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, 만족도 또는 관광지 개수를 기준으로 하는 힙을 이용한 우선순위 큐, 다익스트라 알고리즘 등을 활용해 구현할 수 있는 능력을 배양한다.

② 1차 문제 해결 결과

우선 우리는 문제에서 제시한 조건에 주목할 필요가 있다.

- **주어진 시간 내**에 가장 **많은 관광지**를 가거나 가장 **큰 만족도**를 얻을 수 있어야 하고

- **지역 중복 방문**은 허용되지만, **관광지 중복 방문**은 허용하지 않는다.

문제 해결을 위해 살펴보아야 할 중요한 변수로는 **주어진 시간 내**, **많은 관광지, 가장 큰 만족도, 지역 중복 방문,** 그리고 **관광지 중복 방문**이 있다.

우리는 지난 1차 보고서에서 이러한 변수들로 인해 일반적인 그래프에서 활용 가능한 알고리즘들을 이번 문제에 단순히 적용하기는 어려운 이유를 살펴보았다. 그 이유는 아래 표에서 간략히 정리해놓았다.

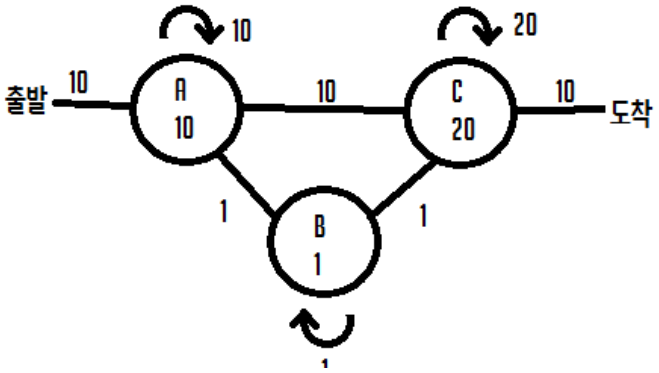
알고리즘	특징	적용하기 어려운 이유
다익스트라 알고리즘	1. 그래프에서 노드 사이의 최단 경로를 찾는 알고리즘 2. 일반적으로 가능한 적은 비용으로 가장 빠르게 해답에 도달하는 문제에 적용	단순히 최단 경로의 정보만을 제공
크루스칼 알고리즘	1. 일반적으로 그래프의 모든 노드를 방문하는 최소 비용의 신장 트리를 찾기 위해 사용	주어진 시간 내에 최대 관광지 방문과 최대 만족도 획득을 찾기 위한 경로 탐색 알고리즘으로 사용하기에는 부적절
프림 알고리즘		
플로이드 알고리즘	1. 그래프에서 모든 노드 사이의 최단 경로의 거리를 구하는 알고리즘 2. 최소 비용뿐만 아니라 최소 비용 경로까지도 구할 수 있다	하지만 문제에서 주어진 그래프가 완전 그래프가 아니고 모든 노드 사이의 최단 경로를 구할 필요가 없기 때문에, $O(n^3)$의 복잡도를 가진 플로이드 알고리즘만을 활용하기에는 효율성이 떨어짐

[표 1] 일반적인 그래프에서 활용 가능한 알고리즘의 사용이 어려운 이유

따라서 우리는 기존의 알고리즘들을 응용하고 발전시켜 새로운 알고리즘을 찾아야 한다.

ⓐ 문제 사항 1: 주어진 시간 내에 가장 많은 관광지를 방문하는 경우

우선 주어진 시간 내에 가장 많은 관광지를 방문하는 경우를 살펴본다. 여기서 기본적으로 가정해야 할 점은 다음과 같다.



[그림 1] 예시로 사용할 그림.

---

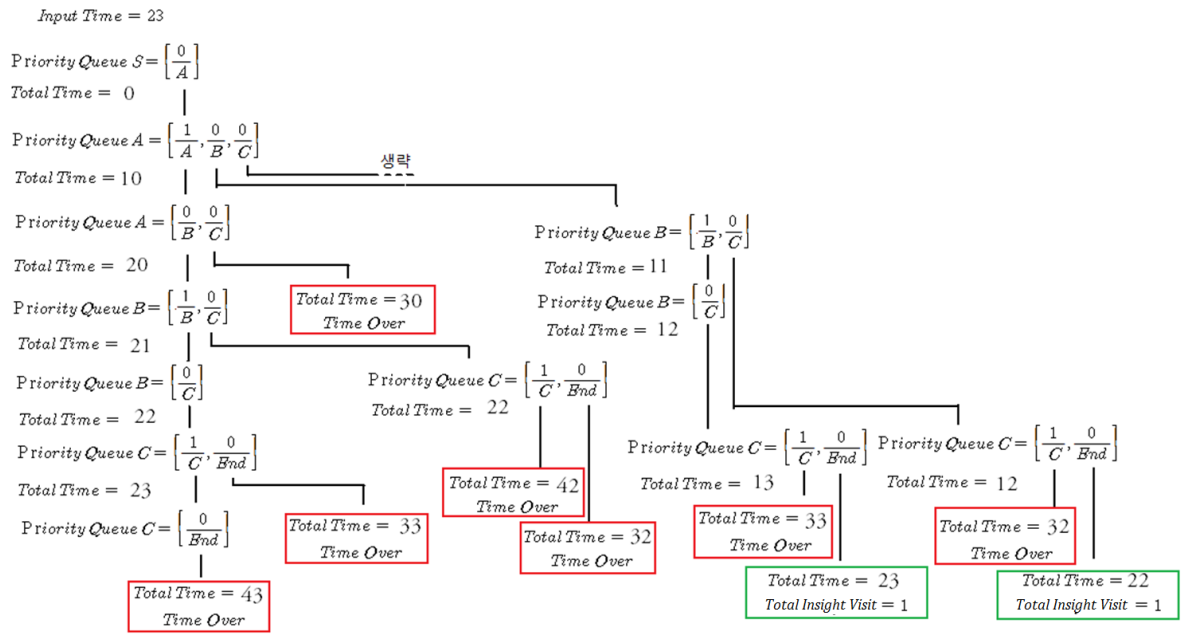
(1) 우선순위 큐는 1회 이동으로 획득할 수 있는 **관광지의 개수**를 Primary Key로 선택한다.

(2) 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.

(3) 일반적인 경우에 이미 방문한 노드는 더 많은 관광지를 방문할 방법을 구하기 위하여 다시 우선순위 큐에 넣지 않는다.

중요한 점은 가정 (1)은 **지역에서 관광지를 1회 방문하는 것을 자기 자신으로 향하는 Edge로 취급한다는 특징**을 바탕으로 이루어진다는 것이다. 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.



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

ⓑ 문제 사항 2: 주어진 시간 내에 가장 많은 만족도를 얻는 경우

다음으로 주어진 시간 내에 가장 큰 만족도를 얻을 수 있는 경우에 대해 살펴보자. 이 때 우리는 가장 많은 관광지를 방문하는 경우의 알고리즘을 응용하여 이용할 수 있다.

---

(1) 우선순위 큐가 1회 이동으로 획득할 수 있는 **만족도**를 Primary Key로 선택한다.

(2) 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.

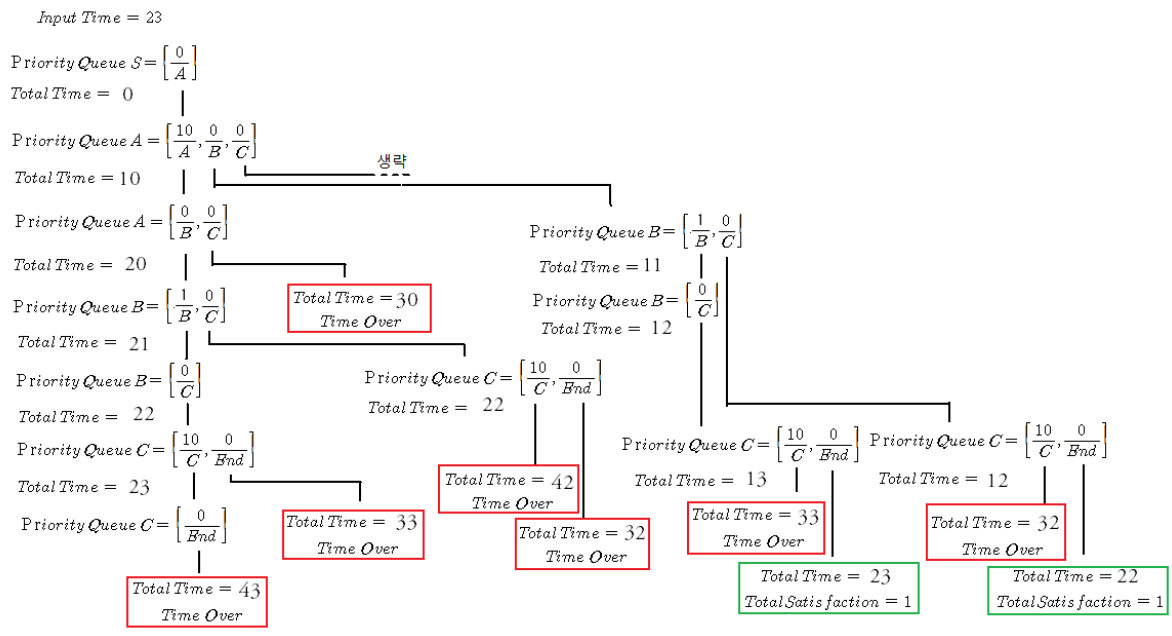
중요한 점은 이전 알고리즘의 특징이었던 가장 많은 관광지를 방문하는 것이 아닌, 1회 이동으로 얻을 수 있는 만족도가 가장 높은 경우를 생각한다는 것이다.

이 외에 고려할 수 있는 특징으로는 다음과 같은 것들이 있다.

(3) 사용자의 입력 시간을 저장하는 변수를 1회 이동마다 갱신하여, 현재까지 이동에 걸린 누적 시간이 사용자 입력 시간보다 클 경우에는 현재 열려있는 우선순위 큐 항목을 닫는다.
나아가, 우선 순위 큐의 효율을 증가시키기 위해, 현재 위치에서 남은 시간 내에 도착지까지 도달할 수 없다면 현재 위치는 유망하지 않다고 판단 가능하다.

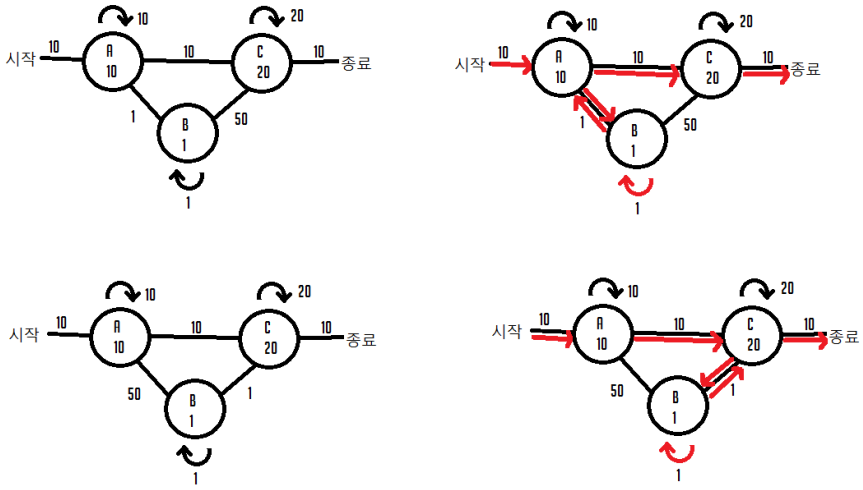
(4) 사용자의 입력 시간이 출발지에서 도착지까지 최단 경로(즉, 관광지를 아무것도 들리지 않고 가는 경우)로 이동하는 데 걸리는 시간보다 작다면, 계산 할 필요 없이 예외 처리한다.

만약 사용자 입력 시간이 23이고, 위에서 든 예시 그래프를 이러한 방법을 이용해 최적의 경로를 탐색하다 보면, [그림 3]와 같은 과정과 결론이 나온다.



[그림 3] 시간 내에 도착한 경로 중에서 관광지를 방문했을 때 얻을 수 있는 만족도가 높은 것을 선택한다.

따라서 사용자 입력 시간이 23일 때, Start – A – B – Sights of B – C – End로 가는 경로가 최선이라고 볼 수 있다.



[그림 4] 지역 재방문이 필요한 경우

하지만 여기에서 지역 재방문의 경우는 고려되지 않았다. [그림 4]와 같은 예시를 보자. 사진과 같은 그래프가 있을 때, 사용자 입력 시간이 33인 경우이다. 간단한 그래프이니만큼 직접 계산해보자면, 왼쪽에 그려진 각각의 그래프는 그 오른쪽에 있는 경로로 이동해야 만족도 1을 얻은 채 탐색이 끝날 수 있다. 따라서 위에서 언급한 방법만을 그대로 사용하면 한 번 들린 지역을 다시 방문하지 않기 때문에, 최적의 경로를 탐색할 수 없다.

따라서 우리는 다른 노드를 거쳐가더라도 최적의 효율을 발생시킬 수 있는 경로를 찾아 사용해야 한다. 그러므로 우리는 중복 방문이 있을 경우를 대비해 출력용 큐를 따로 구현하여야 한다.

이때, 인접한 노드들을 방문할 때, 다른 노드를 거쳐 가는 것이 비용 면에서 이득이라면 그 노드로 가는 최단 거리를 구하기 위해 **다익스트라 알고리즘(Dijkstra Algorithm)**을 사용한다. 여기에서 다익스트라 알고리즘이 사용 가능한 이유는 노드에 포함된 만족도와 같은 가중치를 고려하지 않고, 순수히 출발지(현재 유저가 있는 노드)에서 도착지(인접 노드)까지 가장 짧은 경로만을 탐색하기 때문이다.

위 내용을 정리하자면 다음과 같다.

(6) 중복 방문을 위해 프로그램 상 이동 경로와 출력용 경로를 따로 구현한다.

(7) 인접 노드를 방문할 때 다익스트라 알고리즘을 사용한다.

③ 2차 문제 해결 방법

우리는 문제를 해결하기 위해 다음과 같은 자료구조와 알고리즘을 필요로 한다.

자료구조 또는 알고리즘	사용한 이유
무방향 그래프	노드 간 이동 경로의 가중치가 대칭적이고 이동이 자유롭다. 따라서 단순 연결선을 갖는 무방향 그래프를 사용한다.
힙을 이용한 우선순위 큐	우선순위 큐를 재귀적으로 계속 호출하기 때문에, 우선순위 큐의 효율을 높이고자 한다. 힙을 이용하면 일반적인 경우의 시간 복잡도 O(1)보다 효율적인 O(logN)을 사용할 수 있다.
다익스트라 알고리즘	남은 시간 내에 도착지에 도달이 가능한 지 계산할 때, 인접 노드로 이동하는 최소 경로를 판단할 때, 노드의 유망성을 판단할 때 등 다양하게 사용된다.
되추적(Back Tracking)	모든 경우를 탐색하는 깊이 우선 검색은 효율적이지 않은 모든 경우의 수를 탐색하기 때문에 유망한 것만을 탐색해서 효율을 높인다.

[표 2] 사용한 자료구조와 알고리즘

```
class App
{

private:
    int LeadTime[29][29];
    vector<Location*> LocationInfo;
    vector<Node*> nodes;
    vector<pair<int, int>> adj[29];
```

본격적으로 구체적인 해결 방법을 찾아보자. App 클래스는 기본적인 프로그램의 중추

가 되는 클래스로, 여기에서 프로그램 실행 시 메모리에 데이터를 로드하고, 사용자의 입력을 처리하고, 경로 탐색을 진행한다.

```
void loadLocationInfoFile();
void loadLeadTimeFile();

void searchWay_Insights(int _start, int& here, int _destination, int
void searchWay_Satisfaction(int UserTime);
void searchWay_Dijkstra(int start, int destination);
int searchWay_Dijkstra_PriorityQueue(int _start, int _destination);
```

현재 App에 구현하고 있는 다양한 경로 탐색 알고리즘들이다.

```
class Location
{
private:
    int LocationIndex;
    float CoordX, CoordY;
    Insight InsightOfThisLocation;
```

```
class Insight
{
protected:
    int InsightIndex;
    int Satisfaction;
    int Time;
```

다음은 Location 클래스와 Insight 클래스다. Location은 기본적인 지역 정보를 담고 있다. 인덱스와 X, Y좌표를 멤버 변수로 가진다. 특이한 점은 내부에 Insight 객체를 멤버 변수로 갖고 있다는 점인데, 관광지가 Location에 위치한다는 점을 고려했다. 각각의 인덱스는 차후 방문 여부를 확인하거나, 특정 인덱스의 객체에 접근이 필요할 때 사용된다.

$$Priority\ Queue\ A = \left[\frac{10}{A}, \frac{0}{B}, \frac{0}{C}\right]$$
$$Total\ Satisfaction = 0$$

```
class Node : public Location
{
private:
    Location* location;
    int priority;
    bool operator>(Node& node);
    bool operator<(Node& node);
```

우리는 우선순위 큐를 이용하기 위해 특별히 노드라는 클래스를 정의하기로 하였다. 노드의 멤버 변수는 Location 정보와 Priority 정보를 담고 있다. 우리가 위 부분에서 제시한 우선순위 큐에 들어가는 그 항목을 노드로 정의한 것이다. 우선순위 큐에 적용시키기 위해서는 노드 간의 대소 비교를 할 연산자를 오버로딩 해야 하는데, 그 대소 비교의 역할을 Priority가 담당할 것이고 location은 이동할 Location의 정보를 담고 있다.

ⓐ 문제 사항 1

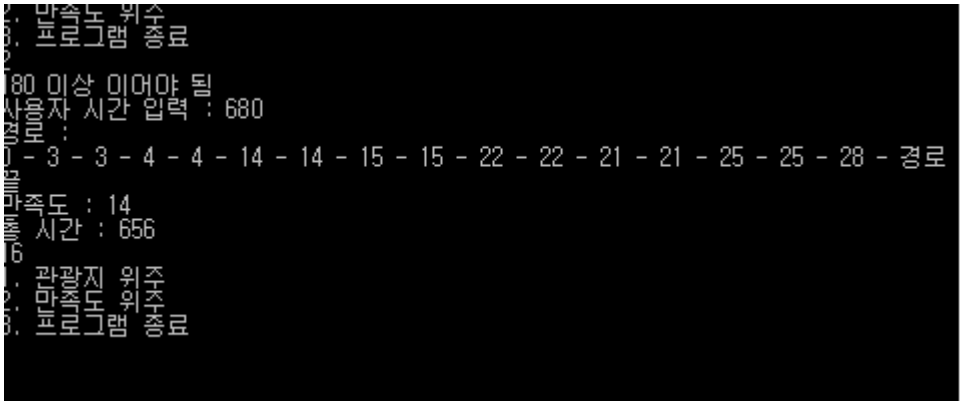
최대한 많은 관광지를 들리는 경우는 1회 이동으로 **방문할 수 있는 관광지**를 Primary Key로 활용하여 적용한다.

```
Function(시작 위치, 현재 위치, 도착 위치, 사용자 시간, 누적 시간, 방문 여부 표시 배열, 경로 저장 배열)
현재 위치까지 걸린 시간을 갱신
if(방금 내가 이동하면서 시간이 오버되었다면 또는 남은 시간 내에 도착할 수 없으면)
        경로 삭제 후  return
 if(현재 위치 == 도착)
        if(방문 표시한 배열에 표기한 방문 숫자가 최대 )
          갱신
 else
        이동경로 갱신
        우선순위 큐 선언
        if (방문하지 않은 인접 노드/자기자신 이 있다면)
              for(방문하지 않은 인접 노드 수 만큼 반복)
                    노드를 생성하고 우선순위 큐에 넣음
        우선순위 큐에 있는 것을 pop                    ;
        다익스트라로 최적의 이동 경로 찾음
        Function(시작 위치, 우선순위 큐 Top 도착 위치, 사용자 시간, 누적 시간, 방문 여부 표시 배열, 경로 저장 배열)
```

ⓑ 문제 사항 2

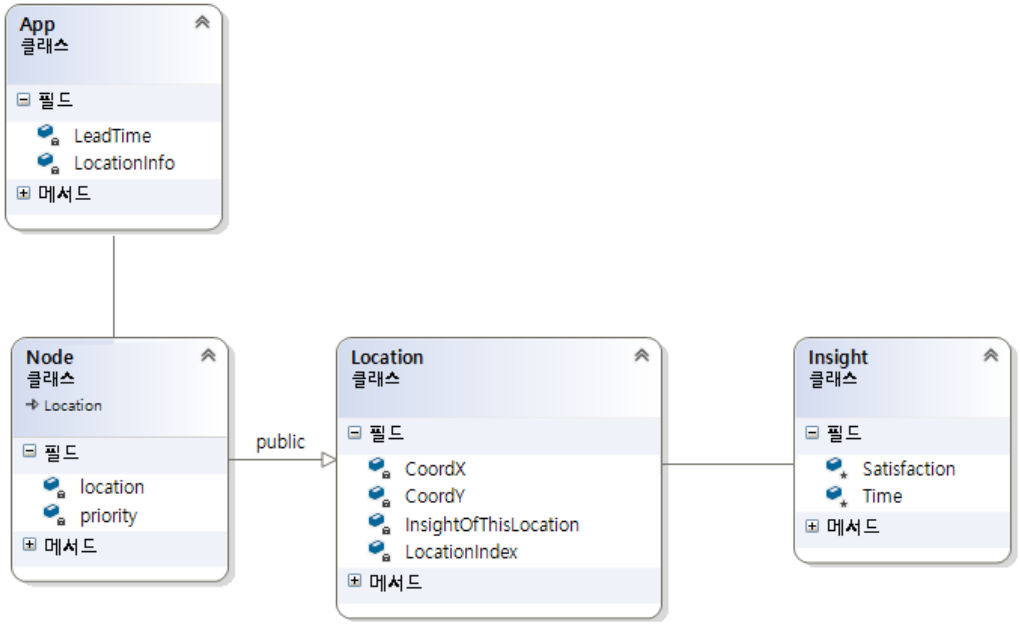
최대한 많은 만족도를 들리는 경우는 1회 이동으로 **얻을 수 있는 만족도**를 Primary Key로 활용하여 적용한다.

```
Function(시작 위치, 현재 위치, 도착 위치, 사용자 시간, 누적 시간, 방문 여부 표시 배열, 경로 저장 배열)
현재 위치까지 걸린 시간을 갱신
if(방금 내가 이동하면서 시간이 오버되었다면 또는 남은 시간 내에 도착할 수 없으면)
        경로 삭제 후  return
 if(현재 위치 == 도착)
        if(방문한 경로들로부터 얻은 최대 만족도보다 크면)
          갱신
 else
        이동경로 갱신
        우선순위 큐 선언
        if (방문하지 않은 인접 노드/자기자신 이 있다면)
              for(방문하지 않은 인접 노드 수 만큼 반복)
                    노드를 생성하고 우선순위 큐에 넣음
        우선순위 큐에 있는 것을 pop                    ;
        다익스트라로 최적의 이동 경로 찾음
        Function(시작 위치, 우선순위 큐 Top 도착 위치, 사용자 시간, 누적 시간, 방문 여부 표시 배열, 경로 저장 배열)
```



- 사용자 입력이 최소 경로(180)미만인 경우는 예외 처리하도록 하였음
- 경로가 중복 표시 된 것은 그 지역의 관광지를 방문한 것을 표시한 것
- 경로와 함께 획득한 만족도, 총 시간 등을 출력하게 구현했음
- 부분적으로 동작하기는 하지만 아직 완전히 구현되지 않은 부족한 부분이 많아 알고리즘 부분을 좀 더 심도 깊게 보완할 예정

④ 클래스 다이어그램



위에서 설명한 내용을 클래스 다이어그램으로 나타내면 위와 같다.

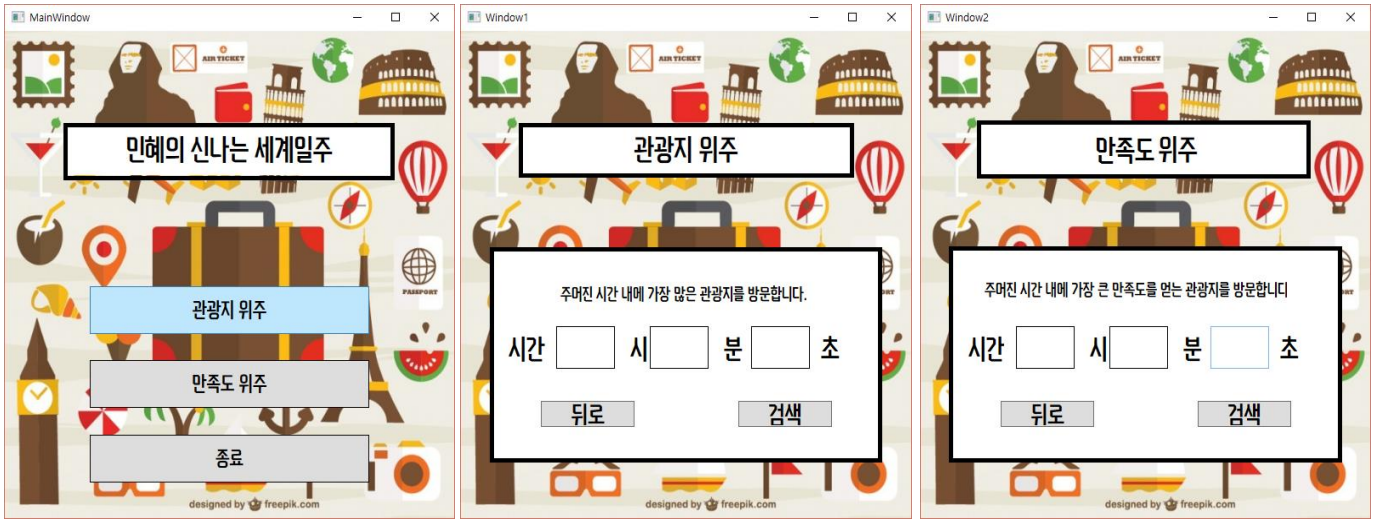
⑤ UI 구현

그래픽 유저 인터페이스는 사용하기에 최대한 직관적이어야 한다. 메인 화면에서는 프로그램 제목과 함께 관광지 위주, 만족도 위주, 종료 버튼이 있다. 배경은 이미지 파일을 로드하여 삽입하였다. 사용자가 특정 버튼을 누르면, 그에 맞는 새 창으로 넘어가게 된다.

버튼을 선택함에 따라 관광지 위주, 만족도 위주의 검색을 시행하기 위한 사용자 시간을 입력받는다. 시간을 입력한 후 검색을 시행하거나, 다시 메인 화면으로 돌아갈 수 있다. 시간은 숫자만 입력 받도록 예외처리 하였다.

우리는 이러한 그래픽 유저 인터페이스를 위해, C#과 XAML을 이용해 WPF를 구현했다.

[그림 6] 1차 보고서에서 유저 인터페이스로 제출한 모습



[그림 5] C#과 WPF를 이용해 구현한 유저 인터페이스

```
namespace DesignProjectC
{
    /// <summary>
    /// MainWindow.xaml에 대한 상호 작용 논리
    /// </summary>
    public partial class MainWindow : Window
    {
        public MainWindow()
        {
            InitializeComponent();
        }

        private void type1_Click(object sender, RoutedEventArgs e)
        {
            Window1 window1 = new Window1();
            this.Close();
            window1.ShowDialog();
        }
        private void type2_Click(object sender, RoutedEventArgs e)
        {
            Window2 window2 = new Window2();
            this.Close();
            window2.ShowDialog();
        }

        private void exit_Click(object sender, RoutedEventArgs e)
        {
            this.Close();
        }
    }
}
```

```
<Window x:Class="DesignProjectC.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:DesignProjectC"
        mc:Ignorable="d"
        Title="MainWindow" Height="600" Width="545">
    <Grid>

        <Grid.Background>
            <ImageBrush Stretch="Fill"  ImageSource="travel.jpg" />
        </Grid.Background>

        <Label x:Name="title" FontSize ="35" Margin="70,108,70,395" Background="Wh
        <Button Name="type1" FontSize="25" Margin="100,294,100,212" Click="type1_C
        <Button Name="type2" FontSize="25" Margin="100,379,100,127" Click="type2_C
        <Button Name="exit" FontSize="25" Margin="100,465,100,41" FontFamily="Kove

    </Grid>
</Window>
```