**설계 프로젝트 C: 민혜의 신나는 세계일주^O^/**

2015104197 이민혜 / 2015104220 정종윤

1. 시스템 개요 및 목표

ⓐ 개요

최적의 여행 루트를 얻기 위한 경로 탐색 프로그램을 만든다. 유저는 출발장소에서 시작하여 주어진 시간 내에 도착지에 도달해야 한다. 출발 장소와 도착 장소 사이에는 **지역**이 존재하며, 각 지역에는 **관광지**가 존재한다. 관광지는 방문 할 경우에 시간을 소모하지만, **만족도**를 얻을 수 있다.

프로그램은 최적의 경로를 탐색하기 위해, 사용자에게서 탐색 가능한 시간을 입력 받아 다음과 같은 기능을 수행할 수 있다. 첫 번째로 **주어진 시간 내에 가장 많은 관광지**를 가는 경로를 탐색하거나, 두 번째로 **주어진 시간 내에 가장 큰 만족도**를 얻을 수 있는 경로를 탐색하는 방법이다. 이때, 지역은 중복으로 지나갈 수 있지만 지역에 있는 관광지를 중복 방문 할 수 없다.

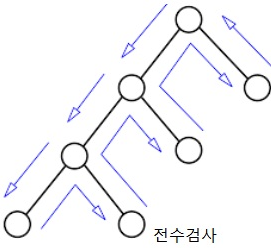
ⓑ 목표

시나리오에 명시된 기능들을 모두 포함하는 최적의 경로 탐색 프로그램을 GUI가 구현된 프로그램으로 작성하고, 각각의 조건을 어떻게 최적의 알고리즘과 자료구조로 표현 가능한지 고민해본다.

객체 간의 관계를 표현하는 그래프를 프로젝트에 적용하고, 우선순위 큐를 사용한 Dijkstra 알고리즘, Floyd-Warshall 알고리즘, 만족도 또는 관광지 개수를 기준으로 하는 Greedy 알고리즘, 또는 Backtraking 알고리즘 등을 활용해 구현할 수 있는 능력을 배양한다.

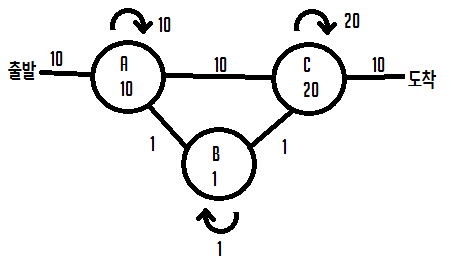
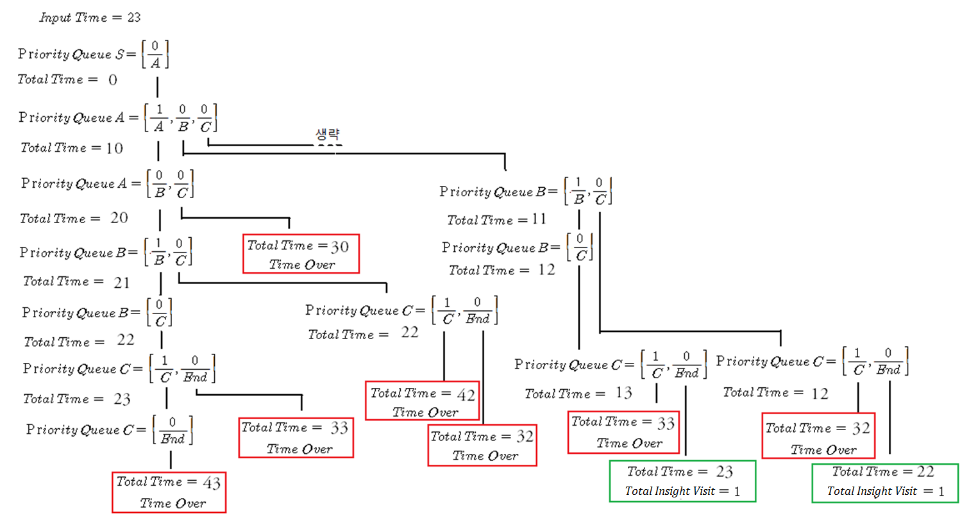
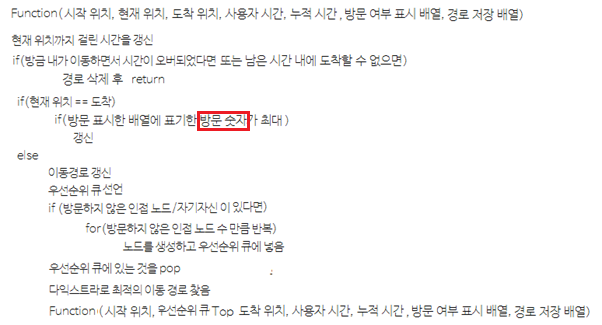
1. 문제사항 1번: 주어진 시간 내에 가장 많은 관광지를 들려야 한다.

ⓐ 1차 문제 해결 방법:

우리는 1차 해결 방법에서 힙으로 구현된 우선 순위 큐를 사용한 Backtracking 방법을 언급하였다. 우리는 문제를 풀기 위해 다음의 가정을 두었다.

1. 일반적으로 Backtracking은 모든 경우의 수를 방문하여 최적의 해를 구할 수 있다.
2. 하위 노드는 1회 이동으로 방문할 수 있는 인접 노드를 의미한다.
3. 1회 이동은 우선순위 큐를 이용해 획득할 수 있는 **관광지의 개수**를 Primary Key로 정의한다.
4. 중요한 점은 가정 (3)에서 **지역에서 관광지를 1회 방문하는 것을 자기 자신으로 향하는 Edge로 취급한다는 특징**을 바탕으로 이루어진다는 것이다.
5. 사용자의 입력 시간 내에 이동 가능한 경로가 여러 개 생길 수 있기 때문에, 함수 실행 전에 변수를 선언하여 **누적된 관광지 방문 횟수**가 가장 높은 것으로 계속 갱신한다.
6. 일반적인 경우에 이미 방문한 노드는 더 많은 관광지를 방문할 방법을 구하기 위하여 다시 우선순위 큐에 넣지 않는다.

이동한 노드에서 다시 방문할 노드를 검색하고, 이를 함수로 구현해 재귀적으로 계속 반복하고자 한다. 다시 말해, 유망한 경우를 계속 찾는 되추적(Back Tracking) 방법을 이용한다.

[그림 2]에서 임의로 분자에 적힌 1은 방문 가능 플래그로써, 1회 이동으로 관광지 방문이 가능함을 의미하고 0은 1회 이동으로 관광지 방문이 불가능함을 말한다. 분모는 1회 이동 시 사용자가 위치하는 노드를 의미한다.

백트래킹을 이용해 구현한 수도 코드

[그림 2] 가장 많은 관광지를 찾는 되추적 방법을 트리로 설명한 것. 도착지까지 도달 가능한 경로 중에서 방문한 관광지가 가장 많은 것을 선택한다.

[그림 1] 예시로 사용할 그래프

ⓑ 2차 문제 해결 방법

우리는 지난 보고서에서는 힙을 이용한 우선순위 큐를 사용한 Backtracking 알고리즘을 적용할 것이라고 설명하였다. 하지만 이번 2차 문제 해결 방법에서는 Greedy 알고리즘을 통해서 문제를 접근해 보았다.

먼저 우리는 주어진 시간 내에 최대의 많은 관광지를 가기 위해서 세 가지 조건을 가정한다.

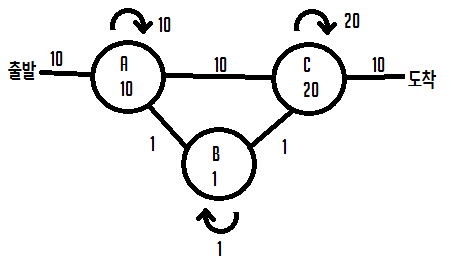
**(1)** 현재 노드에서 다음 노드로 이동하는 이동시간과 관광지 소요시간을 더한 시간이 가장 작은 노드를 우선으로 이동한다.

**(2)** 그 다음 노드로 이동했을 시에 남은 시간 안에 도착지에 도착이 가능해야 한다.  
**(3) 따라서 1회 이동을 현재 위치에서 이동할 노드까지 최단 경로로 가는 데 걸리는 시간   
+ 도착한 노드의 관광지를 방문하는 시간으로 정의한다.**

먼저, (1)번의 가중치를 얻기 위해서 인접행렬을 새로 만들었다. 인접한 노드 뿐만이 아니라 자신을 제외한 모든 노드 사이의 거리와 그 관광지를 방문하여 소요되는 시간까지 합한 시간을 새로운 **인접행렬**을 만들어 선언하였다. 노드와 노드의 거리는 다익스트라 알고리즘을 사용하여 최단 경로를 구하였고, 플로이드-워셜 알고리즘을 통해 인접하지 않은 노드들의 최단거리를 구할 수 있다. 여기서 중요한 점은, 현재 노드에서 인접한 노드만을 생각하는 것이 아니라 전체 노드와 비교한다는 점이다.

(2)번 조건에서 다음으로 갈 노드가 결정이 되었다는 것은 현재 노드에서 다음 노드로 최단 경로로 이동 후 노드의 관광지를 방문한다는 것을 뜻한다. 우리는 그 노드로 이동 후에 도착지에 도착 가능한 충분한 시간이 남아 있어야 하므로 위의 (2)번 조건을 만족하여야 한다. 만약, 도착지에 도착 가능한 충분한 시간이 불가능 하다면, 다음 노드로 이동이 불가능 하다는 뜻이므로, 현재 노드까지의 경로에서 도착지로 가는 것이 최선의 답이다.

우리는 크게는 위 두 가지 조건을 만족하는 알고리즘을 Greedy하게 구현하였다.



[그림 3] 예시로 사용할 그림.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | 출발 | A | B | C |
| 출발 | 0 | 20 | 12 | 22 |
| A | 10 | 10 | 2 | 22 |
| B | 11 | 11 | 1 | 21 |
| C | 22 | 12 | 2 | 20 |

**[표1] [그림3]을 인접행렬로 나타낸 것**

1. 먼저, 현재 위치에서 가장 적은 소요시간이 드는 다음 노드를 찾는다.

이때, 출발지와 목적지, 자기자신으로의 이동은 예외처리를 함으로써 중복적으로 방문하는 것과, 출발지나 목적지로 향하는 것을 방지한다.

(2) 만약, 그 선정된 노드를 방문하지 않았고, 노드로 이동하는 시간이 사용자가 입력한 시간보다 크지 않고, 지금까지 걸린 시간과 선정된 노드로 부터 도착지까지의 최단 거리를 합한 시간이 입력시간보다 크지 않다면 경로에 다음 노드로 추가한다.

(3) 위의 (1),(2)를 반복하다가 만약, 선정된 노드에서 도착지로 이동하는 시간이 사용자가 입력한 시간보다 크다면 그 노드를 포함하지 않은, 이전까지의 경로에서 도착지로 이동한 것이 해답이다.

위의 [그림3]의 예를 들어, 사용자가 30의 시간을 입력했다고 하자. [표1]을 보면 출발에서부터 가장 적은 소요시간이 드는 노드는 출발을 제외한 B이다. (1)

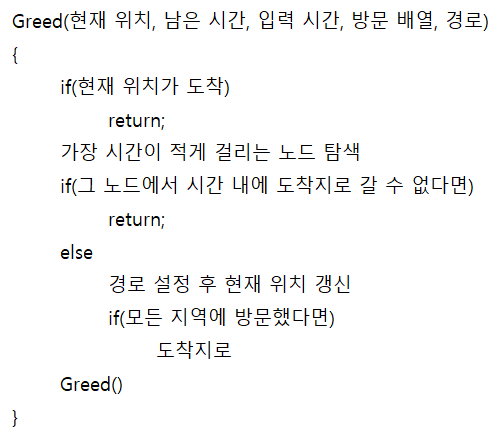
아직 B를 방문하지 않았고, 출발로부터 B까지 걸리는 시간은 12이므로 입력시간인 30보다 작다. 또한 B로부터 도착까지 걸리는 최단거리는 11이므로 (출발) – 관광지 B – (도착)까지의 소요시간은 23로 입력시간인 30보다 작다. 그러므로 경로에 추가가 가능하다. (2)

이제 현재위치를 B로 바꾼 후 B에서 가장 적은 소요시간이 드는 다음 노드를 찾는다. 표를 보면 현재 위치인 B와 출발을 제외한 A가 다음 노드로 적합하다. 현재까지 소요시간인 12와 B에서 A로 가는 시간인 11을 더하면 23으로 입력시간인 30보다 작다. 하지만 A에서 도착까지 가는 최단 거리인 12까지 더한다면 35로 입력시간보다 초과 되었다. 이는 A가 다음 노드로 적합하지 않는다는 뜻이므로 A를 포함하지 않은 이전까지의 경로에서 도착지로 이동한 것이 주어진 시간 안에 최대한의 관광지를 방문한 경로이다. (3)

경로 : (출발) – 관광지B – (도착) / 소요 시간 : 23



그리드 알고리즘은 현재 위치, 남은 시간, 사용자 입력 시간, 방문 여부 배열, 그리고 경로 저장 벡터를 파라미터로 사용하며, 구현된 수도 코드는 다음과 같다.



ⓒ 문제 해결 방법 비교

1차 문제 해결 방법과 2차 문제 해결 방법의 가장 큰 차이는 알고리즘에 있다. 1차 문제 해결 방법에는 Backtracking 알고리즘을 사용하였고, 2차 문제 해결 방법에서는 Greedy 알고리즘을 사용하였다.

|  |  |  |
| --- | --- | --- |
| 알고리즘 | Backtracking | Greedy |
| 구현 | 어렵다 | 쉽다 |
| 이해 | 어렵다 | 쉽다 |
| 속도 | 느리다 | 빠르다 |
| 해 | 최적 | 근사 |

**[표2] Backtracking알고리즘과 Greedy알고리즘의 비교 표**

1차 문제 해결 때 사용한 Backtracking의 경우에는 우선순위 큐를 사용해서 재귀적으로 함수를 호출하는 방식이었다. 하지만 Greedy 알고리즘으로 바뀌면서 답은 최적의 해가 아닌 근사값이 되었지만 구현이나 속도에서 훨씬 쉽고 빨라질 수 있었다.

|  |  |
| --- | --- |
| 1차 | 2차 |
| 1. 현재 위치에서 인접한 노드 중 방문하지 않은 노드를 대상으로 함 2. 1회 이동 = “현재 위치에서 해당 지역으로 이동” 또는 “해당 지역의 관광지 방문” | 1. 방문하지 않은 모든 노드를 대상으로 함 2. 1회 이동 = “현재 위치에서 해당 지역으로 이동해 관광지 방문하는 시간을 포함” |

1회 이동의 정의 또한 바뀌었다.

**[표3] 1차 문제 해결 방법과 2차 문제 해결 방법의 조건 비교**

1차 방법에서는 현재 위치에서 인접한 노드만을 생각해 가중치를 판단하였다면, 2차 방법에서는 인접하지 않은 모든 노드들의 가중치까지도 계산하였다. 또한 지난 1차 방법에서는 1회 이동이 지역으로의 이동과 관광지로의 이동으로 따로 보았다. 하지만 2차에서는 1회 이동을 지역이동과 해당지역의 관광지 방문으로 묶어서 1회 이동을 하였다.

1. 문제사항 2번 : 주어진 시간 내에 가장 높은 만족도를 얻어야 한다.

ⓐ 1차 문제 해결 방법

주어진 시간 내에 가장 큰 만족도를 얻을 수 있는 경우에 대해 살펴보자. 우리는 ②-ⓐ 에서 언급 했던 가장 많은 관광지를 방문하는 경우의 알고리즘을 응용하여 적용하였다.

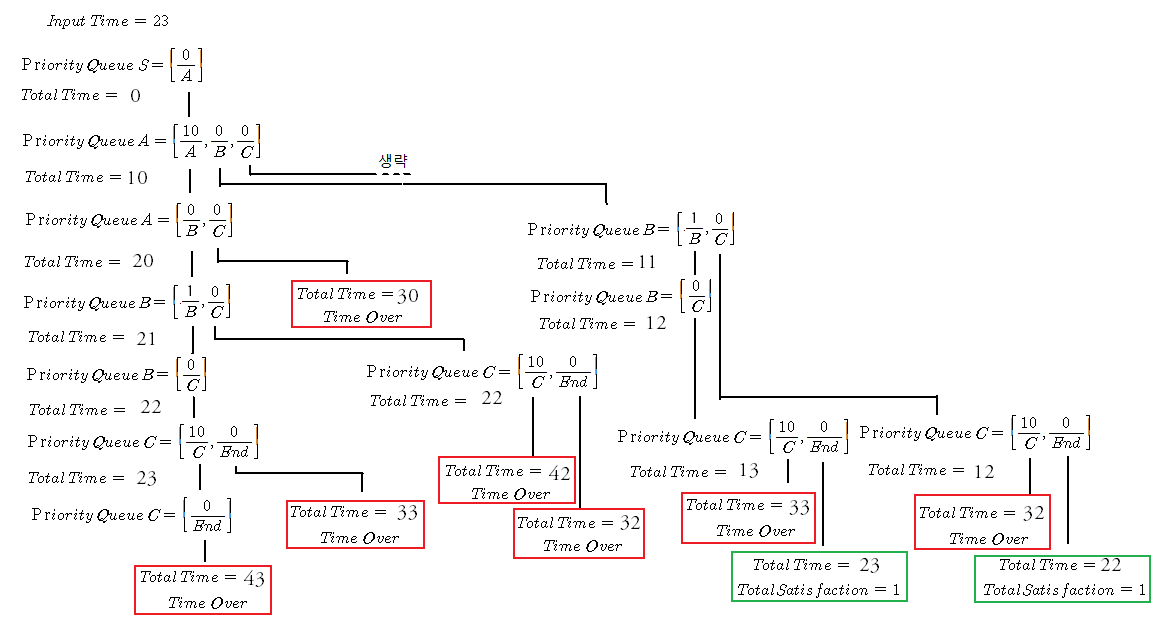
1. 일반적으로 Backtracking은 모든 경우의 수를 방문하여 최적의 해를 구할 수 있다.
2. 하위 노드는 1회 이동으로 방문할 수 있는 인접 노드를 의미한다.
3. 우선순위 큐가 1회 이동으로 획득할 수 있는 **만족도**를 Primary Key로 선택한다.
4. 사용자의 입력 시간 내에 이동 가능한 경로가 여러 개 생길 수 있기 때문에, 함수 실행 전에 변수를 선언하여 **누적된 만족도가 가장 높은 것**으로 계속 갱신한다.

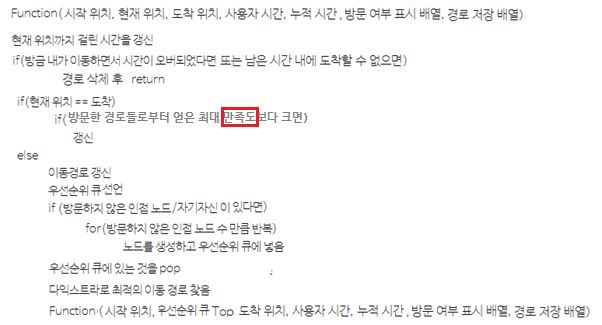
중요한 점은 이전 알고리즘의 특징이었던 가장 많은 관광지를 방문하는 것이 아닌, 1회 이동으로 얻을 수 있는 만족도가 가장 높은 경우를 생각한다는 것이다.

이 외에 고려할 수 있는 특징으로는 다음과 같은 것들이 있다.

(5) 현재 위치에서 남은 시간 내에 도착지까지 도달할 수 없다면 현재 위치는 유망하지 않다고 판단 가능하다.

(6) 사용자의 입력 시간이 출발지에서 도착지까지 최단 경로(즉, 관광지를 아무것도 들리지 않고 가는 경우)로 이동하는 데 걸리는 시간보다 작다면, 계산 할 필요 없이 예외 처리한다.

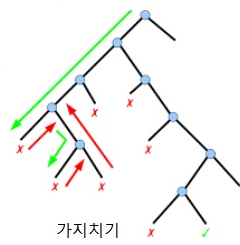
만약 사용자 입력 시간이 23이고, 위에서 든 예시 그래프를 이러한 방법을 이용해 최적의 경로를 탐색하다 보면, [그림 4]와 같은 과정과 결론이 나온다.

**[그림 4] 시간 내에 도착한 경로 중에서 관광지를 방문했을 때 얻을 수 있는 만족도가 높은 것을 선택한다**.

백트래킹을 이용한 수도 코드

ⓑ 2차 문제 해결 방법

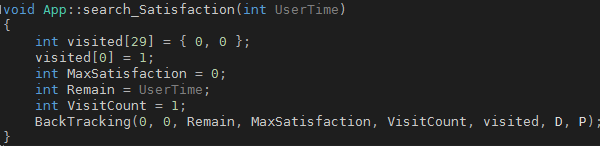
이번 2차 문제 해결 방법에서는 Back Tracking 방법을 계속 이용해 문제에 접근해 보고자 한다.

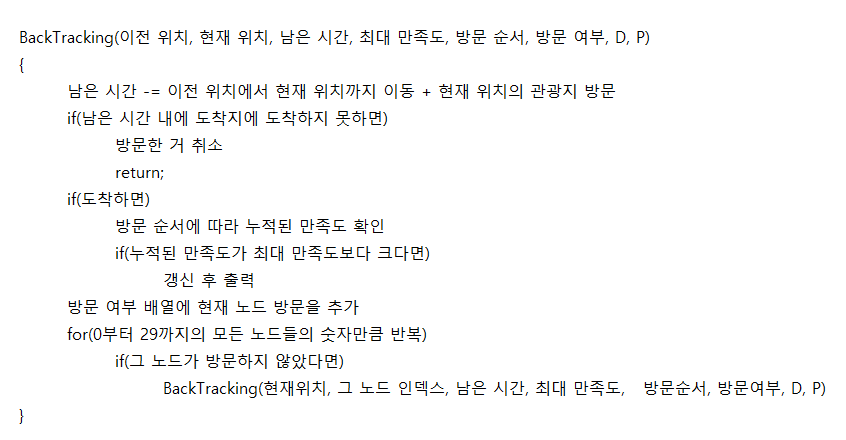
먼저 우리는 주어진 시간 내에 최대의 많은 만족도를 획득 위해서 세 가지 조건을 가정한다.

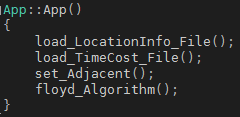
1. 1회 이동은 위에서 언급한 바와 같이 현재 위치에서 이동할 노드까지 최단 경로로 가는 데 걸리는 시간 + 도착한 노드의 관광지를 방문하는 것으로 정의한다.
2. 트리의 노드를 방문했을 때, 유망하지 않는 노드라면 그 이후는 볼 필요 없다.
3. 방문하지 않은 모든 경우의 노드에 대해서 재귀적으로 함수를 시행한다.

**우선 함수를 시행하기 전에 고려되어야 할 사항들은 다음과 같은 것들이 있다.**

|  |  |
| --- | --- |
| **파라미터** | **사용된 이유** |
| **Int 사용자 입력 시간** | **시간 내에 도착해야 하므로** |
| **Int 방문 배열[29]** | **1회 이동이 관광지 방문을 포함하기  때문에, 재방문이 포함되어서는 안되므로** |
| **Int 최대 만족도** | **도착했을 때 획득한 최대만족도가 기존보다 크다면 더 최적인 결과이므로** |
| **Int 이전 위치** | **1회 이동으로 소모되는 시간이 현재 위치와 이전 위치의 관계로 계산되므로** |
| **Int 현재 위치** | **1회 이동마다 현재 위치가 달라지므로** |
| **Int 남은 시간** | **1회 이동마다 남은 시간이 달라지므로** |
| **D** | **플로이드 알고리즘을 통해 구해진 거리값** |
| **P** | **플로이드 알고리즘을 통해 구해진 경로** |



**만족도를 이용한 검색에서는 먼저 사용자 입력을 파라미터로 받은 후, 나머지 지역 변수들을 초기화한다. 이를 이용해 구현된 함수의 수도 코드는 다음과 같다.**

ⓒ 문제 해결 방법 비교

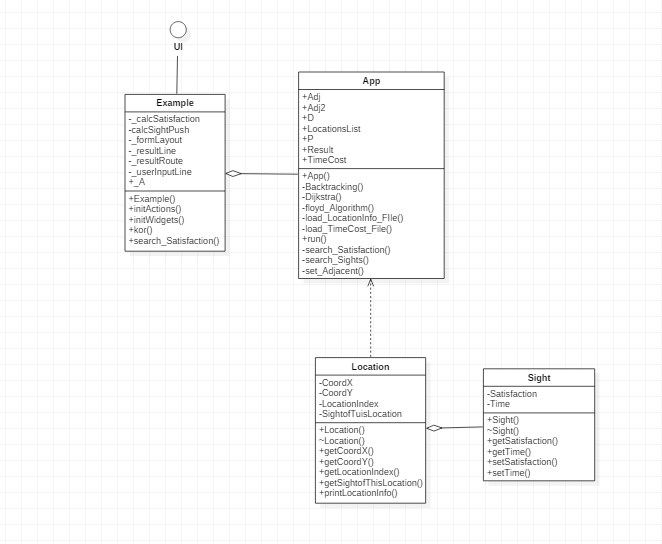
App의 생성자에서는 플로이드 알고리즘을 호출하여 저장한다.

플로이드 알고리즘을 활용한 것 또한 중요하다. 이전의 방식에서는 그래프에서 각 노드가 물리적으로 연결되어 있는 상태의 인접한 노드만을 검사했기 때문에, 다익스트라 알고리즘만을 활용했다. 하지만 알고리즘을 변경하면서, 모든 노드 사이의 거리와 시간을 비교할 필요가 생겼기 때문에, 다익스트라 알고리즘을 계속 사용하는 것보다는 프로그램 시행 시에 플로이드 알고리즘을 1회 호출하여 미리 저장해 놓는 편이 낫다고 생각했다.

|  |  |  |
| --- | --- | --- |
| 알고리즘 | 특징 | 적용한 이유 |
| 플로이드 알고리즘 | 1. 그래프에서 모든 노드 사이의 최단 경로의 거리를 구하는 알고리즘 2. 최소 비용뿐만 아니라 최소 비용 경로까지도 구할 수 있다 | 주어진 그래프 내에서 **모든 노드 사이의 최단 경로를 비교**할 필요가 있기 때문에, **O(**의 복잡도를 가진 플로이드 알고리즘을 프로그램 호출 실행 시 1회 호출하는 것으로도 최단 거리 및 경로를 구할 수 있음 |

**또한 모든 경우의 트리를 탐색하는 방법을 이용한다면, O(N!)의 비효율적인 알고리즘이 구현될 수 있다. 왜냐하면 노드 수가 많고, 각각의 노드에서 방문하지 않은 모든 경우의 수를 탐색하기 때문이다. 따라서 유망하지 않은 노드를 구분해야 하는데, 이것이 주어진 시간 내에 현재 노드에서 도착지까지 도착 가능한지의 여부로 판단한다. 이때 다익스트라 알고리즘을 이용해 함수를 호출할 때 마다 거리를 구하는 것보다는 미리 구해놓은 플로이드 알고리즘을 이용하면 단순히 O(1)의 비교만으로 유망한 지의 여부를 확인 가능하다.**

1. 클래스 다이어그램   
   클래스 다이어그램은 다음과 같다.

App클래스 , Location 클래스, Sight 클래스는 알고리즘을 구현하는 단계에서 만든 객체이다. Location클래스는 SightofLocation 변수를 가짐으로써 Sight클래스의 함수들을 관리가능하고, App클래스는 LocationsList 변수를 가짐으로써 이를 모두 관리할 수있다.

Example클래스는 이를 GUI로 나타내기 위한 클래스로 A변수로 APP클래스를 불러와 알고리즘을 사용하였다.

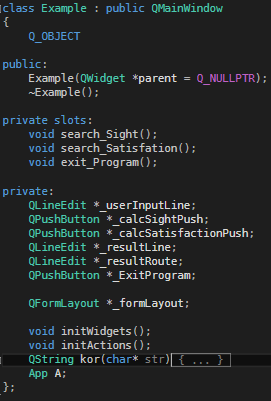
1. 결과

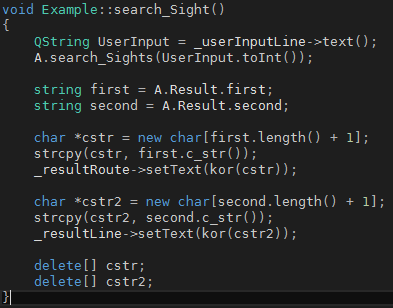
그래픽 유저 인터페이스(GUI)는 사용하기에 최대한 직관적이어야 한다. 메인 화면에서는 프로그램 제목과 함께 숫자를 입력 받을 수 있는 Line Edit과 종류에 따른 검색 버튼, 그리고 결과 출력Line Edit과 종료 버튼이 있다. 사용자가 숫자를 입력하고 특정 버튼을 누르면, 그에 맞는 결과가 즉시 출력되게 된다.

우리는 이러한 그래픽 유저 인터페이스를 위해, C++을 기반으로 한 프로그램을 GUI로 구현할 수 있는 Qt를 이용했다.

Qt는 컴퓨터 프로그래밍에서 GUI 프로그램 개발에 널리 쓰이는 크로스 플랫폼 프레임워크로, 네트워크와 데이터베이스를 제외한 순수 GUI 개발 에디션을 지원하고 있다. Qt는 C++를 주로 사용하기 때문에, 이를 시각화하기에 적절하다고 판단했다.



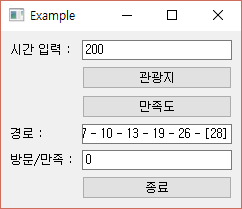
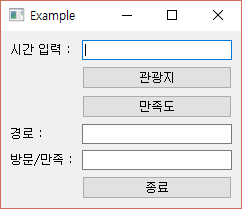




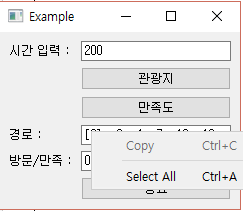
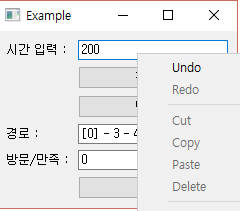
프로그램의 미적인 디자인 역시 중요하지만 그래도 가장 중요한 길찾기 기능 결과가 우선되어야 하기에, 실제 결과가 출력이 되는 것을 선택하였다.



초기 디자인이었지만, 정작 입력 처리 부분 구현이 매우 복잡해서 시간 내 구현이 어려울 것 같았다. 따라서 기능이 우선적으로 작동하는 프로그램을 구현하기로 하였다.



시간을 입력하고 관광지나 만족도 버튼을 누르면,   
경로가 아래에 출력되고 방문한 관광지나 만족도가 출력된다.



시간을 입력 창은 사용자 입력이 가능한 Line Edit이지만,   
출력하는 부분은 사용자가 임의로 설정할 수 없다.