

ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
Khoa Công nghệ Thông tin



CSC14003 – Cơ sở trí tuệ nhân tạo

Report

Solve N-Queens problem by searching algorithms

21127021 - Trương Văn Chí
21CLC06

Giảng viên hướng dẫn

Nguyễn Ngọc Thảo

Lê Ngọc Thành

Nguyễn Trần Duy Minh

Hồ Chí Minh - 07/2023

MỤC LỤC

DANH MỤC ẢNH	i
DANH MỤC BẢNG	ii
1 Tỷ lệ hoàn thành công việc	1
2 Tổng quan	1
2.1 Bài toán N-Queens	1
2.2 Giải quyết bài toán bằng các thuật toán searching	1
2.3 Tính toán thời gian chạy thuật toán	1
2.4 Tính toán bộ nhớ sử dụng trong thuật toán	1
3 Các định nghĩa trong bài toán	2
3.1 Biểu diễn trạng thái	2
3.2 Class Problem	2
3.3 Class Search	2
3.4 Hàm heuristic	2
3.5 Biểu diễn Node trong frontier và population	2
4 Kết quả thực thi	3
4.1 Cấu hình máy tính	3
4.2 Màn hình output	3
4.3 Thống kê kết quả	4
5 Nhận xét	4
6 Chi tiết thuật toán	5
6.1 Thuật toán UCS	5
6.1.1 Biểu diễn NODE trong frontier	5

6.1.2	Biểu diễn NODE trong Explored Set	5
6.1.3	Tính toán bộ nhớ	5
6.1.4	Khởi tạo thuật toán	6
6.1.5	Cài đặt	6
6.1.6	Cải tiến	7
6.1.7	Độ phức tạp	8
6.1.8	Các lần chạy khác	9
6.2	Thuật Toán A*	9
6.2.1	Biểu diễn Frontier và Explored Set	9
6.2.2	Tính toán bộ nhớ	9
6.2.3	Khởi tạo thuật toán	9
6.2.4	Cài đặt	9
6.2.5	Vấn đề về hàm heuristic	9
6.2.6	Độ phức tạp	10
6.2.7	Xem xét về việc cải tiến	11
6.2.8	Các lần chạy khác	11
6.3	Genetic Algorithms	11
6.3.1	Biểu diễn population	11
6.3.2	Hàm fitness	11
6.3.3	Hàm selection	12
6.3.4	Hàm crossOver	12
6.3.5	Hàm mutation	12
6.3.6	Hàm reGeneral	12
6.3.7	Giải thuật	12
6.3.8	Độ phức tạp	13
6.3.9	Tối ưu thuật toán	13
6.3.10	Các lần chạy khác	14

7	Tổng kết	15
7.1	UCS	15
7.2	A*	15
7.3	GA	15

DANH MỤC HÌNH ẢNH

Hình 2.1	Magic command đo lường thời gian của một hàm	1
Hình 4.2	Màn hình output với $N = 8$, thuật toán GA	4
Hình 6.3	Tính toán chi phí không gian	6
Hình 6.4	Pseudo Code giải thuật UCS	7
Hình 6.5	Cải tiến thuật toán UCS	8
Hình 6.6	Chạy $N=100$, thuật toán A^*	10
Hình 6.7	Pseudo Code Genetic Algorithms	13
Hình 6.8	Tối ưu Genetic Algorithms	14

DANH MỤC BẢNG BIỂU

Bảng 1.1	Bảng biểu tỉ lệ hoàn thành công việc	1
Bảng 4.2	Tổng quan thời gian chạy	4
Bảng 4.3	Tổng quan dung lượng bộ nhớ	4
Bảng 6.4	Thông kê chi tiết giải thuật UCS	9
Bảng 6.5	Thông kê chi tiết giải thuật A*	11
Bảng 6.6	Thông kê chi tiết giải thuật GA	15

1 Tỷ lệ hoàn thành công việc

Bảng 1.1 Bảng biểu tỷ lệ hoàn thành công việc

Algorithms	Tỷ lệ hoàn thành
Uninformed-Cost Search	100%
A* Search	100%
Genetic Algorithms	100%

2 Tổng quan

2.1 Bài toán N-Queens

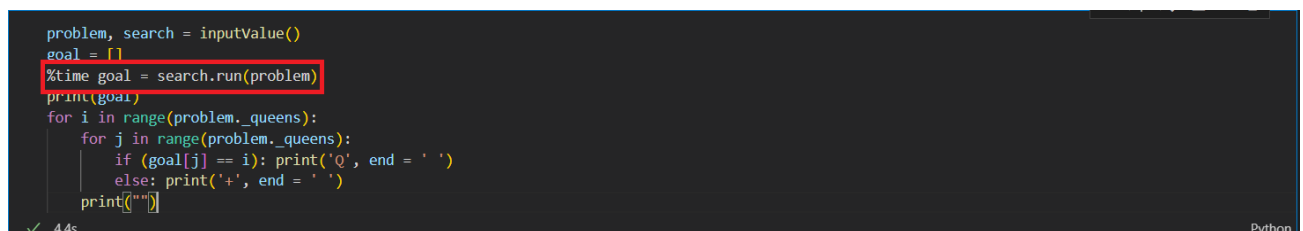
Bài toán N-Queens là bài toán đặt N quân hậu trên bàn cờ vua kích thước $N \times N$ sao cho không có quân hậu nào có thể "ăn" được quân hậu khác, hay nói khác đi không quân hậu nào có thể di chuyển theo quy tắc cờ vua.

2.2 Giải quyết bài toán bằng các thuật toán searching

Trong phạm vi bài lab này ta sẽ sử dụng 3 thuật toán tìm kiếm cơ bản trong AI để giải quyết. Lần lượt là Uniform-Cost Search(UCS), A* search(A*) và Genetic Algorithms(GA).

2.3 Tính toán thời gian chạy thuật toán

Ta chạy thuật toán trên kernel python của jupyter nên có thể dùng magic command `%time` để đo thời gian chạy cho từng thuật toán.



```

problem, search = inputValue()
goal = []
%time goal = search.run(problem)
print(goal)
for i in range(problem.queens):
    for j in range(problem.queens):
        if goal[j] == i: print('Q', end = ' ')
        else: print('+ ', end = ' ')
    print("")

```

Hình 2.1 Magic command đo lường thời gian của một hàm

2.4 Tính toán bộ nhớ sử dụng trong thuật toán

- Thuật toán UCS và A*: Dung lượng bộ nhớ dùng chủ yếu được gây ra bởi frontier và explored. Khi ta tìm thấy goal state ta sẽ đo bộ nhớ từ 2 cấu trúc này là chủ yếu.

- Thuật toán GA: Dung lượng bộ nhớ tiêu tốn chủ yếu ở các cấu trúc lưu trữ các population và việc gán, sinh mới các population.

3 Các định nghĩa trong bài toán

3.1 Biểu diễn trạng thái

Chúng ta chạy giải thuật thông qua các trạng thái của chúng. Một trạng thái được biểu diễn bằng một list có độ dài bằng n (với n là số quân hậu). Mỗi giá trị phần tử trong list biểu diễn hàng mà con hậu đó đang đứng.

Ví dụ một list $[4, 3, 1, 2]$ có ý nghĩa rằng:

- Con hậu của cột 1 đứng ở hàng 4
- Con hậu của cột 2 đứng ở hàng 3
- Con hậu của cột 3 đứng ở hàng 1
- Con hậu của cột 4 đứng ở hàng 2

3.2 Class Problem

Là một lớp chứa các thông tin về số quân hậu của bài toán, state khởi tạo và loại search ta sẽ dùng để chạy. Mặc định số quân hậu là 8 và thuật toán search là UCS.

3.3 Class Search

Là một lớp trừu tượng được khởi tạo với 1 Problem(). Mặc định số quân hậu là 8 và dùng thuật toán UCS search.

3.4 Hàm heuristic

- Input: Nhận vào một state

- Output: trả về một số nguyên là số cặp hậu tấn công lẫn nhau, tối đa là $\frac{n(n-1)}{2}$.

Ban đầu khởi tạo $h = 0$. Với mỗi con hậu ở cột i , kiểm tra xem các con hậu trước nó có tấn công nó hay không, nếu có tăng lên 1 với mỗi cặp hậu tấn công. 2 con hậu tấn công nhau khi cùng nằm trên 1 hàng hoặc nằm trên đường chéo. Giả sử, ta xét 2 con hậu tại vị trí i và j ($j < i$). 2 con hậu này tấn công nhau khi và chỉ khi $state[i] = state[j]$ (nằm trên cùng một hàng) hoặc $i - j = |state[i] - state[j]|$ (nằm trên cùng đường chéo).

3.5 Biểu diễn Node trong frontier và population

- Frontier trong thuật UCS và A* là một hàng đợi ưu tiên. Ta sử dụng thư viện `queue.PriorityQueue()` để cài đặt hàng đợi này. Trong python, hàng đợi ưu tiên luôn là min heap nên ta phải biểu diễn state một cách hợp lý để hàng đợi này có thể chạy tốt. Chính

vì vậy mỗi Node trong frontier là một tuple, tuple này có dạng **<độ dài đường đi, trạng thái đang xét>**.

- Tương tự như frontier thì population trong GA cũng có cấu trúc tương tự. Điều này có thể không cần thiết phải lưu cả độ dài đường đi nhưng vì thuật GA tốn khá ít bộ nhớ nên có thể dùng một ít chi phí không gian để lưu lại heuristic của mỗi state. Điều này giúp tiết kiệm chi phí thời gian khi tại mỗi state ta không cần phải kiểm tra lại heuristic của state đó với độ phức tạp $O(n^2)$

4 Kết quả thực thi

4.1 Cấu hình máy tính

Chương trình được chạy trên jupyter với kernel python 3.11.1

CPU: intel core i5-8265U

RAM: 16GB dual channel

Hard disk: 512GB SSD + 1TB HDD

4.2 Màn hình output

Màn hình output sẽ cho ta biết được một số thông tin quan trọng:

- Dòng 1: Dung lượng bộ nhớ sử dụng
- Dòng 2 + 3: Thời gian hàm chạy trên CPU và tổng thời gian hàm chạy
- Dòng 4: Kết quả goal state
- Từ dòng 5 đến hết: Ma trận biểu diễn goal state tương ứng

```

Data usage: 0.0763Mb
CPU times: total: 46.9 ms
Wall time: 35.7 ms
[2, 5, 7, 0, 4, 6, 1, 3]
+ + + Q + + + +
+ + + + + + Q +
Q + + + + + + +
+ + + + + + + Q
+ + + + Q + + +
+ Q + + + + + +
+ + + + + Q + +
+ + Q + + + + +

```

Hình 4.2 Màn hình output với $N = 8$, thuật toán GA

4.3 Thống kê kết quả

Bảng 4.2 Tổng quan thời gian chạy

	Running time (ms)		
Algorithms	$N = 8$	$N = 100$	$N = 500$
UCS	Intractable	Intractable	Intractable
A*	6	Intractable	Intractable
GA	30	1 200 000	Intractable

Bảng 4.3 Tổng quan dung lượng bộ nhớ

	Memory (MB)		
Algorithms	$N = 8$	$N = 100$	$N = 500$
UCS	Intractable	Intractable	Intractable
A*	0.0376	Intractable	Intractable
GA	0.0035	0.1526	Intractable

5 Nhận xét

Ta thấy rõ rằng thuật UCS không chạy được với số lượng quân hậu dù là nhỏ nhất. A* có thể chạy tốt hơn GA ở những trường hợp nhỏ nhưng sẽ không chạy được các trường hợp N lớn. GA chạy tốt trong đa số trường hợp nhưng ở trường hợp bài toán nhỏ chạy lâu hơn A*.

Về phần dung lượng bộ nhớ:

- UCS: Ta tạm thời chưa kết luận vì không có dữ liệu
- A*: Dung lượng tiêu tốn tương đối cao, nguyên nhân chủ yếu từ frontier và explored chứa một lượng lớn các node mà ta đã xét và đang chờ được xét.
- GA: Dung lượng tiêu tốn tương đối ít. Thuật toán GA chỉ dùng bộ nhớ cho việc lưu các population, các population này luôn được giữ ở một số lượng cố định, không mở rộng cùng với thời gian chạy thuật toán.

6 Chi tiết thuật toán

6.1 Thuật toán UCS

6.1.1 Biểu diễn NODE trong frontier

Như đã đề cập ở phần 3.5, frontier trong thuật UCS được biểu diễn bởi một hàng đợi ưu tiên. ta dùng thư viện queue của python và cài đặt `queue.PriorityQueue()`. Với mỗi NODE ta biểu diễn bằng 1 tuple có dạng **<Path Cost, State>**.

6.1.2 Biểu diễn NODE trong Explored Set

Explored set trong thuật toán UCS nắm giữ vai trò lưu lại các state đã đi. Chính vì thế trong explored set chỉ chứa các state mà không cần phải lưu thêm chi phí đường đi cho mỗi state. Ta dùng kiểu dữ liệu List mặc định của python để biểu diễn cấu trúc này.

6.1.3 Tính toán bộ nhớ

Thuật toán này dùng 2 cấu trúc dữ liệu chính có chi phí cao là frontier và explored. Từ đó nên ta có thể kết luận được rằng chi phí không gian của thuật toán này chủ yếu nằm ở 2 cấu trúc trên.

Ta dùng thư viện `sys.getsizeof()` của python để tính toán dung lượng đã sử dụng.

Tổng dung lượng = dung lượng của explored + dung lượng của frontier

Dung lượng explored: Vì là kiểu list nên `sys.getsizeof()` có thể trả về đúng số byte mà list này chiếm.

Dung lượng frontier: Vì là hàng đợi ưu tiên nên dung lượng của frontier được tính bằng cách: Tổng số NODE * số byte 1 NODE

```
# get data usage by byte
dataUse = sys.getsizeof(explored) + sys.getsizeof(node)*frontier.qsize()
# convert byte to Mb
dataUse = dataUse / (1024**2)

print(f"Data usage: {round(dataUse, 4)}Mb")
```

Hình 6.3 Tính toán chi phí không gian

6.1.4 Khởi tạo thuật toán

Bài toán được khởi tạo với các giá trị:

- Trạng thái khởi tạo: Đặt N quân hậu trên bàn cờ tại hàng thứ 0
- Frontier: Khởi tạo một queue.PriorityQueue() rỗng
- Explored: Khởi tạo một empty list
- Thêm NODE đầu tiên với giá trị <0, init state> vào frontier

6.1.5 Cài đặt

Thuật toán cài đặt theo đúng pseudo code trong slide **trang 25 - Lecture03 - P2**

Uniform-cost search (UCS)

```

function UNIFORM-COST-SEARCH(problem) returns a solution, or failure
  node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
  frontier ← a priority queue ordered by PATH-COST, with node as the element
  explored ← an empty set
  loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier) /* chooses the lowest-cost node in frontier */
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
      child ← CHILD-NODE(problem, node, action)
      if child.STATE is not in explored and not in frontier then
        frontier ← INSERT(child, frontier)
      else if child.STATE is in frontier with higher PATH-COST then
        replace that frontier node with child

```

25

Hình 6.4 Pseudo Code giải thuật UCS

6.1.6 Cải tiến

Việc kiểm tra một successor state có đang nằm trong frontier hay không (giống như pseudo code) gây lãng phí một chi phí thời gian vô cùng lớn. Vấn đề thuật toán đã mất rất nhiều thời gian cho việc duyệt hết tất cả các state nên ta có thể cải tiến chỗ này.

Cơ sở cho việc cải tiến này là ta dùng một chi phí không gian để tối ưu cho chi phí thời gian. Cụ thể trong trường hợp này, với mỗi successor state ta không cần kiểm tra xem nó đã nằm trong frontier hay chưa mà ta có thể thêm luôn vào frontier. Ta sẽ kiểm tra nó khi lấy nó ra khỏi frontier và xem nó đã nằm trong explored set hay chưa. Nếu nó đã nằm trong đó rồi thì ta chỉ việc bỏ qua nó và xét đến NODE kế tiếp. Giả sử có 2 state giống nhau nhưng khác chi phí đường đi. Hàng đợi ưu tiên sẽ lấy ra state có chi phí thấp hơn trước và xét nó sau đó thêm vào explored. Khi hàng đợi lấy ra state bị trùng lặp kia, ta kiểm tra thấy nó đã tồn tại trong explored rồi, ta chỉ việc bỏ qua (vì là hàng đợi ưu tiên nên chắc chắn state có đường đi tốt hơn mới là state được xét trước).

```

while (not frontier.empty()):
    node = frontier.get()
    nodeCost = node[0]
    state = node[1]

    if (state in explored):
        continue

    # Goal state -> heuristic = 0
    if (heuristic(state) == 0):
        # get data usage by byte
        dataUse = sys.getsizeof(explored) + sys.getsizeof(node)*frontier.qsize()
        # convert byte to Mb
        dataUse = dataUse / (1024**2)

        print(f"Data usage: {round(dataUse, 4)}Mb")
        return state
    # Add state to explored
    explored.append(state)

    # number = node.State
    number = len(state)
    for queen in range(number):
        for i in range(number):
            childState = state[:]
            if i != state[queen]:
                childState[queen] = i
                if (childState not in explored):
                    frontier.put((nodeCost+1, childState))

```

Hình 6.5 Cải tiến thuật toán UCS

Để đánh giá sự cải tiến này ta thấy rõ việc kiểm tra successor trong frontier tốn chi phí thời gian $O(m)$ với m là số state đang trong frontier và sau khi cải tiến ta tốn một chi phí thời gian $O(n)$ (với n là số phần tử trong explored, $n < m$) và chi phí không gian cho một số NODE mới trong frontier. Sự đánh đổi này là hợp lý bởi vì thuật toán UCS tiêu tốn chi phí thời gian rất nhiều với những trường hợp N nhỏ, chi phí thời gian này có thể vượt xa chi phí không gian.

6.1.7 Độ phức tạp

Như trong bài giảng ta biết được độ phức tạp của UCS với chi phí bằng nhau sẽ tiệm cận với độ phức tạp của thuật toán BFS tức là b^d . Xét trường hợp $N = 8$, số successor state ta có thể có là $b = n(n-1) = 56$ từ đó ta suy ra sẽ có 56^8 state mà ta phải xét trong trường hợp xấu nhất.

Với bài toán 8 hậu và việc khởi tạo init state như đề cập phía trên, để chúng ta tìm ra được goal state đòi hỏi phải có ít nhất 7 bước di chuyển. Số state có thể có là $56^7 = 1\,727\,094\,849\,536$. Giả sử theo lý thuyết 1 máy tính có thể thực hiện 10 000 000 phép tính trên 1 giây thì phải mất tận 48 giờ mới có thể cho ra được kết quả. Một con số quá lớn đối với $N = 8$.

6.1.8 Các lần chạy khác

Bảng 6.4 Thống kê chi tiết giải thuật UCS

UCS	N = 4	N = 5	N = 6
Running time(ms)	7	243	201 000
Memory(Mb)	0.03	0.4467	10.6228

Nhìn vào bảng biểu ta có thể thấy rằng với sự tăng trưởng của N thì thời gian và bộ nhớ cũng tăng trưởng theo đó với một tốc độ rất nhanh. Theo đó ta có thể dự đoán được thuật toán này không thể chạy nhanh trên hầu hết các trường hợp $N \geq 8$

6.2 Thuật Toán A*

6.2.1 Biểu diễn Frontier và Explored Set

Biểu diễn giống như cách biểu diễn trong thuật toán UCS (xem thêm mục 6.1)

6.2.2 Tính toán bộ nhớ

Giống như thuật toán UCS (xem thêm ở mục 6.1.3)

6.2.3 Khởi tạo thuật toán

Khởi tạo giống thuật toán UCS (xem thêm ở mục 6.1.4)

Có sự khác biệt trong cách khởi tạo NODE đầu tiên trong frontier: Thay vì $pathCost = 0$ thì sẽ là $0 + heuristic(initState)$.

6.2.4 Cài đặt

Cài đặt giống như thuật toán UCS (xem thêm mục 6.1.5). Có một sự khác biệt: thay vì pathCost tăng 1 sau mỗi bước đi thì pathCost được tính bằng công thức:

$$f(n) = g(n) + heuristic(n) + 1$$

6.2.5 Vấn đề về hàm heuristic

Một trong những nguyên nhân gây ra việc không thể chạy được thuật toán này với $N = 100$ và $N = 500$ đến từ hàm heuristic. Ta sẽ đề cập và phân tích sâu hơn về hàm heuristic ở đây.

Tính admissible của hàm heuristic: Như trong slide có đề cập, một hàm heuristic được xem là admissible khi $\forall n, h(n) \leq h^*(n)$ với $h^*(n)$ là chi phí thực tế từ state N đến goal.

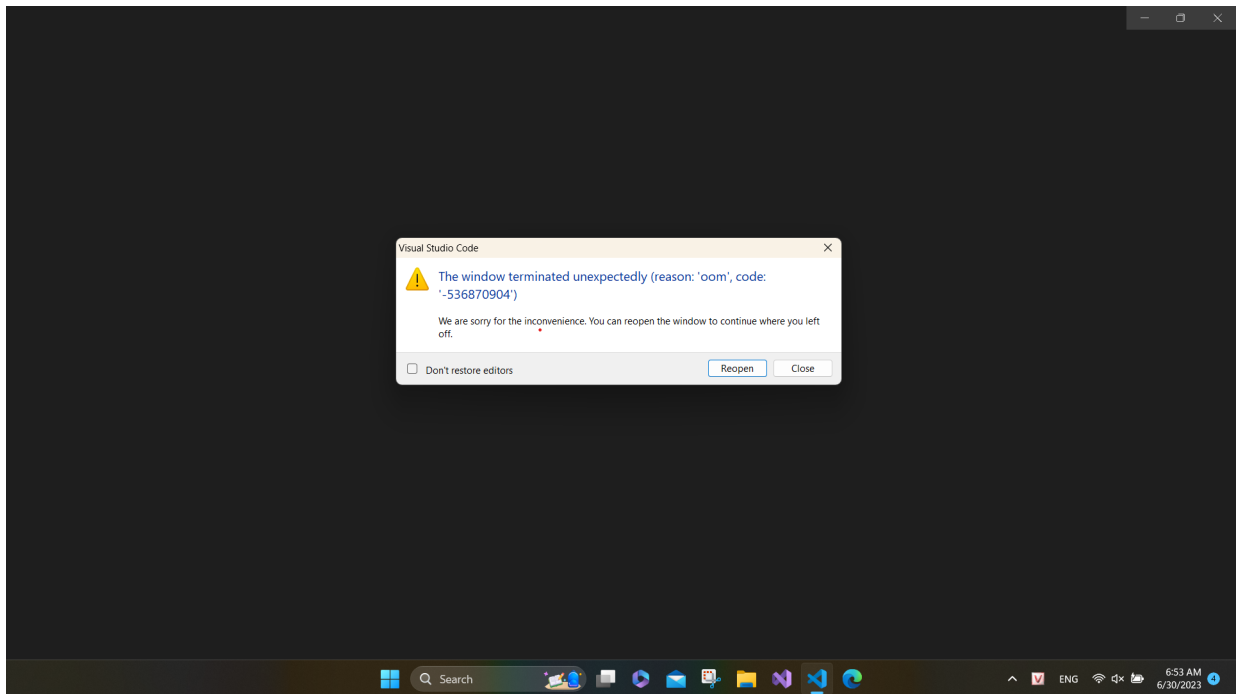
Xét trường hợp $N = 4$: Ta có 1 goal state $g = [2, 4, 1, 3]$ với heuristic = 0, xét state

$n = [2, 4, 1, 2]$ có heuristic = 3. Ta nhận thấy rằng $h^*(n) = 1$ (di chuyển con hậu ở cột 4) trong khi $h(n) = 3 \Rightarrow h(n) > h^*(n)$. Vậy ta kết luận hàm heuristic này không admissible và nó làm tốn nhiều chi phí hơn để đạt đến goal state.

Cũng xét như theo ví dụ trên, ta thấy rằng có thể đi đến goal state bằng state n phía trên nhưng với A* thì những state nào có heuristic < 3 sẽ được xét trước mặc dù chưa chắc nó đã có thể đi đến được goal state hay nói cách khác là ta đang bị mắc kẹt trong vùng cực tiểu địa phương.

6.2.6 Độ phức tạp

Vì là informed search nên độ phức tạp của thuật toán A* sẽ có cải thiện hơn nhiều so với thuật toán UCS nhờ vào hàm heuristic. Nhưng độ phức tạp vẫn còn rất lớn đối với các trường hợp bài toán có số quân hậu lớn. Thực nghiệm chạy thuật toán A* 2 lần với lần 1 kéo dài trong 17 tiếng và lần 2 kéo dài trong 13 tiếng nhưng vẫn không thu về được goal state.



Hình 6.6 Chạy N=100, thuật toán A*

Hình ảnh phía trên thu được khi chạy thực nghiệm $N = 100$ với thời gian 17 tiếng. Lỗi hiển thị ra màn hình theo em tìm hiểu là do sự thiếu hụt RAM của máy tính.

6.2.7 Xem xét về việc cải tiến

Cải tiến thời gian truy cập frontier: Giống như cách cải tiến của thuật UCS (xem thêm mục 5.1.6)

Chúng ta nhận được một lỗi do sự thiếu hụt RAM \Rightarrow có thể cải tiến để dùng ít bộ nhớ hơn. Em đã tìm hiểu và có một kết luận rằng khi ta muốn cải tiến để giảm thiểu tổn thất của chi phí không gian thì phải đánh đổi lại bằng một chi phí thời gian tương ứng. Vậy thì tổng thiệt hại có thể sẽ không đổi. Sau khi tham khảo ý kiến của thầy Lê Ngọc Thành thì thầy có đưa ra một giải pháp là ta sẽ xâm phạm vào ổ cứng. Tuy nhiên biện pháp này cũng không thể thực hiện được trong hoàn cảnh bài toán. Lí do chính là việc truy xuất vùng nhớ ổ cứng có thời gian lâu hơn truy xuất từ RAM nhưng thuật toán của ta đã chạy quá thời gian cho phép là 12 tiếng ngay cả khi truy cập trên RAM \Rightarrow Không khả thi.

6.2.8 Các lần chạy khác

Bảng 6.5 Thống kê chi tiết giải thuật A*

A*	N = 8	N = 20	N = 40
Running time(ms)	5.55	2 300	1 319 200
Memory(Mb)	0.0376	3.2172	1000.4958

Cùng với sự tăng trưởng của số lượng quân hậu, ta nhận xét chi phí thời gian và không gian tăng lên nhanh nhưng vẫn ít hơn thuật toán UCS. Từ đó có thể đưa ra dự đoán về các trường hợp lớn hơn như $N \geq 100$.

6.3 Genetic Algorithms

6.3.1 Biểu diễn population

Ta biểu diễn các tập hợp trạng thái(population) bằng cách dùng list trong python. Ta để cố định 1 population chứa $k = 100$ trạng thái và điều này không thay đổi trong suốt quá trình chạy thuật toán.

Với mỗi state trong population ta biểu diễn bằng cấu trúc **<heuristic(state), state>**, xem thêm về lí do biểu diễn ở mục 3.5

6.3.2 Hàm fitness

- Input: population
- Output: tỉ lệ chọn của các Trạng dựa vào heuristic. Trạng thái nào có heuristic càng thấp(số cặp hậu không tấn công nhau cao) thì có tỉ lệ chọn cao hơn. Hàm này xây dựng dựa

trên ý tưởng Russian Wheel.

6.3.3 *Hàm selection*

- Input: population, fitness function
- Output: 1 trạng thái được random dựa trên hàm fitness.

Ta dùng hàm **random.choices()** thuộc thư viện **random** để lựa chọn mẫu dựa trên tỉ lệ trả về từ hàm fitness.

6.3.4 *Hàm crossOver*

- Input: 2 trạng thái cha
 - Output: 1 trạng thái con được kết hợp từ mã gen của 2 trạng thái cha
- Cách thức hoạt động:

- Bước 1: Ta sử dụng hàm **random.randint()** thuộc thư viện **random** để random ra vị trí mà ta thực hiện việc ghép. Vị trí này ta tạm gọi là position
- Bước 2: từ vị trí position đã random từ bước 1 ta tạo ra một trạng thái mới bằng cách ghép 2 trạng thái cha. Trạng thái cha thứ nhất ta lấy từ đầu đến position, cha thứ 2 ta lấy từ position đến hết.

6.3.5 *Hàm mutation*

- Input: Một trạng thái bất kì
- Output: Một trạng thái mới đã trải qua quá trình đột biến gen.

Khi nhận vào một trạng thái, trạng thái này sẽ có xác suất **80%** bị biến đổi ở một **cột** bất kì thành một **giá trị** bất kì. Ta random để lựa chọn xem trạng thái này có bị biến đổi hay không (tỉ lệ 80%) và nếu trạng thái đó vượt qua được bước này sẽ random tiếp cột và giá trị mới. Giá trị tại cột sẽ được thay bằng giá trị mới.

6.3.6 *Hàm reGeneral*

- Input: new population
- Output: old population = new population

Công việc của hàm này chỉ đơn giản là gán lại các state cũ thành các state mới được sinh ra.

6.3.7 *Giải thuật*

Giải thuật được cài đặt theo đúng pseudo code trong slide (**trang 38 - Lecture 04 - LocalSearch**)

Genetic algorithms

```

function GENETIC-ALGORITHM(population, FITNESS-FN)
  returns an individual
  inputs: population, a set of individuals
  FITNESS-FN, a function that measures the fitness of an individual
  repeat
    new_population  $\leftarrow$  empty set
    for i = 1 to SIZE(population) do
      x  $\leftarrow$  RANDOM-SELECTION(population, FITNESS-FN)
      y  $\leftarrow$  RANDOM-SELECTION(population, FITNESS-FN)
      child  $\leftarrow$  REPRODUCE(x, y)
      if (small random probability) then child  $\leftarrow$  MUTATE(child)
      add child to new_population
    population  $\leftarrow$  new_population
  until some individual is fit enough, or enough time has elapsed
  return the best individual in population, according to FITNESS-FN

```

38

Hình 6.7 Pseudo Code Genetic Algorithms

6.3.8 Độ phức tạp

Về thời gian: GA sử dụng cho mình rất nhiều lần random để thực hiện giải thuật, các random này trả về kết quả ngẫu nhiên, có thể xấu cũng có thể tốt. Chúng ta không kiểm soát được các trạng thái mới được sinh ra có chắc chắn tốt hơn các trạng thái cũ hay không vì thế giải thuật sẽ chạy rất lâu nếu như ta không may mắn.

Về không gian: GA sử dụng ít bộ nhớ, chủ yếu cho việc lưu các population và sinh ra các population mới. Các không gian này không mở rộng theo thời gian của giải thuật mà nó chỉ mở rộng theo độ phức tạp và số quân hậu giải thuật phải chạy. Vì vậy nên ta có thể nói chi phí không gian của thuật toán này là tuyến tính theo độ lớn của bài toán chứ không theo thời gian chạy thuật toán.

6.3.9 Tối ưu thuật toán

Như đã nói ở trên thì thuật toán GA sử dụng random để random ra các thế hệ sau. Với mỗi thế hệ sau thì các phần tử trong đó chưa chắc đã tốt hơn so với thế hệ trước, vì vậy ta có thể tối ưu thuật toán ở điểm này.

Ý tưởng tối ưu (ĐÃ XIN Ý KIẾN CÔ NGUYỄN NGỌC THẢO VÀ ĐƯỢC CHO

PHÉP): Ta tối ưu bằng cách thay vì lựa chọn hết tất cả các phần tử trong thể hệ sau được tạo ra thì ta kết hợp lại giữa 2 thể hệ cũ và mới. Sau đó ta lựa chọn ra lại một thể hệ mới từ những phần tử **tốt nhất** ở 2 thể hệ với xác suất 30% ở thể hệ cũ và 70% ở thể hệ mới. Các xác suất lựa chọn phần tử này là độc lập nhau.

Để làm được điều đó ta lựa chọn cách sửa lại hàm **reGeneral**. Ta sử dụng `queue.PriorityQueue()` để lấy những trạng thái tốt nhất (heuristic nhỏ nhất). Ta có thể thay thế hàng đợi ưu tiên bằng cách dùng `sort` lên chính 2 list này nhưng như thế sẽ gây lãng phí thêm chi phí thời gian ($O(n \log_2(n))$) **cho thuật toán sort so với** $O(\log_2(n))$ **cho thời gian dùng hàng đợi ưu tiên**). Bù lại ta phải trả thêm chi phí không gian $O(n)$.

```
def reGeneral(self, newPopulation):
    childs = queue.PriorityQueue()
    parents = queue.PriorityQueue()
    for node in self._state:
        parents.put(node)
    for state in newPopulation:
        childs.put((heuristic(state), state))

    count = 0
    newGeneral = []
    while (count < self.numberState):
        if (random.random() < 0.3):
            newGeneral.append(parents.get())
        else:
            newGeneral.append(childs.get())
        count += 1

    self._state = newGeneral
```

Hình 6.8 Tối ưu Genetic Algorithms

6.3.10 Các lần chạy khác

Mặc dù giải thuật GA có thể chạy với N lớn hơn nhưng ta đã thống kê ở mục 4, vì vậy ta sẽ không thống kê lại ở đây mà lựa chọn một hệ quy chiếu giống với giải thuật A* để có thể so sánh sự khác biệt.

Bảng 6.6 Thống kê chi tiết giải thuật GA

GA	N = 8	N = 20	N = 40
Running time(ms)	282	664	75 000
Memory(Mb)	0.0763	0.1129	0.1755

Với sự tăng trưởng của số lượng quân hậu thì chi phí thời gian tăng lên nhưng không nhiều. Chi phí không gian tăng rất ít so với thuật toán A*.

7 Tổng kết

7.1 UCS

Thuật toán chạy chậm trong đa số trường hợp kể cả số lượng quân hậu nhỏ.

Sử dụng nhiều bộ nhớ để lưu frontier và explored set.

Không cho kết quả tốt được như 2 thuật A* và GA.

7.2 A*

Thuật toán có thể chạy với số lượng quân hậu nhỏ và trung bình.

Sử dụng nhiều bộ nhớ nhưng ít hơn UCS để lưu frontier và explored set.

Tuy nhiên thuật toán phải dựa rất nhiều vào độ tốt của hàm heuristic.

Nhìn chung thuật toán tốt hơn UCS và tệ hơn thuật GA trong phạm vi bài toán này.

7.3 GA

Thuật toán có thể chạy tốt trong hầu hết các trường hợp.

Sử dụng ít bộ nhớ vì không có sự mở rộng của việc lưu trữ các trạng thái.

Đánh giá tốt hơn 2 thuật toán A* và UCS trong mô hình bài toán N-Queens.

Có thể cải thiện thời gian nhiều hơn nhờ vào các thuật toán random khác hợp lý hơn.