Table Of Contents

Swarm Intelligence

Giới thiệu

Swarm Intelligence là mô hình đa tác tử thông minh được xây dựng dựa trên tập tính bầy đàn của động vật trong tự nhiên. Swarm Intelligence thường được sử dụng trong các chương trình độ họa, các hệ thống tự sắp xếp hay trong một số bài toán tìm kiếm tối ưu

Ba hướng phát triển chính của Swarm Intelligence là Swarm Robotic, Computer Graphic và Tìm kiếm đồ thị

Các mô hình

Các mô hình được xây dựng dựa trên mô hình Swarm Intelligence là:

- Mô hình Boids
- Thuật toán Stochastic Diffusion Search
- Nhánh thuật toán Ant Colony Optimization
- Particle swarm optimization

Tài liệu tham khảo

Wikipedia: Swarm Intelligence

Boids

Giới thiệu

- Là mô hình được xây dựng để mô phỏng lại tập tính bầy đàn của động vật
- Được phát triển vào năm 1986, trong điều kiện là các chương trình mô phỏng còn hạn chế. Thường không có các tác nhân như lực tác động hay tính toán va chạm
- Mô hình chỉ đơn giản là mô tả làm sao cho giống chuyển động của bầy cá hoặc bầy chim trong tự nhiên

Mô tả

Boids là một tác tử đại diện cho một con vật trong đàn

Mỗi tác tử trên tuân theo ba luật lệ sau:

- **Separation** : Lái nghiêng đi để tránh đám đông
- **Aligment** : Hướng về hướng chung của bầy đàn
- **Cohension** : Bay sát về phía trung tâm của bầy đàn

Mỗi tác tử có khả năng truy cập trực tiếp vào tất cả yếu tố đồ họa của khung hình. Tuy nhiên tập tính bầy đàn chỉ có phép tác tử phản ứng cục bộ ở bên trong vùng lân cận.

Vùng lân cận được xác định bởi một bán kính khoảng cách xung quanh tác tử và góc nhìn (trừ đi khoảng mù phía sau của tác tử đó).

Pseudocode

> Nguồn: KFish

Trong mỗi khung hình chương trình sẽ thực hiện hai công việc:

- `draw_boids()` : Vẽ các tác tử đại diện cho động vật
- `move_all_boids_to_new_positions()` : Di chuyển các tác tử tới vị trí mới

```
initialise_positions()

LOOP
    draw_boids()
    move_all_boids_to_new_positions()
END LOOP
```

Việc di chuyển các tác tử tới vị trí mới phải tuân theo ba luật lệ là `rule1, rule2, rule3`. Kết quả thu được là ba vector vận tốc `v1, v2, v3`.

Vận tốc của tác tử `Boid b` sẽ bằng tổng của vận tốc ban đầu cộng với vận tốc thu được từ việc thực hiện ba luật trên. Sau đó vị trí của các tác tử sẽ được cập nhật

```
PROCEDURE move_all_boids_to_new_positions()

    Vector v1, v2, v3
    Boid b

    FOR EACH BOID b
        v1 = rule1(b)
        v2 = rule2(b)
        v3 = rule3(b)

        b.velocity = b.velocity + v1 + v2 + v3
        b.position = b.position + b.velocity
    END

END PROCEDURE
```

Luật thứ 1: Các tác tử sẽ cố gắng bay để tránh đám đông

Tác tử sẽ cố gắng bay tránh những tác tử ở gần mình

```
PROCEDURE rule1(boid bJ)

    Vector c = 0;

    FOR EACH BOID b
        IF b != bJ THEN
            IF |b.position - bJ.position| < 100 THEN
                c = c - (b.position - bJ.position)
            END IF
        END IF
    END

    RETURN c

END PROCEDURE
```

Luật thứ 2: Các tác tử sẽ bay theo hướng của cả đàn

```
PROCEDURE rule2(boid bJ)

    Vector pvJ

    FOR EACH BOID b
```

```
        IF b != bJ THEN
            pvJ = pvJ + b.velocity
        END IF
    END

    pvJ = pvJ / N-1

    RETURN (pvJ - bJ.velocity) / 8

END PROCEDURE
```

Luật thứ 3: Các tác tử sẽ cố gắng bay sát theo đàn, hướng về tâm của đàn

```
PROCEDURE rule1(boid bJ)

    Vector pcJ

    FOR EACH BOID b
        IF b != bJ THEN
            pcJ = pcJ + b.position
        END IF
    END

    pcJ = pcJ / N-1

    RETURN (pcJ - bJ.position) / 100

END PROCEDURE
```

Ngoài ra chúng ta cũng có thể cài đặt thêm các tính năng là bay theo tác tử dẫn đầu hoặc bay theo mục tiêu nhất định, tránh gió

Cải tiến

Vì trong phát biểu của mình về mô hình `boids` tác giả cũng không nói rõ về cách thức hoạt động và mô hình vật lý mà chỉ đưa ra phát biểu tương đối nên chúng ta cũng có thể dựa vào các luật trên để cải tiến thuật toán như sẽ nêu trong phần demo

Demo

> Demo được xây dựng dựa trên Flocking Fish Demo

Chương trình demo mô phỏng tập tính bầy đàn như trong mô hình `boids` tuy nhiên có nhiều thay đổi

Giống như trong mô hình `boids` demo cũng xây dựng các tác tử dựa trên mô hình luật:

- **Separation**: Các tác tử không được bay quá gần nhau
- **Cohension**: Các tác tử có khả năng kết hợp với những tác tử bay xung quanh tạo thành đàn
- **Aligment**: Các tác tử sẽ cố gắng bay cùng hướng đối với các tác tử cùng đàn
- **Persuit**: Các tác tử có khả năng bay theo đuổi mục tiêu
- **Avoid Obstacles**: Tránh chướng ngại vật từ xa

Những khó khăn gặp phải:

- Một là khác với các mô hình `boids` khác mô hình boids trong demo có đưa các yếu tố vật lý là va chạm giữa các tác tử vào. Do đó việc lập trình cho tác tử sẽ khó hơn
- Hai là hình dạng các tác tử cũng rất đặc biệt (hình tam giác). Vì vậy hai tác tử bay theo cùng một hướng nếu va chạm thì đầu chúng sẽ chếch về phía nhau rất khó tách ra. Vấn để tương tự cũng xảy ra khi tác tử va vào chướng ngại vật

Demo sử dụng những phương pháp sau đây

Xác định vận tốc

Mỗi tác tử sẽ có hai vector vận tốc một là vector vận tốc mong muốn `targetVelocity` hai là vector vận tốc thực tế `velocity`

Dựa vào các luật chúng ta có thể tính được vector `targetVelocity`. Tuy nhiên vector này có thể cùng phương, có thể khác phương với chuyển động của của tác tử. Vector này cũng có thể cùng chiều, cũng có thể ngược chiều với phương chuyển động. Giá trị của vector này cũng có thể vượt quá vận tốc tối đa mà tác tử có thể chuyển động

Trong khi đó tác tử chỉ có thể chuyển động thẳng với vector vận tốc tiếp tuyến và quay sử dụng vector quay

Do đó để tính toán vận tốc thực tế chúng ta làm như sau

- Giới hạn độ lớn của `targetVelocity`
- Tính toán độ lớn của vector tiếp tuyến bằng cách lấy hình chiếu. Thực tế của việc lấy hình chiếu là nhân với `cos` góc xen giữa. Tuy nhiên việc này lại gây ra vận tốc âm. Do đó thực tế chúng ta sẽ tính bằng công thức `(cos(alpha) + 1) * |targetVelocity|` sẽ ra vận tốc tiếp tuyến. Nhờ đó vận tốc tiếp tuyến sẽ không âm
- Góc quay tỉ lệ thuận với góc xen giữa và bị giới hạn bởi tốc độ quay tối đa

Việc tính `targetVelocity` được mô tả dưới đây

Duy trì khoảng cách giữa các tác tử

Mỗi tác tử sẽ xác định một bán kính lân cận và một khoảng cách an toàn

Tác tử sẽ duy trì khoảng cách với các tác tử khác trong vùng lân cận bằng một vận tốc duy trì

Vận tốc duy trì là hàm phụ thuộc vào hiệu giữa `khoảng cách từ tác tử này đến tác tử kia` và `khoảng cách an toàn`

Việc cài đặt hàm này thế nào phụ thuộc vào chúng ta. Độ biến thiên càng lớn thì khả năng duy trì khoảng cách an toàn càng cao

Sắp xếp để cùng hướng với các tác tử xung quanh

Vận tốc sắp xếp này thường được tính là bằng vận tốc trung bình của các tác tử xung quanh nó. Tuy nhiên việc này dẫn đến việc vận tốc bị gấp đôi lên theo lũy thừa. Do đó

nên vận tốc này thường được nhân với một tỉ lệ phần trăm nào đó

Tránh chướng ngại vật

Việc xây dựng khả năng tránh vật cản cho tác tử tương tự như việc chúng ta xây dựng khả năng duy trì khoảng cách cho các tác tử. Khi tác tử nhận thấy nó tới gần chướng ngại vật nào đó. Nó có thể tính toán một vận tốc đẩy cùng phương với đường nối tâm từ vật cản tới tác tử. Độ lớn của vector phụ thuộc vào khoảng cách giữa tác tử và vật cản

Đuổi theo mục tiêu

Vận tốc đuổi theo mục tiêu nên được đặt là vận tốc cố định vào có hướng từ tác tử tới mục tiêu

Vận tốc mong muốn `targetVelocity` sẽ được tính bằng tổng các vận tốc trên cộng lại

Mã nguồn của demo có thể tham khảo tại: Boids Demo

Tài liệu tham khảo

Homepage: red3d

Wiki:

- Boids

Pseudocode:

- KFish

Source code:

- Flocking Fish Demo

Ant Colony Optimization

Giới thiệu

Là một trong những thuật toán đại diện cho Swarm Intelligence được Marco Dorigo đề xuất vào năm 1992

Thường Ant Colony Optimization (ACO) được gọi là một nhánh thuật toán vì cũng có khá nhiều thuật toán mở rộng khác nhau được xây dựng trên nền thuật toán này. Bản thân Ant Colony Optimization cũng được xây dựng trên nền mô hình Ant System nhưng được phát triển với mục đích riêng là giải quyết bài toán trên đồ thị

Các thuật toán mở rộng có thể kể đến đó là:

- Max-min ant system (MMAS)
- Ant colony system
- Rank-based ant system (ASrank)
- Continuous orthogonal ant colony (COAC)

Pseudocode

> Nguồn: Wikipedia

```
procedure ACO_MetaHeuristic
    while(not_termination)
        generateSolutions()
        daemonActions()
        pheromoneUpdate()
    end while
end procedure
```

Mô tả

Thuật toán tận dụng thế mạnh của thuật toán ngẫu nhiên kết hợp với thuật toán tìm kiếm cục bộ để giải các bài toán

Tổng quan bài toán gồm các bước như sau:

1. Khởi tạo trọng số trên các tuyến đường
2. Thực hiện một số lần hữu hạn các vòng lặp

- Đưa ra một loạt các kết quả thông qua việc tìm kiếm cục bộ kết hợp với tìm kiếm ngẫu nhiên trên đồ thị
- Chọn ra tuyến đường tốt nhất và cập nhật
- Cập nhật lại trọng số trên đồ thị dựa vào kết quả thu được

Đối với bài toán cụ thể là Travelling Salesman Problem trọng số phụ thuộc vào hai yếu tố đó là pheromone và độ dài tuyến đường.

Ban đầu pheromone chưa biết ta có thể khởi tạo pheromone bằng nhau. Bài toán giống với Nearest Neighbor (NN). Tuy nhiên điểm khác giữa ACO và Nearest Neighbor là Nearest Neighbor sẽ chọn điểm gần nhất tại mỗi điểm, ACO không chọn cạnh có trọng số lớn nhất để đi luôn mà nó sẽ chọn ngẫu nhiên theo xác suất. Cạnh nào có trọng số lớn hơn thì xác suất được chọn lớn hơn

Sau mỗi vòng lặp pheromone trên các cạnh sẽ bay hơi một phần. Kiến cũng để lại pheromone trên các cạnh mà nó đi qua dựa vào đánh giá của nó về chi phí của tuyến đường.

So sánh với NN

Việc lựa chọn ngẫu nhiên các cạnh khiến cho ACO trông có vẻ bất lợi hơn thuật toán NN. Tuy nhiên trong một vòng lặp, ACO không chỉ đưa ra một kết quả mà cho nhiều con kiến đi để tìm ra các kết quả khác nhau. Điều này giúp cho ACO lợi thế hơn thuật toán NN là chỉ tìm ra quãng đường có chi phí cục bộ

Ngoài ra việc cập nhật pheromone trên các tuyến đường giúp cho thuật toán ACO có khả năng thay đổi trọng số các cạnh, đánh giá tuyến đường và tối ưu hóa.

Chúng ta cũng có thể sử dụng ACO để tìm ra nhiều tuyến đường khác nhau mà chi phí tương đương nhau

Việc giải bài toán bằng thuật toán ACO có thể được ví như `việc để mọi thứ diên ra theo tự nhiên`

Nhược điểm của ACO là nếu chỉ xét trên bài toán Travelling Salesman Problem thì đối với NN hay 2OPT việc code ACO sẽ phức tạp hơn mà hiệu quả hơn không đáng kể

Đối với kích thước bài toán to hơn thì hiệu năng của thuật toán cũng giảm đáng kể và yêu cầu tài nguyên cũng tăng cao

Một số thuật toán mở rộng có thể giúp mở rộng thuật toán đối với những bài toán lớn hơn như Ant Colony System nhưng cũng yêu cầu thuật toán phức tạp hơn

Chương trình demo

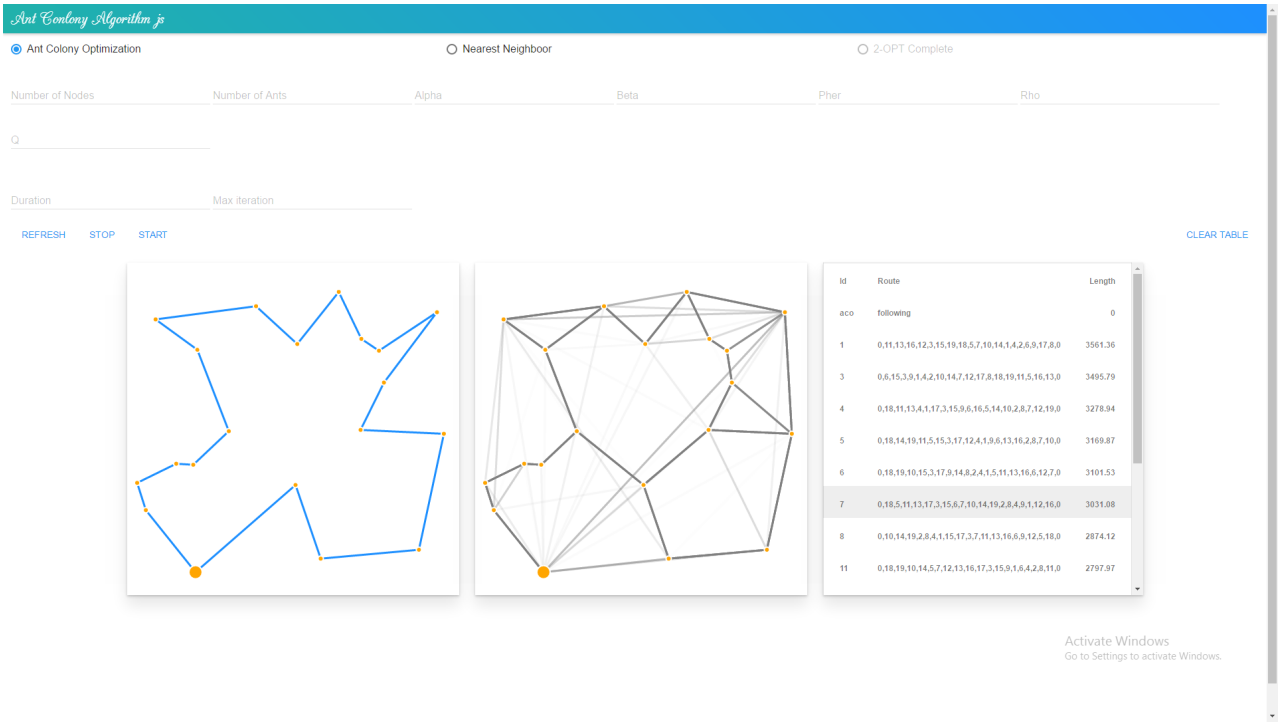
Giới thiệu

Chương trình là demo của thuật toán Ant Colony Optimization (ACO) trên nền bài toán Travelling Salesman Problem (TSP)

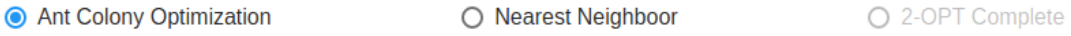
Bên cạnh đó chương trình cũng lấy hai thuật toán là Nearest Neighbor (NN) và 2-OPT Complete để làm tiêu chuẩn so sánh

Demo được xây dựng dựa trên demo aco-js của GordyD

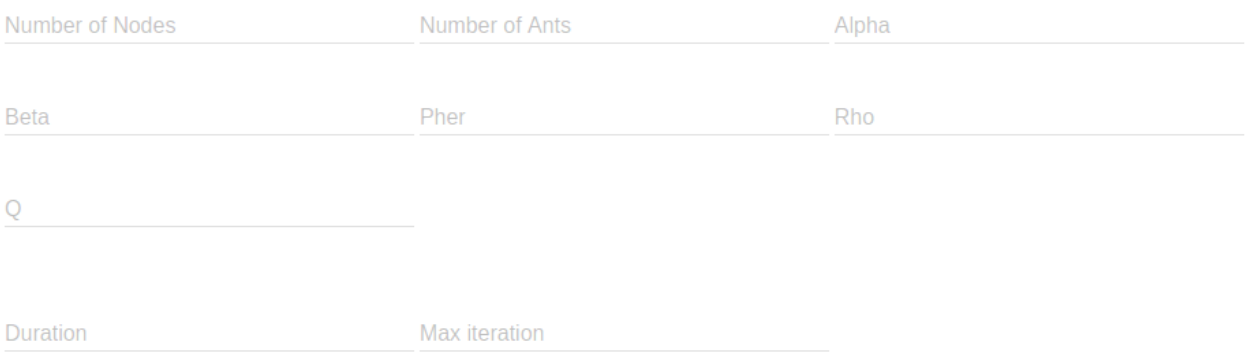
Giao diện chính của chương trình



Lựa chọn thuật toán



Các giá trị cài đặt cho thuật toán ACO



Nếu bị bỏ trống chương trình sẽ chạy với các giá trị mặc định sau

```
+ Số đỉnh: 20
+ Số con kiến: 20
+ Alpha: 1, Beta: 1
+ Rho: 0.1
+ Q: 100

+ Iteration: 200
+ Duration: 100
```

Hiển thị kết quả

Phần hiển thị kết quả gồm có 3 ô với nhiệm vụ của từng khung là:

- Ô bên trái hiển thị tuyến đường.
- Ô ở giữa hiển thị giá trị pheromone trên các tuyến đường.
- Ô bên phải là bảng các tuyến đường và chi phí tương ứng.

Ta có thể kích vào dòng bất kì để hiển thị lại tuyến đường của dòng đó.

Các phím chức năng

- **Refresh**: Khởi tạo lại sử dụng khi số đỉnh thay đổi
- **Stop**: Dừng lại sử dụng khi thuật toán ACO chạy quá lâu
- **Start**: Để bắt đầu thực hiện Demo
- **Clear Table**: Xóa bảng lưu trữ tuyến đường khi bảng quá dài và chứa nhiều dữ liệu cũ

Tài liệu tham khảo

Research Gate:

- High-level pseudo-code for the ACO algorithm
- Using Ant Colony Optimization (ACO) on Kinetic Modeling of the Acetoin Production in Lactococcus Lactis C7

Wikipedia:

- Ant
- Travelling salesman problem
- Ant colony optimization algorithms

Source-code:

- aco-metaheuristic
- aco-js