Aufgabe 1: Schiebeparkplatz

Team-ID: 00067

Team-Name: Panic! at the Kernel

Bearbeiter/-innen dieser Aufgabe:
Christopher Besch

19. November 2021

Inhaltsverzeichnis

1 Lösungsidee **1**
1.1 Berechnung der Tabellen 2

2 Umsetzung und Quellcode **2**

3 Beispiele **3**

1 Lösungsidee

Fuer jedes Hotel laesst sich sagen, ob es innerhalb von d Tagen erreicht werden kann. Ist dies fuer ein d der Fall, kann fuer die minimale Bewertung aller auf dem optimalen Weg zu ihm liegenden Hotels angegeben werden. Dieser optimale Weg benoetigt genau d Tage. Zudem ist er in dem Sinne optimal, dass kein anderer Weg existiert, der in d Tagen das Hotel erreicht und eine besserer minimale Bewertung aufweist. Diese Werte koennen in einer Tabelle—der Minimumstabelle—dargestellt werden:

```
            min rating at day:
idx rating  0   1   2   3   4   5
=== ======  =   =   =   =   =   =
0   inf     inf
1   4.3         4.3
2   4.8         4.8 4.3
3   2.7         2.7 2.7 2.7
4   2.6         2.6 2.6 2.6 2.6
5   3.6             3.6 3.6 2.7 2.6
6   0.8             0.8 0.8 0.8 0.8
7   4.4             2.7 3.6 3.6 2.7
8   2.8                 2.7 2.8 2.8
9   2.6                 2.6 2.6 2.6
10  2.1                     2.1 2.1
11  2.8                     2.7 2.8
12  3.3                         2.7
13  inf                         2.7
```

Neben den Hotels werde hier mit dem Index 0 der Startort und mit 13 das Ziel gelistet. Dessen Bewertung ist nie schlechter als die eines beliebigen Hotels, weshalb sie als unendlich angesehen wird. Da eine Fahrt, die laenger als fuenf Tage dauert nicht zulaessig ist, werden diese Optionen weggelassen.

Anhand dieser Daten laesst sich recht leicht die minimale Bewertung auf dem Weg zu jedem beliebigen Hotel ausgeben. Interessant ist dieser Wert fuer das Ziel. Es muss lediglich das Minimum aller Werte in der letzten Zeile bestimmt werden.

Um nun die einzelnen Hotels, die optimal zum Ziel fuehren, zu bestimmen, wird eine zweite Tabelle—die Vorgaengertabelle—benoetigt. Diese weist die gleichen Dimensionen auf, gibt allerdings fuer jedes Hotel x und Tag d das Hotel, das am vorherigen Tag zuletzt besucht wurde, an. So kann vom Ziel am optimalen Tag ausgehend immer das zuletzt besuchte Hotel bestimmt werden. Dieser Vorgang endet, wenn man am Anfang angekommen ist.

1.1 Berechnung der Tabellen

Als Taglaenge wird die Strecke verstanden, die an einem Tag zurueckgelegt werden kann. Zu jedem Hotel x kann man am Tag d von allen Hotels, die maximal eine Taglaenge vor a liegt und innerhalb von genau $d-1$ Tagen erreichbar sind, gelangen. Diese Hotels bilden die Menge Y. Alle Hotels, die hinter x liegen, sind nicht zu verwenden. Die minimale a Bewertung zum Hotel x nach d Tagen entspricht mit

- dem Minimum b der Werte aus der Minimumstabelle fuer aller Hotels aus Y und den Tag $d-1$ und
- der Bewertung des Hotels x:

$$a = b + c$$

a wird in der Minimumstabelle gespeichert. Der entsprechende Wert in der Vorgaengertabelle entspricht dem Index des Hotels dessen

Da so immer nur die Information von Hotels, die vor dem aktuellen liegen, benoetigt werden, kann das Prinzip der dynamischen Programmierung verwendet werden. Hierbei werden die Tabellen anfangend beim „Start“ Hotel fuer Hotel gefuellt. Jedes weitere Hotel benoetigt ausschliesslich die bereits berechnete Information.

2 Umsetzung und Quellcode

Die Loesungsidee wird in C++ implementiert. Zur Repraesentation der Tabellen werden mehrdimensionale vectors benutzt:

```
// min_ratings[x][y] -> min rating till hotel x requiring y days to get to
// -1 -> impossible
std::vector<std::vector<float>> min_ratings(n + 2, std::vector<float>(DAYS + 1, -1));
std::vector<std::vector<int>>   last_hotel(n + 2, std::vector<int>(DAYS + 1, -1));
```

Nun kann mit der Funktion `populate_tables` diese Tabellen der Loesungsidee nach fuellen.

```
void populate_tables(
    const std::vector<std::pair<int, float>>& hotels,
    std::vector<std::vector<float>>&          min_ratings,
    std::vector<std::vector<int>>&            last_hotel)
{
    // the "first" (virtual) hotel never has the worst rating
    min_ratings[0][0] = inf;
    // go through all but "first" hotels
    for(auto cur = hotels.begin() + 1; cur != hotels.end(); ++cur) {
        int cur_idx = cur - hotels.begin();
        // first hotel ‘cur‘ can be reached from
        auto prev = std::lower_bound(hotels.begin(), hotels.end(),
                                     std::make_pair(cur->first - DAY_LEN, .0f));
        for(; prev != cur; ++prev) {
            int prev_idx = prev - hotels.begin();
            // go through past days when prev can be reached
            // call ride off after ‘DAYS‘ days
            for(int prev_day = 0, cur_day = 1; prev_day < DAYS; ++prev_day, ++cur_day) {
                // if the hotel ‘prev‘ can’t be reached in ‘prev_day‘ days
                if(min_ratings[prev_idx][prev_day] == -1)
                    continue;
                float new_min = std::min(min_ratings[prev_idx][prev_day], cur->second);
                // update when better
                if(new_min > min_ratings[cur_idx][cur_day]) {
                    min_ratings[cur_idx][cur_day] = new_min;
                    last_hotel[cur_idx][cur_day]  = prev_idx;
                }
            }
```

```
        }
    }
}
```

Wenn die Tabellen produziert wurde, koennen mit `get_best_day` die Wert des „Zielhotel“ in der Minimumstabelle linear nach dem Optimum durchsucht werden:

```
int get_best_day(const std::vector<std::vector<float>>& min_ratings, int n)
{
    int   best_day {-1};
    float best_min_rating {-1};
    for(int i = 0; i < DAYS + 1; ++i) {
        if(min_ratings[n + 1][i] > best_min_rating) {
            best_min_rating = min_ratings[n + 1][i];
            best_day        = i;
        }
    }
    return best_day;
}
```

Wurde festgestellt, dass es einen Weg zum „Zielhotel“ innerhalb von fuenf Tagen gibt, kann der optimale Weg mithilfe der Funktion `construct_path` bestimmt werden:

```
void construct_path(
    const std::vector<std::vector<int>>& last_hotel,
    int                                  best_day,
    int                                  n,
    std::vector<int>&                    path)
{
    int cur_idx = last_hotel[n + 1][best_day];
    for(int cur_day = best_day - 1; cur_day; --cur_day) {
        path.push_back(cur_idx);
        cur_idx = last_hotel[cur_idx][cur_day];
    }
    std::reverse(path.begin(), path.end());
}
```

Hierbei wird `cur_idx` immer so aktualisiert, dass immer das jeweilig vorhergehende Hotel referenziert wird.

Zum Schluss werden die beiden Tabellen und der finale, optimale Weg ausgegeben.

3 Beispiele