Simulated Annealing을 통한 외판원 문제의 근사

20110 김채완
20514 유재원

목차

I. 서론

1. P, NP

P, NP 문제의 정의를 알아보겠습니다.
어떠한 결정 문제 A를 다항시간에 해결하는 것이 가능하다면 A ⊂ P입니다.

Q) 크기 n의 수열 x가 주어질 때 x의 원소 중 t가 존재하는지 확인하라.
A가 위와 같을 때 문제의 답은 참 거짓 중 하나입니다.
따라서 결정 문제이며 아래와 같이 풀이했을 때 O(n)에 확인할 수 있기 때문에
위 문제는 P에 속합니다.

```
int solve(int n, int x[], int t) {
        int answer = 0;
        for(int i=0; i<n; i++) if(x[i] == t) answer = 1;
        return answer;
}
```

NP 문제는 문제와 답을 알고있을 때 이 답을 다항시간안에 검증할 수 있는
문제를 뜻합니다. 모든 NP 문제를 어떠한 문제 A로 환원할 수 있다면
A⊂NP-hard이며 A⊂NP-hard, A⊂NP를 모두 만족한다면 A⊂NP-Complete
입니다. NP-Complete 문제의 대표적인 예로는 Circuit-Sat, TSP(외판원 문제),
Subset-Sum 문제 등이 존재합니다.
어떠한 문제가 NP-Complete임을 증명할 때에는 대체로 그 문제가 NP에
속하며 3-Sat 문제로 환원됨을 보입니다.

2. 근사 알고리즘

최적화 문제에 대해 최적해에 가까운 해를 구하는 알고리즘을 뜻합니다.
항상 최적해를 구해내는 것이 보장되지 않지만 그에 가까운 해를 구합니다.
이 프로젝트는 NP-Complete인 외판원 문제를 근사 알고리즘을 통해
빠르게 해결하고 이 과정을 시각화하는 것을 목표로 합니다.

II. TSP

정의

TSP는 Traveling Salesman Problem의 약자로 우리말로 외판원 문제이며
가중치 있는 완전 그래프 G = (V, E)가 있을 때, G의 한 정점에서 출발해 모든
정점을 방문하고 시작점으로 돌아오는 최소 가중치 합을 구하는 최적화 문제입니다.

Solution 1) 동적 계획법

D[now][state]
:= 현재 now 정점에 위치하며 정점들의 방문 상태가 state일 때 최소 비용
위처럼 수열 D를 정의하여 O($N \times 2^N$)시간에 최적해를 구합니다.
$N = 16$ 정도가 이 풀이를 사용할 수 있는 상한선입니다.

```
ll f(int x,int state){
       ll &ret=dp[x][state];
       if(ret!=-1)return ret;
       if(state==(1<<N)-1 and w[x][0]==0)return
ret=0x3f3f3f3f;
       if(state==(1<<N)-1 and w[x][0])return ret=w[x][0];
       ret=0x3f3f3f3f;
       for(int i=0;i<N;i++){
           if((state&(1<<i))==0 and w[x][i]){
                rmin(ret,f(i,state|(1<<i))+w[x][i]);
             }
    }
    return ret;
}
```

II. TSP

Solution 2) 분기 한정법

외판원 문제를 보고 떠올릴 수 있는 직관적인 풀이는 다음과 같습니다.

1. 방문할 정점의 순서를 $N!$개 모두 나열해본다.
2. 나열한 모든 순서에 대해 비용을 더해본다.

위 풀이는 대략 O($N \times N!$)의 시간복잡도를 가집니다.
O($N \times N!$)은 굉장히 느리기 때문에 조금의 최적화를 진행합니다.

1. 다음 경로의 가중치 합의 상한선을 구한다.
2. 현재의 최적해보다 상한선이 작으면 이를 채택한다.
3. {1, 2}를 반복한다.

위의 분기 한정법을 이용하면 개선의 여지는 있지만
여전히 빠르다 할 수 없습니다.
$N = 20$ 정도가 분기 한정법을 사용할 수 있는 상한선입니다.

II. TSP

Simulated Annealing

Simulated Annealing, 담금질 기법은 고려해야할 상태공간이 클 때
Global Minima를 찾는 등 근사치를 찾는 휴리스틱 알고리즘입니다.

경사하강법(Gradient Descent)은 현재 상태에서 평가함수의 미분계수를 구한 후
다음 상태의 후보 중 평가함수의 미분계수의 절대값이 낮은 방향으로 속도를 줄이며
나아가는 최적화 알고리즘입니다. 경사하강법은 극값이 다수 존재할 때
지역 최소점에 빠질 수 있는 문제점이 존재합니다.

Simulated Annealing은 랜덤을 이용해 경사하강법의 문제점을 해결할 수 있습니다.
Simulated Annealing의 기본적인 동작은 다음과 같습니다.

1. 초기 상태를 만든다.
2. 다음 상태의 후보 중 하나를 골라 선택했을 때 이득이 될 확률 A를 계산한다.
3. 0%~100%사이의 랜덤한 확률을(B) 뽑아서 A > B일 때 상태전이.
4. {2, 3}을 적당히 반복

다음 상태를 선택했을 때 이득이 될 확률을 구하는 방법은 맥스웰 볼츠만 분포에
근거합니다. 다음 상태를 선택했을 때 평가함수의 값 = e2,
현재의 평가함수 값 = e1이라 정의합니다. (평가함수의 값은 클수록 좋은 상태)

$$P(e1,\ e2)\ =\ e^{(e2\ -\ e1)\ /\ (K\ \times\ T)}$$

K는 볼츠만 상수, T는 온도를 의미하며 온도에 따른 에너지 분포 확률을 나타내는
맥스웰 볼츠만 분포의 에너지에 평가함수의 차를 대입해 다음 상태로 전이했을 때
이득일 확률을 계산합니다.

$e1\ \leq\ e2$인 경우 P는 항상 1이상이기 때문에 전이합니다. 이 때 지역 최소점에 빠지는
것을 방지하기 위해 P가 1 미만일 때에도 랜덤하게 선택한 확률보다 크다면 전이합니다.
$e1\ >\ e2$라면 e의 지수가 음수이기 때문에 K, T가 작아질수록 P는 커집니다.
$(0\ \leq\ P\ <\ 1)$ 이 때 온도 T를 낮춰가면서 진행하기 때문에 $e1\ >\ e2$일 때
전이할 확률은 점점 낮아집니다.

II. TSP

설계

생각해야할 것이 3가지 있습니다.

1. 인접 상태 정의
2. 평가 함수
3. 볼츠만 상수, 시작 온도, 온도 감률

1. 인접 상태 정의

일단 상태는 Sol 2와 같이 정점을 방문하는 순열로 관리합니다.
그렇다면 인접 상태로 정해볼만한 것이 몇가지 있습니다.

1. 방문 순서에서 랜덤한 두 원소 swap
2. 방문 순서에서 랜덤한 인접한 두 원소 swap
3. 방문 순서에서 구간 뒤집기
4. k-opt (랜덤하게 구간 k개로 분할해 랜덤한 순서로 조립, 일반적으로 k=3)

인접 상태는 말 그대로 “인접한", 가까운 상태가 좋습니다.
1번 방법은 교체되는 간선이 4개, 2번 방법은 3개,
3번 방법은 2개, 4번 방법은 k=3일 때 3개입니다.
따라서 3번 방법이 가장 좋은 정확도와 성능을 보여줄 것이라 생각할 수 있습니다.

(각 방법을 직접 비교해본 결과는 [결과]에 정리해두었습니다)

II. TSP

2. 평가 함수

TSP에서의 평가함수는 간단히 정의할 수 있습니다.
TSP 최적해의 값이 작을 수록 좋으니
(평가함수 = $-1 \times$ 최적해)로 정의할 수 있습니다.

3. 볼츠만 상수, 시작 온도, 온도 감률

$$P(e1, e2) = e^{(e2 - e1) / (K \times T)}$$

P의 값에 $(e2 - e1) / (K \times T)$가 큰 영향을 주기 때문에
지역 최소에 빠지지 않도록 랜덤하게 이득이 아닌 경우도
잘 살펴보도록 전이해야합니다. 그에 맞게 $(e2 - e1)$의 기대값과
$(K \times T)$의 비율을 적당히 조절해야 합니다.

III. 구현

1. Render

- Canvas를 사용해 정점과 간선을 그렸고 Generator function을 사용해 상태 전이가 있을 때마다 간선을 새로 그렸습니다.

- 1 frame을 100ms로 설정해 상태전이를 하나하나 눈으로 확인할 수 있도록 구현했습니다.

```
function nodeDraw(n: number, mp: {x: number, y: number}[]) {
    const cvs = document.getElementById("asdf")! as HTMLCanvasElement;
    const ctx = cvs.getContext('2d')!;
    const __nodeDraw = (p : number, q : number) => {
        ctx.beginPath();
        ctx.arc(p * R + BASE, q * R + BASE, 10, 0, Math.PI * 2, false);
        ctx.stroke();
        ctx.fillStyle = 'blue'; ctx.fill();
    };
    for(let i=0; i<n; i++) __nodeDraw(mp[i].x, mp[i].y);
}

function edgeDraw(mp: {x: number, y: number}[], edges: Array<[number, number]>) {
    const cvs = document.getElementById("asdf")! as HTMLCanvasElement;
    const ctx = cvs.getContext('2d')!;
    const __edgeDraw = (edge: [number, number]) => {
        const currentMps = [mp[edge[0]], mp[edge[1]]];
        ctx.beginPath();
        ctx.moveTo(BASE + currentMps[0].x * R, BASE + currentMps[0].y * R);
        ctx.lineTo(BASE + currentMps[1].x * R, BASE + currentMps[1].y * R);
        ctx.strokeStyle = 'black';
        ctx.lineWidth = 4;
        ctx.stroke();
    };
    for(const edge of edges) __edgeDraw(edge);
}
```

III. 구현

1. Render

```
function getCanvas() {
    const cvs = document.getElementById("asdf")! as HTMLCanvasElement;
    const ctx = cvs.getContext('2d')!;
    ctx.clearRect(0, 0, cvs.width, cvs.height);
}

function render
    (
        n: number,
        mp: {x: number, y: number}[],
        edge: Array<[number, number]>
    )
{
    getCanvas();
    edgeDraw(mp, edge);
    nodeDraw(n, mp);
}

function frame
    (
        gen: Generator<Array<[number, number]>, void, unknown>,
        n: number,
        mp: {x: number, y: number}[]
     )
{
    const edge = gen.next();
    if(edge.value)
        render(n, mp, edge.value);
    setTimeout(() => {
        frame(gen, n, mp);
    }, 100);
}
```

III. 구현

2. Simulated Annealing

dist(i, j) = 정점 i와 정점 j의 거리
Swap(i, j) = 방문순서[i, j]를 flip
cost() = perm에 저장된 순서로 방문할 때 비용
edge = frame마다 그릴 간선의 목록을 저장

[구현]의 코드는 구간 flip로 상태전이를 진행하는 구현입니다.

III. 구현

2. Simulated Annealing

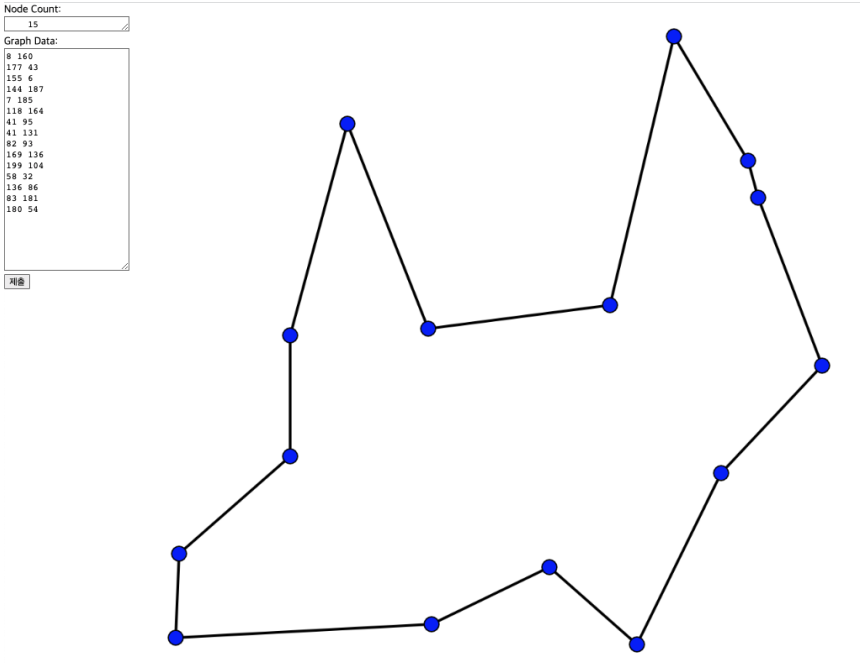
```
function* simulated_annealing(n: number, mp: { x: number, y: number }[]) {
    let T = n / 2;
    const K = 1;
    const D = 0.9999;
    let e1 = 0;
    let e2 = 0;
    const perm: number[] = [];
    mp.sort((a : {x : number, y : number}, b : {x : number, y : number})=>a.x - b.x);
    for (let i = 0; i < n; i++) perm[i] = i;
    const getidx = (idx: number) => { return (idx + n) % n; };
    const dist = (p: number, q: number) => {
        p = getidx(p); q = getidx(q);
        const a = mp[perm[p]].x - mp[perm[q]].x;
        const b = mp[perm[p]].y - mp[perm[q]].y;
        return Math.sqrt(a * a + b * b);
    };
    for(let i=0; i<n; i++) for(let j=i+1; j<n; j++) T = Math.max(T, dist(i, j));
    const Swap = (p: number, q: number) => {
        p = getidx(p); q = getidx(q);
        if (p > q){
            const tmp = p;
            p = q;
            q = tmp;
        }
        for (let i = 0; p + i < q - i; i++){
            const tmp = perm[p + i];
            perm[p + i] = perm[q - i];
            perm[q - i] = tmp;
        }
    };
    let edge: [number, number][] = [];
    e1 = 0;
    for (let i = 0; i < n; i++) e1 += dist(i - 1, i);
    const cost = () => {
        let ret = 0;
        for(let i=0; i<n; i++) ret += dist(i - 1, i);
        return ret;
    };
    for(let tc=0; tc<IT; tc++) {
        for(let i=0; i<1000; i++) {
            const p: number = Math.floor(Math.random() * (n));
            const q: number = Math.floor(Math.random() * (n));
            e1 = cost();
            Swap(p, q);
            e2 = cost();
            const newE : [number, number][] = [];
            for(let i=0; i<n; i++) {
                newE.push([perm[getidx(i - 1)], perm[i]]);
            }
            const x: number = Math.exp((e1 - e2) / (K * T));
            if (e2 < e1 || x >= Math.random()) edge = newE;
            else Swap(p, q);
            T *= D;
        }
        yield edge;
    }
}
```

IV. 결과

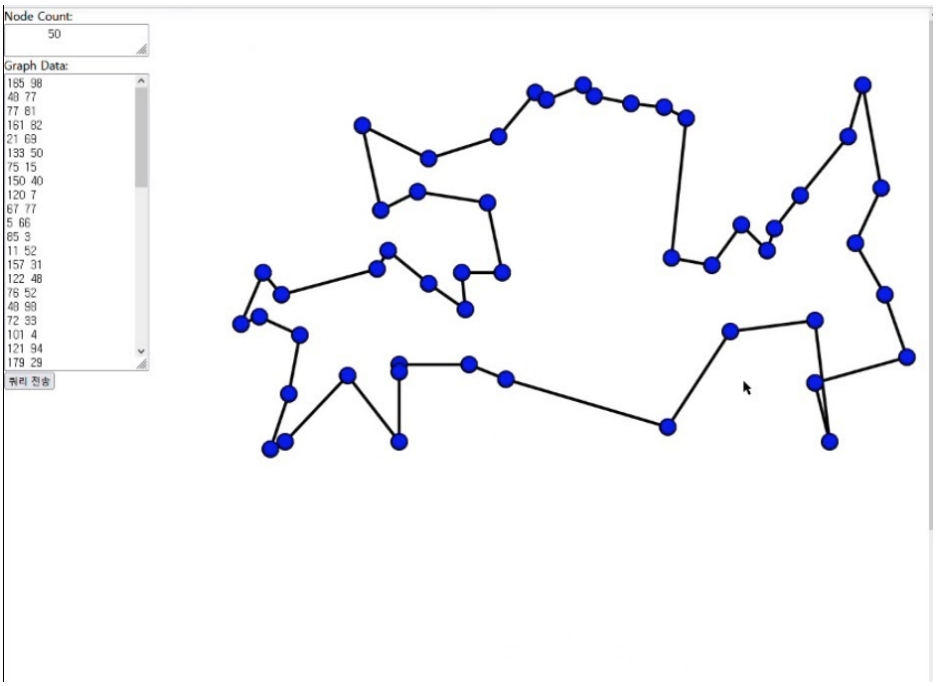
1. 화면의 크기 상 N = 1000정도까지만 테스트해봤으나
   N = 1000일 때에도 굉장히 높은 정확도를 보였습니다.

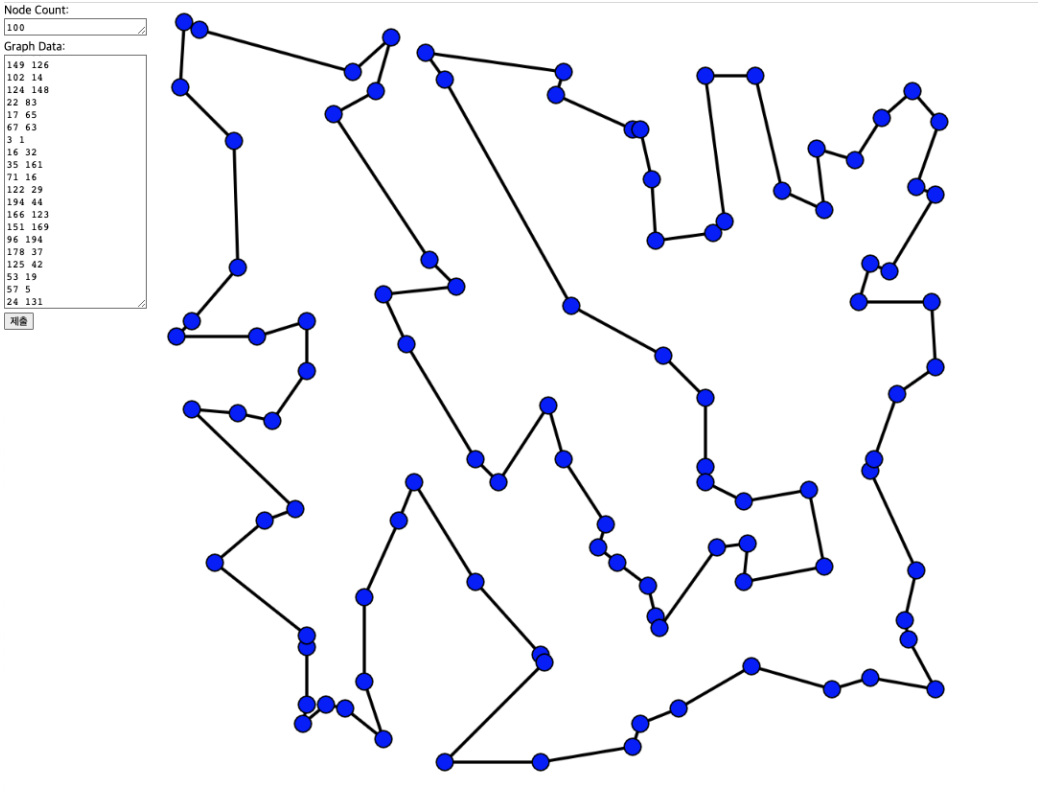
2. {
   인접한 두 원소 swap,
   랜덤한 두 원소 swap,
   K-Opt,
   랜덤한 구간 flip
   }
   순으로 랜덤한 구간 flip이 압도적으로 좋은 성능과
   정확도를 보였습니다.

3. 시연 결과



IV. 결과


Node Count:
50
Graph Data:
165 98
48 77
77 81
161 82
21 69
133 50
75 15
150 40
120 7
67 77
5 66
85 3
11 52
157 31
122 48
76 52
48 98
72 33
101 4
121 94
179 29
쿼리 전송


Node Count:
100
Graph Data:
149 126
102 14
124 148
22 83
17 65
67 63
3 1
16 32
35 161
71 16
122 29
194 44
166 123
151 169
96 194
178 37
125 42
53 19
57 5
24 131
제출