

[Project #1]

Traveling Salesman Problem 소스코드 설명서

[Team Name (9 팀)]

	이름	학번	학년	E-mail
팀장	이창우	2019		
조원 1	이재호	2019		
조원 2	정준서	20192873	3	juns1s@soongsil.ac.kr
조원 3				



TSP를 위한 GA 초기 모집단 생성 알고리즘



23/04/13 00:27

목차

목차
1. 개요
1-1. 기본
2. GA 구현 상세
2-0. 주요 파라미터
2-1. 부모 선택
2-2. 교차 연산
2-3. 돌연변이 연산
2-4. 적합도 산출
3. 영역 Branch & Bound + Heuristic
3-1. 영역 분할
3-2. Backtracking
3-3. Branch & Bound
3-3. Heuristic : 경로 전처리
4. 영역 Convex Hull Insertion
4-1. 초기 아이디어
4-2. Convex Hull Insertion
4-2. 구현 상세
5. 클러스터링 영역 기반의 Greedy 알고리즘
5-1. 개요
5-2. 구현 상세
6. 클러스터링 영역 기반의 Convex Hull Insertion
6-1. 개요
6-2. 구현 상세
7. 전역 Convex Hull Insertion
7-1. 개요
7-2. 구현
8. 성능 측정
8-1. 무작위 + 단순 GA
8-2. 영역 단위 Branch & Bound + Heuristic
8-3. 클러스터 영역의 Greedy Algorithm
8-4. 영역 단위 Convex Hull Insertion
8-5. 클러스터 단위의 Convex Hull Insertion
8-6. 전역 Convex Hull Insertion

1. 개요

1-1. 기본

- 1000개의 2차원 좌표로 이루어진 도시 정점을 입력으로 받아 모든 도시를 순회하는 가장 작은 total cost를 찾는 TSP(Traveling Salesman Problem)을 해결하는 프로그램이다
- 세부 문제 상세

- 도시의 좌표는 최소 ($0.0f$, $0.0f$), 최대 (100.0 , $100.0f$)이다.
- 도시의 초기 간선은 주어지지 않는다.

2. GA 구현 상세

2-0. 주요 파라미터

- 종류
 - a. `populationSize` : 매 세대의 최대 모집단 개수
 - b. `maxCrossoverRate` : crossover 연산의 최대 길이 비율 (백분율)
 - c. `maxMutateRate` : mutate 연산의 최대 길이 비율 (백분율)
 - d. `genThres` : 최대 generation 수
- 디폴트 값

```
const int populationSize = 50;
const int maxCrossoverRate = 65;
const int maxMutateRate = 5;
const int genThres = 500000;
...
mutate 확률 : 25% //main.cpp
부모 랜덤 선택 개수 : 10 //GeneticAlgorithm::selectParent의 randParentCnt
crossover하는 개수 : tspSolver->getPopulationSize(); //main.cpp의 메인루프 2번째 반복
```

2-1. 부모 선택

- 염색체 표현 : 도시의 순열. 즉, 실제 경로를 표현
 - 따라서 유전자는 각 도시가 됨.
- **순위 기반 선택** + **임의 선택**
 - 상위 `populationSize` 개 집단을 선택
 - fitness를 기반으로 오름차순 정렬한 다음, 상위 `populationSize` 개를 제외하고 제거
 - `randParentCnt`
 - `populationSize` 중에서 다양성을 위해 상위 집단이 아닌 하위 집단에서 고를 염색체 수
 - 소스 코드

```
void GeneticSearch::selectParents(vector<Chromosome>& population)
{
    //fitness에 따라 오름차순 정렬
    fitness(population);

    sort(population.begin(), population.end(), compChromosome);

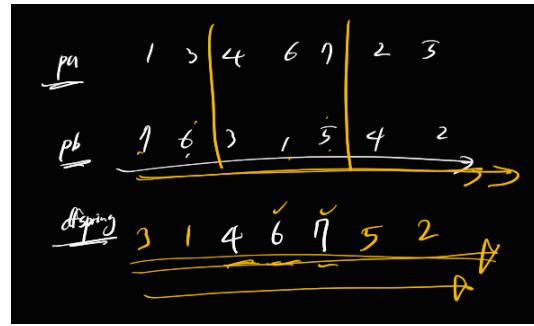
    //순위 기반 선택 -> populationSize 만큼의 상위 집단을 고름
    static const int randParentCnt = 10;
    if(population.size() >= populationSize + randParentCnt);
        population.erase(population.begin() + populationSize, population.end());

    for(int i=0; i<randParentCnt; i++)
    {
        int eraseIdx = getRandomIntVal(populationSize, population.size()-1);
        population.erase(population.begin() + eraseIdx);
    }
}
```

2-2. 교차 연산

- `populationSize` 번 진행, `[0, populationSize-1]`과 랜덤한 idx의 염색체가 대상
- **순열 교차** : 부모 염색체의 일부 구간을 선택하여 자식을 생성하는 방식

```
Chromosome GeneticSearch::crossover(const Chromosome& p1, const Chromosome& p2)
```



- 범위를 지정해서 pa의 부분을 offspring의 같은 영역에 지정
- pb를 기준으로 처음부터 순회하며 offspring의 앞에서부터 채워줌

- 해당 연산을 선택한 이유?**

- 교차된 구간에서 중복이 발생하지 않아 일점, 다점 교차와는 달리 구현이 편함
- 교차된 구간에서의 노드의 순서가 유지되며, 부모의 정보를 최대한 유지하면서 새로운 조합을 생성할 수 있음
 -) 단점? : 두 부모가 같을 경우, 자식도 같은 염색체가 나옴

2-3. 돌연변이 연산

- 범위 기반 돌연변이, `maxMutationRate` 만큼의 길이가 범위
- crossover된 newChild가 연산의 대상이며, 일정 %에 따라 연산 유무를 결정

```
//50% 확률의 mutate 연산
if(tspSolver->getRandomIntVal(1, 100) <= 25)
    tspSolver->mutate(population[cIdx].gene);
tspSolver->repair(newChild);
population.push_back(newChild);
```

- 2가지 돌연변이 연산을 제공
 - swap mutation : 2개의 노드를 무작위로 선택하여 순서를 바꿈
 - inversion mutation : 2개의 노드를 무작위로 선택하여 그 사이의 노드들의 순서를 반대로 뒤집음

- 해당 연산을 선택한 이유?**

- 공통 : local minimum으로부터 벗어날 수 있음
- swap mutation : 무작위의 swap를 통해 특정 구간을 최적화 할 수 있음
- inversion mutation : 무작위의 Inversion은 Twisted 구간을 푸는 효과가 있음

- 소스 코드**

```
bool GeneticSearch::mutate(vector<Node> &child)
{
    if(getRandomIntVal(1, 100) < 90) return inverseMutate(child);
    else return swapMutate(child);
}

bool GeneticSearch::swapMutate(vector<Node> &child)
{
    int rIdxA, rIdxB, t; //random index, try count
    const int maxMutationLength = cities.size() * maxMutationRate / 100;

    rIdxA = getRandomIntVal(1, child.size()-2);
    rIdxB = getRandomIntVal(rIdxA, min((int)child.size()-1, rIdxA + maxMutationLength));
    swap(child[rIdxA], child[rIdxB]);

    return true;
}

bool GeneticSearch::inverseMutate(vector<Node>& child)
{
    int rIdxA, rIdxB, t; //random index, try count
```

```

const int maxMutateLength = cities.size() * maxMutateRate / 100;

rIdxA = getRandomIntVal(1, child.size()-2);
rIdxB = getRandomIntVal(rIdxA, min((int)child.size()-1, rIdxA + maxMutateLength));
if(rIdxA == rIdxB) return false;
reverse(child.begin()+rIdxA, child.begin()+rIdxB);
return true; //success
}

```

2-4. 적합도 산출

- 각 염색체의 실제 totalCost를 유클리드 거리 기반으로 산출
- 소스 코드

```

void GeneticSearch::fitness(vector<Chromosome>& population)
{
    //모집단의 염색체 하나씩을 돌면서 fitness 계산
    for(auto &ch : population)
    {
        vector<Node> child = ch.gene;
        double fitnessSum = 0.0f;

        Node prev = child[0];
        for(int idx = 0; idx < child.size(); idx++) //총 경로 cost 계산
        {
            fitnessSum += getDistance(child[idx], child[(idx+1) % child.size()]);
        }
        ch.fitnessVal = fitnessSum;
        currFitnessAvgValue += fitnessSum;
        minFitnessValue = min(minFitnessValue, ch.fitnessVal);
    }
    currFitnessAvgValue /= population.size();
}

```

3. 영역 Branch & Bound + Heuristic

3-1. 영역 분할

- 한 변의 길이가 20인 정사각형 25개로 나누어 주었음. 즉, 5*5로 영역을 분할
- `TreeRouteFinder`의 생성자와 `initAreaAdjList()` 함수 내에서 영역 데이터를 초기화
- `initAreaAdjList()` 함수는 각 정점과 가까운 최대 `MAX_BRANCH_COUNT` 개의 간선을 잇는다.

3-2. Backtracking

- `TreeRouteFinder::findAreaMinimumRoute()` 함수에서 진행
- 세부 파라미터는 아래와 같다.

```

void TreeRouteFinder::findAreaMinimumRoute
(
    const int areaId           //현재 영역 ID
    , int curr                 //탐색할 정점
    , vector<Node> currState //지금까지 탐색한 route 정보
    , double currCost          //지금까지 탐색한 route의 totalCost
){...}

```

- 모든 상태 트리에 대한 탐색을 시작
→ 그러나, 노드가 가장 많은 영역은 100개가 넘음, 100!의 상태 노드의 수

3-3. Branch & Bound

- `TreeRouteFinder::findAreaMinimumRoute`
- 영역 분할 단계에서 최대 branch 개수를 `MAX_BRANCH_COUNT`로 한정 시켜놓음
또한 `currCost` 가 현재까지 찾은 `minCost` 보다 크다면, `pruning` 을 진행

- 소스 코드

```

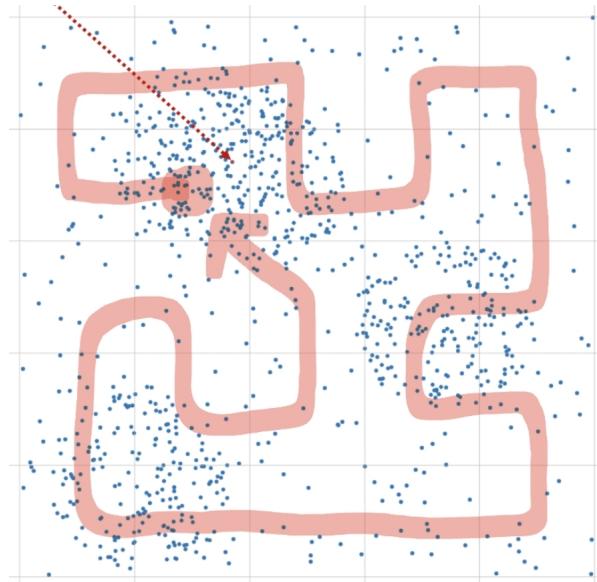
void TreeRouteFinder::findAreaMinimumRoute(const int areaId, int curr, vector<Node> currState, double currCost)
{
    ...
    for(auto &nextInfo : adj[curr])
    {
        const int next = nextInfo.second;
        const int cost = nextInfo.first;

        if(visited[next]) continue;
        if(currCost + cost > minCost[areaId]) break; //Pruning : 현재까지 찾은 minCost보다 크다면 가지를 침
        ...
    }
}

```

3-3. Heuristic : 경로 전처리

- `areaID` 를 순서대로 하게 된다면, 인접한 영역이 아닌 거리가 먼 영역과의 간선이 연결된다.
 - 따라서, 영역간 순서를 초기에 강제하여 최악의 간선이 연결되는 것을 방지한다.



- 소스코드

```

const int tufuOrder[25] = { 8, 3, 4, 9, 14
                          , 13, 18, 19, 24, 23
                          , 22, 17, 16, 21, 20
                          , 15, 10, 5, 0, 1
                          , 2, 7, 6, 11, 12};

for(int idx = 0; idx < subRouteFinder->getTotalAreaCount(); idx++)
{
    const int areaId = tufuOrder[idx];
    vector<Node> areaMinRoute; //영역당 최소 경로가 들어갈 벡터
    const int startIdx = subRouteFinder->getAreastartIndex(areaId); //영역 내 시작 idx

    //각 영역의 부분해를 구한다.
    subRouteFinder->findAreaMinimumRoute(areaId, startIdx, areaMinRoute, 0);
    areaMinRoute = subRouteFinder->getMinRoute(areaId);

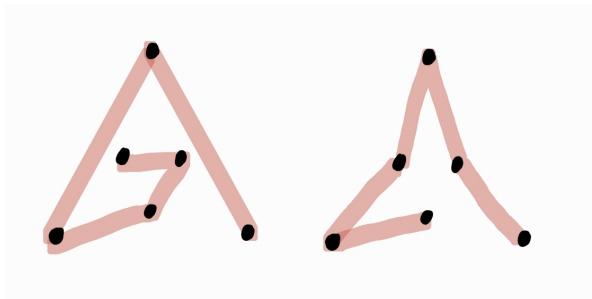
    //초기 염색체에 등록
    initialChromosome.gene.insert(initialChromosome.gene.end(), areaMinRoute.begin(), areaMinRoute.end());
}

```

4. 영역 Convex Hull Insertion

4-1. 초기 아이디어

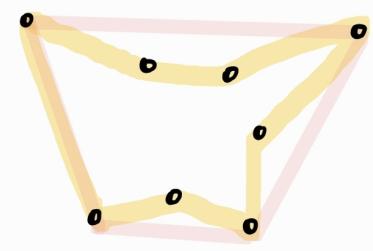
- Convex Hull을 활용해서 회오리 모양으로 그래프를 구성?
 - 아래와 같은 문제가 있음
 - Convex Hull은 가장 외곽의 정점들을 활용해 볼록 껍질을 구성



- 순차적으로 안쪽으로 말려들어가는 모양을 구성하게 되면, 중간을 들르는 것보다 cost가 더 높은 현상이 빈번하게 발생
- 외곽 볼록껍질을 구성한 다음, 하나씩 안으로 굽혀나가는건 어떨지?

4-2. Convex Hull Insertion

- Convex Hull을 통해 TSP를 최적화 하는 한 방법



- 교차 선을 아예 없앨 수 있다.
- 방법

- Convex Hull를 만든다. 구성 정점이 들어있는 집합 S.
- 간선 후보 배열 insertDataList를 선언
- S에 포함되지 않은 노드마다 반복 auto &k : S.
 - S의 간선을 끊어보며, 중간에 k를 연결했을 때 최소 cost가 되는 간선 e(a, b)을 찾는다.
 - c_ak + c_kb - c_ab가 최소가 되는 간선
 - insertDataList에 {k, e(a, b)} 쌍을 넣고 다음 k를 시도
- insertDataList 배열의 원소를 순회하며
 - insertDataList 원소 중에, {c_ak + c_kb}/c_ab가 최소가 되는 원소를 찾음
 - e(a, b)를 끊고, e(a, k)와 e(k, b)를 Convex Hull 배열 S에 추가한다.
- S에 모든 정점이 포함될 때까지 3, 4를 반복

4-2. 구현 상세

a. Graham's Scan

- createInitialConvexHull(areaId)

```
vector<Node> TreeRouteFinder::createInitialConvexHull(const int& areaId)
{
    vector<NodeCH>& currArea = citiesGroup[areaId];
    visited = vector<bool>(currArea.size(), false);

    sort(currArea.begin(), currArea.end(), compNode); //y좌표 -> x좌표 정렬
```

```

for(int i=1; i<currArea.size(); i++) //상대 위치 초기화
{
    currArea[i].p = currArea[i].node.y - currArea[0].node.y;
    currArea[i].q = currArea[i].node.x - currArea[0].node.x;
}

NodeCH st = currArea[0]; //pivot을 설정
sort(currArea.begin() + 1, currArea.end(), [&st](const NodeCH& a, const NodeCH& b) {
    int ccwVal = getCCwValue(st.node, a.node, b.node);
    if(ccwVal == 0) return GeneticSearch::getDistance(st.node, a.node) < GeneticSearch::getDistance(st.node, b.node);
    return ccwVal > 0;
}); //CCW 값을 활용하여 피봇을 제외한 정점들을 반시계로 정렬

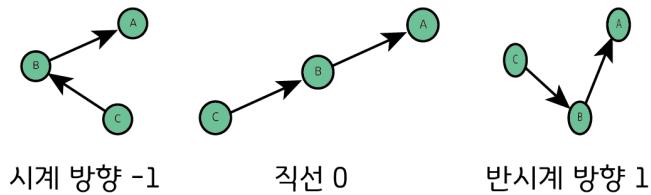
stack<int> stk; stk.push(0); stk.push(1); //스택에 초기 점 2개를 넣음

for(int next = 2; next < currArea.size(); next++)
{
    while(stk.size() >= 2)
    {
        int first, second;
        first = stk.top(); stk.pop();
        second = stk.top();

        //스택의 최상단 점 2개와 다음 정점의 관계가 CCW일때까지
        if(getCCwValue(currArea[second].node, currArea[first].node, currArea[next].node) > 0)
        {
            stk.push(first);
            break;
        }
    }
    stk.push(next); //다음 점을 스택에 Push
}
... //stk의 내용을 convexHull로 옮긴다
return convexHull;
}

```

- 본 알고리즘은 Pivot을 선정하여 이 정점을 기준으로 반시계 방향으로 Scan을 진행
 - 볼록 꼭지에 해당하는 정점인지 포함 여부를 판단하며 감싸는 과정을 반복
 - CCW(Counter Clock Wise)
 - 위 알고리즘은 3개의 점을 이은 직선의 방향을 알고자 할 때 유용한 기하 알고리즘



출처 : <https://snowfleur.tistory.com/98>

b. Convex Hull Insertion

```

vector<Node> TreeRouteFinder::createConvexHullRoute(const int& areaId)
{
    //Convex Hull를 만든다. 구성 정점이 들어있는 집합 S.
    vector<Node> convexHull = createInitialConvexHull(areaId);
    vector<NodeCH>& currArea = citiesGroup[areaId];

    //끼워넣기 할 데이터
    vector<pair<int, int> > insertDataList;

    //모든 정점이 convex hull 배열에 포함될 때까지 반복
    while(convexHull.size() != currArea.size())
    {
        //Convex Hull 배열에 없는 정점을 들여
        for (int i = 0; i < currArea.size(); i++)
        {
            if (visited[i]) continue;
            const Node &targetNode = currArea[i].node;
            double minLength = 1e6f;

            pair<int, int> insertData;
            for (int j = 0; j < convexHull.size(); j++)
            {
                if (minLength >= currArea[i].length)
                {
                    insertData.first = j;
                    insertData.second = currArea[i].length;
                    minLength = currArea[i].length;
                }
            }
            insertDataList.push_back(insertData);
        }
    }
}

```

```

{
    const Node &p = convexHull[j];
    const Node &q = convexHull[(j + 1) % convexHull.size()];

    //중간에 k를 연결했을때의 cost에 e(p, q)의 cost를 뺀 값
    double dist = GeneticSearch::getDistance(p, targetNode)
        + GeneticSearch::getDistance(targetNode, q)
        - GeneticSearch::getDistance(p, q);

    if (dist < minLength)
    {
        insertData = {i, j};
        minLength = dist;
    }
}
//candidate에 {k, e(a, b)} 쌍을 넣고 다음 k를 시도
insertDataList.push_back(insertData);
}

double minLength = 1e6f;
int insertPos = -1, insertTargetIdx = -1;

for (auto &c: insertDataList) //insertDataList 배열의 원소를 순회하며
{
    if (visited[c.first]) continue;
    const Node &targetNode = currArea[c.first].node;

    const Node &a = convexHull[c.second];
    const Node &b = convexHull[(c.second + 1) % convexHull.size()];

    //얼마나 볼록껍질의 선분에 가까운지?
    double dist = (GeneticSearch::getDistance(a, targetNode)
        + GeneticSearch::getDistance(targetNode, b))
        / GeneticSearch::getDistance(a, b);

    if (dist < minLength) //c_ak + c_kb)/c_ab가 최소가 되는 원소를 찾음
    {
        insertPos = c.second;
        insertTargetIdx = c.first;
        minLength = dist;
    }
}
... //insert 작업 e(a, k)와 e(k, b)를 Convex Hull 배열 S에 추가한다.
}
return convexHull;
}

```

- 설명은 주석과 구현 순서로 대체

c. 그 외

- [] 와 같이 Convex Hull이 처리된 영역들의 순서를 전처리

```

for(int i=0; i<25; i++)
{
    vector<Node> convexHull = subRouteFinder->createConvexHullRoute(tufuOrder[i]);
    ... //시작정점에 대한 처리
    initialChromosome.gene.insert(initialChromosome.gene.end(), convexHull.begin(), convexHull.end());
}

```

5. 클러스터링 영역 기반의 Greedy 알고리즘

5-1. 개요

- 주어진 노드들을 opencv2의 Kmeans 클러스터링을 사용한다. 이후 각 클러스터내부에서 돌아오지 않는 경로를 구성한 뒤 각 클러스터를 연결하여 돌아오는 전체 경로를 구한다. 이렇게 구해진 경로는 유전알고리즘의 초기 모집단이 된다.

5-2. 구현 상세

a. 주요 파라미터

```

- int k : 클러스터의 개수이다.
- cv::Mat clusteredLabel : 클러스터 결과로 opencv2/opencv.hpp의 Mat형식이다.

```

b. K-means 클러스터링

- `convertNodesToMat`

```
v::Mat KmeansGeneticSearch::convertNodesToMat(const vector<Node> &nodes)
{
    //행: 도시의 개수(1000), 열: y x좌표(2)
    cv::Mat nodeMat(1000, 2, CV_32F);
    for(int i=0; i<1000; i++)
    {
        nodeMat.at<float>(i,0) = nodes[i].y;
        nodeMat.at<float>(i,1) = nodes[i].x;
    }
    return nodeMat;
}
```

- Node들을 클러스터링 하기 위해 cv::Mat으로 변환하는 함수이다.

- `kmeansClustering`

```
void KmeansGeneticSearch::kmeansClustering(cv::Mat &clusteredLabel, cv::Mat &centers, vector<Node> cities, int k)
{
    if(cities.size()==0) return;

    //cv::Mat으로 변환
    const cv::Mat &mat = convertNodesToMat(cities);

    //클러스터링 알고리즘 수행 최대 반복횟수, 임계값 지정
    cv::TermCriteria termCriteria = cv::TermCriteria(cv::TermCriteria::EPS
                                                       + cv::TermCriteria::COUNT, 10, 0.1);

    //kmeans 클러스터링 실행
    cv::kmeans(mat, k, clusteredLabel, termCriteria,
               3, cv::KMEANS_PP_CENTERS, centers);
}
```

- 읽어온 Node들을 K-means 클러스터링하여 각 Node 별 클러스터 번호를 부여
- 클러스터들의 중심 위치들을 저장하고, 클러스터링 시 K-means++알고리즘을 통해 임의의 값을 통해 중심점을 찾는 것 이 아닌 입력 값을 중심으로 더욱 효율적으로 클러스터링 한다.

c. 클러스터 별 Greedy Algorithm을 통한 초기 모집단 형성

- `i initPopulation`

```
void KmeansGeneticSearch::initPopulationWithGreedy(vector<Chromosome> &population)
{
    //[[groupNum][idx]] : groupNum 영역의 idx번째 도시 (node.id가 저장)
    vector<int> citiesGroup[k];

    //클러스터 내 인접 리스트 [u][v] : t
    vector<vector<pair<double, int>> adj;

    //클러스터 내 그리디 적용 시중복 체크
    vector<bool> visited(cities.size(), false); //클러스터링 실행
    kmeansClustering(clusteredLabel, centers, cities, k); //k개의 군집 생성
    vector<Node> group[k];

    for(int i=0; i<cities.size(); i++) {...} //클러스터 별 ID 초기화 및 그룹화
    adj.resize(cities.size()+1); //클러스터 내 인접행렬 생성.

    //u - v 모든 쌍을 시도하고, 각 행렬을 dist순으로 정렬
    for(int groupNum=0; groupNum<k; groupNum++)
    {
        vector<int>& currArea = citiesGroup[groupNum];
        for(int i=0; i<currArea.size(); i++)
        {
            ...
            sort(adj[currArea[i]].begin(), adj[currArea[i]].end()); //dist 순으로 정렬
        }
    }

    vector<Node> route[k]; //클러스터 별 그리디알고리즘 실행
    for(int i=0; i<k; i++)
    {

```

```

vector<int>& currGroup = citiesGroup[i];
route[i].push_back(cities[citiesGroup[i][0]]);
//경로의 시작 지점을 그룹의 첫 노드로 설정.
int curr = currGroup[0];
visited[curr] = true;
while(route[i].size() < currGroup.size())
{
    int next = -1;
    //연결된 노드 중에서 가장 짧은 거리를 가진 노드를 다음 노드로 선택
    for(auto& edge : adj[curr])
    {
        int node = edge.second;
        if(!visited[node])
        {
            next = node;
            break;
        }
    }
    // 다음 노드를 현재 노드로 업데이트하고 경로에 추가
    curr = next;
    route[i].push_back(cities[curr]);
    visited[curr] = true;
}
// 클러스터의 순서 랜덤 배열 초기화
...
}

```

- kmeansClustering을 통해 클러스터링 한 결과를 받아 각 클러스터 별 Greedy 알고리즘을 통해 초기 경로를 설정한다.
- 클러스터 별 인접 리스트를 만들어 각 Node당 거리와 해당 노드를 기록한 후 거리에 따라 오름차순으로 정렬한다. 또한 방문 여부를 Node의 id에 의해 확인한다.
- 이후 각 클러스터의 첫 Node를 시작점으로 하여 Greedy 알고리즘을 통해 각 클러스터 별 초기 경로를 설정한다.
- 마지막으로 클러스터 순으로 붙여 초기 경로를 설정한다.

6. 클러스터링 영역 기반의 Convex Hull Insertion

6-1. 개요

- 위에서 기술했던 [] 과 동일 테크닉 적용. 5*5의 정사각 영역이 아닌 클러스터 단위로 바꿔었음.

6-2. 구현 상세

- 세부 구현은 [] 의 내용을 참고

a. *initPopulationWithConvexHull*

```

void KmeansGeneticSearch::initPopulationWithConvexHull(vector<Chromosome>& population)
{
    vector<vector<NodeCH>> citiesGroup(k); //groupNum[idx] : groupNum 영역의 idx번째 도시 (node.id가 저장)
    vector<vector<pair<double, int>> adj; //클러스터 내 인접행렬 [u][v] : t
    vector<bool> visited(cities.size(), false); //중복 체크

    //클러스터링 실행
    kmeansClustering(clusteredLabel, centers, cities, k);

    //k개의 군집 생성
    vector<Node> group[k];
    for(int i=0; i<cities.size(); i++){...}
    int firstClusterNum = clusteredLabel.at<int>(0,0);

    ...
    //Convex Hull 알고리즘 실행
    for(int i=0; i<k; i++)
    {
        vector<Node> convexHull = createConvexHullRoute(i, citiesGroup);
        if(i==firstClusterNum){...} //시작 정점인 경우
        initialChromosome.gene.insert(initialChromosome.gene.end(), convexHull.begin(), convexHull.end());
    }
    population.clear();
    for(int i=0; i<populationSize; i++)
        population.push_back(initialChromosome);
}

```

- k-means로 초기화한 클러스터 단위로 Convex Hull 알고리즘을 적용시켜 초기 경로를 설정
- 만약 시작 정점을 포함하고 있는 클러스터의 경우 되돌아오도록 설정

7. 전역 Convex Hull Insertion

7-1. 개요

- 영역 단위가 아닌, 전체 단위의 Convex Hull Insertion 연산

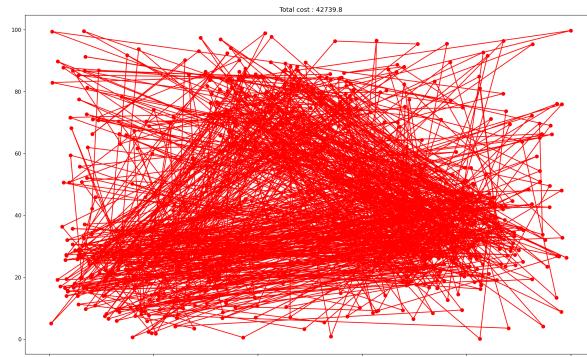
7-2. 구현

- [] 의 영역 개수를 1개로 지정 후 실행

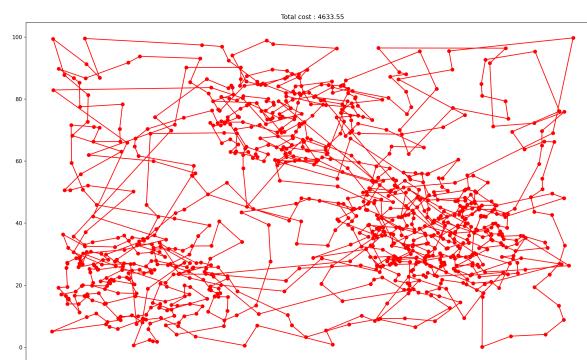
8. 성능 측정

8-1. 무작위 + 단순 GA

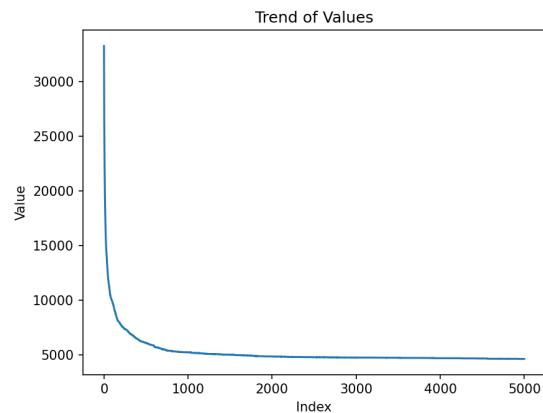
- 최초 세대 : **42848**
 - `std::shuffle` 을 통해 무작위로 순열 생성



- 50만 세대 결과 : **4633**

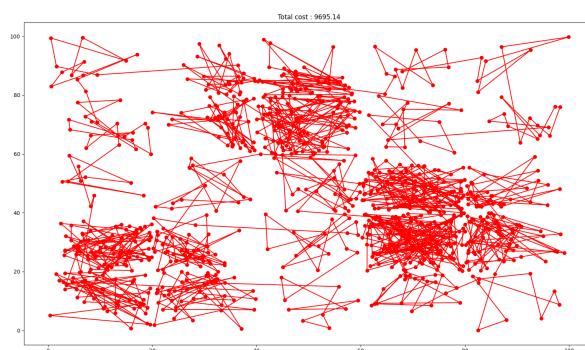


- Fitness 감소 분석

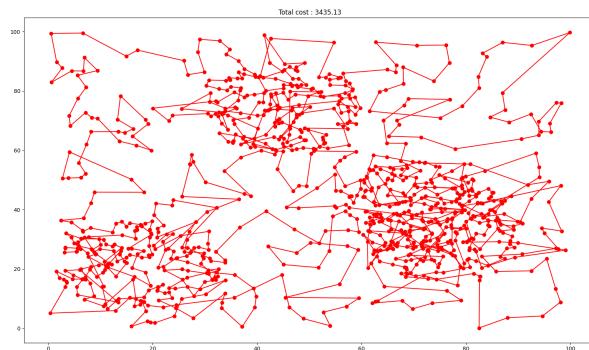


8-2. 영역 단위 Branch & Bound + Heuristic

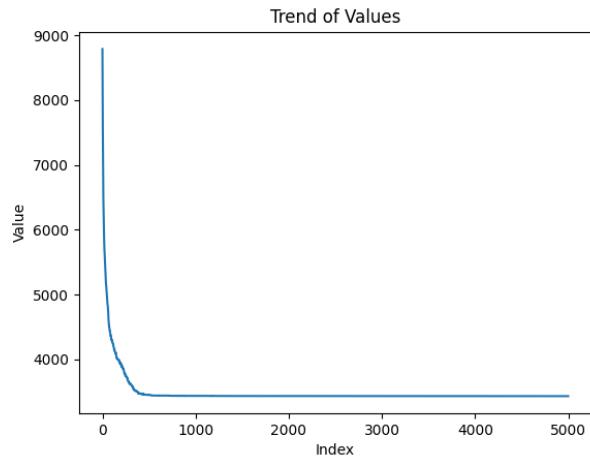
- 최초 세대 : 9695



- 50만 세대 결과 : 3435

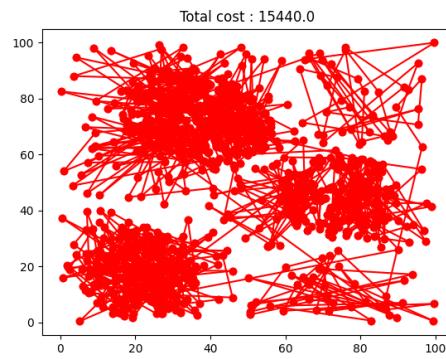


- fitness 감소 분석

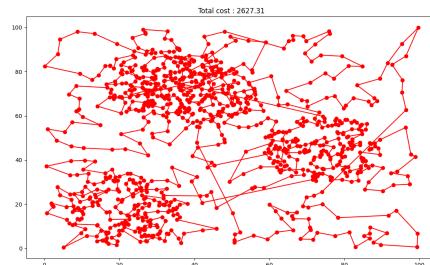


8-3. 클러스터 영역의 Greedy Algorithm

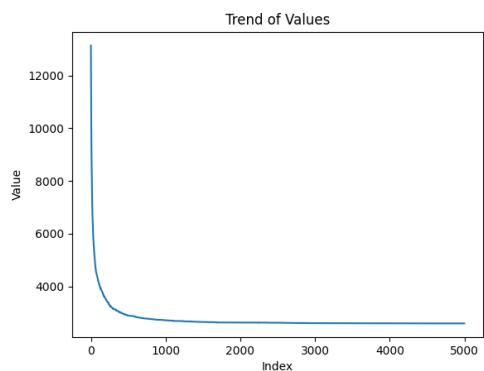
- 최초 세대: 15440



- 50만 세대: 2627.31

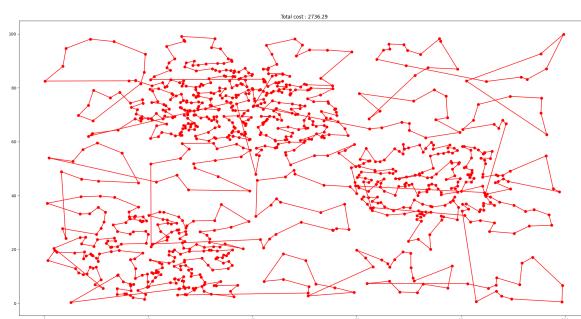


- fitness 감소 분석

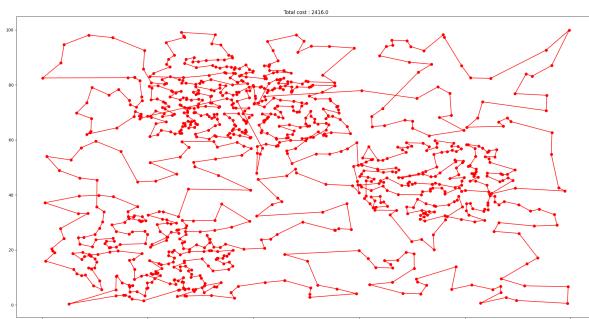


8-4. 영역 단위 Convex Hull Insertion

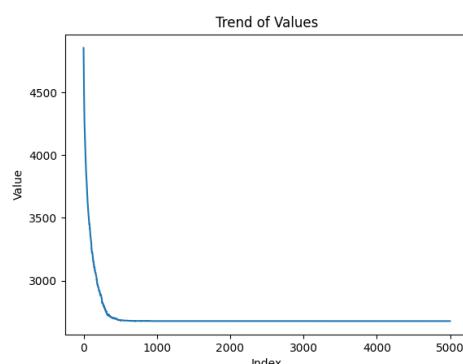
- 최초 세대 : 2736



- 5만 세대 결과 : 2416

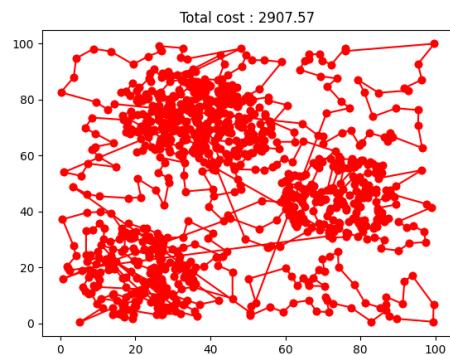


- fitness 감소 분석

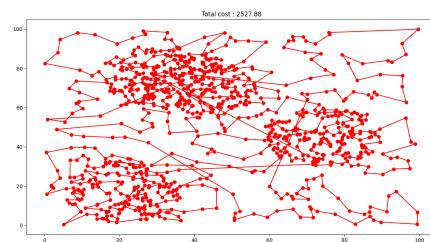


8-5. 클러스터 단위의 Convex Hull Insertion

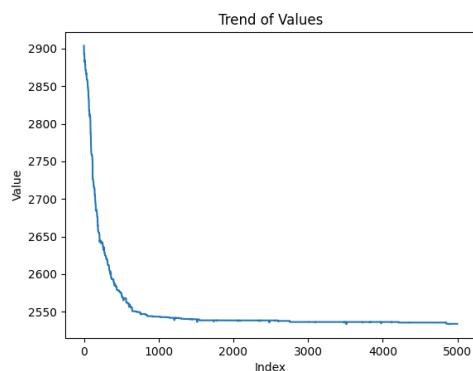
- 초기 세대 : 2907.57



- 50만 세대 : 2527.88

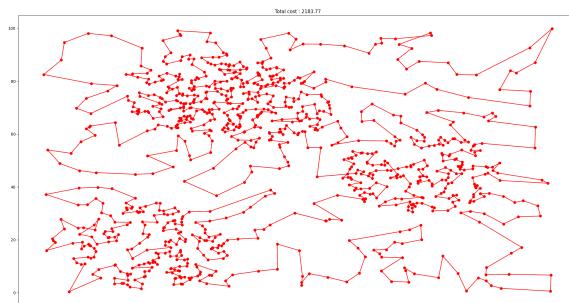


- fitness 감소 추이

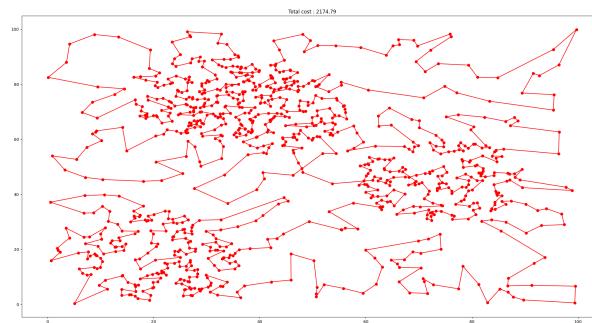


8-6. 전역 Convex Hull Insertion

- 초기 세대 : 2183



- 10만 ~ 50만세대 : 2174.79



- fitness 감소 추이

