REPORT

국민대학교
KOOKMIN UNIVERSITY

과목명 | 알고리즘

담당교수 | 임은진

학과 | 컴퓨터공학

학년 |
학번 |
이름 |
윤찬우(20153202)(팀장),
김선필(20143038),
김명진(20153156),
김민재(20153157)

제출일 | 2019년 11월 27일

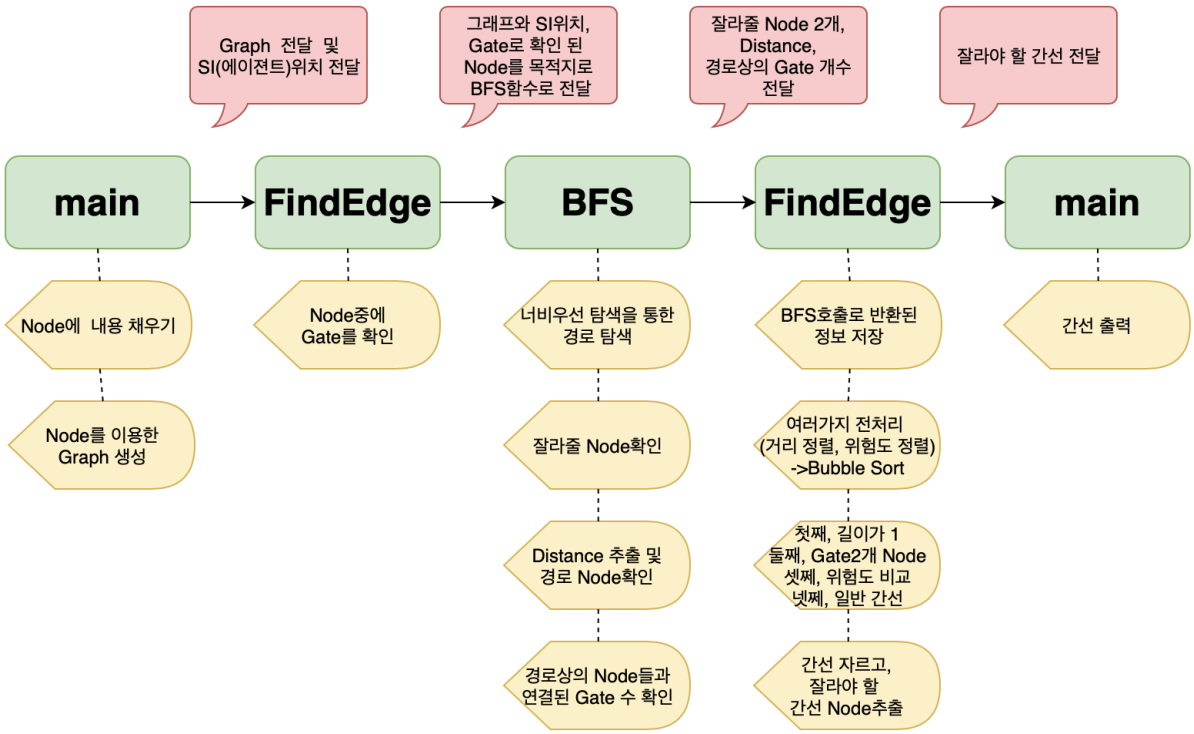
Skynet Revolution - Episode 2

Our solution

일단 문제를 요약하면 네트워크상에 존재하는 에이전트가 출구를 빠져나가지 못하게 연결을 끊는 것이 게임의 핵심입니다.

저희의 알고리즘은 크게 Node 를 가지고 에이전트의 위치에서 너비 우선 탐색을 통해 거리와 Node 의 index 그리고 경로의 출구 수를 얻습니다. 그 후, 첫번째로 거리가 1 인 것을 먼저 확인하고 두번째로 출구가 2 개인 노드와의 거리가 짧은 2 개를 받아와 경로상의 출구 수를 비교하며 접근하여 간선을 제거합니다. 세번째로 2 개의 출구가 연결된 Node 가 없을 경우 가까이 있는 출구와의 간선을 제거합니다. 이러한 규칙들을 통해 에이전트가 출구를 빠져나가지 못하게 하였습니다.

〈알고리즘 구성도〉



0. 전체적인 구성도

```
#include <vector>
#include <algorithm>
#include <queue>

using namespace std;

/**
 * Auto-generated code below aims at helping you parse
 * the standard input according to the problem statement.
 **/

vector<int> gateway;    //  set of vertex for the gateway

struct Node{...};

//  count gate for edge about the Node
void NumOfGate(vector<Node> &arr){...}

//  output = vertex, gateway, ditance, risk(number of gateway)
vector<int> BFS(vector<Node> arr, int st, int dst){...}

//  vertex processor(sort)
void SortDistance3(vector<Node> arr, vector<vector<int>> &tempo2){...}

//  vertex processor(sort)
void SortDistance2(vector<Node> arr, vector<vector<int>> &tempo2){...}

//  vertex processor(sort)
void SortDistance(vector<Node> arr, vector<vector<int>> &distance){...}

//  output = vertex, gateway
//  find two vertex to cut edge
vector<int> FindEdge(vector<Node> &arr, int start){...}

int main() {...}
```

1. Node(Structure)

```
vector<int> gateway;    //  set of vertex for the gateway

struct Node{
    vector<int> edge;
    int parent; //  front Node to find load
    int gate;   //  check the gate
    int gateCnt;//  check the number of gate for load
    Node() : gate(0), parent(-1), gateCnt(0){}
    void SortEdge(){    //  ascend vertax of edge in this node
        sort(edge.begin(), edge.end());
    }
};

//  count gate for edge about the Node
void NumOfGate(vector<Node> &arr){
    for(int i = 0; i < arr.size(); i++){
        int cnt = 0;
        if(arr[i].gate == 0){
            for(int j = 0; j < arr[i].edge.size(); j++){
                if(arr[arr[i].edge[j]].gate == 1)
                    cnt++;
            }
            arr[i].gateCnt = cnt;
        }
    }
}
```

Graph 를 얻기위해서 구조체로 Node 를 구현하였습니다.
변수로는 parent, gate, gateCnt, Edge 를 만들었습니다.
Parent 는 나중에 경로를 구하기 위해서 앞에 있던 Node 의 index 를 알기 위해서 만들었습니다.
Gate 는 bool 타입으로 만들어도 무방하나 0 과 1 을 이용하여 출구인지 아닌지 flag 역할을 합니다.
gateCnt 는 자신과 연결된 출구의 수입니다. 그리고 Method 인 NumOfGate 함수를 호출하여 gateCnt 를 구합니다.
Edge 를 통해 자신과 연결된 Node 들을 확인하고 Sort 함수를 호출하여 오름차순으로 정렬 합니다.

2.Main(Function)

```
int main()
{
    int N; // the total number of nodes in the level, including the gateways
    int L; // the number of links
    int E; // the number of exit gateways
    cin >> N >> L >> E; cin.ignore();
    vector<Node> arr(N);
    for (int i = 0; i < L; i++) {
        int N1; // N1 and N2 defines a link between these nodes
        int N2;
        cin >> N1 >> N2; cin.ignore();
        arr[N1].edge.push_back(N2);
        arr[N2].edge.push_back(N1);
    }
    for (int i = 0; i < E; i++) {
        int EI; // the index of a gateway node
        cin >> EI; cin.ignore();
        arr[EI].gate = 1;
        gateway.push_back(EI);
    }
    for (int i = 0; i < N; i++) {
        arr[i].SortEdge();
    }
    // game loop
    while (1) {
        NumOfGate( &arr);
        int SI; // The index of the node on which the Skynet agent is positioned this turn
        cin >> SI; cin.ignore();
        vector<int> result = FindEdge( &arr, SI);
        printf("%d %d\n", result[0], result[1]);
    }
}
```

Node 타입의 vector 배열을 만들어 Graph 를 만들어 줍니다.
Node 배열인 arr 에 EI 를 받아 Index 로써 gate 변수를 1 로 업데이트 함으로써 출구를 표시합니다.
각각의 노드에 연결된 Node 를 SortEdge 함수를 호출하여 오름차순으로 정렬합니다.
NumOfGate 함수를 호출하여 각 Node 의 연결된 gateway 의 수를 업데이트합니다.
Int 타입의 result 벡터에 FindEdge 함수를 호출하여 반환된 잘라 줘야할 Node 두 개를 받습니다.
결과를 출력 하여 간선을 잘라줍니다.

3.BFS(Function)

```
//  output = vertex, gateway, ditance, risk(number of gateway)
vector<int> BFS(vector<Node> arr, int st, int dst){
    int flag = 0;
    vector<int> ret;
    vector<int> visit(arr.size(), x: 0);
    int depth = -1;
    queue<int> a;
    a.push(st);
    while(!a.empty()){
        depth++;
        int qSize = a.size();
        for(int i = 0; i < qSize; i++){
            int temp = a.front();
            a.pop();
            visit[temp] = 1;
            for(int i = 0; i < arr[temp].edge.size(); i++){
                if(arr[temp].edge[i] == dst ){
                    ret.push_back(temp);
                    ret.push_back(arr[temp].edge[i]);
                    arr[arr[temp].edge[i]].parent = temp;
                    flag = 1;
                    break;
                }
                if(visit[arr[temp].edge[i]] == 0 && arr[arr[temp].edge[i]].gate != 1){
                    visit[arr[temp].edge[i]] = 1;
                    a.push(arr[temp].edge[i]);
                    arr[arr[temp].edge[i]].parent = temp;
                }
            }
            if(flag == 1)
                break;
        }
        if(flag == 1)
            break;
    }
    if(flag == 1) {
        ret.push_back(depth);
        int sum = 0, ll = ret[1];
        while(ll != -1){
            sum += arr[ll].gateCnt;
            ll = arr[ll].parent;
        }
        ret.push_back(sum);
        return ret;
    } else{
        return ret;
    }
}
```

Graph 와 SI(에이전트의 현재 위치), 임의의 게이트를 전달받습니다.
너비 우선 탐색(BFS)을 진행하면서, Node 의 parent 를 1 로 바꿔주고 Queue 를 사용해서 경로를 탐색하고 거리를 구합니다.
Flag 는 BFS 를 통해 목적지를 찾았을 경우 연산을 줄이기 위해 만들었습니다.
탐색이 끝나고 나서, 목적 Node 에서 parent 를 통해 경로를 얻고 경로에서의 Gateway 의 수를 구합니다.
목적 Node index, 목적 Node 하고 연결된 일반 Node index, 거리, 경로상에서의 Gateway 개수를 반환합니다.

3.FindEdge(Function)

우선은 BFS 함수를 호출하여 잘라줄 간선의 정보를 얻습니다.
잘라줄 간선의 두 Node 와 gateway 까지의 거리, 그 gateway 까지 경로 상에서의 gateway 개수를 담아두어 후보군을 만들어 줍니다.

첫번째, 거리가 1 인 후보가 있는지 확인을 합니다.
일단, SortDistance 함수를 호출하여 정렬을 합니다.
거리가 1 이면 다음 차례에 갈 수 있음으로 제거합니다.

두번째, 거리가 1 이 아닌 경우는 2 개의 gateway 가 연결된 정보를 얻습니다.
거리가 1 이 아니면 우선으로 2 개의 gateway 를 가진 Node 를 잘라야 합니다.

세번째, 후보군을 위험도로 정렬하고 거리로 다시 정렬하여 에이전트와 가까운 2 개의 후보만 비교를 합니다.
2 개의 후보중에서 위험도가 높은 것을 먼저 제거합니다.

네번째, 2 개의 gateway 를 가진 Node 가 하나일 경우 다음 우선으로 제거합니다.

다섯번째, 가까운 gateway 에 연결된 Node 를 제거합니다.

마지막으로, 출력하기 위해 두 Node 를 반환합니다.

```cpp
//  output = vertex, gateway
//  find two vertex to cut edge
vector<int> FindEdge(vector<Node> &arr, int start){
    vector<vector<int>> distance;
    int flag = -1;
    for(int i = 0; i < gateway.size(); i++){
        vector<int> tp = BFS(arr, start, gateway[i]);
        if(tp.size() > 2) {
            distance.push_back(tp);
        }
    }
    SortDistance(arr,  & distance);    // sort distance of gateway
    for(int i = 0; i < distance.size(); i++){   // find (distance = 1)
        if(distance[i][2] < 1){
            flag = i;
        }
    }
    if(flag != -1){ //  distance is 1
        vector<int> result;
        result.push_back(distance[flag][0]);
        result.push_back(distance[flag][1]);
        for(int i = 0; i < arr[distance[flag][0]].edge.size(); i++){
            if(arr[distance[flag][0]].edge[i] == distance[flag][1]) {
                arr[distance[flag][0]].edge.erase(arr[distance[flag][0]].edge.begin() + i);
            }
        }
        for(int i = 0; i < arr[distance[flag][1]].edge.size(); i++){
            if(arr[distance[flag][1]].edge[i] == distance[flag][0]) {
                arr[distance[flag][1]].edge.erase(arr[distance[flag][1]].edge.begin() + i);
            }
        }
        return result;
    }
    else {  //  distance is not 1 -> count numbers of gate for load and find node to have two gate
        vector<int> result;
        vector<vector<int>> tempo2;
        for(int i = 0; i < arr.size(); i++){
            if(arr[i].gateCnt == 2 && arr[i].gate != 1){
                vector<int> tempo;
                tempo.push_back(i); //  index0 = node to have two gate
                vector<int> tempo3 = BFS(arr, start, i);    //  use BFS
                tempo.push_back( - tempo3[2] - tempo3[3]); //  index1 = distance - risk(number of gate in load)
                tempo.push_back(tempo3[2]); //  index2 = distance
                tempo2.push_back(tempo);    //  in conclude, only use tempo2
            }
        }
        if(tempo2.size() != 0){ //  priority of node having two gateway
            SortDistance3(arr,  & tempo2); //  sort for risk
            SortDistance2(arr,  & tempo2); //  sort for distance
            int flag1 = 0;  //  reduce to calculate
            if(tempo2.size() > 1){  //  only compare two Node
                if(tempo2[0][1] < tempo2[1][1]){
                    for(int i = 0; i < gateway.size(); i++){
                        for(int j = 0; j < arr[gateway[i]].edge.size(); j++){
                            if(arr[gateway[i]].edge[j] == tempo2[0][0]){
                                result.push_back(tempo2[0][0]);
                                result.push_back(gateway[i]);
                                flag1 = 1;
                                break;
                            }
                        }
                        if(flag1 == 1)
                            break;
                    }
                    for (int i = 0; i < arr[result[0]].edge.size(); i++) {
                        if (arr[result[0]].edge[i] == result[1])
                            arr[result[0]].edge.erase(arr[result[0]].edge.begin() + i);
                    }
                    for (int i = 0; i < arr[result[1]].edge.size(); i++) {
                        if (arr[result[1]].edge[i] == result[0])
                            arr[result[1]].edge.erase(arr[result[1]].edge.begin() + i);
                    }
                    return result;
                }
                else{
                    for(int i = 0; i < gateway.size(); i++){
                        for(int j = 0; j < arr[gateway[i]].edge.size(); j++){
                            if(arr[gateway[i]].edge[j] == tempo2[1][0]){
                                result.push_back(tempo2[1][0]);
                                result.push_back(gateway[i]);
                                flag1 = 1;
                                break;
                            }
                        }
                        if(flag1 == 1)
                            break;
                    }
                    for (int i = 0; i < arr[result[0]].edge.size(); i++) {
                        if (arr[result[0]].edge[i] == result[1])
                            arr[result[0]].edge.erase(arr[result[0]].edge.begin() + i);
                    }
                    for (int i = 0; i < arr[result[1]].edge.size(); i++) {
                        if (arr[result[1]].edge[i] == result[0])
                            arr[result[1]].edge.erase(arr[result[1]].edge.begin() + i);
                    }
                    return result;
                }
            }
            else{   //  only use edge to have two gate
                for(int i = 0; i < gateway.size(); i++){
                    for(int j = 0; j < arr[gateway[i]].edge.size(); j++){
                        if(arr[gateway[i]].edge[j] == tempo2[0][0]){
                            result.push_back(tempo2[0][0]);
                            result.push_back(gateway[i]);
                            flag1 = 1;
                            break;
                        }
                    }
                    if(flag1 == 1)
                        break;
                }
                    if(flag1 == 1)
                        break;
                }
                for (int i = 0; i < arr[result[0]].edge.size(); i++) {
                    if (arr[result[0]].edge[i] == result[1])
                        arr[result[0]].edge.erase(arr[result[0]].edge.begin() + i);
                }
                for (int i = 0; i < arr[result[1]].edge.size(); i++) {
                    if (arr[result[1]].edge[i] == result[0])
                        arr[result[1]].edge.erase(arr[result[1]].edge.begin() + i);
                }
                return result;
            }
        }
        else {  //  the last is only to use edge to have one gateway
            vector<int> result;
            result.push_back(distance[0][0]);
            result.push_back(distance[0][1]);
            for(int i = 0; i < arr[distance[0][0]].edge.size(); i++){
                if(arr[distance[0][0]].edge[i] == distance[0][1]) {
                    arr[distance[0][0]].edge.erase(arr[distance[0][0]].edge.begin() + i);
                }
            }
            for(int i = 0; i < arr[distance[0][1]].edge.size(); i++){
                if(arr[distance[0][1]].edge[i] == distance[0][0]) {
                    arr[distance[0][1]].edge.erase(arr[distance[0][1]].edge.begin() + i);
                }
            }
            return result;
        }
    }
}
```

4.여러가지 Sort 함수들

```
//  vertex processor(sort)
void SortDistance(vector<Node> arr, vector<vector<int>> &distance){
    //  the first is to sort for number of gateway
    for(int a = distance.size() - 1; a > 0; a--){
        for(int b = 0; b < a; b++){
            if(arr[distance[b][0]].gateCnt < arr[distance[b + 1][0]].gateCnt){
                vector<int> tp = distance[b];
                distance[b] = distance[b + 1];
                distance[b + 1] = tp;
            }
        }
    }
    int twoGate = 0;    //  count number of two gateway
    for(int i = 0; i < distance.size(); i++){
        if(arr[distance[i][0]].gateCnt == 2){
            twoGate++;
        }
    }
    //  the seconds is to sort for distance in number of two gateway
    for(int a = distance.size() - 1 - twoGate; a > 0; a--){
        for(int b = 0; b < a; b++){
            if(distance[b][2] > distance[b+1][2]){
                vector<int> tp = distance[b];
                distance[b] = distance[b + 1];
                distance[b + 1] = tp;
            }
        }
    }
    //  the thirds is to sort for distance in number of one gateway
    for(int a = distance.size() - 1; a > twoGate; a--){
        for(int b = twoGate; b < a; b++){
            if(distance[b][2] > distance[b+1][2]){
                vector<int> tp = distance[b];
                distance[b] = distance[b + 1];
                distance[b + 1] = tp;
            }
        }
    }
}
```

```
//  vertex processor(sort)
void SortDistance3(vector<Node> arr, vector<vector<int>> &tempo2){
    for(int a = tempo2.size() - 1; a > 0; a--){
        for(int b = 0; b < a; b++){
            if(tempo2[b][1] > tempo2[b + 1][1]){
                vector<int> tp = tempo2[b];
                tempo2[b] = tempo2[b+1];
                tempo2[b+1] = tp;
            }
        }
    }
}

//  vertex processor(sort)
void SortDistance2(vector<Node> arr, vector<vector<int>> &tempo2){
    for(int a = tempo2.size() - 1; a > 0; a--){
        for(int b = 0; b < a; b++){
            if(tempo2[b][2] > tempo2[b + 1][2]){
                vector<int> tp = tempo2[b];
                tempo2[b] = tempo2[b+1];
                tempo2[b+1] = tp;
            }
        }
    }
}
```

간선들을 제거하기 위한 간선들의 후보군들을 정렬하기 위해 필요한 함수들을 강의시간에 교수님이 알려주신 Bubble Sort 알고리즘을 사용하여 정렬합니다.
각각의 Sort 함수들은 문제의 해결을 위해 직접 구현하였습니다.