دانشگاه صنعتی امیرکبیر
دانشکده مهندسی کامپیوتر

پروژه درس ساختمان داده‌ها و الگوریتم‌ها
(Routing with Graph Algorithms)

استاد درس: دکتر باقری

زمستان ۹۹

فهرست

مقدمه

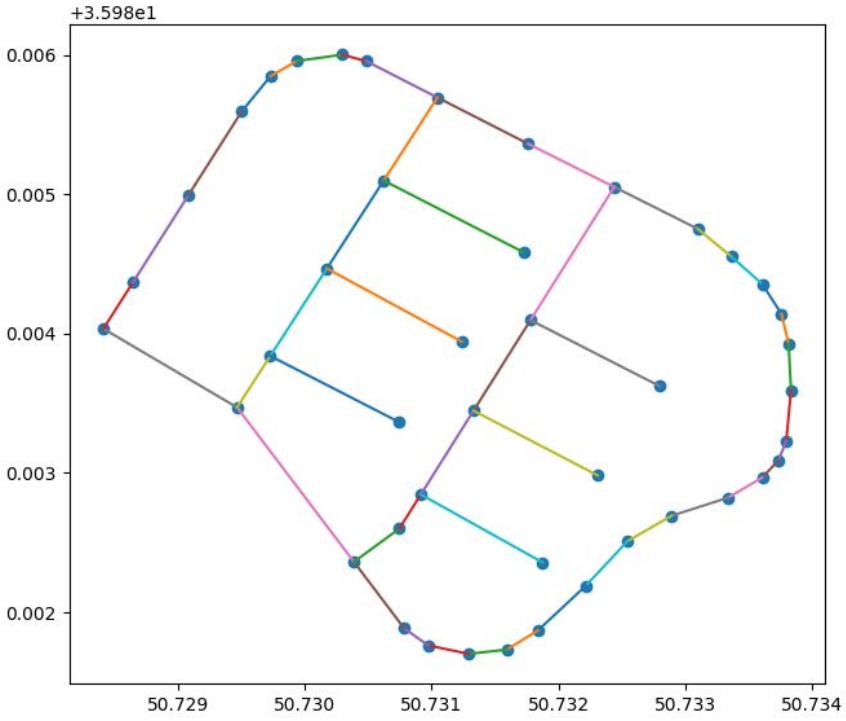
یکی از کاربردهای الگوریتم‌های Shortest Path که توی درس با اون‌ها آشنا شدیم، مسیریابی در نقشه است. برنامه‌هایی مانند Google Map و Waze و نمونه‌های ایرانی مثل «بلد» و «نشان»، چنین سرویس‌هایی رو ارائه می‌دن. در این برنامه‌ها، کاربرها مبدأ و مقصدشون رو مشخص می‌کنن و برنامه باید بهترین مسیر ممکن رو (از لحاظ کوتاه بودن مسیر، ترافیک و...) مشخص کنه.

توی این پروژه قصد داریم نمونه‌ی ساده‌ای از این برنامه‌های مسیریابی رو پیاده‌سازی کنیم و بر روی داده‌های واقعی (بخش‌هایی از نقشه تهران) اجراش کنیم.

مدلسازی مسئله به صورت الگوریتم گراف

فرض کنیم نقشه‌ای که می‌خواهیم با اون کار کنیم، به صورت زیر هستش:

این نقشه رو می‌شه به صورت یک گراف ذخیره کرد:



پس مسئله‌ی ما میشه پیدا کردن کوتاه‌ترین مسیر بین دو گره‌ی دلخواه در این گراف، که به کمک الگوریتم‌های Shortest Path می‌تونیم انجامش بدیم.

بخش اول

الگوریتم Dijkstra

ما توی درس با الگوریتم Floyd-Warshall آشنا شدیم که برای پیدا کردن کوتاه‌ترین مسیر بین تمام جفت گره‌های یک گراف به کار میاد. توی مسئله‌ی ما، با توجه به وجود ترافیک، وزن یال‌ها در طی زمان تغییر می‌کنه، در نتیجه وقتی یه کاربر درخواست میده، باید همون لحظه الگوریتم کوتاه‌ترین مسیر رو اجرا کنیم (یعنی نمی‌تونیم یک بار واسه همیشه، اول برنامه بیایم و کوتاه‌ترین مسیرها رو پیدا کنیم).

همچنین، نکته ای که وجود داره اینه که برای پاسخ به یک کاربر، نیازی نداریم بهترین مسیر بین تمام گره‌ها رو حساب کنیم. در نتیجه استفاده از Floyd-Warshall منطقی نیست.

بجاش میایم از الگوریتم Dijkstra استفاده کنیم که توی لینک زیر خیلی خوب توضیح داده شده:

How Dijkstra's Algorithm Works

همونطور که توی ویدیو دیدید، شبه‌کد این الگوریتم به شکل زیر هستش:

```
for each vertex v:
  dist[v] = ∞
  prev[v] = none
dist[source] = 0
set all vertices to unexplored
while destination not explored:
  v = least-valued unexplored vertex
  set v to explored
  for each edge (v, w):
    if dist[v] + len(v, w) < dist[w]:
      dist[w] = dist[v] + len(v, w)
      prev[w] = v
```

برای انتخاب گره‌ی بعدی‌ای که می‌خواهیم explore کنیم (خط ۷ شبه‌کد بالا)، اگه بخواهیم خیلی ساده عمل کنیم، میایم و کل آرایه رو پیمایش می‌کنیم تا گره unexplored ای که کمترین فاصله رو از گره مبدا داره، انتخاب کنیم. این کار هر بار از مرتبه $O(n)$ زمان می‌گیره و خوب نیستش. برای بهبود این قضیه، از یک Min-Heap که توی درس باهاش آشنا شدیم، استفاده می‌کنیم.

کافیه گره‌های unexplored رو در یک Min-Heap نگه‌داری کنیم و هر بار در $O(1)$، به بهترین گره دسترسی پیدا کنیم. اما نکته‌ای که وجود داره اینه که در طی اجرای الگوریتم، مقادیر تخمینی ما برای فاصله گره‌ها از مبدا، آپدیت می‌شن؛ در نتیجه این نیاز وجود داره که گره‌های موجود در Heap رو هم آپدیت کنیم. این آپدیت کردن شامل دو مرحله میشه:

۱- ابتدا باید گره‌ای که قراره آپدیتش کنیم رو توی هیپ پیدا کنیم. اگه بخوایم سرچ کنیم توی کل هیپ، بازم $O(n)$ طول می‌کشه و خوب نیست. پس برای این کار، در کنار هیپ یک آرایه یا Hash Table نگه داریم که برای هر گره‌ای که unexplored هستش، index اون گره توی هیپ رو نگه‌داری کنیم، تا بتونیم توی $O(1)$ گره رو توی هیپ پیدا کنیم.

۲- پس از اینکه index گره مورد نظرمون رو توی هیپ پیدا کردیم، ابتدا اون رو delete می‌کنیم و بعد با مقدار فاصله‌ی آپدیت‌شده، دوباره insert می‌کنیم. فقط حواسمون باید باشه که با ایجاد هر تغییری توی هیپ، باید اون آرایه یا Hash Table ای که در قسمت ۱ گفتیم رو هم آپدیت کنیم.

ورودی برنامه

ورودی برنامه بخش‌هایی از نقشه‌ی تهران هستش که ما از سایت Open Street Map استخراج کردیم. این نقشه‌ها رو به صورت لیستی از گره‌ها و یال‌های گراف و در قالب فایل متنی در اختیار دارید.

در خط اول، به ترتیب دو عدد n (تعداد گره‌ها) و m (تعداد یال‌ها) قرار دارند.

در n خط بعدی، هر خط مختص یک گره است که به ترتیب در آن id منحصر به فرد آن گره، latitude گره (y نقطه) و longitude گره (x نقطه) آورده شده است.

در m خط بعدی، هر خط یک یال است که id گره‌های دو سمت آن نوشته شده است. برای سادگی، در این مسئله فرض کرده‌ایم که تمامی مسیرها دوطرفه هستند (و در نتیجه گراف بدون جهت است).

با گرفتن ورودی، گراف را به صورت لیست همسایگی ذخیره کنید.

با توجه به اینکه id ها ترتیب خاصی ندارند، برای نگاشت id گره‌ها به اندیس آرایه در هنگام ذخیره گراف، از ساختمان داده Hash Table بهره می‌بریم. مثلا در پایتون تایپ داده dict و در جاوا کلاس HashMap، پیاده‌سازی‌هایی از Hash Table هستند.

درخواست کاربر

پس از دریافت ورودی‌های مرحله قبل، برنامه شما باید درخواست کاربرها رو از کامند لاین دریافت کنه. هر درخواست در یک خط و به صورت زیر داده می‌شه:

request_time start_id destination_id

قسمت اول، زمانی است که درخواست کاربر دریافت شده است. این زمان به دقیقه است و اولین درخواست، زمانش 0 هستش. قسمت دوم id گره مبدا و قسمت سوم، id گره مقصد می‌باشد.

خروجی برنامه

به ازای هر درخواست کاربر، خروجی ما به او، گره‌های بهترین مسیری که پیدا کرده‌ایم، می‌باشد (گره‌ها باید به ترتیب چاپ بشن). همچنین، زمان لازم برای پیمودن این مسیر نیز باید در یک خط جداگانه چاپ بشه. توضیحات نحوه محاسبه اون در ادامه آورده شده است.

برای مشخص کردن بهترین مسیر، باید دو پارامتر رو برای محاسبه وزن یال‌ها لحاظ کنیم: یکی فاصله‌ی مکانی بین گره‌هاست و دیگری ترافیکی که در مسیرها وجود داره. پس وزن هر یال، ترکیبی از این دو باید باشه. بنابراین، اون رو به کمک رابطه زیر محاسبه می‌کنیم:

$$\text{Weight} = \text{Length} \times (1 + \text{Traffic-Factor} \times \text{Traffic})$$

Weight: وزن یال

Length: فاصله اقلیدسی بین دو گره (که از latitude و longitude گره‌ها بدست میاد)

Traffic-Factor: ضریب ترافیک (ما آن را 0.3 در نظر گرفتیم)

Traffic: تعداد کاربرانی که مسیر آن‌ها از این یال به خصوص می‌گذرد.

فاصله مکانی بین گره‌ها، مقداری ثابت است؛ اما میزان ترافیک در طی زمان تغییر می‌کند. برای هندل کردن این قضیه، پس از اینکه برای یک کاربر مسیری رو انتخاب کردیم، ترافیک کل یال‌های اون مسیر رو بعلاوه ۱ می‌کنیم. در نتیجه، این موضوع در درخواست‌های کاربران بعدی تاثیر خواهد گذاشت. همچنین، هنگامی که کاربر به مقصد رسید، ترافیک کل یال‌های اون مسیر رو منهای ۱ می‌کنیم. (به حالت قبلش برمی‌گردونیم).
حال، با داشتن وزن یال، باید الگوریتم Dijkstra را اجرا کنیم و بهترین مسیر رو به کاربر معرفی کنیم.

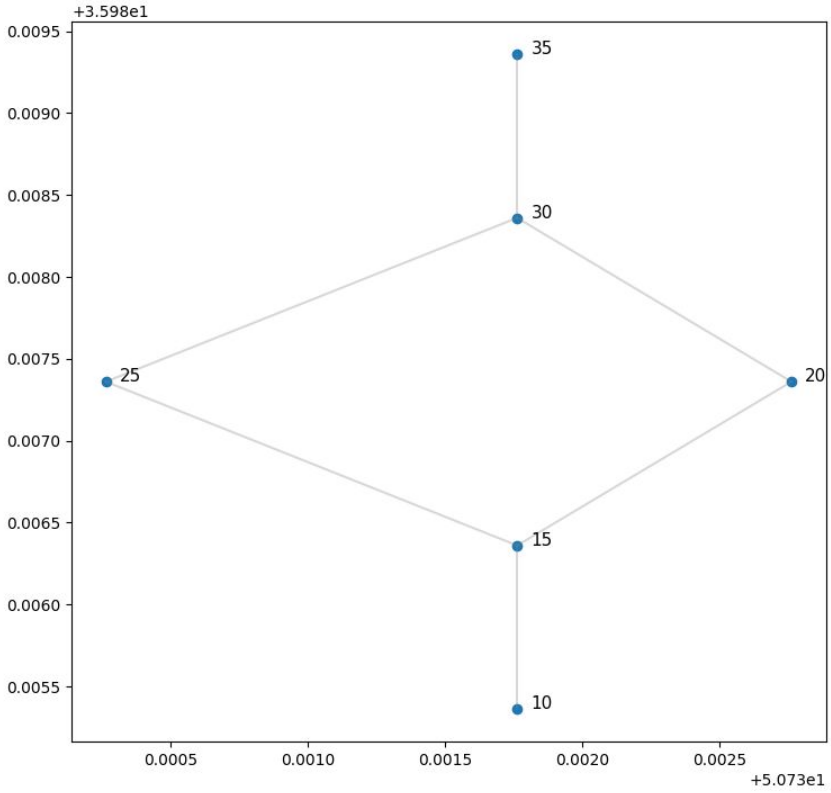
در نهایت، برای محاسبه زمان لازم (بر حسب دقیقه) برای پیمایش یک مسیر توسط کاربر، از فرمول زیر استفاده می‌کنیم:

$$\text{Time} = 120 \times (\Sigma_{edge \in path}\ edge.weight)$$

که این رابطه با توجه به مقیاس نقشه و با فرض اینکه کاربر مسیر رو با سرعت ۵۰ کیلومتر بر ساعت طی می‌کنه، بدست اومده.

مثال

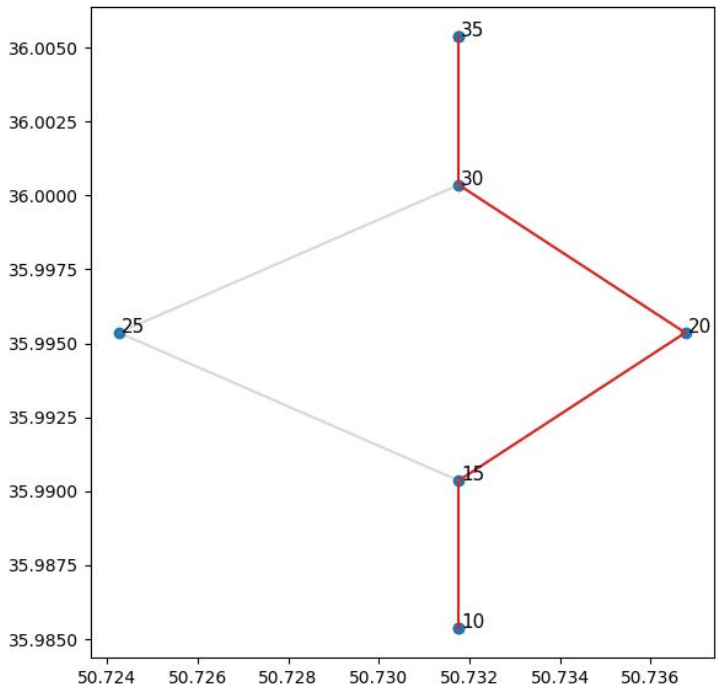
نقشه‌ی زیر رو در نظر بگیریم:



به ترتیب کامندهای زیر رو از کاربران دریافت می‌کنیم:

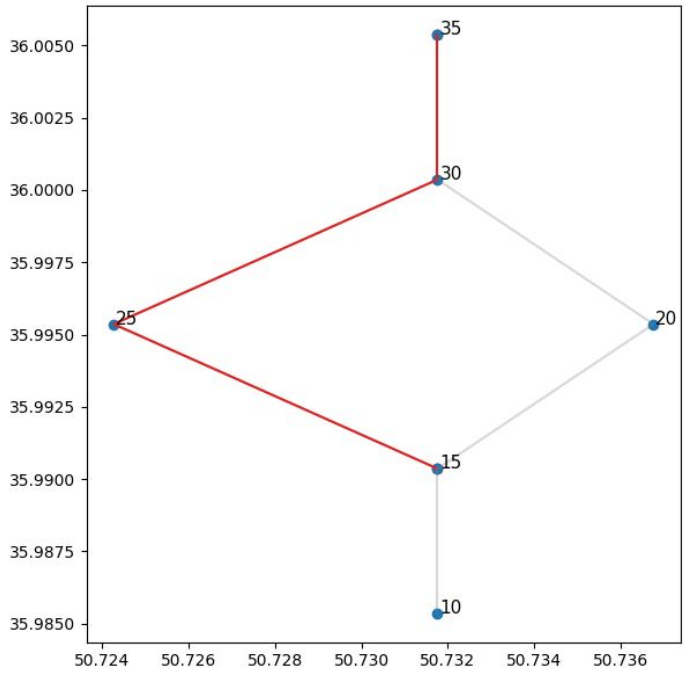
0 10 35
2 15 35
7 15 35

جواب ما به کاربر اول، مسیر زیر می‌باشد:



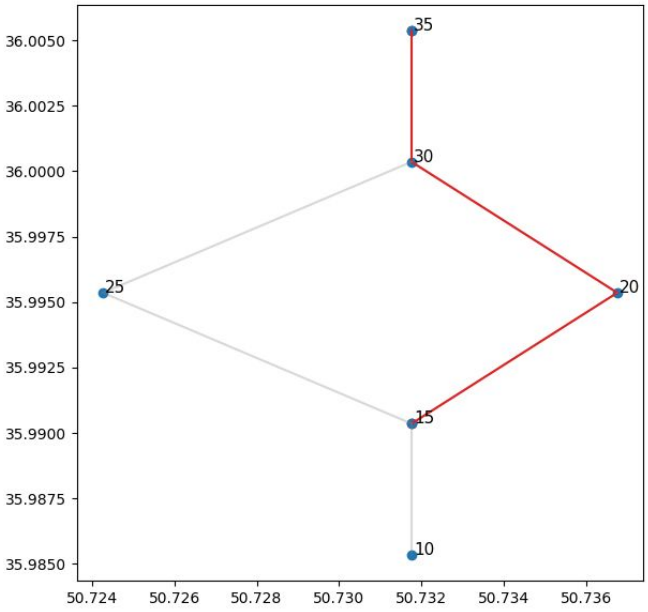
طبق رابطه‌ی گفته شده در قبل، پیمایش این مسیر 2.89 دقیقه طول می‌کشه.

کاربر دوم در دقیقه 2 درخواست داده که در اون لحظه، مسیر کاربر قبلی دارای ترافیک ۱ است (یعنی پارامتر ترافیک برای تمام یال‌های قرمز تصویر بالا، برابر با ۱ هستش). در نتیجه، اون یکی مسیر توسط الگوریتم Dijkstra ترجیح داده میشه. پس مسیر کاربر دوم به شکل زیر هستش:



که طی کردنش، 2.94 دقیقه طول می‌کشه.

در نهایت، کاربر سوم در دقیقه 7 درخواست داده که در اون زمان، کاربرهای قبلی به مقصد رسیدن و در نتیجه ترافیک تمامی یال‌ها صفر هستش. پس، مسیر زیر برگردونده میشه:



که طی کردن این مسیر، 2.29 دقیقه زمان می‌گیره.

بخش دوم (اختیاری)

۱- هنگامی که مسیری رو به یک کاربر معرفی کردیم، ترافیک کل یال‌های مسیر رو بعلاوه ۱ کردیم و پس از رسیدن کاربر به مقصد، ترافیک کل یال‌های اون مسیر رو منهای ۱ کردیم. برای اینکه ترافیک را دقیق‌تر مدل کنیم، می‌تونیم ترافیک هر یال در مسیر حرکت کاربر رو به طور جداگانه حساب کنیم. یعنی اگر مسیر کاربر شامل ۲ یال است، هنگامی که کاربر در یال اول قرار گرفت، ترافیک اون رو بعلاوه ۱ کنیم و هنگامی که به یال دوم وارد شد، ترافیک یال اول رو منهای ۱ کرده و دومی رو بعلاوه ۱ کنیم. در نهایت، با رسیدن کاربر به مقصد، ترافیک یال دوم رو نیز منهای ۱ کنیم. در اینصورت ترافیک یال‌ها به صورت واقعی‌تری محاسبه میشن.

۲- پلات کردن نقشه و مسیر انتخاب شده برای کاربران (مثلا توی پایتون می‌تونید از کتابخونه matplotlib استفاده کنید).

چند نکته

- پیاده‌سازی بخش اول 1 نمره و پیاده‌سازی بخش دوم 0.2 نمره دارد. پس در مجموع، پیاده‌سازی پروژه 1.2 نمره امتیازی دارد.
- لازم است فرمت ورودی رعایت شود اما فرمت خروجی خیلی مهم نیست، زیرا پروژه توسط کوئرا جاج نمی‌شود.
- فقط کد پروژه خود را در کوئرا بارگذاری نمایید.
- پروژه تحویل مجازی دارد.