**서론:**

이번 과제에서 TSP문제를 해결하기 위해 백트래킹과 분기 한정 방식을 사용하여 두 알고리즘을 비교하였습니다. TSP는 주어진 도시들을 한 번씩 방문하고 시작점으로 돌아오는 최소 경로를 찾는 문제입니다.

**Backtracking:**

백트래킹은 가능한 모든 경로를 탐색하여 최적의 해를 찾는 알고리즘입니다. 이 알고리즘은 재귀적으로 모든 경로를 시도하고, 현재 경로가 최소 경로인지 확인합니다.

A screen shot of a computer code

Description automatically generated

백트래킹을 위해서 구현한 재귀함수로 현재 경로와 방문 상태를 업데이트하며 가능한 모든 경로를 탐색합니다. 즉 모든 노드를 방문했을 때 현재 경로의 길이를 계산하고, 최소 경로와 비교하며 갱신합니다.

A screen shot of a computer code

Description automatically generated

Backtracking함수에서는 초기화 작업을 수행하고 재귀적으로 Backtrack함수를 호출해서 결과를 얻어냅니다. 시작 도시를 0번으로 설정하고, 방문하지 않은 노드를 하나씩 방문하며 경로를 기록 합니다. 모든 노드를 방문하면 경로의 길이를 계산하고, 최소 경로와 비교하고 재귀적으로 반복해서 모든 경로를 탐색합니다.

**Best-First Search Branch and Bound:**

분기 한정 알고리즘은 백트래킹의 단점을 보완하여 불필요한 경로 탐색을 줄이는 방법입니다. 이는 현재 경로의 하한 값을 계산하고, 최소 경로보다 큰 경우 해당 경로를 pruning합니다.

A screen shot of a computer code

Description automatically generated

Bound함수에서는 현재 경로의 하한 값을 계산하여서 현재 경로가 최소 경로가 될 가능성의 여부를 확인합니다. 먼저 path 배열에 저장된 현재 경로를 따라 노드 간 이동 비용을 더합니다. 방문하지 않은 각 노드간의 이동의 최소 이동 비용을 계산하기 위해서 graph[i][j]의 값이 최소인 값을 선택하여서 미래에 발생하게 될 비용의 하한값을 예측합니다.

A screen shot of a computer code

Description automatically generated

우선순위 큐를 사용하여 최적 경로를 찾습니다. 각 노드는 경로, 방문 상태, 현재 길이 및 하한 값을 포함합니다. 루트 노드를 생성하고 경로의 하한 값을 계산하고 우선순위 큐에 루트 노드를 추가한 후 큐에서 노드를 꺼내 자식 노드를 생성합니다. 각 자식 노드는 Bound함수로 하한 값을 계산하여 최소 경로보다 작은 경우에 큐에 추가해 줍니다. 모든 노드를 방문한 경로의 길이를 계산하고 최소 경로와 비교하는 과정을 큐가 텅 비게 될 때까지 반복합니다.

**사용된 그래프:**

A diagram of a network

Description automatically generatedA diagram of a network

Description automatically generated

<https://csacademy.com/app/graph_editor/>

Graph graph5 = {

{0, 5, 15, 20, 25},

{5, 0, 35, 25, 30},

{15, 35, 0, 30, 20},

{20, 25, 30, 0, 15},

{25, 30, 20, 15, 0}

};

Graph graph10 = {

{0, 75, 80, 35, 45, 15, 45, 50, 40, 55},

{75, 0, 65, 45, 70, 30, 10, 5, 25, 30},

{80, 65, 0, 25, 20, 50, 55, 35, 50, 65},

{35, 45, 25, 0, 60, 50, 30, 60, 40, 15},

{45, 70, 20, 60, 0, 60, 85, 75, 45, 40},

{15, 30, 50, 50, 60, 0, 55, 20, 35, 40},

{45, 10, 55, 30, 85, 55, 0, 65, 35, 25},

{50, 5, 35, 60, 75, 20, 65, 0, 70, 55},

{40, 25, 50, 40, 45, 35, 35, 70, 0, 45},

{55, 30, 65, 15, 40, 40, 25, 55, 45, 0}

};

node[i][j]에 대한 cost로 나타낸 adjacency matrix 형식

**출력결과:A screenshot of a computer

Description automatically generated**

Backtracking (5 nodes) Result:

Cost: 80

Path: 0 1 3 4 2 0

Time: 104µs

BFBnB (5 nodes) Result:

Cost: 80

Path: 0 1 3 4 2 0

Time: 128µs

Backtracking (10 nodes) Result:

Cost: 220

Path: 0 5 7 1 6 9 3 2 4 8 0

Time: 207038µs

BFBnB (10 nodes) Result:

Cost: 220

Path: 0 5 7 1 6 9 3 2 4 8 0

Time: 9898µs

**고찰:**

Fully connected node의 개수가 5개였을 때 Backtracking의 경우가 간소하게 나마 Best-First Search with Branch and Bound 알고리즘을 사용했을 때 빠르다는 것이 확인되었습니다.

반면에 Fully connected node의 숫자가 10개로 늘어난 경우 탐색공간이 매우 커지기 때문에 가능한 모든 경우의 수를 따지는 Backtracking이 Branch and Bound의 경우보다 성능이 저하됨을 확인하였습니다.