Project 1 Dynamic Programming

- 姓名：黃思誠
- 學號：112501533
- Group ID: 1
- Github Repository

目錄

如何編譯及運行

How to run

```
# in /project1/src
$ ../bin/[executable] [input_file_path] [output_file_path]
```

For example:

```
# in /project1/src
$../bin/project1 ../input/public_input1.txt
../output/public_output1.txt
```

How to compile

```
# in /project1/src
$ make
```

It will generate executable file `project1` in `../bin`.

Remove generated file

```
# in /project1/src
$ make clean
```

It will remove all generated file from `make`.

結果

Case 3

```
Total distance: 20
Oredering of cities:
  v1 v5 v7 v9 v3 v4 v6 v8 v2 v1
Execution time: 0.000633 seconds
```

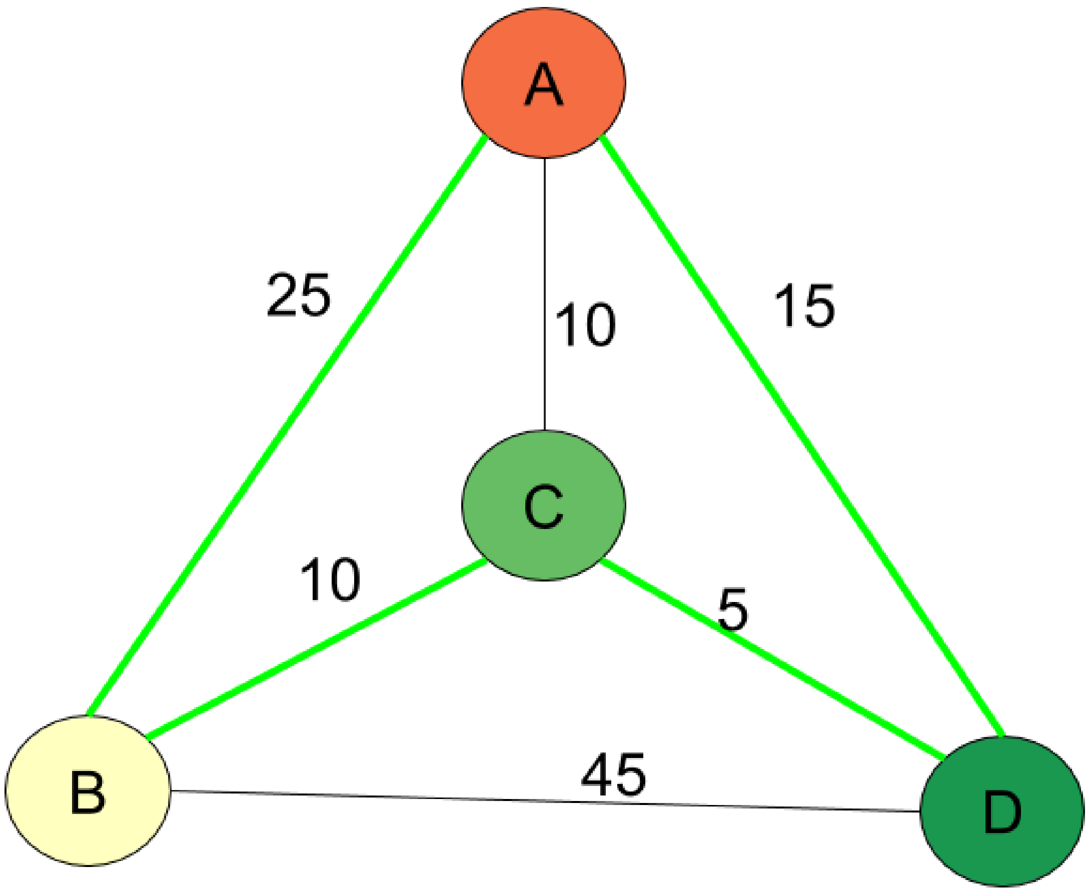
Case 4

```
Total distance: 25
Oredering of cities:
  v1 v10 v8 v6 v5 v7 v9 v3 v4 v11 v2 v1
Execution time: 0.000865 seconds
```

Case 5

```
Total distance: 25
Oredering of cities:
  v1 v10 v5 v12 v8 v6 v4 v11 v3 v9 v7 v2 v1
Execution time: 0.001279 seconds
```

Travelling Salesman Problem (TSP)



TSP 所要做的正是如上圖所顯示的：

- 給定節點及其互相之間的距離
- 找出從起點出發，走到每個節點都只剛好一次的最短的路線

Dynamic Programming (DP)

> ### 主要概念
>
> 通常適合使用 Dynamic Programming 的問題具有下列特性：
>
> - 具有很多重疊的子問題
> - 子問題具有規律性

> 由於這種特性，只要將前面子問題的解儲存起來，在求後續的問題（子問題）的解需要用到前面的解時，即可直接查表使用。所以是一種**用空間複雜度換取時間複雜度**的方式。
>
> 以解 Fibonacci number 當作範例：
>
> $$\begin{aligned}&F_0 = 0\\&F_1 = 1\\&F_n = F_{n-1} + F_{n-2}\end{aligned}$$
>
> 可以看到很明顯後續的解會用到前項的解，因此只要將 F_1、F_2、F_3 ... F_n 一個個儲存起來，即可快速的解出後續的答案。

解題思路

動態規劃方程推導

設 $G = (V, E)$ 為一帶權重的無向圖，$V = (v0, v1, v2, \dots v_{n-1})$ 為頂點集合。

從頂點 s 出發，設 $d(i, V)$ 為從頂點 s 出發經過集合 V 中各頂點各一次且僅一次，最後到達頂點 i 的最短路徑長度。

因此有兩種情況：

1. 當 $V = \varnothing$，代表從 s 到 i。

$$\begin{cases}d(i,V) = & C_{is}, & i \neq s\\ d(i,V) = & 0, & i = s\end{cases}$$

2. 當 $V \neq \varnothing$ ，就是找從前面走到 i 的最短路線，也對子問題求解。

$$d(i,V) = \min(C_{ik} + d(k, V-(k)))$$

   - k 為前一個節點，C_{ik} 為 i 到 k 的距離。

從因此最終可以得到一方程：

$$d(i,V) = \begin{cases}0, & V = \varnothing, i = s\\ C_{is}, & V = \varnothing, i \neq s\\ d(i,V) = \min(C_{ik} + d(k, V-(k))), & k \in V, V \neq \varnothing\end{cases}$$

Example

$$d(0,\{1,2,3\}) = \min\left\{\begin{matrix} C_{01} + d(1,\{2,3\}) \\ C_{02} + d(2,\{1,3\}) \\ C_{03} + d(3,\{1,2\}) \end{matrix}\right\}$$

然後依據同樣邏輯一路計算，直到集合為空，即可求解。

數據結構

dp Table								
index	0	1	2	3	4	5	6	7
i, V	{0}	{1}	{2}	{1, 2}	{3}	{1,3}	{2,3}	{1,2,3}
0								
1								
2								
3								

以總共 4 個點為例，$d(i,V)$ 總共會有上面這幾種可能。可以看到 V 的可能數量剛好是 2^{N-1}，N 為節點數量。因此可以使用 狀態壓縮 來簡單的表示不同的集合：

- {1, 2, 3} = (111) = 7
- {1} = (001) = 1

因此以 C++ Vector 來說，dp[0][7] 就代表最後所求的 $d(0,\{1,2,3\})$。

Bitwise 表達 (C++)

- 只有第 i 位為 1（代表第 i 個節點）：**1 << (i - 1)**
- 判斷 x 的第 i 位是不是 1：**(x & (1 << (i - 1))) == 1**
- 將 X 的第 i 位變為 0（從集合中剔除第 i 個節點）：**x = x ^ (1 << (i - 1))**

Pseudo Code

```
dp_table[0][all_node] = distance_matrix[0][all_node]

for state in states_of_sets:
        for node not in state:
                dp_table[state][node]
                        = min(dp_table[state_except_i][i] +
```

```
dis_matrix[i][node])
                                          for i in state
```

最短路徑求解

已知最短路徑長，通過動態規劃方程逆向計算即可：

$$d(0,\{1,2,3\}) = \min \begin{Bmatrix} C_{01} + d(1,\{2,3\}) \\ C_{02} + d(2,\{1,3\}) \\ C_{03} + d(3,\{1,2\}) \end{Bmatrix}$$

通過 dp*table 已知* $C_{01}+d(1,\{2,3\})$ 最短，再繼續計算，最後即可得到最短路徑。

結論

主要學習到的東西：

- Dynamic Programming 的實作
- 狀態壓縮的概念

主要遇到的困難：

- 太久沒寫 C++，語法忘光
- dp Table 的概念與如何實現
- 狀態壓縮的 Bitwise 操作