

データ構造とアルゴリズム実験レポート

<課題 6 グラフアルゴリズム:>

202213025 – 3 クラス – Can Minh Nghia

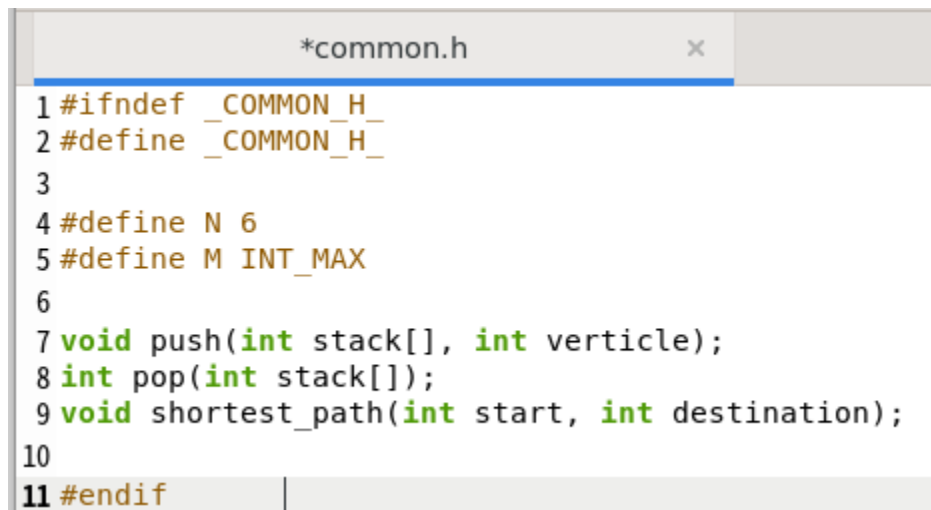
締切日: 2024/1/9

提出日: 2024/1/8

必須課題

まず、基本課題 3 のために、common.h と main ファイルにこのように追加する。

common.h ファイルに、line 7, 8, 9 のように追加する。



```
1 #ifndef _COMMON_H_
2 #define _COMMON_H_
3
4 #define N 6
5 #define M INT_MAX
6
7 void push(int stack[], int verticle);
8 int pop(int stack[]);
9 void shortest_path(int start, int destination);
10
11 #endif
```

さらに、main ファイルに、line 36, 37, 38, 60, 61 のようにを追加する。Line 36, 37, 38 は新しい関数でを宣言する。Line 60, 61 は基本課題3の結果を出力するためのコード。

```
33 int dijkstra(int);
34 void display(char *, int *, int);
35
36 void push(int[], int);
37 int pop(int[]);
38 void shortest_path(int, int);
39
40
41 int main(int argc, char *argv[]) {
42     if (argc != 2) {
43         fprintf(stderr, "Usage: ./main <node ID>\n");
44         return 1;
45     }
46
47     int p = atoi(argv[1]);
48     if (p < 0 || N <= p) {
49         fprintf(stderr, "Node ID %d is out of range: [0, %d].\n", p, N);
50         return 1;
51     }
52
53     for (int i = 0; i < N; i++)
54         S[i] = false;
55
56
57     dijkstra(p);                // ダイクストラ法による最短経路の計算
58     display("Result", d, N);    // 結果表示
59
60     for (int u = 0; u < N; u++)
61         shortest_path(p, u);
62
63     return 0;
64 }
```

課題 6-1

6-1-1 実装の仕様

上記ソースコードにおいて空白になっている関数を実装し、プログラムを完成させなさい。なお、`dijkstra` 関数も必要に応じて修正すること

6-1-2 実装コードおよび実装コードの説明

`dijkstra.c` にある関数を一つずつ解説する。

add 関数

```
15 /**
16  * Add vertex u to vertex set S.
17  * @to add param u Vertex
18  * @param S vertex set
19  * @no return
20  */
21
22 void add(int u, bool S[]) {
23     S[u] = true;
24 }
```

Array `S` は、どの頂点を訪問したかを表示するもの。訪問された頂点を `true`、訪問されていないものを `false` のように表す。

remain 関数

```
26 /**
27  * Check whether there are vertices in the vertex set that have not been added to S.
28  * @return true if there are vertices that have not been added to S, false otherwise
29  */
30 static bool remain() {
31     for (int i = 0; i < N; i++) {
32         if (!S[i]) {
33             return true;
34         }
35     }
36     return false;
37 }
```

この関数は、訪問されていない頂点があるかを確認する関数である。

Line 32: すべての頂点の中で、array S に存在しない頂点があったら true を返す。つまり、訪問されていない頂点があったら true を返す。

一方で、全部の頂点が array S に存在したら、つまり全部の頂点が訪問されたら、false を返す。

Select_min 関数

```
39 /**
40  * Among the vertices whose shortest distance from p is not determined, select the vertex with the minimum d[]
41  * return.
42  * @param None
43  * @return vertices with minimum undetermined d[]
44  */
45 int select_min() {
46     int min = -1;
47     int temp = M;
48     for (int i = 0; i < N; i++) {
49         if (S[i] == false) { //the verticle that its distance from p is not determined
50             if (d[i] <= temp) {
51                 temp = d[i];
52                 min = i;
53             }
54         }
55     }
56     return min;
57 }
58 }
```

この関数は、ある頂点（出発頂点）から、他のいくつかの到達可能頂点の中で、最短路がある到達頂点を返す関数である。

Line 49 ~ 52: d は辺の重みを保存する array。辺の重みが最も小さい頂点を返す。ただし、その頂点はまだ訪問されていないとする場合だけ (line 49)。

Line 57: ある頂点から、到達可能頂点がなかったり、訪問されていない頂点がなかったりしたら、-1 を返す。

member 関数

```
62 /**
63  * True if there is an edge connecting vertex u to vertex x, otherwise returns false.
64  * @param u vertex
65  * @param x vertex
66  * @return true if edge exists, false otherwise
67  */
68 bool member(int u, int x) {
69     if (w[u][x] != M) {
70         return true;
71     }
72     return false;
73 }
```

この関数は、頂点 u から頂点 x まで路があるかどうかを確認する関数。あったら true、なかったら false。

Line 69: w は隣接行列である。

dijkstra 関数

```

76 /**
77  * Calculate the weight of the shortest path from vertex p to each vertex using Dijkstra's method.
78  * @param p starting point
79  * @no return
80  */
81 void dijkstra(int p) {
82     add(p, S);
83
84     for (int i = 0; i < N; i++) {
85         d[i] = w[p][i];
86         if (d[i] != M) { //which vertices can be accessed directly from the start
87             parent[i][0] = p;
88         }
89     }

```

Line 82: 出発頂点を訪問したとして、array S の中に相当する要素を add 関数で更新する(その要素を true にする)。

Line 84: 隣接行列の中、相当する行を array d に格納する。

Line 87: 基本課題 3 に使うもの。今は無視する。

```

91 while (remain()) {
92     int u = select_min();
93     printf("%d\n", u); //check which vertices is being accessed
94     if (u == -1) //no path => break
95         break;
96     else
97         add(u, S);

```

Line 91: まだ訪問されていない頂点があったらという条件文。

Line 93: どの頂点が訪問されるかを表示する。

Line 94~95: 訪問されていない頂点の中で、路がない頂点だったら止める。

Line 96~97: 路がある頂点だったら、add 関数で array S に相当する要素を更新して進む。

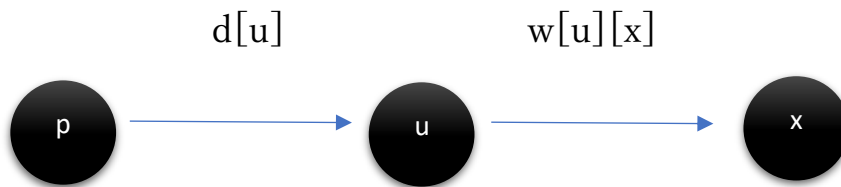
```

99     for (int x = 0; x < N; x++) {
100         if (member(u, x)) {
101             int k = d[u] + w[u][x]; //add path

```

Line 99~101: 頂点 u を中間として、次に頂点 x が到達可能だったら、 p から x までの重みを変数 k に格納する。

図にすると、



$$k = d[u] + w[u][x]$$

```
102 //avoid overflow (M plus something will cause overflow)
103 if (k < 0) {
104     k = M;
105 }
```

Line 102~105: ある正整数を M に足したら、overflow により負整数の結果が出るから、それを防ぐために、負整数の代わりに M のままとする。

```
---
107 if (d[x] == M) {
108     d[x] = k;
109     if (x != u) {
110         parent[x][0] = u; // Update parent
111     }
112 }
113 else if (k < d[x]) {
114     d[x] = k;
115     if (x != u) {
116         parent[x][0] = u; // Update parent
117     }
118 }
```

Line 107~118: p から x までの重み(d[x])が現在の d[x]より小さかったら d[x]を更新する。ただし、Line 109~110 と line 115~116 は基本課題 3 に使うから、今は無視する。

```
120     printf("parent of %d \n", x);
121         for (int checkx = 0; checkx < N; checkx++) {
122             printf("%d ", parent[x][checkx]);
123         }
124         printf("\n");
125     }
126 }
127 printf("\n");
128 }
129
130 }
```

Line 120~128 は基本課題 3 に使うから、今は無視する。

display 関数

```
182 /**
183  * Display the contents of the array to standard output. For outputting results and debugging.
184  * @param name Label (variable name, etc.)
185  * @param ary array
186  * @no return
187  */
188 void display(char* name, int* ary, int length) {
189     printf("%s: [", name);
190     for (int i = 0; i < length; i++) {
191         if (ary[i] == M)
192             printf(" M");
193         else
194             printf(" %d", ary[i]);
195     }
196     printf(" ]\n");
197 }
198 }
```

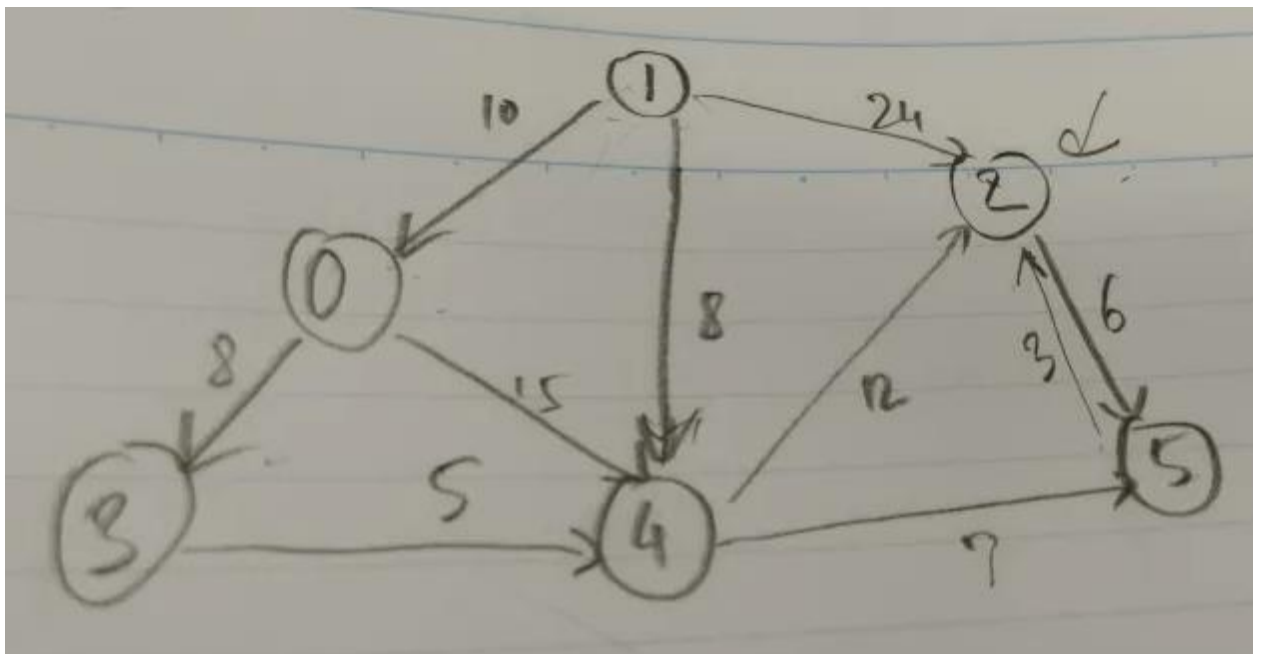

何も更新せず、与えたサンプルコードのままに使う。

6-1-3 実行結果：

まず、サンプルコードの隣接行列はこのようなである：

```
int w[N][N] = {  
  { 0, M, M, 8, 15, M},  
  {10, 0, 24, M, 8, M},  
  { M, M, 0, M, M, 6},  
  { M, M, M, 0, 5, M},  
  { M, M, 12, M, 0, 7},  
  { M, M, 3, M, M, 0}};
```

グラフにしたら、このようになる：



また、ターミナルで実行したら、この結果が得られた：

```
Result: [ 10 0 18 18 8 15 ]
```

課題が求めるものと同じであり、グラフに比べてもあっているの
で、実行が成功することがわかる。

課題 6-2

6-2-1 実装の仕様

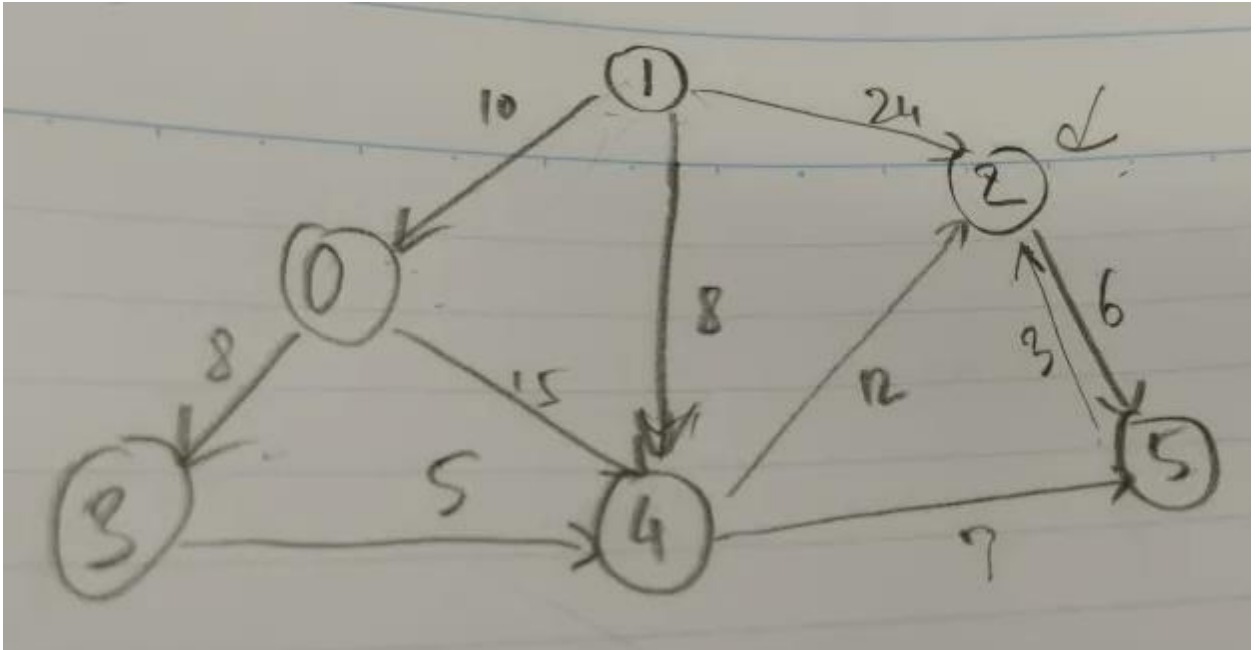
作成したプログラムを，サンプルコード中のグラフおよびサンプル以外のグラフ
（ただし，グラフのサイズは同じかより大きいものとする）に適用しなさい。
レポートでは，

- 確認に使用したグラフ
- 開始点
- 開始点から各頂点まで最短経路の重みの和

を示し，それがプログラムで正しく計算されていることを，二つ以上の開始点で
確認すること。

6-2-2 実行結果

サンプルコード中のグラフ



サンプルコード中のグラフ

開始点：2

Result: [M M 0 M M 6]

つまり、出発頂点が2だったら、到達可能頂点が5でしかない。グラフと比べたら、あっていることがわかる。

開始点：3

Result: [M M 15 0 5 12]

つまり、0と1には到達不可能で、頂点2への重みは15 (3 -> 4 -> 5 -> 2, つまり 5+7+3)。頂点4への重みは5(直接に到達する)。頂点5への重みは12 (3 -> 4 -> 5, つまり 5+7)。

開始点：4

Result: [M M 10 M 0 7]

つまり、0、1や3には到達不可能で、頂点2への重みは10(4->5->2)。頂点5への重みは7(直接に到達する)。

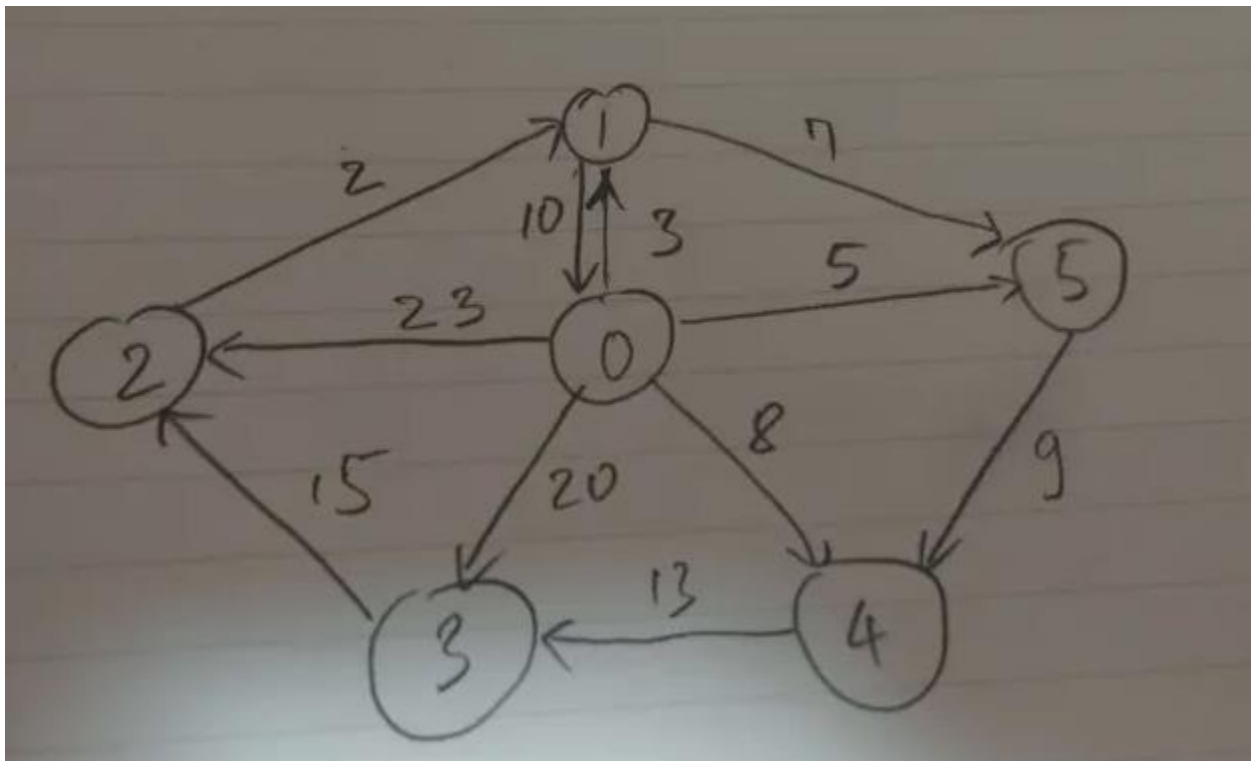
開始点: 5

Result: [M M 3 M M 0]

つまり、頂点2にしか到達できない。

グラフと比べたら、全部あっていることがわかる。

サンプルコード以外のグラフ



main 関数にある隣接行列：

```
7 int w[N][N] = {  
8   { 0, 3, 23, 20, 8, 5},  
9   {10, 0, M, M, M, 7},  
10  { M, 2, 0, M, M, M},  
11  { M, M, 15, 0, M, M},  
12  { M, M, M, 13, 0, M},  
13  { M, M, M, M, 9, 0}};
```

開始点：0

Result: [0 3 23 20 8 5]

つまり、他の頂点へ直接に到達する。

開始点：1

Result: [10 0 33 29 16 7]

つまり、0 には $1 \rightarrow 0 = 10$ 、2 には $1 \rightarrow 0 \rightarrow 2 = 10 + 23 = 33$ 、3 には $1 \rightarrow 5 \rightarrow 4 \rightarrow 3 = 7 + 9 + 13 = 29$ 、4 には $1 \rightarrow 5 \rightarrow 4 = 7 + 9 = 16$ 、5 には $1 \rightarrow 5 = 7$ 。全部あっていることがわかる。

よって、プログラムで正しく計算されていることがわかる。

課題 6-3

6-3-1 実装の仕様

最短路の重みに加えて、出発点から各頂点への最短路も出力するようにプログラムを拡張しなさい。

6-3-2 実装コードおよび実装コードの説明

dijkstra.c ファイル

```
7 extern int w[N][N]; //matrix
8 extern bool S[N]; //an array that shows which vertices have been accessed
9 extern int Scount;
10 extern int d[N]; //an array that shows distance from 1 verticle to others
11
12 int parent[N][N];
```

まず、line 12 のように、新しい array を作る。この array は、頂点の親を格納するものである。

```
132 /*****/
133 //Show Shortest Path
134 /***/
135
136 int STACK[N];
137
138 int top = 0; //pointer of the stack
139
```

次に、dijkstra 関数を作った後、最短路を表示するための push 関数、pop 関数、shortest_path 関数を作る。STACK と top を作っておく。

push 関数

```

140 void push(int stack[], int verticle) {
141     if (top > N-1) {
142         printf("top is %d\n", top);
143         printf("Stack Overflow at push\n");
144         exit(EXIT_FAILURE);
145     }
146     stack[top] = verticle;
147     top++;
148 }

```

教科書に書いてある課題2のサンプルコードと同じものである。ただし、デバッグするため、line 142, line 143 を追加する。

pop 関数

```

150 int pop(int stack[]) {
151     if (top < 0) {
152         printf("Stack Overflow at pop\n");
153     }
154     top--;
155     return stack[top];
156 }

```

教科書に書いてある課題2のサンプルコードと同じものである。ただし、デバッグするため、line 152 を追加する。

shortest_path 関数

```

158 void shortest_path(int start, int destination) {
159     int x;
160
161     if (d[destination] == M) {
162         printf("There is no path to %d.\n", destination);
163     }
164     else {
165         x = destination;
166         printf("Shortest path to %d \n", x);
167         push(STACK, x);
168         while (x != start) {
169             x = parent[x][0];
170             push(STACK, x);
171         }
172         while (top > 0) {
173             printf("%d->", pop(STACK));
174         }
175         printf("END\n");
176     }
177 }
178
179 /*****/

```

Line 161~163: 頂点 start から頂点 destination まで、路がなかったら、それを告げるためのコード。

Line 165~166: 路があったら、まず到達頂点を変数 x に格納し、後はどの頂点を到達するのかを printf 関数を使って表示する。

Line 167: push 関数で、到達頂点を STACK に格納する。

Line 168~171: while x != start という条件を使う。そうしないと無限ループになってしまう。特に、line 169 は追跡みたいに、頂点 x を知っていれば、x の親も知ることができる。それらの親を push 関数で STACK に格納する。

ここで、もう少し詳しく説明する。parent の要素がどこで格納するかというと、それは dijkstra 関数の中である。


```

84  for (int i = 0; i < N; i++) {
85      d[i] = w[p][i];
86      if (d[i] != M) { //which vertices can be accessed directly from the start
87          parent[i][0] = p;
88      }

```

dijkstra 関数、line 87 は、p からある頂点 i に到達することが可能だったら、i の親を p とする。

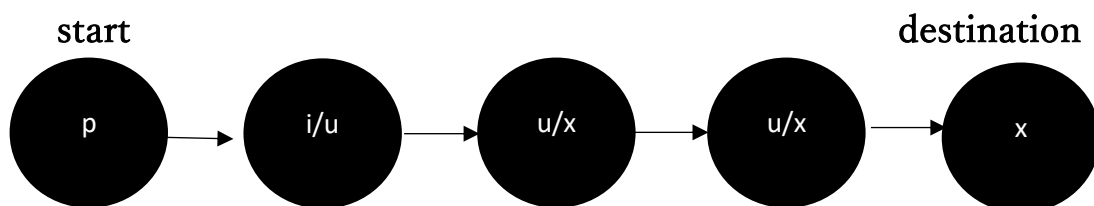
```

---
107  if (d[x] == M) {
108      d[x] = k;
109      if (x != u) {
110          parent[x][0] = u; // Update parent
111      }
112  }
113  else if (k < d[x]) {
114      d[x] = k;
115      if (x != u) {
116          parent[x][0] = u; // Update parent
117      }
118  }

```

dijkstra 関数、line 110 と 116 は、u からある頂点 x に到達することが可能だったら、x の親を u とする。最初のところ、u と i が重なる。

図にすると：



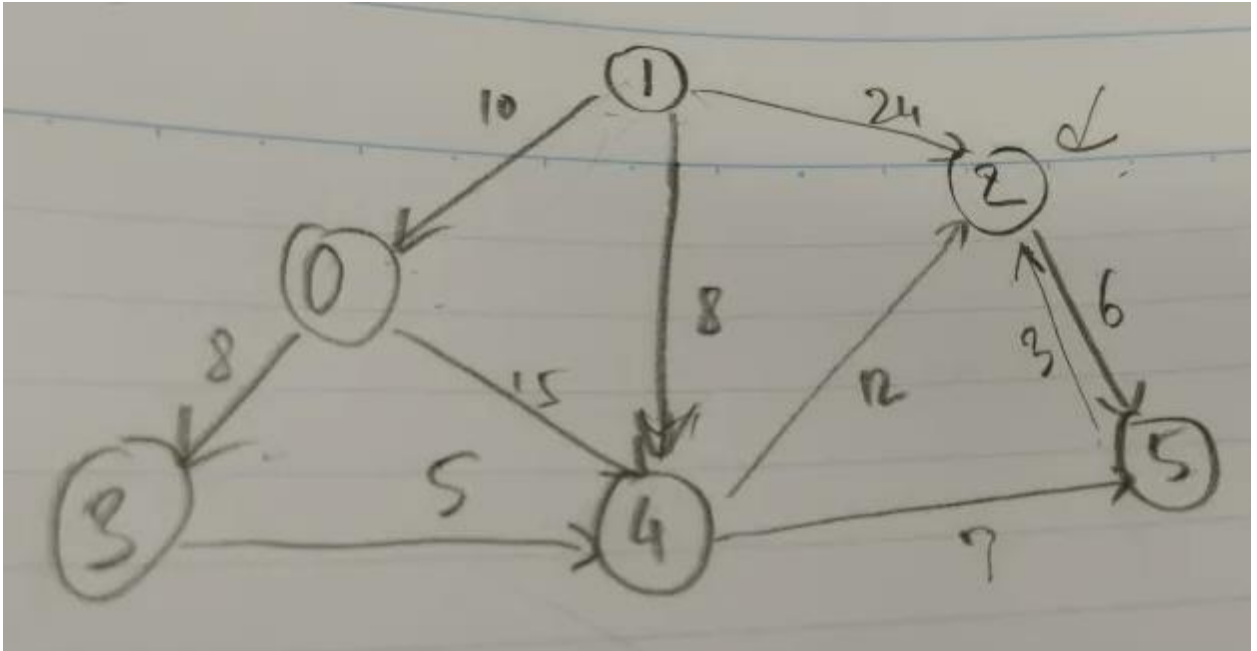
shortest_path 関数に戻る。

```
172         while (top > 0) {
173             printf("%d->", pop(STACK));
174         }
175         printf("END\n");
176     }
177 }
```

Line 172~177: 格納した親を pop 関数で STACK から出力する。

6-3-3 実行結果

サンプルコードの隣接行列を使う。



サンプルコードのグラフ

出発頂点: 1

```
Result: [ 10 0 18 18 8 15 ]  
Shortest path to 0  
1->0->END  
Shortest path to 1  
1->END  
Shortest path to 2  
1->4->5->2->END  
Shortest path to 3  
1->0->3->END  
Shortest path to 4  
1->4->END  
Shortest path to 5  
1->4->5->END
```

出発頂点: 2

```
Result: [ M M 0 M M 6 ]  
There is no path to 0.  
There is no path to 1.  
Shortest path to 2  
2->END  
There is no path to 3.  
There is no path to 4.  
Shortest path to 5  
2->5->END
```

出発頂点: 3

```
Result: [ M M 15 0 5 12 ]  
There is no path to 0.  
There is no path to 1.  
Shortest path to 2  
3->4->5->2->END  
Shortest path to 3  
3->END  
Shortest path to 4  
3->4->END  
Shortest path to 5  
3->4->5->END
```

—

出発頂点: 4

```
Result: [ M M 10 M 0 7 ]  
There is no path to 0.  
There is no path to 1.  
Shortest path to 2  
4->5->2->END  
There is no path to 3.  
Shortest path to 4  
4->END  
Shortest path to 5  
4->5->END
```

出発頂点: 5

```
Result: [ M M 3 M M 0 ]  
There is no path to 0.  
There is no path to 1.  
Shortest path to 2  
5->2->END  
There is no path to 3.  
There is no path to 4.  
Shortest path to 5  
5->END
```

グラフに比べると、全部あっていることがわかる。

感想

今回の課題が 2-simension array を適用する必要があるので、難しかったが、追跡することが面白いと思う。