**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**

**TRƯỜNG ĐẠI HỌC BÁCH KHOA**

**BỘ MÔN: NHẬP MÔN TRÍ TUỆ NHÂN TẠO**

A blue and white logo

Description automatically generated

**BÁO CÁO BÀI TẬP LỚN 1**

**ĐỀ TÀI:**

**ÁP DỤNG GIẢI THUẬT DFS, HILL CLIMBING**

**VÀ GENETIC VÀO CÁC TRÒ CHƠI**

**SUDOKU VÀ MINESWEEPER**

**Giáo viên bộ môn: Vương Bá Thịnh**

**Lớp: L02**

**Thành viên nhóm:**

**Nguyễn Phúc Gia Khiêm 2211573**

**Lê Ngọc Vinh ???????**

**THÀNH PHỐ HỒ CHÍ MINH, THÁNG 9 NĂM 2023**

Contents

**No table of contents entries found.**

# **A. CƠ SỞ LÝ THUYẾT.**

## **I. Giải thuật DFS (Depth First Search).**

Thuật toán DFS (Depth-First Search) là một thuật toán tìm kiếm trong đồ thị, được sử dụng để duyệt một đồ thị theo chiều sâu. Thuật toán này bắt đầu từ một đỉnh bất kỳ và tiếp tục theo các cạnh để tìm các đỉnh kề, sau đó di chuyển xuống đỉnh kề tiếp theo và tiếp tục quá trình này cho đến khi không còn đỉnh nào mới để thăm. Khi gặp một đỉnh chưa được thăm, thuật toán sẽ tiếp tục đi theo một nhánh nào đó cho đến khi đạt được đỉnh cuối cùng trên nhánh đó trước khi quay lại và tiếp tục từ đỉnh trước đó.

Cách triển khai DFS trên một cây:

1. Tạo một mảng visited để đánh dấu một đỉnh đã được duyệt qua hay chưa.
2. Tạo một stack stk để phục vụ cho quá trình duyệt cây bằng DFS.
3. Bắt đầu từ một đỉnh, ta chèn đỉnh đó vào stk.
4. Ta lấy đỉnh trên cùng của stk ra và xem xét các nút có liên kết với đỉnh đó.
5. Nếu nút con đã ở trong visited thì bỏ qua, nếu nút con chưa ở trong visited ta thêm nó vào visited và chèn nó vào stack.
6. Nếu stack rỗng thì tiến tới bước 7, nếu stack chưa rỗng thì quay lại bước 4.
7. Kết thúc thuật toán.

## **II. Giải thuật Hill Climbing dốc nhất.**

Giải thuật Hill Climbing là một thuật toán tìm kiếm cục bộ (local search algorithm) được sử dụng trong các bài toán tối ưu hóa, nhất là trong các bài toán tìm kiếm không gian trạng thái (state space search). Mục tiêu của thuật toán là di chuyển từ một trạng thái hiện tại đến một trạng thái kế tiếp tốt hơn (tối ưu hơn) trong không gian trạng thái.

Thuật toán Hill Climbing dốc nhất hoạt động như sau:

1. Khởi tạo: Bắt đầu từ một trạng thái bất kỳ trong không gian trạng thái.
2. Đánh giá: Sử dụng một hàm đánh giá (evaluation function) để đánh giá chất lượng của trạng thái hiện tại. Hàm đánh giá thường trả về một giá trị số.
3. Tạo lân cận: Tạo ra các trạng thái lân cận của trạng thái hiện tại bằng cách thay đổi một phần của nó.
4. Lựa chọn lân cận tốt nhất: Đánh giá chất lượng của các trạng thái lân cận được tạo ra trong bước trước bằng cách sử dụng hàm đánh giá. Chọn ra trạng thái lân cận có giá trị đánh giá cao nhất (hoặc thấp nhất, tùy thuộc vào bài toán).
5. Kiểm tra điều kiện dừng: Kiểm tra xem trạng thái lân cận tốt nhất có tốt hơn trạng thái hiện tại không. Nếu có, di chuyển đến trạng thái lân cận tốt nhất và quay lại bước 2. Nếu không, dừng lại và trả về trạng thái hiện tại là giải pháp.
6. Kết thúc: Kết thúc thuật toán và trả về giải pháp tìm được.

Thuật toán Hill Climbing là một giải thuật tham lam (greedy algorithm), do vậy không tránh khỏi một số hạn chế như:

1. Dễ bị mắc kẹt ở điểm cực tiểu cục bộ (local minimum) mà không phải là điểm cực tiểu toàn cục (global minimum).
2. Bước vào trạng thái “cao nguyên” tức là các trạng thái lân cận có giá trị lượng giá bằng với trạng thái hiện tại và không thể đi tiếp.

Phương pháp giải quyết các hạn chế này sẽ được đề cập khi tiếp cận bài toán.

## **III. Giải thuật Genetic.**

Thuật toán genetic là thuật toán phỏng theo quá trình chọn lọc tự nhiên trong sinh học. Ta có một quần thể gồm nhiều cá thể (là các trạng thái của bài toán). Cá thể được đánh giá dựa trên fitness score. Cá thể có fitness cao hơn thì khả năng được chọn làm parent và tạo ra cá thể con cao hơn.

Thuật toán bao gồm 5 giai đoạn:

1. Khởi tạo quần thể ban đầu bằng cách các trạng thái ngẫu nhiên.
2. Tính fitness score của từng cá thể trong quần thể.
3. Chọn lọc (selection): Thông thường người ta lấy fitness score làm trọng số để chọn parent. Tuy nhiên 1 cách làm khác là lấy ngẫu nhiên n - p (n: số cá thể trong quần thể, p < n) cá thể từ quần thể, sau đó lấy p cá thể có fitness score lớn nhất.
4. Crossover: thông thường người ta dùng 2 parent chọn 1 điểm crossover để nối chuỗi và tạo nên chuỗi con.
5. Biến dị (mutation): Thay giá trị 1 vị trí bất kỳ trong chuỗi bằng một giá trị mới.

# **B. CÁC BÀI TOÁN.**

## **I. Trò chơi Sudoku.**

### **1. Giới thiệu trò chơi.**

Sudoku được chơi trên 1 bảng 9x9, được chia thành 9 khối 3x3, điền các giá trị vào ô trống sao cho:

- Không có 2 số giống nhau nằm trên cùng 1 hàng.

- Không có 2 số giống nhau nằm trên cũng 1 cột.

- Không có 2 số giống nhau nằm trong cùng 1 khối.

Đối với bài toán sudoku, nhóm chúng em đã sử dụng 2 phương pháp: depth first search và genetic algorithm.

### **2. Giải thuật Depth First Search.**

**2.1 Cách tiếp cận của nhóm:**

Nhóm sử dụng toán tử bitwise cho bài toán thay vì dùng mảng do tốc độ tính toán nhanh hơn.

Định nghĩa không gian trạng thái của bài toán:

* Không gian trạng thái: dùng 3 dictionary để biểu diễn cột, hàng, khối; dùng 1 số nguyên để biểu diễn các ô có giá trị 0.
* Trạng thái khởi đầu: Duyệt bảng ban đầu, và điền vào các dictionary để biểu diễn số x có tồn tại trên hàng i, cột j, block k hay không.
* Trạng thái kết thúc: Khi số ô trống bằng 0 và bảng sudoku đầu vào là hợp lệ.
* Các bước di chuyển hợp lệ: điền 1 số vào ô trống có index nhỏ nhất sao cho không vi phạm luật chơi.

Với cách tiếp cận của nhóm thì độ sâu của lời giải luôn bằng m, với m là số ô trống.

**2.2 Tiêu tốn bộ nhớ:**

Nhóm sử dụng thư viện chuẩn của python tracemalloc để tính toán tiêu tốn bộ nhớ. Thư viện này giúp ta truy vết cấp phát bộ nhớ.

Ta gọi hàm tracemalloc.get\_traced\_memory(). Hàm này trả về 1 tuple gồm 2 số là bộ nhớ được cấp phát tại thời điểm hiện tại và vào lúc đỉnh điểm. Ta quan tâm đến bộ nhớ được cấp phát vào lúc đỉnh điểm.

Ta sẽ cho k chạy từ 0 tới 81. Với k là số ô bị khuyết trên bảng sudoku. Thực hiện lấy nhiều lần nhóm nhận thấy kết quả tiêu tốn bộ nhớ trả về là như nhau. Nên ứng với mỗi k nhóm sẽ lấy kết quả 1 lần duy nhất.

A graph with a line drawn on it

Description automatically generated

Khi lấy kết quả về sự tiêu tốn bộ nhớ của thuật toán, ta thấy được tiêu tốn bộ nhớ của chương trình tăng theo k. Điều này cũng khá dễ hiểu do độ phức tạp về không gian của thuật toán DFS là b\*m với b là breanching factor, m là độ sâu lớn nhất (và với cách tiếp cận của nhóm thì m = k).

**2.3 Thời gian chạy của thuật toán:**

Thực hiện tương tự như khi tính tiêu tốn bộ nhớ, tuy nhiên, ta cho k chạy từ 0 tới 60. Ứng với mỗi k, ta sinh ra 100 puzzle ngẫu nhiên và cho thuật toán giải.

Lý do cho việc lấy tới k = 60 là do khi k tăng thì nhóm gặp 1 số trường hợp giải khá lâu (> 1 giờ). Điều này có thể giải thích là do thuật toán DFS có độ phức tạp thời gian là b\*d. Với b đươc định nghĩa như phần 1.2 và d là độ sâu của lời giải. Và ở đây d = k. Trong tình huống tệ nhất thì thuật toán của chúng ta có thể phải duyệt trường hợp nên việc thuật toán chạy lâu với 1 số trường hợp cũng là điều dễ hiểu.

Trong hầu hết trường hợp, thuật toán chạy với thời gian khá ngắn (< 0.1 giây và có thể là mili giây).

A graph with a line

Description automatically generated

Từ k = 0 tới k = 52 thời gian trung bình của thuật toán gần như bằng nhau. Thuật toán chạy trong thời gian mili giây. Từ k = 53 trờ đi ta thấy thời gian chạy trung bình có xu hướng tăng nhanh, đó là do thuật toán gặp phải những trường hợp phải giải trong thời gian lâu hơn.

A graph with numbers and a triangle

Description automatically generated

Tuy nhiên khi ta cho k chạy từ 75 tới 81 thì thời gian chạy giảm xuống còn mili giây. Đó là do khi số ô trống quá nhiều thì bài toán trở nên vô cùng đơn giản.

Từ dữ liệu thu thập được về thời gian chạy cũng như tiêu tốn bộ nhớ của thuật toán DFS, ta kết luận rằng thuật toán DFS nhìn chung là hiệu quả đối với bài toán Sudoku.

### **3. Giải thuật Genetic.**

**3.1. Cách tiếp cận của nhóm**

Nhóm sử dụng 1 chuỗi có độ dài 81 để biểu diễn bảng sudoku. (1 trạng thái là 1 chuỗi có độ dài 81 chứa giá tri từ 1 đến 9).

Cách tính fitness: Ban đầu gán fitness = MAX\_FITNESS = 216 (do 1 bảng sudoku với tất cả các ô có cùng 1 giá trị sẽ có fitness = 0). Ta vẫn sử dụng 3 dictionary như DFS.

* Duyệt chuỗi
* Nếu số x tồn tại trên hàng i rồi thì giảm fitness 1 đơn vị, ngược lại đánh dấu x tồn tại trên hàng i.
* Làm tương tự đối với cột và khối.

Thuật toán tìm được nghiệm khi fitness = MAX\_FITNESS

Cho đến nay, nhóm đã thử sử dụng 9 cách tiếp cận đối với thuật toán này. Ngoài ra nhóm đã thử chỉnh kích thước quần thể khác nhau, chỉnh mutation rate... tuy nhiên, không có cách nào có thể tìm được lời giải bài toán sudoku.

**3.1.1 Các cách tiếp cận.**

* + - * Cách 1 – 3:

Chuỗi được tạo thành bằng cách biến ma trận 9x9 ban đầu thành 1 mảng 1 chiều.

* Khởi tạo quần thể ban đầu, ta điền các giá trị ngẫu nhiên vào các ô trống.
* Selection: chọn ngẫu nhiên từ quần thể, lấy fitness làm trọng số (fitness càng cao thì khả năng được chọn cao hơn)
* Crossover:
* Cách 1: chọn 1 điểm ngẫu nhiên bất kỳ (gọi là x). Ta thực hiện crossover bằng cách ghép chuỗi parent1[:x] với parent2[x:]
* Cách 2 - 3: dùng nhiều parent hơn. Cách ghép tương tự cách 1.
* Mutation: duyệt chuỗi.
* Cách 1 - 2: nếu tại vị trí i trong bảng ban đầu là trống thì ta thay giá trị tại vị trí i bằng 1 giá trị hợp lệ (nếu có nhiều giá trị hợp lệ thì chọn bằng random).
  + - * Cách 4:

Cách tạo chuỗi tương tự từ cách 1 tới cách 3. Tuy nhiên, ta sẽ giúp thuật toán bằng cách khi khởi tạo quần thể ban đầu ta sẽ random các giá trị sao cho trên mỗi hàng, không có vị trí nào bị vi phạm.

* Crossover tương tự cách 1.
* Mutation: swap 2 giá trị tại 2 vị trí bất kỳ với điều kiện 2 vị trí đó là 2 vị trí trống trên bảng sudoku ban đầu.
  + - * Cách 5:

Đổi cách crossover, ta chọn m hàng từ parent1 và 9 - m hàng từ parent2

* + - * Cách 6:

Dùng nhiều parent hơn khi crossover, logic tương tự cách 5

* + - * Cách 7:

Dùng mutation rate thay đổi theo thời gian. Nhóm đã tiếp cận theo 2 cách:

* Dùng mutation rate tăng theo thời gian
* Mutation rate cao tại những iteration đầu, tăng độ đa dạng của quần thể.
  + - * Cách 8:

Nhóm đổi cách tạo chuỗi. Chuỗi sẽ được tạo thành bằng cách ghép 9 khối lại với nhau, trong đó mỗi khối 3x3 sẽ được trải phẳng thành mảng 1 chiều 9 phần tử.

Logic cách tạo quần thể ban đầu tương tự cách 4, tạo chuỗi sao cho trong mỗi khối, mỗi giá trị chỉ xuất hiện 1 lần.

Nhóm đổi cách Selection. Thay vì dùng fitness làm trọng số, nhóm sẽ chọn 100\*p% cá thể có fitness cao nhất trong quần thể, 100\*(1-p)% còn lại được chọn ngẫu nhiên.

* Crossover: dùng 2 parent. Cũng tương tự cách 5, ta chọn m khối từ parent1 và 9 - m khối từ parent2.
* Mutation: Không thay đổi so với cách gần nhất.

Với việc thay đổi cách Selection, nhóm nhận thấy rằng chất lượng của quần thể giảm theo thời gian. Nguyên nhân có thể là do 100 \* (1 - p)% còn lại được chọn hoàn toàn ngẫu nhiên dẫn đến việc những cá thể có fitness thấp được chọn làm parent.

* + - * Cách 9:

Đổi cách selection trở lại giống các phiên bản trước.

Các siêu tham số của hàm:

* n\_iters: số lần lặp trước khi kết thúc giải thuật. Thuật toán genetic sẽ dừng khi tìm được nghiệm, để thuật toán dừng trong thời gian hữu hạn ta sẽ thiết lập sao cho thuật toán dừng sau 1 số lần lặp nhất định.
* population\_size: kích thước quần thể, mặc định 1000
* mutation\_rate: tỉ lệ đột biến, mặc định 0.2
  + - * Cách 10 (Hiện tại):

Crossover: ứng với mỗi khối ta chọn ngẫu nhiên 1 của của 1 trong 2 parent

Ta thử cho thuật toán với tham số mặc định, số lần lặp bằng 5000 và kết quả thu được:

A graph of blue and orange lines

Description automatically generated

**3.1.2 Giải thích nguyên nhân thuật toán không tìm được nghiệm**

Dưới đây là một số dự đoán của nhóm về nguyên nhân thuật toán không tìm được lời giải:

* Thuật toán phụ thuộc vào hàm fitness và dễ rơi vào các điểm tối ưu cục bộ. Hàm mutation không đủ tốt để thoát khỏi các điểm tối ưu cục bộ.
* Sudoku là 1 bài toán phức tạp đối với thuật toán này. Các cách tiếp cận khác như DFS, constraint satisfaction problem (CSP)... là phù hợp hơn.

**3.2 Thời gian chạy và tiêu tốn bộ nhớ của thuật toán.**

Vì thuật toán không tìm được nghiệm nên ta sẽ chỉ phân tích thời gian chạy và sự tiêu tốn của bộ nhớ trên câu đố này:

A screen shot of a computer screen

Description automatically generated

**3.2.1 Thời gian chạy của thuật toán**

Ta sẽ chọn các tham số khác là tham số mặc định và xét ảnh hưởng của số lần lặp lên thời gian chạy của thuật toán. Ta lần lượt thử số lần lặp bằng 1000, 2000, 3000, 4000, 5000:

A graph with a line

Description automatically generated

Hiển nhiên, thời gian chạy của thuật toán là thời gian từ lúc bắt đầu đến khi kết thúc thuật toán (và do đó tỉ lệ với số lần lặp).

Bây giờ, chúng ta sẽ để số lần lặp bằng 1000, vẫn giữ các tham số khác là mặc định và xét ảnh hưởng của kích thước quần thể lên thời gian chạy:

A graph with a line

Description automatically generated

Từ đồ thị ta thấy được thời gian chạy của thuật toán tỉ lệ thuận với kích thước quần thể. Điều này khá dễ hiểu do ở mỗi lần lặp ta phải duyệt cả quần thể để tính fitness score và sinh quần thể mới.

Mutation rate thay đổi sẽ không làm tăng đáng kể thời gian chạy của thuật toán. Giả sử mutation\_rate = 1 thì ứng với mỗi cá thể con được tạo ra ta tốn O(1) => cả quần thể sẽ tốn O(population\_size).

Kết luận: độ phức tạp thời gian là O(n\_iters \* population\_size)

**2.2.2 Tiêu tốn bộ nhớ của thuật toán**

Ta sẽ chọn các tham số khác là tham số mặc định và xét ảnh hưởng của số lần lặp lên sự tiêu tốn bộ nhớ của thuật toán. Ta lần lượt thử số lần lặp bằng 1000, 2000, 3000, 4000, 5000:

A graph with a line

Description automatically generated

Thực chất, sự tiêu tốn bộ nhớ của thuật toán không phụ thuộc vào thời gian chạy. Tuy nhiên, vì nhóm có lưu lại thông tin là fitness của cá thể tốt nhất và tệ nhất trong mỗi lần lặp nên ta thấy bộ nhớ là 1 hàm tuyến tính theo số lần lặp. Sự tiêu tốn bộ nhớ thực sự của thuật toán này nằm ở kích thước quần thể:

A graph with a line

Description automatically generated

Tương tự như độ phức tạp thời gian, mutation rate không gây ảnh hưởng nhiều đến độ phức tạp không gian của thuật toán.

Kết luận: Độ phức tạp không gian của thuật toán là O(population\_size + n\_iters) nếu lưu thông tin về fitness score, O(population\_size) nếu trái lại.