N)]
    for i in range(self.N):
        if board[i*2] == 0 and board[i*2+1] == 0:
            left_list[i] = 1
        elif board[i*2] == 2 and board[i*2+1] == 2:
            right_list[i] = 1
    left_func = lambda x, i: x * 10**(self.N - 1 - i)
    right_func = lambda x, i: x * 10**(self.N - 1 - i)
    for i in range(self.N):
        h += left_func(left_list[i], i) * 15
        h += 2/(right_func(right_list[i], i) + 1)
        h -= right_func(right_list[i], i) ** 3
    return h
```

この方法では、右岸、左岸のペア数をカウントするのではなく、indexが小さいペアから順に、別々に現在位置を確認し、indexが小さいペアの情報を重視するような計算関数を使用して、重み付けしている。これによって、できるだけindexが小さいペアから右岸に移動する解を優先的に探索するようなアルゴリズムが実現される。

この設定においてターミナル上で以下のコマンドを実行した。

```
python main.py > result_02_6.txt
```

この時のノード展開数は220、解の深さは31である。この結果から、解の深さを31と最適解に近い値に保ったまま、ノード展開数を大幅に削減し、深さ優先探索よりも小さい展開に抑えることに成功したことがわかる。

ここから発想を変えてさらに探索数を減らす方法を見つけたい。ここで鍵となるのは、深さ優先探索が安定して、少ない展開数に抑えられているということである。

そこで、深さが大きいノードの方がゴールに近いという自然な感覚から、深さの情報の重みを大きくしてhの計算に組み込む。

この時のh値の計算方法が以下である。

```python
def calc_h_island(self, board):

    h = 0

    left_list = [0 for _ in range(self.N)]

    right_list = [0 for _ in range(self.N)]

    for i in range(self.N):

        if board[i*2] == 0 and board[i*2+1] == 0:

            left_list[i] = 1

        elif board[i*2] == 2 and board[i*2+1] == 2:

            right_list[i] = 1

    left_func = lambda x, i: x * 10**(self.N - 1 - i)
```

```
        right_func = lambda x, i: x * 10**(self.N - 1 - i)

        for i in range(self.N):

            h += left_func(left_list[i], i) * 15

            h += 2/(right_func(right_list[i], i) + 1)

            h -= right_func(right_list[i], i) ** 3

        h -= board[-2]**10

        return h
```

この設定においてターミナル上で以下のコマンドを実行した。

```
python main.py > result_02_7.txt
```

この時のノード展開数は46、解の深さは39である。この結果から、解の深さが若干大きくなったが、ノード展開数を大幅に削減し、ほとんど無駄な探索をすることなく、解を見つけることができていることがわかる。

3. (3)について

ここでは、main関数を以下のように設定した。

```
def main():

    newPuzzle = Puzzle(4)

    newPuzzle.search_island("A")
```

続いて、ターミナル上で以下のコマンドを実行した。

```
python main.py > result_03.txt
```

この結果から、真ん中に島がある場合、ボートを直接一方の岸から対岸に移動させる手

順を含む解が存在することが示された。

4. **(4)について**

ここでは、main関数を以下のように設定した。

```
def main():

   newPuzzle = Puzzle(5)

   newPuzzle.search("A")
```

続いて、ターミナル上で以下のコマンドを実行した。

```
python main.py > result_04_1.txt
```

この結果から、真ん中に島が存在しない場合、N=5の解は存在しないことがわかった。

続いて、main関数を以下のように設定した。

```
def main():

   newPuzzle = Puzzle(5)

   newPuzzle.search_island("A")
```

続いて、ターミナル上で以下のコマンドを実行した。

```
python main.py > result_04_2.txt
```

この結果から、真ん中に島が存在すれば、N=5の場合にも解が存在することがわかった。

講義へのコメント

哲学やパラドックスの話が豊富で個人的にはすごく面白いと思います。強化学習や進化計算などが、現実世界でどのように応用され、活用されているかの実装例を、よりたくさん提示していただければさらに興味深いと感じました。