ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
KHOA CÔNG NGHỆ THÔNG TIN

BÁO CÁO MÔN HỌC
CƠ SỞ TRÍ TUỆ NHÂN TẠO

Lab 01: N-queens problem

- **Sinh viên thực hiện**
  Võ Nguyễn Hoàng Kim – 21127090

PHỤ LỤC

A. THỐNG KÊ

1. Các thuật toán đã cài đặt được

Algorithms	Tình trạng
UCS	Hoàn thành
A*	Hoàn thành
GA	Hoàn thành

2. Thống kê

	Running time (ms)			Memory (MB)		
Algorithms	N = 8	N = 100	N = 500	N = 8	N = 100	N = 500
UCS	Intractable	Intractable	Intractable	Intractable	Intractable	Intractable
A*	22.29	Intractable	Intractable	0.14	Intractable	Intractable
GA	54.83	544782.55	Intractable	0.0075	0.28	Intractable

Giới hạn chờ đợi: số lần lặp không quá N*1000 (với N là số quân hậu được nhập từ bàn phím)

B. MỘT SỐ HÀM BỔ TRỢ

1. Hàm **isValid**(*checkingState*)
   - Hàm **isVailid** nhận vào tham số là *checkingState* – trạng thái cần kiểm tra và trả về giá trị True/ False.
   - Công dụng của hàm dùng để kiểm tra liệu rằng *checkingState* có phải là trạng thái đích (Goal State) hay không.
   - Nếu như *checkingState* thỏa điều kiện rằng tất quân hậu không tấn công nhau (nếu chúng không ở cùng hàng và không cùng đường chéo) → Hàm sẽ trả về giá trị True.
   - Ngược lại, chỉ cần tồn tại một cặp hậu tấn công nhau, hàm sẽ lập tức trả về giá trị False.

2. Hàm **calcHeuristic**(*checkingState*)
   - Hàm **calcHeuristic** nhận vào tham số là *checkingState* và trả về giá trị là số nguyên.
   - Công dụng của hàm dùng là tính tổng số lượng cặp hậu tấn công nhau trong *checkingState*.

3. Hàm **calcFitness**(*checkingState*)
   - Hàm **calcFitness** nhận vào tham số là *checkingState* và trả về giá trị là số nguyên.
   - Công dụng của hàm dùng là tính độ thích nghi mà *checkingState* có được.
   - Trong hàm, em sử dụng lại cách tính heuristic ở hàm **calcHeuristic** để tính tổng số lượng cặp hậu tấn công nhau trong *checkingState* và gán vào biến *atkPairs*.
   - Kết quả mà hàm trả về sẽ là giá trị của phép tính **n*(n-1)/2 – attackingPairs** (với n*(n-1)/2 là số lượng tối đa cặp hậu tấn công nhau)

C. BÁO CÁO CHI TIẾT VỀ THUẬT TOÁN

1. UCS (Uniform-cost Search)

a. Chi tiết cài đặt

Với thuật toán UCS, em sử dụng Priority Queue để làm frontier, lưu trữ tổng chi phí để từ trạng thái ban đầu (initial state) đến với trạng thái hiện tại.

Em đặt trạng thái ban đầu (cố định) là n quân hậu cùng nằm trên một hàng đầu tiên và có chi phí (cost) là 0 và thêm chùng vào frontier.

Vòng lặp while để thực thi thuật toán với điều kiện dừng là khi trong frontier không còn phần tử nào (frontier rỗng) hoặc đặt đến giới hạn cho phép:

- Lấy trạng thái có chi phí thấp nhất trong frontier để xét (đặt trạng thái được xét là curState, có chi phí là cost)
- Kiểm tra xem liệu curState có phải là Goal State hay không, nếu đúng vậy thì hàm sẽ trả về curState và dừng thuật toán. Nếu không là Goal State, thuật toán sẽ tiếp tục chạy
- Xét từng quân hậu của trạng thái curState, với những vị trí mà quân hậu có thể đến (không quan tần rằng liệu trạng thái đó có các cặp hậu tấn công nhau hay không), em ghi nhận lại trạng thái của bài toán lúc đó và chi phí của trạng thái đó khi quân hậu được di chuyển (là nextState và nextCost). Và sau đó thêm chúng vào frontier. Thao tác này sẽ thực hiện cho N quân hậu.
- Khi đã xét hết N quân hậu, thuật toán sẽ lặp lại các thao tác trong vòng lặp while cho đến khi thỏa điều kiện dừng.

b. Đánh giá

- Với cách cài đặt như trên, em đã thử với N=8, tuy nhiên, thuật toán không thể chạy ra được kết quả trong giới hạn cho phép.
- Em đã thử giảm mức N xuống để kiểm tra độ đúng đắn của thuật toán, và nhận - thấy ra N = 5 là con số lớn nhất mà thuật toán có thể trả ra kết quả như hình.

```
Number of algorithms: 1
Number of queens: 5
-------------- UCS ---------------------------
[0, 2, 4, 1, 3]
Q * * * *
* * * Q *
* Q * * *
* * * * Q
* * Q * *
total time : 12474.941798 ms
total memory:  91.03531169891357 mb
```

- Có thể thấy, tuy N rất nhỏ, nhưng thời gian trả ra kết quả khá lâu và bộ nhớ mà nó sử dụng rất cao.

- Theo em, nguyên nhân dẫn đến thời gian xử lý lâu cũng như sử dụng bộ nhớ lớn là do số lượng trạng thái kế thừa (next successor) mà mỗi quân hậu sinh ra là khá nhiều (có đến n-1 bước kế tiếp), dẫn đến với lượng N quân hậu như đề bài yêu cầu thì số lượng trạng thái mà frontier lưu trữ vô cùng lớn. Đồng thời, để có thể duyệt hết lượng trạng thái mà frontier lưu trữ sẽ tốn một khoảng thời gian khá lâu cũng như sử dụng một lượng lớn bộ nhớ.

2. A*

a. Chi tiết cài đặt

Note: Trong thuật toán A*, em sử dụng các chữ cái đại diện cho các chi phí như h(n): chi phí để có được từ n đến đích (goal); g(n): chi phí để đạt được nút n → f(n) = g(n) + h(n).

Trong A*, em sử dụng Priority Queue làm frontier với các giá trị lưu trữ lần lượt là h, c, state (h, c là các chi phí cho state tương ứng). Ngoài ra, em còn có một checkedList để ghi lại các trạng thái đã xét trong quá trình thuật toán thực thi.

Khởi tạo các giá trị ban đầu

- Có cost = 0 (đại diên cho g), pathCost = 0 (đại diện cho f)
- Em khởi tạo trạng thái ban đầu (cố định) là các quân hậu cùng nằm trên hàng đầu tiên
- Thêm vào frontier lần lượt là h (thông qua việc gọi hàm **caclHeuristic** cho trạng thái ban đầu), cost, trạng thái ban đầu

Bắt đầu vòng lặp while với điều kiện dừng khi trong frontier không còn phần tử nào (frontier rỗng) hoặc đặt đến giới hạn cho phép:

- Lấy trạng thái có heuristic thấp nhất trong frontier để xét (lần lượt được lưu trữ trong các biến h, c, curState)
- Kiểm tra xem liệu rằng curState có phải là trạng thái đích (Goal State) hay không. Nếu đây là Goal State, thuật toán sẽ trả về curState và dừng lại. Nếu không, thuật toán sẽ tiếp tục các thao tác ở sau.
- Kiểm tra xem liệu rằng curState đã được xử lý qua chưa bằng cách kiểm tra xem nó đã tồn tại trong checkedList chưa (đây là bước tiết kiệm chi phí cho thuật toán, tránh trường hợp các trạng thái giống nhau được duyệt lại). Nếu curState đã tồn tại trong checkedList, thuật toán sẽ quay lại để lấy trạng thái khác trong frontier. Còn nếu chưa, nó sẽ thêm curState vào checkedList để đánh dấu.
- Xét từng quân hậu, với mỗi vị trí mà quân hậu đó có thể di chuyển, em ghi nhận trạng thái đó là nextState. Thêm nextState vào danh sách các trạng thái kế thừa (successorList). Lặp lại cho đến khi duyệt hết N quân hậu
- Duyệt trong successorList, nếu như successorList[i] chưa tồn tại trong checkedList (điều kiện này tránh các trường hợp sản sinh ra các trạng thái cũ) thì tiến hành tính h và pathCost của trạng thái đó (với h được tính thông qua hàm calcHeuristic, pathCost = h + c +1. Cuối cùng, thêm vào frontier lần lượt là pathCost, c + 1, successorList[i].

- Khi đã duyệt xong successorList, thuật toán sẽ lặp lại các thao tác trong vòng lặp while cho đến khi thỏa điều kiện dừng.

b. Đánh giá

Đối với N = 8, A* xử lý khá nhanh để cho ra kết quả. Tuy nhiên, bộ nhớ mà nó sử dụng để thực thi thuật toán có thể xem là tương đối nhiều.

Em đã thử nâng mức N = 20 để kiểm tra nhận định của mình, kết quả được trả về như hình bên dưới (thời gian khá lâu đối với N=20).

```
Number of algorithms: 2
Number of queens: 20
-------------- A* --------------------------
[13, 10, 0, 11, 6, 2, 12, 14, 18, 8, 15, 4, 16, 3, 17, 9, 7, 5, 19, 1]
* * Q * * * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * * Q
* * * * * Q * * * * * * * * * * * * * * *
* * * * * * * * * * * * * Q * * * * * * *
* * * * * * * * * * * Q * * * * * * * * *
* * * * * * * * * * * * * * * * * * Q * *
* * * * Q * * * * * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * Q * * *
* * * * * * * * * Q * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * Q * * *
* Q * * * * * * * * * * * * * * * * * * *
* * * Q * * * * * * * * * * * * * * * * *
* * * * * * Q * * * * * * * * * * * * * *
Q * * * * * * * * * * * * * * * * * * * *
* * * * * * * Q * * * * * * * * * * * * *
* * * * * * * * * Q * * * * * * * * * * *
* * * * * * * * * * * * Q * * * * * * * *
* * * * * * * * * * * * * * Q * * * * * *
* * * * * * * * Q * * * * * * * * * * * *
* * * * * * * * * * * * * * * * * * * Q *
total time : 21069.026666 ms
total memory:  38.87222766876221 mb
```

3. GA (Genetic Algorithms)

a. Chi tiết cài đặt

Để sử dụng thuật toán GA, em đã cài đặt thêm một số hàm để sử dụng trong quá trình cài đặt thuật toán

- **createPopulation**(*size*): để tạo ra một tập quần thể với kích thước tương ứng với size. Tập quần thể được tạo ra bằng cách chọn ngẫu nhiên các vị trí cho N quân hậu cho từng cá thể (trạng thái) (sử dụng hàm random cho thao tác này). Đồng thời trong quá trình tạo quần thể, em sẽ tính độ thích nghi của từng cá thể. Cuối cùng là thêm độ thích nghi và cá thể tương ứng vào quần thể.
- **chooseParents**(*population*): chọn cá thể cha mẹ từ quần thể để sử dụng trong phép lai tạo phía sau. Để duy trì điểm thích nghi trong quần thể không bị quá thấp, em chọn ra tập cha mẹ gồm cá thể có độ thích nghi cao nhất và cá thể có độ thích nghi thấp nhất. Cách chọn này sẽ giúp cho phép lai tạo trong việc sản sinh ra cá thể con có độ thích nghi tương đối (độ thích nghi không trở nên quá thấp). Sau đó xóa cá thể có độ thích nghi thấp nhất khỏi tập quần thể.

- **crossOver**(*par1, par2*): lai tạo giữa 2 cá thể cha mẹ. Em sử dụng hàm random để chọn ra một số nguyên (đặt là c) trong khoảng từ 0 đến n-1. Sau đó tìm kiếm xem trong par1 hoặc par2 có cá thể nào mà c xuất hiện nhiều hơn 1 lần hay không (tương ứng là có 2 quân hậu nằm cùng một hàng). Nếu có thì sẽ tạo một biến k bằng cách tìm kiếm vị trí của c xuất hiện và cộng thêm 1 (tương ứng lấy k là số phần tử đầu để lai tạo), còn nếu c không xuất hiện trong cả par1 và par2 quá 1 lần thì k sẽ giữ nguyên giá trị của c. Bắt đầu lai tạo: cá thể con thứ nhất (child1) được sinh ra bằng cách lấy k phần từ đầu của par1, n-k phần từ còn lại sẽ lấy từ par2; cá thể con thứ 2 (child2) thì sẽ lấy k phần từ đầu của par2, n-k phần tử còn lại sẽ lấy từ par1. Hàm sẽ trả về 2 cá thể con là child1 và child2.
- **mutation**(*child*): đột biến xảy ra với cá thể con. Ta sẽ random tỉ lệ khi gọi hàm, nếu tỉ lệ được random < 0.8 thì đột biến sẽ xảy ra, nếu không sẽ trả về lại giá trị cá thể con ban đầu (child). Quá trình đột biến ta sẽ chọn ngẫu nhiên vị trí xảy ra đột biến (pos) cũng như giá trị(number). Sau đó gán cho vị trí đã được chọn trong child giá trị tương ứng và trả về cá thể con đã được xử lý đột biến.

Bước vào thuật toán GA, em tạo quần thể với số lượng cá thể bằng với số quân hậu, và có thể một biến gens được đặt bằng 0 để tính số lượng thế hệ mà thuật toán sản sinh ra (gens sẽ được tăng mỗi khi tạo một quần thể mới).

Vào vòng lặp while, nếu như gens đạt tới giới hạn nhất định nhưng thuật toán vẫn chưa tìm được goalState thì sẽ kết thúc thuật toán (đây là điều kiện dừng):

- Do quần thể (population) lưu trữ cả fitness (độ thích nghi) và state (trạng thái) nên để kiểm tra xem goalState đã xuất hiện trong population chưa, ta chỉ cần sử dụng hàm max để lấy ra cá thể có độ thích nghi cao nhất và so độ thích nghi đó với độ thích nghi tối đa là N*(N-1)/2. Nếu thỏa, thuật toán sẽ trả ra goalState và dừng. Nếu không, thuật toán sẽ tiếp tục các bước phía dưới
- Sao chép quần thể vào biến tmp. Tiến hành vòng lặp while với điều kiện dừng là số lượng cá thể trong tmp chỉ còn lại 1:
  - Ta chọn cá thể cha mẹ từ quần thể tmp bằng hàm **chooseParents**.
  - Thực hiện phép lai bằng hàm **crossOver** cho cá thể cha mẹ → cho ra 2 cá thể con
  - *Note:* Em cho trạng thái có độ thích nghi cao nhất lai tạo với các trạng thái còn lại để sản sinh ra một tập cá thể con (số lượng cá thể con lớn hơn size) có độ thích nghi đa dạng → Từ đó dễ dàng chọn lọc các cá thể cho quần thể mới
  - Em muốn rằng một cặp cha mẹ bắt buộc lai tạo 2 cá thể con và bắt buộc chúng sau khi đột biến phải là những cá thể khác với tập cá thể trong quần thể population. Do đó nếu như không thỏa, chúng sẽ tiếp tục đột biến (đối với cá thể con ban đầu được sản sinh từ việc lai tạo) cho tới khi số lượng số lượng cả thể con đột biến thành công đủ số lượng là 2. Thêm cá thể con đột biến thành công vào tập successorList và tính độ thích nghi tương ứng của từng cá thể.

- Sau khi đã thực hiện xong việc lai tạo cho tất cả các cá thể trong quần thể tmp, ta đặt lại quần thể population về trạng thái rỗng, thực hiện chọn lọc các cá thể con trong successorList có độ thích nghi cao nhất và thêm chùng vào tập population cho đến khi số lượng cá thể mới bằng với size.
- Thuật toán sẽ lặp lại cho đến khi tìm được goalState hoặc đạt đến giới hạn cho phép

b. Đánh giá

Với cách cài đặt như trên, thuật toán gặp khó khăn để trả ra kết quả với N = 500. Tuy nhiên, thuật toán đã xử lý tốt với N = 100 trở xuống.

Mặc dù thời gian xử lý của GA chỉ ở mức tương đối, nhưng nó có thể xử lý với các giá trị N lớn, đồng thời, bộ nhớ mà nó sử dụng rất ít so với các thuật toán khác.

Theo em, lí do để GA xử lý thuật toán khá lâu có thể do nhiều nguyên nhân khác nhau:

+ Do quần thể ban đầu được tạo ra với những trạng thái xấu (có độ thích nghi thấp).

+ Trong quá trình xử lý, phần lớn cách thức chạy của thuật toán này dựa vào sự ngẫu nhiên (được thể hiện qua các phần random), dẫn đến các quá trình như lai tạo hoặc đột biến sẽ sản sinh ra các cá thể có độ thích nghi xấu → khiến thuật toán phải thực hiện nhiều lần để có thể kiếm được Goal State.

Lựa chọn ngẫu nhiên, mặc dù, khiến thuật toán tốn nhiều thời gian để chạy, nhưng bù lại bộ nhớ mà nó sử dụng rất ít. Theo em, nhờ vào tính chất random, để thực thi đã giúp cho thuật toán tiết kiệm bộ nhớ sử dụng, không tốn quá nhiều chi phí để xử lý hoặc lưu trữ.

D. NHẬN XÉT

Sau khi thực hiện chương trình để giải quyết bài toán N-queens problem, em rút ra được một số nhận định như sau

- UCS là thuật toán dễ cài đặt, tuy nhiên hiệu quả lại không cao, tốn nhiều thời gian để xử lý cũng như sử dụng bộ nhớ cao.
- A* là lựa chọn tốt nhất để xử lý các đầu vào nhỏ. Thời gian mà A* cho ra kết quả tốt hơn so với các thuật toán còn lại. Tuy nhiên ta phải đánh đổi với việc A* sử dụng bộ nhớ nhiều trong quá trình thực thi.
- GA tuy không cho ra kết quả trong thời gian nhanh nhất, tuy nhiên, đây là thuật toán khả thi đối với các đầu vào lớn. Bên cạnh đó, ưu điểm của GA còn nằm ở việc chỉ sử dụng một lượng nhỏ bộ nhớ để xử lý.