개요

외판원 문제는 여러 도시들이 있고 한 도시에서 다른 도시로 이동하는 비용이 모두 주어졌을 때, 모든 도시들을 단 한 번만 방문하고 원래 시작점으로 돌아오는 최소 비용의 이동 순서를 구하는 것이다. 그래프 이론의 용어로 엄밀하게 정의한다면, "각 변에 가중치가 주어진 완전 그래프에서 가장 작은 가중치를 가지는 해밀턴 순환을 구하라" 라고 표현할 수 있다. 이 문제는 반드시 시작점으로 돌아와야 한다는 제약조건을 없애도 계산 복잡도는 변하지 않는다. 외판원 문제(Traveling Salesman Problem)또는 순회 외판원 문제는 조합 최적화 문제의 일종이다. 이것은 줄여서 TSP라고도 하며, 이 문제는 NP-난해에 속하며, 흔히 계산 복잡도 이론에서 해를 구하기 어려운 문제의 대표적인 예로 많이 다룬다.

특히 이 문제가 어려운 이유가 완전 연결 그래프라는 점이다. 단순히 완전 탐색으로 진행을 하면 O(n!)이라는 시간복잡도가 나온다. 이는 10개만 탐색해도 3,628,000이라는 값이 나오며, 20개는 2,432,902,008,176,640,000개라는 상상하기도 힘든 값이 나온다. 완전 탐색으로도 풀이는 가능하지만 왠만하면 지양해야 하는 방법이다.

TSP문제는 0번 도시부터 시작해서 모든 도시를 순회한 후 최종적으로 다시 0번으로 돌아오는 코드이다. 이는 무식하게 모든 경우의 수를 일일히 다 계산하는 방법이다. 그러나 앞서 말했듯, 이러한 방법으로는 15개 이상부터는 어마어마한 시간이 걸린다. 그래서 완전 탐색 말고 DP를 사용하는 방법을 이번 실험에서는 알아볼 것이다.

1. TSP(Traveling Salesman Problem)의 단계적 해석

동적 계획법으로 이를 구현하려면 탐색하는 과정을 부분 문제로 쪼개야한다. 완전 탐색으로 구현했을 때 사용하는 값들은 지금까지의 경로를 저장한 배열, 방문여부를 저장한 배열, 지금까지 연결된 거리이다. 동적 계획법은 기본적으로 각각 독립적인 부분 문제로 쪼개지므로 지금까지 연결된 거리는 필요가 없고, 현재점을 기준으로 연산을 해야 한다. 즉 아래와 같은 점화식으로 풀 수 있다.

$$TSP(here, visit) = \min\left(TSP(next, visit + next) + W[here][next]\right)$$

여기서 n은 0부터 next까지로, visit을 검사해서 방문한 적이 없는 값으로만 조사를 진행해야한다. 시작점과, 현재 방문한 도시를 기준으로 모든 도시를 순회해서 얻을 수 있는 최소 거리를 출력하는 방식이다.

그렇다면, 거리에 대한 정보를 어떻게 메모이제이션을 해야 하는가? 또 해당 시작점으로 출발하는 경로는 메모이제이션 할 때 어떻게 표현해야 하는가?

이에 대한 답으로 비트마스킹을 말할 수 있다. 바로 비트로 도시의 방문 여부를 체크하는 것이다. 예를 들어 0번 도시와 2번 도시를 방문했다면, 00101 = 5로 체크가 될것이고, 0번, 3번, 4번도시를 방문했다면 11001 = 25가 될 것이다. 즉 0번 도시에서 0, 1, 2번 건물을 방문한 것은 D[0][7]에 메모이제이션 하면 될 것이다. int형은 4byte, 총 32bit이기 때문에 int형 변수만으로도 32개의 도시를 표현할 수 있다. 그렇다면 코드를 작성하기 전 비트연산을 간단하게 알아보자.

- 1 ⟨⟨ n : 1을 n만큼 왼쪽으로 시프트한다. 즉 1 ⟨⟨ 4 이면 10000이다.
- a = a | (1 ⟨⟨ n) : a라는 변수의 n번째 비트를 켠다.
- a & (1 ⟨⟨ n) : a라는 변수의 n번째 비트가 켜져있으면 1⟨⟨ n을, 꺼져있으면 0을 리턴한다.
- a -= (1 ⟨⟨ n) : a라는 변수의 n번째 비트를 끈다. 단 무조건 켜져 있을 때만 사용해야한다.
- a &= ~(1 ⟨⟨ n) : a라는 변수의 n번째 비트를 끈다. 꺼져 있으면 꺼진 상태로 유지한다.
- a ^= (1 ⟨⟨ n) : a라는 변수의 n번째 비트를 토글한다. 켜져있으면 끄고, 꺼져있으면 킨다.

즉 위의 비트연산을 이용하면 visit & (1 ⟨⟨ next)는 next번째 도시의 방문 여부를 확인하는 것이고, 또한 재귀 호출의 visit인자로 visit | (1 ⟨⟨ next)를 전달 하는 것은, 해당 도시의 방문 여부를 체크해서 전달하는 것이다. 코드로 구현하면 다음과 같다.

```
/*
 D[here][(1 << MAX) - 1] = 도시를 모두 방문, 현재 here위치
 TSP(here, visit) = min(TSP(next, visit+next) + distance[here][next]))
 */
int TSP(int here, int visit) {
    int ret = D[here][visit];
```

```
    // 다 방문시, 100000 - 1 = 011111
    if (visit == (1 << MAX) - 1) return W[here][0];

    if (ret != 0) return ret;

    ret = INF;
    for (int i = 0; i < MAX; i++) { // i == 다음에 방문할 노드
        // visit & (1 << i) == visit의 i번째 비트가 1이면 1 << i, 0이면 0을 반환
        // 이미 방문했으면 다음 지점을 선택
        if (visit & (1 << i)) continue;
        // 길이 없을 경우
        if (W[here][0] == 0) continue;
        // visit | (1 << i) == visit의 i번째 비트를 1로 바꾼다.
        ret = min(ret, TSP(i, visit | (1 << i)) + W[here][i]);
    }
    D[here][visit] = ret;
    return ret;
}
```

[코드 1] TSP( )

완전 탐색으로 푸는 TSP는 O(n!)인 것에 반해 DP는 $O(2^n * n^2)$이다. DP로 풀어도 빠른 속도는 아니지만 완전 탐색에 비해 현저하게 빨라진 것을 알 수 있다.

이제 최적의 순환 경로를 찾았으니, 위 소스코드에서 메모이제이션한 값으로 경로를 추적해보도록 하자. 방법은 간단하다. 이 소스코드는 0이라는 위치부터 시작하므로 전체 거리 – 0부터 다른 경로까지의 거리 = D[k][masking + (1 〈〈 k)](k에서 masking에 켜져있는 비트의 도시들과 k번째 도시를 방문한 값)을 만족하는 것을 찾으면, k가 바로 다음 경로가 되는 것이다. 이 k를 경로를 저장하기 위한 path배열에 저장하자. 그 후 k를 다음 경로를 찾기 위해 비교하기 위한 변수 piv에 넣는다. 그리고 비교할 거리를 masking에 켜진 도시들과 k를 방문했으며, k부터 시작하는 거리인 D[k][masking + (1 〈〈 k)]으로 바꾼 후, k번째 비트를 방문했다는 표시로 켜준 뒤 전과 같은 연산을 반복적으로 진행하면 된다. 이를 코드로 구현하면 다음과 같다.

```
void printPath(int distance) {
    int piv = 0;
    int masking = 1;
    int index = 1;

    for(int j = 0; j < MAX; j++) {
        for(int k = 0; k < MAX; k++) {
            if(masking & (1 << k)) continue;
            if(distance - W[piv][k] == D[k][masking + (1 << k)]) {
                // 다음 경로 저장
                path[index++] = k;
                distance = D[k][masking + (1 << k)];
                piv = k;
                masking += (1 << k);
            }
        }
    }

    for(int i = 0; i < MAX; i++) {
        printf("(%d)->", path[i]);
    }
    printf("(0)");
}
```

[코드 2] printPath( )

이 코드들을 실행하면 최종적으로 결과는 다음과 같다.

```
29
(0)->(1)->(2)->(4)->(3)->(0)
```

[그림 1] 실행 결과

2. TSP.c

```
#include <stdio.h>
#define MAX 5
#define INF 987654321
int path[MAX] = {0};
int W[MAX][MAX] = {
   {INF, 8, 13, 18, 20},
   {3, INF, 7, 8, 10},
   {4, 11, INF, 10, 7},
   {6, 6, 7, INF, 11},
   {10, 6, 2, 1, INF}
};
int D[MAX][1 << MAX] = {0}; // [5][2^4+2^3+2^2+2^1+2^0 = 16 + 8 + 4 + 2 + 1 = 31]
void printPath(int distance);
int TSP(int here, int visit);
int main() {
   printf("%d\n", TSP(0, 1));
   printPath(29);
   return 0;
}
/*
 D[here][(1 << MAX) - 1] = 도시를 모두 방문, 현재 here위치
 TSP(here, visit) = min(TSP(next, visit+next) + distance[here][next]))
 */
int TSP(int here, int visit) {
   int ret = D[here][visit];
   // 다 방문시, 100000 - 1 = 011111
   if (visit == (1 << MAX) - 1) return W[here][0];
   if (ret != 0) return ret;
   ret = INF;
   for (int i = 0; i < MAX; i++) { // i == 다음에 방문할 노드
      // visit & (1 << i) == visit의 i번째 비트가 1이면 1 << i, 0이면 0을 반환
      // 이미 방문했으면 다음 지점을 선택
      if (visit & (1 << i)) continue;
      // 길이 없을 경우
      if (W[here][0] == 0) continue;
      // visit | (1 << i) == visit의 i번째 비트를 1로 바꾼다.
      ret = min(ret, TSP(i, visit | (1 << i)) + W[here][i]);
   }
   D[here][visit] = ret;
   return ret;
}

void printPath(int distance) {
   int piv = 0;
   int masking = 1;
   int index = 1;

   for(int j = 0; j < MAX; j++) {
      for(int k = 0; k < MAX; k++) {
         if(masking & (1 << k)) continue;
         if(distance - W[piv][k] == D[k][masking + (1 << k)]) {
            // 다음 경로 저장
            path[index++] = k;
            distance = D[k][masking + (1 << k)];
            piv = k;
            masking += (1 << k);
         }
      }
   }

   for(int i = 0; i < MAX; i++) {
      printf("(%d)->", path[i]);
   }
   printf("(0)");
}
```