

# Semanhasht

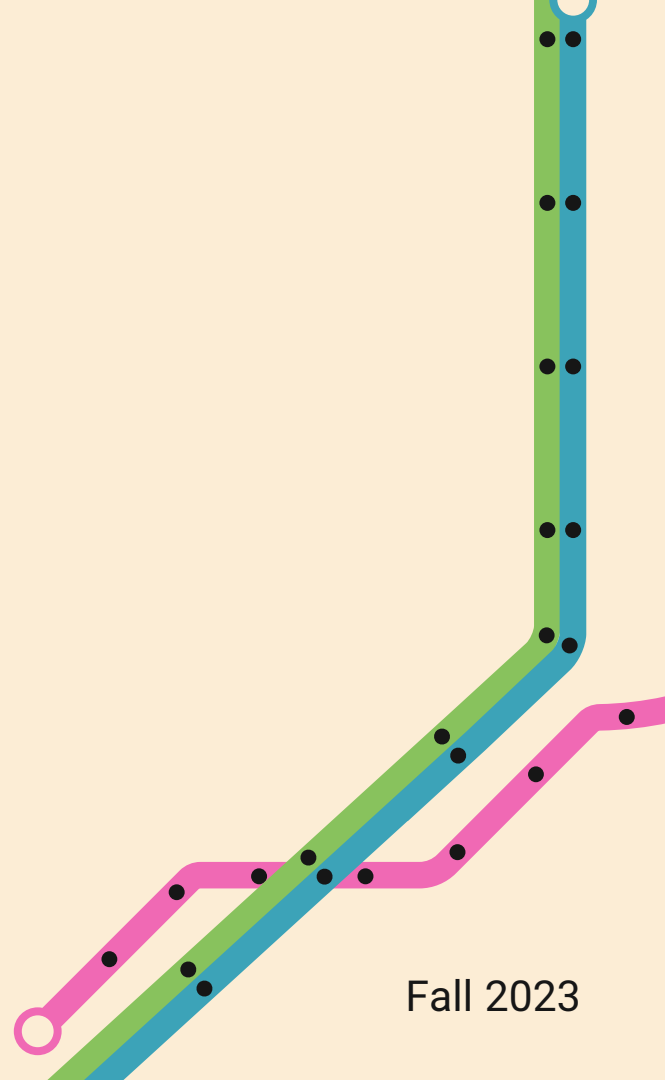
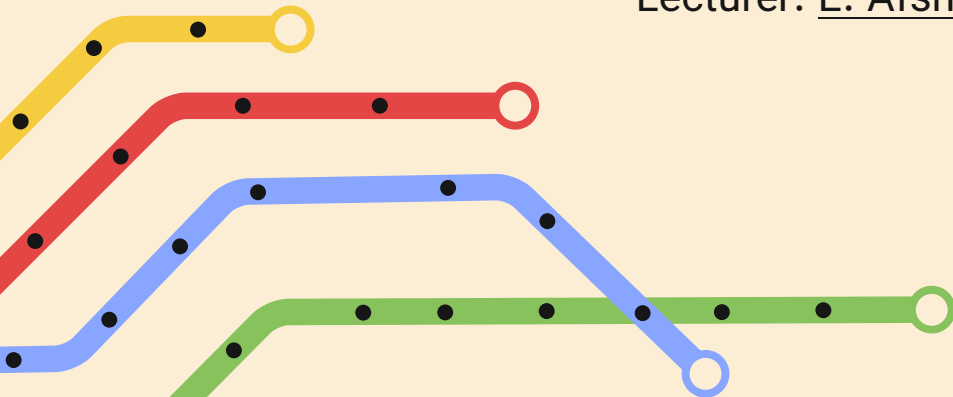
---

DS final Project

Mohammad Matin Parian  
Amir Hossein Jalili  
Amir Mohammad Mousavi

Lecturer: E. Afshar

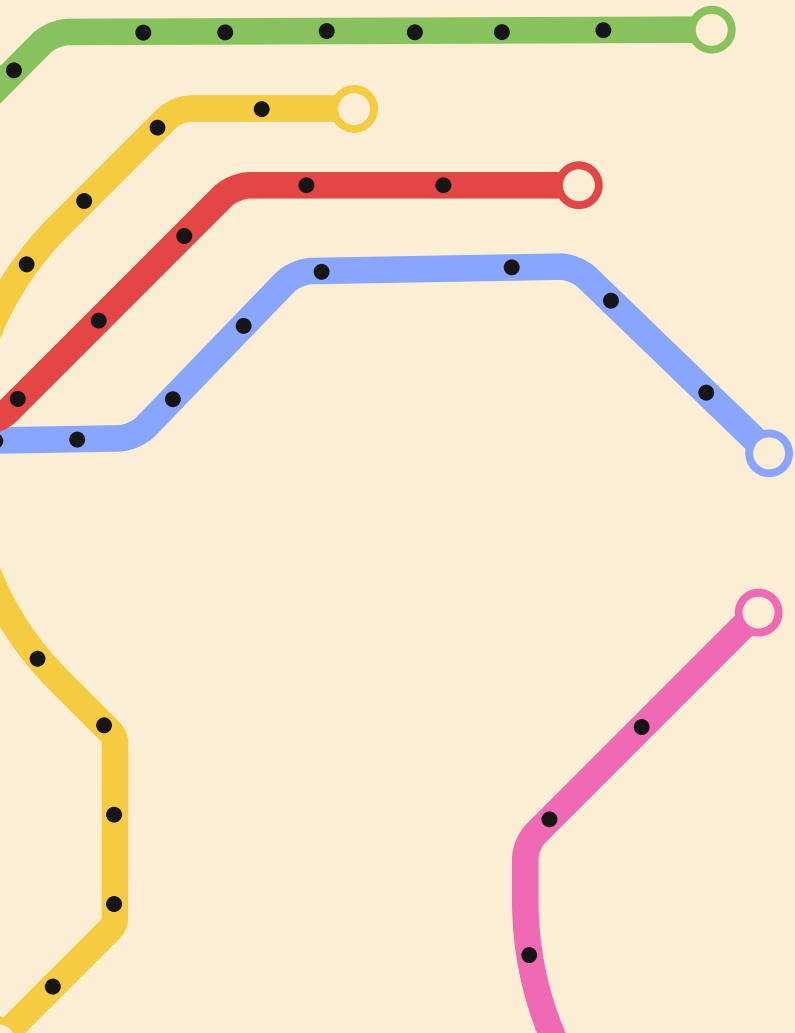
Fall 2023





# فهرست مطالب

فهرست	2
مقدمه	3
بدنه ی پروژه	6
کلاس ها	8
الگوریتم ها	18
سخن پایانی	<u>24</u>



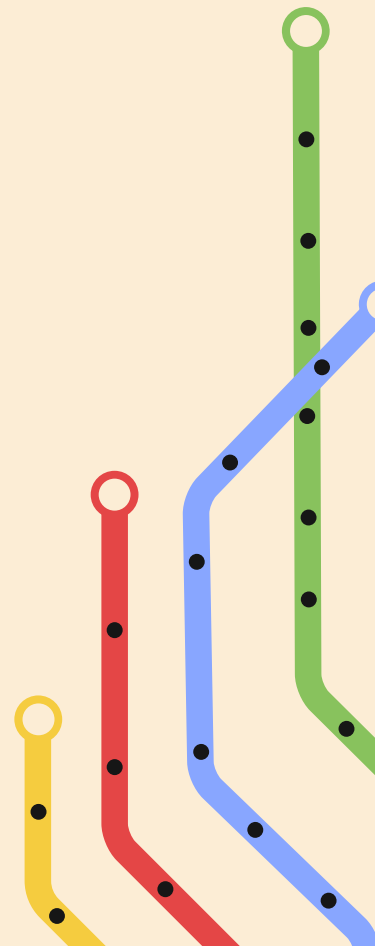
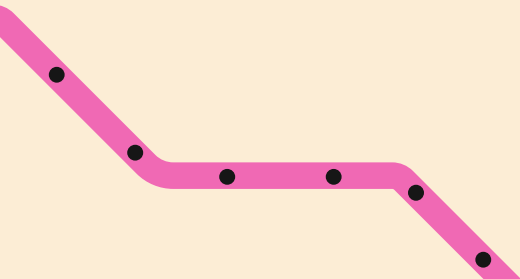
03

مقدمه



# موضوع پروژه

موضوع پروژه، سامانه مدیریت حمل و نقل شهروندی تهران (سمنحشت) می‌باشد.  
در پیاده سازی این سامانه، سعی بر آن بوده، که تمامی انواع وسایل نقلیه پوشش داده شود.  
جهت سهولت کار از نقشه ی کوچک تر استفاده شده و نقشه در مقیاس واقعی نیست.





# اهداف پروژه

## ساختمان های داده

این پروژه به منظور پیاده سازی و استفاده ی بهینه از ساختمان های داده ی متفاوت نظیر آرایه، لیست، صف، گراف و هش ساخته شده است.

## مشارکت و کار تیمی

در وهله دوم سعی بر مشارکت حداکثری تمام اعضای تیم و تقویت روحیه کار تیمی و هماهنگی درونی اعضا با یکدیگر بوده است.



06

بدنه پروژه

# سخن از کارآیست!

C++

Performance



OOP



Qt

Multi-platform



UI-oriented



C++ & Qt

به منظور حفظ کارآیی بالا و رعایت اصول شیء گرایی، این برنامه تماماً به زبان C++ نوشته شده است. برای طراحی و پیاده سازی محیط کاربری برنامه، از Qt بهره گرفته ایم.

# Classes

## Semanhasht

برای محاسبه کمترین مسافت، هزینه و کوتاه ترین زمان، کلاس هایی به همین نام ها تعریف شده است. از هر کدام یک شی ساخته میشود. و سپس وکتوری برای ذخیره نتیجه حاصل شده از الگوریتم بخش مربوطه تعریف شده است. (مثلا نتیجه کوتاهترین مسافت که در کانستراکتور کلاس مسافت محاسبه میشود در وکتور مربوط به آن ذخیره میشود.) همچنین زیر هر مورد یک شی از Qstring ساخته شده است که برای چاپ مسیر مربوطه است. مورد بعدی استک است که برای ذخیره سازی نود های یک مسیر به ترتیب مناسب است. ریست پث برای پاکسازی مسیر های بدست آمده است.

کلاس سمنحشت، وظیفه برقراری ارتباط بین منطق و رابط کاربری را بر عهده دارد. اولین دیتا ممبر این کلاس وکتوری به نام objects است. این وکتور حاوی لیست مجاورت یال های گرافیکی در qml است. در ادامه دو مپ داریم، یکی برای دادن ایندکس ایستگاه و گرفتن نام آن و دیگری بلعکس. مورد بعدی، تنها لیست مجاورت مورد استفاده در این برنامه است. این وکتور در کانستراکتور سمنحشت پر میشود.




```
31
32 private:
33     std::vector <std::vector<QObject*>>objects = std::vector <std::vector<QObject*>>(59);
34
35     std::unordered_map <std::string, int> stationToIndex;
36     std::unordered_map <int, std::string> indexToStation;
37
38     std::vector<std::vector<path>> distance_data;
39
40     Distance distance;
41     std::vector <path> distance_path;
42     QString q_distance_path; //prints the path
43
44     Cost cost;
45     std::vector <std::vector<std::pair<path, int>>> cost_path;
46     QString q_cost_path; //prints the path
47
48     Best_time best_time;
49     std::vector <std::vector<std::pair<path, int>>> time_path;
50     QString q_time_path; //prints the path
51
52     std::stack<path> print_pathS;
53
54     ResetPath RP;
55
56 };
57
58 #endif // SEMANHASHT_H
```

---

دو تابع "show\_path" و "print\_path" به ترتیب برای نمایش مسیر به صورت گرافیکی و بعدی برای چاپ مسیر در خروجی برنامه است.

پس از مشخص شدن مبدا و مقصد توسط کاربر و کلیک دکمه "مسیرها" تابع "direction" فراخوانی میشود. در اولین خط شیی از کلاس دست ساز "TTime" برای ذخیره زمان شروع سفر ساخته میشود. سپس هر سه تابع دایجسترا برای محاسبه بهترین حالت از هر نظر فراخوانی میشود.

The background features several stylized, colorful paths (yellow, pink, blue, red, green) with black dots, resembling a map or a network diagram. The paths are drawn with thick lines and have small black dots at various points along their length. The paths are set against a light beige background.

```
void Semanhasht::direction (int src, int end, const int &th, const int &tm){  
    TTime tt(th, tm); //saves the start time  
    distance_path = distance.dijkstra(distance_data, src, end);  
    cost_path = cost.dijkstra(distance_data, src, end);  
    time_path = best_time.dijkstra(distance_data, src, end, tt);  
    show_path (src, end, 3);  
    print_path(src, end, tt);  
}
```

## Distance

سپس استراکت "مقایسه جفت ها" برای مقایسه طول یال ها با هم و ایجاد صف اولویت براساس آن است. در نهایت خود کلاس مسافت تعریف شده، که شامل تابع الگوریتم دایجسترا و تابعی دیگر برای جمع کلی طول یال ها و محاسبه طول کل مسیر تعریف شده است، طبیعتاً دیتا ممبری برای ذخیره این مفهوم نیز تعریف شده است.

برای پیاده سازی کلاس مسافت ابتدا استراکتی تعریف کرده ایم، که مشخصات یال ها را نگه دارد. هر نمونه از این استراکت شامل: شماره نود شروع و پایان یال، طول یال و همچنین نوع و لاین وسیله نقلیه مورد نوع نظر در یال است. رقم اول tp نوع و رقم دوم لاین است.

```
1  #ifndef DISTANCE_H
2  #define DISTANCE_H
3
4  #include <vector>
5  #include <fstream>
6  #include <iostream>
7  #include <sstream>
8
9  struct path {
10     int start, end, length, tp;
11     path(){}
12     path (int s){
13         start=s ; end=s ; length=0; tp=0;
14     }
15     bool operator<(const std::pair<int , path> & other1) const {
16         return true;
17     }
18 };
19
20 struct comparePairs {
21     bool operator()(const std::pair<int, std::pair<path, int>>& p1, const std::pair<int, std::pair<path, int>>& p2) {
22         return p1.first > p2.first; // Sort in descending order
23     }
24 };
25
26 class Distance
27 {
28 public:
29     std::vector<path> dijkstra(const std::vector<std::vector<path>>&, const int &, const int &);
30     int get_total_distance();
31 private:
32     int total_distance{0};
33 };
34
35 #endif // DISTANCE_H
```

## Cost

در بخش دیتا ممبر ها سه متغیر عددی  
برای ذخیره هزینه هر یک نوع از انواع  
وسایل نقلیه و یکی برای ذخیره هزینه  
کلی تعریف شده است.

در کلاس هزینه، ابتدا الگوریتم  
دایجسترای مخصوص هزینه آمده  
است. سپس تابعی برای محاسبه  
هزینه کلی سفر را تعریف کرده ایم.

```
include > C cost.h
1  #ifndef COST_H
2  #define COST_H
3
4  #include <vector>
5  #include "distance.hpp"
6
7
8  class Cost
9  {
10 public:
11     Cost();
12     std::vector<std::vector<std::pair<path, int>>> dijkstra(std::vector<std::vector<path>>, const int&, const int&);
13     int get_total_cost();
14
15 private:
16     int bus_cost;
17     int subway_cost;
18     int taxi_cost;
19
20     int total_cost(0);
21 };
22
23 #endif // COST_H
24
```

## Best\_time

در بخش دیتا ممبر، برای هر یک از انواع وسایل نقلیه متغیری تعریف شده، تا زمان سپری شده مختص به آن وسیله را ذخیره کند. سپس مقادیر (ثابتی) برای اضافه کردن تاخیر هنگام تعویض لاین یا وسیله نقلیه تعریف شده است.

در این بخش تمهیدات لازم برای محاسبه بهترین مسیر از بعد زمان اندیشیده شده است. در این کلاس هم الگوریتمی از جنس دایجسترا، که برای محاسبه زمان بهینه سازی شده است، را استفاده کرده ایم. پس از تعریف این تابع، متغیری عددی برای ذخیره سازی زمان کلی صرف شده، تعریف شده است.



include > C:best\_time.h

```
1  #ifndef BEST_TIME_H
```

```
2  #define BEST_TIME_H
```

```
3
```

```
4  #include <vector>
```

```
5  #include "distance.hpp"
```

```
6  #include "ttime.h"
```

```
7
```

```
8  class Best_time
```

```
9  {
```

```
10 public:
```

```
11     Best_time();
```

```
12     std::vector<std::vector<std::pair<path, int>>> dijkstra(const std::vector<std::vector<path>>&, const int&, const int&, Tt
```

```
13     int get_time_cost();
```

```
14 private:
```

```
15     int bus_time;
```

```
16     int taxi_time;
```

```
17     int subway_time;
```

```
18
```

```
19     int bus_dilay = 15;
```

```
20     int taxi_dilay = 5;
```

```
21     int subway_dilay = 8;
```

```
22
```

```
23     int time_cost=0;
```

```
24 };
```

```
25
```

```
26 #endif // BEST_TIME_H
```

```
27
```

# Algorithms

## Distance

تابع محاسبه مسافت وکتوری از جنس  
یال ها که بالا به آن اشاره شد را  
برمیگرداند. در این الگوریتم از صف  
اولویت استفاده میکنیم که پیچیدگی  
زمانی را کاهش میدهد و بسیار به  
بهینگی نرم افزار کمک میکند.  
برای پیاده سازی زمان نیازی به تغییر  
الگوریتم دایجسترا نبوده است.

```

11 vector <path> Distance::dijkstra(const vector<vector<path>>&distance_data, const int &src, const int &des){
12     int V = 59;
13     vector <path> ans(V);
14
15     map <int, bool> visited;
16
17     priority_queue<pair<int, pair<path, int>>, vector<pair<int, pair<path, int>>>, comparePairs> z;
18
19     path x(src);
20
21     z.push(make_pair(0, make_pair(x, 0)));
22
23
24     while (!z.empty()) {
25
26         if (visited [z.top().second.first.end] == false){
27             ans[z.top().second.first.end] = (z.top().second.first);
28             visited [z.top().second.first.end] = true;
29             visited [z.top().second.first.start] = true;
30             if (z.top().second.first.end == des && total_distance==0) total_distance = z.top().first;
31         }
32         //the number of edges of the desired vertex
33         int edg_num = distance_data[z.top().second.first.end].size();
34         for (int i=0 ; i<edg_num; i++) {
35             int cost_distance = distance_data[z.top().second.first.end][i].length;
36             cost_distance += z.top().first;
37             if (visited [distance_data[z.top().second.first.end][i].end] == false ){
38                 z.push(make_pair(cost_distance, make_pair(distance_data[z.top().second.first.end][i], z.top().second.first.tp
39             )
40         }
41         z.pop();
42     }
43     return ans;
44 }
45

```

## Cost

در مقایسه یال ها اگر نوع و لاین آنها برابر باشد، هزینه را دیگر جمع نمیکنیم، زیرا این حالت نشان میدهد گذر از چند گره تنها با یک سفر و بدون تغییر لاین رخ داده است. لازم بذکر است، در صف اولویت نوع وسیله های نقلیه ی استفاده شده، تا رسیدن به حالت فعلی را نگهداری میکنیم که برای نمایش صحیح گرافیکی لازم است.

در این بخش، بر خلاف نسخه ی معمولی دایجسترا، که تنها ملاقات یک گره را چک میکند، وضعیت گره را هنگام ملاقات لحاظ میکنیم. (منظور از وضعیت نوع وسیله نقلیه نود و لاین آن است.) ابتدا چک میکنیم که اگر گره ای ملاقات نشده است، آنرا ملاقات کنیم. سپس در حلقه هر یالی که ملاقات نشده است را به وکتور از پیش تعیین شده اضافه میکنیم. سپس با توجه به نوع یال (که در استراکت یال) مشخص شده بود، هزینه را با فرمول مخصوص به خود محاسبه میکنیم.

```

while (!z.empty()) {

    if (visited [make_pair(z.top().second.first.end, z.top().second.first.tp)] == false){
        ans[z.top().second.first.end].push_back (make_pair((z.top().second.first) , z.top().second.second));
        visited [make_pair(z.top().second.first.end, z.top().second.first.tp)] = true;
        visited [make_pair(z.top().second.first.start, z.top().second.first.tp)] = true;
        if (z.top().second.first.end == des && total_cost==0) total_cost = z.top().first;
    }

    //the number of edges of the desired vertex
    int edg_num = distance_data[z.top().second.first.end].size();
    for (int i=0 ; i<edg_num; i++) {
        bool inline = (distance_data[z.top().second.first.end][i].tp != z.top().second.first.tp || z.top().second.first.tp/
        int cost;
        switch (distance_data[z.top().second.first.end][i].tp/10){
            case 1:
                cost = bus_cost;
                break;
            case 2:
                cost = distance_data[z.top().second.first.end][i].length * taxi_cost;
                break;
            case 3:
                cost = subway_cost;
                break;
        }
        cost = cost * inline + z.top().first;
        if (visited [make_pair(distance_data[z.top().second.first.end][i].end, distance_data[z.top().second.first.end][i].tp)] == false)
            z.push(make_pair(cost, make_pair(distance_data[z.top().second.first.end][i] , z.top().second.first.tp)));
    }
    z.pop();
}
return ans;
}

```

# Time


در بخش کنترل ترافیک با استفاده از کلاس `time` زمان سفر را لحاظ میکنیم تا بتوانیم با ضرایب مشخص شده هزینه و زمان سفر را افزایش دهیم.

در این بخش نیز مانند هزینه با استفاده از نوع وسیله نقلیه ( که رقم اول `tp` است.) زمان صرف شده با ضرب کردن در ضریب خاص هر وسیله به دست می آید.

```

while (!z.empty()) {
    if (visited [make_pair(z.top().second.first.end, z.top().second.first.tp)] == false){
        ans[z.top().second.first.end].push_back (make_pair((z.top().second.first) , z.top().second.second));
        visited [make_pair(z.top().second.first.end, z.top().second.first.tp)] = true;
        visited [make_pair(z.top().second.first.start, z.top().second.first.tp)] = true;
        if (z.top().second.first.end == des && time_cost==0) time_cost = z.top().first;
    }
    //the number of edges of the desired vertex
    int edg_num = distance_data[z.top().second.first.end].size();
    for (int i=0 ; i<edg_num; i++) {
        int cost_time;
        switch (distance_data[z.top().second.first.end][i].tp/10){
            case 1:
                cost_time = distance_data[z.top().second.first.end][i].length * bus_time;
                break;
            case 2:
                cost_time = distance_data[z.top().second.first.end][i].length * taxi_time;
                break;
            case 3:
                cost_time = distance_data[z.top().second.first.end][i].length * subway_time;
                break;
        }
        int traffic_time = tt.traffic_time(z.top().first);
        if (traffic_time==2 && distance_data[z.top().second.first.end][i].tp/10 == 2){
            cost_time = cost_time * 2;
        }
        else if (traffic_time==1){
            if (distance_data[z.top().second.first.end][i].tp/10 == 1){
                cost_time = cost_time * 2;
            }
        }
    }
}

```



امیدواریم پیاده سازی این پروژه، اولاً لحظاتی  
شیرین در خاطرم‌ان و دوماً گامی در جهت ارتقای  
سطح دانش ما بوده باشد.

— با افتخار تیم سمنحشت