**Computer Algorithm Homework #2**

**<Traveling Salesperson Problem>**

**Class Number : 061**

**201524582 정희석**

이 문제의 graph를 5x5 matrix로 옮기면 전자기기, 벽이(가) 표시된 사진

자동 생성된 설명이다.

이 문제는 최소 값을 가지는 경로가 두 개 있습니다.

<A->E(8)->D(6)->C(9)->B(7)->A(3) : 33 & A->C(9)->B(7)->E(6)->D(6)->A(5) : 33>

**1-a. Greedy Search** : 35 | A->B(6)->E(6)->D(6)->C(9)->A(8) : 35

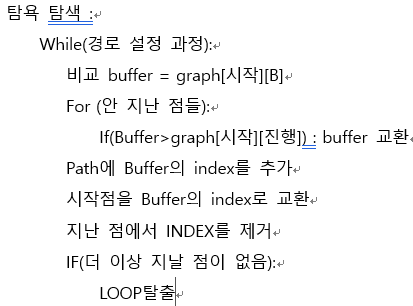
* Greedy search는 A에서 가장 가까운 길은 B로 가는 길인 6(graph(0,1))이고 그 다음 B에서 갈 수 있는 가장 가까운 길은 E로 가는 길(6(graph(1,4))), 그 다음은 D로 가는 길(6(graph(4,3))), 그리고 D에서 C로 (9(graph(3,2))), 마지막으로 C에서 A로 가는 경로(8(graph(2,0)))를 선택하게 된다. => 각 vertex에서 갈 수 있는 가장 빠른 길을 선택하게 된다.

**1-b. BackTracking** : 33 | A->C(9)->B(7)->E(6)->D(6)->A(5) : 33

* BackTracking은 제일 첫 vertex인 A에서 최저 경계 값을 정하고 다음 vertex로 진행한다. 해당 문제에서 graph의 각 행의 최소값들을 다 더한 값을 기본 bound로 정하였다. 그리고 Depth First Search 방식을 차용해서 B로 진행, B에서의 Bound를 계산하고, C로 진행. 이때 Bound는 이전 vertex의 bound에 vertex진행 때 증가한 수 만큼 더하여 계산한다.(ex. Bound(A) = 27, Bound(B) = Bound(A) + graph(0,1) - minimumOf(graph[0]) = 27)
* 해당 방식으로 진행하면 맨 처음 최저 값은 49로 설정되고(A->B->C->D->E), 49를 기준으로 Bound가 크면 dead-end, 작으면 진행하는 방식이다. 최저 값이 설정되었으면 제일 최근의 분기(A->B->C)로 가서(Back) bound를 계산하고 진행, 값이 최저 값과 동일 하므로 dead-end, 그 다음 C분기가 다 끝났으므로 B분기로 돌아 감(Back), B분기에서 D분기로 이동, D분기에서의 bound를 계산하고 최소 값과 비교, 이와 같은 방식으로 진행한다.
* 이렇게 진행하면 제일 먼저 설정되는 최소값은 Greedy Search와 같은 결과가 나오고 그 다음으로 나오는 최소 결과는 A->C->B->E->D->A : 33이 된다. 이후 A->E->D->C->B->A 결과와 만나지만 bound와 minimum이 같으므로 고려하지 않는다. 이 경로를 고려하기 위해서는 A->B분기 끝나고 A->E분기로 가는 backtracking 탐색을 해야 한다. 해당 탐색은 분기 선택 시 bound가 낮은 분기 우선 진행하게 하면 탐색할 수 있다.

**1-c. Branch-and-Bound** : 33 | A->C(9)->B(7)->E(6)->D(6)->A(5) : 33

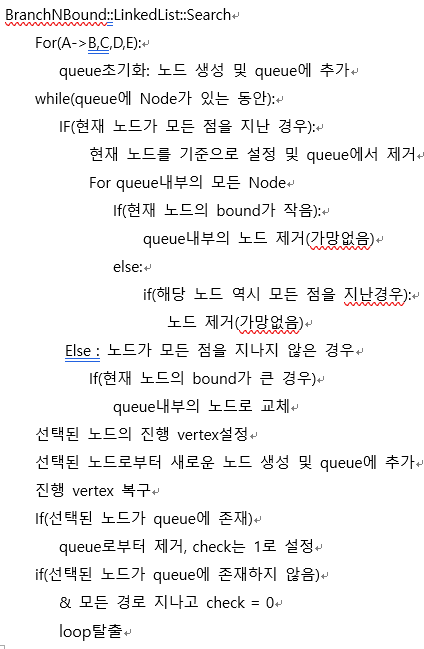
* Branch-and-Bound : 역시 제일 첫 vertex에서 bound값을 설정, 다음 vertex로 진행한다. Backtracking과 달리 다음 vertex의 bound값을 설정하고 다시 상위 분기로 복귀, 다음 vertex로 이동하는 Breadth-First Search방식을 채용한다. 모든 상위 vertex를 진행하면 해당 vertex중 가장 bound가 작은 vertex로 진행하며 진행 한 vertex의 하위 vertex역시 bound를 설정하고 다시 상위 vertex로 복귀한다. 그리고 각 vertex의 최하위 vertex들의 bound값을 비교하여 진행한다. Ex) A : 27, A->B : 27, A -> C : 30, A->D : 31, A->E : 29에서 A->B:27을 먼저 진행, A->B->C : 36, ->D : 31, ->E : 30을 더해서 비교 => A->E:29로 진행 해당 방식으로 계속 아래로 진행해 나가면 제일 먼저 A->C->B->E->D->A : 33가 나오고 그다음 A->E->D->C->B->A:33가 나온다. 가장 먼저 나온 최소 값을 타 진행경로의 bound와 비교해서 최소값인지 비교하게 된다.

****텍스트, 영수증이(가) 표시된 사진

자동 생성된 설명**2-a. Greedy Search** : Greedy Search는 직관적으로 선택한 경로를 저장할 list : Path와 앞으로 지날 점을 포함하는 list : listPointer을 선언, while문을 선언한 후 각 행에서 listPointer에 있는 index의 graph값 중 최소 값을 탐색 -> 저장, path에 append, listPointer에서 remove로 진행 -> listPointer가 빈 list가 되는 경우(모든 점을 거친 경우) 마지막 vertex에서 시작점으로 오는 length를 더하여서 구현함.

2-b. BackTracking : BackTracking에서 최소 값이 정해질 때 greedy search와 같은 결과가 중간에 도출 되므로 사전에 minimum을 greedy search의 최소 값으로 설정하였다. BackTracking은 Node와 Linked-list를 이용하여 class로 구현. Node는 해당 vertex에서 다음 vertex로 진행할 때의 판단 기준이 되는 **Boolean list : deadEnd**, 그리고 해당 Node의 현재 위치 **vertex**를 담고있다. 그리고 모든 다음 vertex가 끝났는지 여부를 return 하는 **isDead함수**, vertex가 dead-end인 경우를 설정하는 **dead함수**를 정의함. 실질적 모든 동작은 linked-list에 구현하였으며 이 class에는 **경로, 지나갈 vertex, 최소값, graph의 각 행의 최소 값**들을 담게 했다. 다음 vertex의 dead-end여부를 결정할 함수로 **promising**함수를 정의, 해당 함수는 지나온 path의 length와 앞으로 지날 vertex에서 갈 수 있는 최소 길이를 합해서 return한다. 이는 다음 vertex로 진행할 때 if-else문에서 check되어 진행 여부를 결정하게 된다. 그리고 이는 마지막 최종 값을 저장할 때도 응용하기 위해서 값을 직접 return하게 했다. 그리고 외부에서는 **search함수** 하나만 실행하면 되게 하였고 DFS이므로 search함수에서는 for문을 사용하여 4번 진행 하도록(A->B,C,D,E)하였다. 실질적 비교를 실행하는 함수는 **down함수**로 구현하여 down하면 인자로 받은 vertex를 경로에 추가, 새로운 노드를 생성하고 promising을 확인후 dead이면 deadEnd에 dead설정, promising을 만족하면 또 down 함수를 호출하여 재귀적으로 진행. 만약 down함수가 가장 하위 텍스트이(가) 표시된 사진

자동 생성된 설명단계에서 진행된다면 해당 경로의 길이를 확인해서 가장 최소 거리인 경우 저장하도록 하였고 함수가 종료됨으로써 상위 노드로 돌아 가게 된다.

2-c. Branch-and-Bound : Branch-and-Bound에서는 Node, Linked-list, queue(를 가장한 list)를 이용하여 구현하였다. Node에서는 현재의 vertex, 진행된 path, 그리고 중복 경로를 제외한 graph, 그리고 부모 노드의 bound를 인자로 받아서 저장한다. 중복 경로 제외를 위해 모든 과정은 Node생성 및 초기화 과정에서 진행한다. Graph의 대각선을 엄청 큰 값(sys.maxsize)로 설정, bound에 reduced\_graph의 마지막으로 지난 점에서 현재 vertex로의 값을 추가하고 ,path로 지나온 점의 행을 INF로 설정, 현재 vertex의 열을 INF로 설정한 후, 각 행의 최솟값이 0이 아니면 최솟값을 0으로 만들고 0으로 만들기 위해 뺀 값을 bound에 추가한다. 모든 행이 최솟값으로 0을 가지면 이번엔 모든 열의 최솟값을 0으로 만고 역시 bound에 추가한다. 이 과정을 거친 Node들을 Linked-list에서 Queue에 넣어서 최솟값들을 비교, 진행하게 된다. 이번 branch and Bound 역시 외부에서는 Search함수만 실행하도록 하였고 실질적 비교과정은 Search함수에서 실행한다. Search함수에서는 먼저 queue를 생성하기 위해 가장 먼저 A->B,C,D,E를 지나는 경로의 Node를 생성하여 queue가 차 있는 동안 작동하는 while문 에서 어떠한 Node가 지나온 경로가 모든 점을 포함하는 경우가 발생 하면 queue에 있는 모든 노드와 bound를 비교하여 더 작은 노드가 있으면 계속 진행하고 더 작은 노드가 없으면 loop를 탈출 하도록 하였고, 모든 점을 포함 하지 않으면 queue 내부의 노드들과 bound를 비교하여 탐색하도록 하였다.

텍스트이(가) 표시된 사진

자동 생성된 설명3. 결과 분석

A의 greedy search는 빠르게 선택할 수 있는 장점이 있지만, 항상 선택할 수 있는 최소값만 따라가다 보면 최소값이 아닌 최소값을 선택하게 되어 항상 최소값을 가진다고는 할 수 없게 됩니다.

B와 C의 BackTracking과 Branch-and-Bound는 bound값을 어떻게 설정하는 지, 결과가 어떤 노드에 위치해 있는지, 어떤 규칙으로 분기 진행을 할 지에 따라 성능차이와 결과가 다르다는 것을 확인 할 수 있었습니다.

B의 BackTracking은 사전에 minimum을 정할 수 있으면 훨씬 빠르게 진행할 수 있고, 앞에서 부터 순차적으로 하위 노드까지 모든 경우를 진행 하다 보니 상대적으로 진행 과정이 많아 진 경향이 있다. 그리고 재귀적 호출이 필요해서 구현하는데 상당히 고생을 했다. 해당 algorithm은 단순한 경로를 찾는(순서적으로 앞 쪽 tree에 존재하는 답)데 좋은 알고리즘이라고 판단할 수 있었다. 만약 훨씬 복잡한 경로를 찾게 된다면 Time Complexity 측면에서 큰 이득은 없을 것 같다.

C의 Branch-and-Bound는 사전에 minimum을 정할 필요가 없고 재귀적 호출이 없어도 되는게 큰 장점으로 BackTracking보다 탐색 시간이 짧고 간단하던 복잡하던 평균적으로 성능을 낼 수 있다. 이 판단의 이유는 Node들의 Queue를 (Priority Queue역시 사용가능) 사용하면 진행할 필요 없는 Node들(path)을 간추리기 용이하기 때문이다. 대신 단점은 queue를 사용함으로써 공간적으로 BackTracking보다 더 많은 공간을 요구하고, Node에 많은 정보가 들어가서 Node를 관리하는데 복잡하다는 단점이 있다. 그리고 bound를 설정할 때 어떻게 설정하느냐에 따라서 답이 갈릴 수 있고, 얼마나 더 좋은 성능을 내는 지 나타난다고 판단된다.

해당 과제를 수행하면서 함수의 재귀적 호출로 인해(Backtracking) 고생을 많이 했습니다. Python의 class사용법을 익히고 알고리즘을 짜고 수정하고 다시 짜고 다시 수정하고 많은 과정을 거쳐서 지금 결과를 얻을 수 있었습니다. 그리고 수업시간에 (best-first branch-and bound search강의에서의) 진행한 예시를 작동했을 때도 정상적으로 작동 하는 것을 확인 할 수 있었습니다. 4페이지 안에 모든 내용을 담고자 하니 너무 적어서 아쉽습니다.

자세한 내용은 첨부된 python코드를 확인 바랍니다. 고생 많으셨습니다.