11　迷路の経路探索

11.1　到達可能性判定

図 37 のように碁盤上に障害物が置かれているものとする。この時左上の S(スタート) から右下の G(ゴール) まで障害物を避けて移動できるかどうかを判定する問題を考える。現在いるマス目に隣接するマス目 (左右上下) に障害物がなければ、そのマス目へ移動できるものとする。また、マップの外へは移動出来ないものとする。

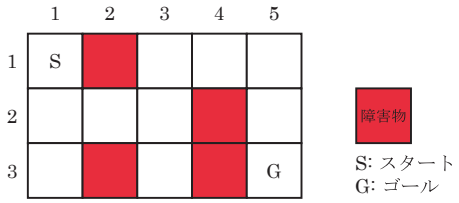


図 37: 迷路を表すマップ (迷路 1)

11.1.1　マップの表現

入力データ

例えば図 37 のマップに対して、プログラムへの入力は

$$
\begin{array}{ccccc}
5 & 3 & & & \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0
\end{array}
$$

のように行うものとする。ここで 1 行目はマップの幅と高さを与えており、続く 3 行のデータはマップの各行のデータを示しており、1 は障害物有り、0 は障害物無しを意味している。なお、スタート地点は左上隅、ゴール地点は右下隅に固定するものとし、入力データとしては与えないものとする。

一般的にマップの入力データを記載すると

$$
\begin{array}{cccc}
w & h & & \\
m_{1,1} & m_{1,2} & \cdots & m_{1,w} \\
m_{2,1} & m_{2,2} & \cdots & m_{2,w} \\
 & \cdots & & \\
m_{h,1} & m_{h,2} & \cdots & m_{h,w}
\end{array}
$$

の形である。ここで、w はマップの幅、h はマップの高さ、$m_{i,j}$ はマップの i 行 j 列の位置のデータで 1 は障害物があることを表し 0 は障害物が無い事を表している。

内部データ

このようなマップのデータは、プログラム内では２次元配列を用いて表現できる。例えば、図 37 に示す例の場合、整数型の２次元配列 `map[3][5]` を用意し、$m_{i,j}$(あるいは $-m_{i,j}$) を `map[i-1][j-1]` に格納する方法が考えられる。この場合、障害物を避けてゴールに至る経路を探す際に、マップの領域からはみでないように注意する必要がある[65]。

「マップの外へは移動できない」ことを表すためには、このマップの周囲は全て障害物で囲まれていると考えれば取扱が容易になる (図 38 参照)。例えば、この例の場合、整数型の２次元配

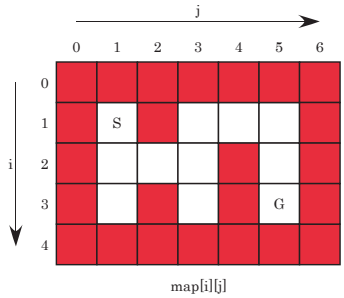


図 38: 周囲を障害物で囲った迷路 1 のマップ

列 `map[5][7]` を用い、`i` 行 `j` 列の位置が障害物であれば`map[i][j] = -1` とし、障害物でなければ `map[i][j] = 0` とすることで、表現出来る。この例では、`(i,j)`$\in$`{(1,1), (1,3), (1,4), (1,5), (2,1), (2,2), (2,3), (2,5), (3,1), (3,3), (3,5)}` に対して`map[i][j] = 0` であり、それ以外の `(i,j)` (但し $0 \leq$ `i` ≤ 4, $0 \leq$ `j` ≤ 6) に対して `map[i][j] = -1` である。前述の**入力データ**の形でデータが与えられる場合、`map[`i`][`j`]` に $-m_{i,j}$ を格納し、`map[0][j] = map[h+1][j] = -1` ($0 \leq j \leq w+1$)、`map[i][0] = map[i][w+1] = -1` ($0 \leq i \leq h+1$) とすれば良い[66]。

プログラム例 27 は、与えられた**入力データ**に基づいて、マップの周囲を障害物で囲ったうえでそれを表示するプログラムである[67]。このプログラムで扱うことが出来る最大の幅と高さは記号定数 `MAXWH` で指定しており、その値は 100 と定義されている。入力データは、`scanf` を用いて標準入力 (キーボード) から読み込むように作られている。したがって、入力データをファイル `mapdata.txt` に保存[68]しておくと、入力のリダイレクション機能を用いて、`./showmapg < mapdata.txt` のように実行すればキーボードから入力データを打ち込む必要は無い。

プログラム例 27

```
// display maze by hgcc
// Filename = showmapg.c (2015.07.01 version)
// Compile : hgcc -o showmapg showmapg.c
// Execution : ./showmapg < mapdata.txt

#include <stdio.h>
#include <handy.h>
```

[65] マップの領域からはみ出た場所のデータにアクセスすると segmentation fault 等のエラーになる可能性が高い。
[66] ここでは、後述 (11.2 節) の説明の都合上、内部データとして-1 が障害物を表すものとしている。
[67] マップの表示には HandyGraphic を利用
[68] ファイルは emacs で作成すること。「テキストエディット」や「NeoOffice」や「LibreOffice」等で作成すると余分な制御コードが含まれ scanf では正常に読み込めない場合がある。

```
#ifdef HgSetFillColor
#define MyFillColor(x) HgSetFillColor((x))  // for HG_VERSION 0.6
#else
#define MyFillColor(x) HgSetPaintColor((x)) // for HG_VERSION 0.5
#endif

#define MAXWH 100          // マップの幅と高さのとして最大 100 まで可能とする
#define MAXBLOCKSIZE 64    // マップの１ブロックの大きさの最大値
#define MAXWINDOWSIZE 700  // マップを表示するウィンドウの大きさの最大値
int map[MAXWH+2][MAXWH+2]; // マップの周囲を障害物で囲むため幅、高さともに２増やす
int w,h; // マップの幅と高さ (周囲は除く)

int main()
{
  int i,j, d,size, maxwh, ret;

  scanf("%d %d", &w, &h); // マップの幅と高さを入力
  for(i=1; i<=h; i++)
    for(j=1; j<=w; j++){
      scanf("%d", &d);    // i 行 j 列の位置のデータを読み込む
      map[i][j] = -d;     // 配列には、障害物を-1 で記憶
    }
  // マップの周囲を障害物で囲む
  for(j=0; j<=w+1; j++)
    map[0][j] = map[h+1][j] = -1; // 上側と下側
  for(i=0; i<=h+1; i++)
    map[i][0] = map[i][w+1] = -1; // 左側と右側
  maxwh = (w > h ? w : h);        // 幅と高さのうち大きい方を maxwh とする
  size = MAXWINDOWSIZE/(maxwh+2); // 周囲を障害物で囲むので maxwh+2 で割る
  if(size > MAXBLOCKSIZE) size = MAXBLOCKSIZE; // マップの１ブロックの大きさを決定
  // マップの周囲を障害物で囲むことを考慮してウィンドウを開く
  HgOpen(size*(w+2), size*(h+2));
  HgSetColor(HG_BLACK);
  HgSetTitle("w=%d h=%d", w, h);  // ウィンドウのタイトルとして幅と高さを表示
  for(i=0; i<=h+1; i++){
    for(j=0; j<=w+1; j++){
      if(map[i][j]) MyFillColor(HG_RED); // 障害物は赤で表示
      else MyFillColor(HG_WHITE);        // 障害物が無いところは白で表示
      HgBoxFill(size*j, size*(h+2-i-1), size,size,0);
    }
  }
  HgSetColor(HG_CYAN); // ブロックの境界が分かるようにシアン色の線を引く
  for(i=0; i<=h+2; i++)
    HgLine(0, i*size, (w+2)*size, i*size); // 水平線
  for(j=0; j<=w+2; j++)
    HgLine(j*size, 0, j*size, (h+2)*size); // 垂直線
  HgGetChar(); // 何か入力されるまでウィンドウを表示
  HgClose();
  return 0;
}
```

#ifdef から#endif の部分は、HandyGraphic のバージョンが 0.5X でも 0.6X でも本プログラムを使用出来るようにするためのものである。式「w > h ? w : h」の値は、条件「w > h」が成立するならば値「w」となり、不成立なら値「h」となる[69]。また、if 文は、一般に「if (条件) 式;」

[69]一般に、「条件 ? 式 1 : 式 2」の形で用いられ、「条件」が成立する時は「式 1」の値、不成立の時は「式 2」の値となる。

の形で記述され、「条件」が成立した時に「式」が実行されるが、「条件」の部分には任意の式[70]を記述することが出来、式の値が 0 の場合は条件不成立、それ以外は条件成立と判定される。

11.1.2 経路の探索

S から G へ移動できるかどうかを探索するには、S から障害物を避けながら移動できる場所を深さ優先探索で探索すれば良い。移動できる場所を探す際に、例えば、右→下→左→上の順に探すものとする。

まず (i,j)=(1,1) とする (S)(スタートから探索開始)(①)。右隣り (1,2) は障害物 (②) なので、下へ移動する (③(2,1))。

マス目 (2,1) から右隣り (2,2) へ移動 (④) し、さらに右へ移動 (⑤(2,3)) へ移動する。

マス目 (2,3) の右隣 (2,4) は障害物 (⑥) なので、下へ移動する (⑦(3,3))。

マス目 (3,3) では、右隣 (⑧(3,4))、下隣 (⑨(4,3))、左隣 (⑩(3,2)) は障害物であり、移動できるのは上隣 (⑪(2,3)) であるが、マス目 (2,3) は既に訪問 (⑤) しているため、そこから先の探索は行わない[71]。マス目 (3,3) に隣接するマス目を全てチェックしたので、一つ戻ってマス目 (2,3)(⑤) から探索を継続する。

マス目 (2,3) では、その右と下は既にチェック済みなので、左隣 (⑫(2,2)) をチェックするが、マス目 (2,2) は既に訪問済み (④) なので、そこから先の探索は行わず、上へ移動 (⑬(1,3)) して探索を続ける。

このような探索を続けると、S から G へ到達可能な場合は、やがて G(⑲(3,5)) に達することになる。

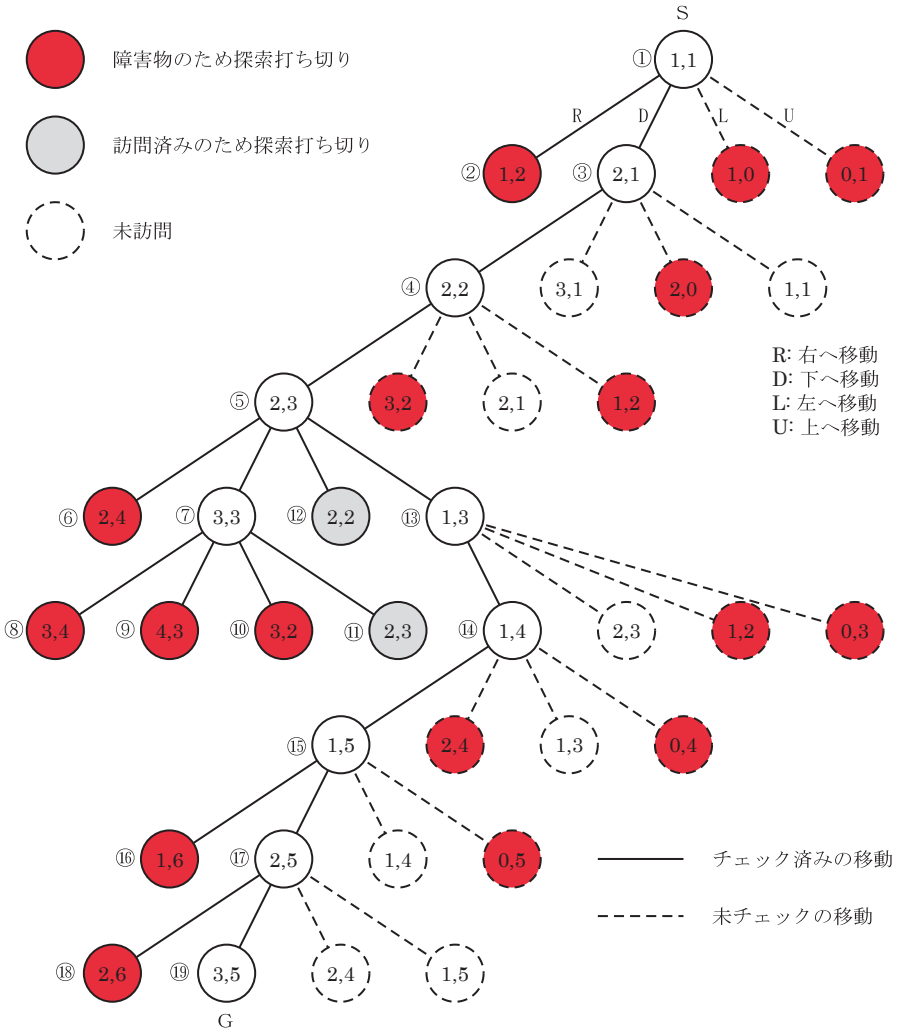
上記で説明した探索方法は、**深さ優先探索 (depth first search)** と呼ばれる。深さ優先探索では、既に訪問済みのマス目を再度訪問した場合、その先への探索を打ち切り戻る必要がある。既に訪問済みかどうかを判定するには、初めてマス目に到着した際にそのマス目に印をつけておく (例えば、map[i][j] = 1;) ことにより、再度の訪問かどうかを判定することが出来る。

図 41 は、図 40 に示すマップに対する深さ優先探索を表す。図 41 では、a → b →· · · · → y の順に探索が行われるが、探索しても G((3,5)) は現れないため S から G へは到達不能である。

深さ優先探索は再帰的な関数を使うことにより、容易に実現できる。その概略を以下に示す。main 関数から dfs(1,1) を呼び出すと、到達可能な場合は dfs 関数の中で「到達可能」と出力されプログラムが終了するが、到達不能な場合はプログラムが終了することなく main 関数に戻ってくるので、その時点で「到達不能」と出力してプログラムを終了すればよい。

```
void dfs(int i, int j)
{   もしマス目 (i,j) がゴール (G) ならば、「到達可能」と出力して終了。
    map[i][j] が-1(障害物) なら戻る。
    map[i][j] が 1(マス目 (i,j) は訪問済み) なら戻る。
    マス目 (i,j) に訪問済みの印をつける (map[i][j] = 1;)。
    // 右、下、左、上の順 (別の順でも良い) に dfs を再帰的に呼び出す。即ち
    dfs(i, j+1); // 右
    dfs(i+1, j); // 下
    dfs(i, j-1); // 左
    dfs(i-1, j); // 上
}
```

[70] 変数のみも式である。
[71] 探索を続けるとループになり探索が終了しない。



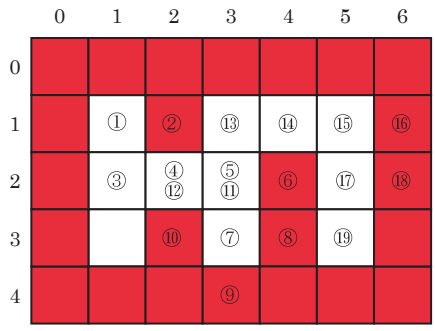


図 39: 迷路 1 の深さ優先探索

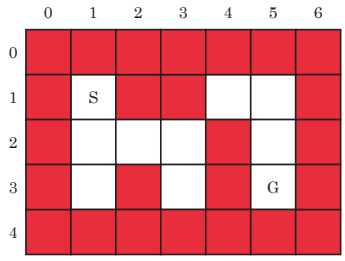


図 40: 到達不能なマップ (迷路 2)

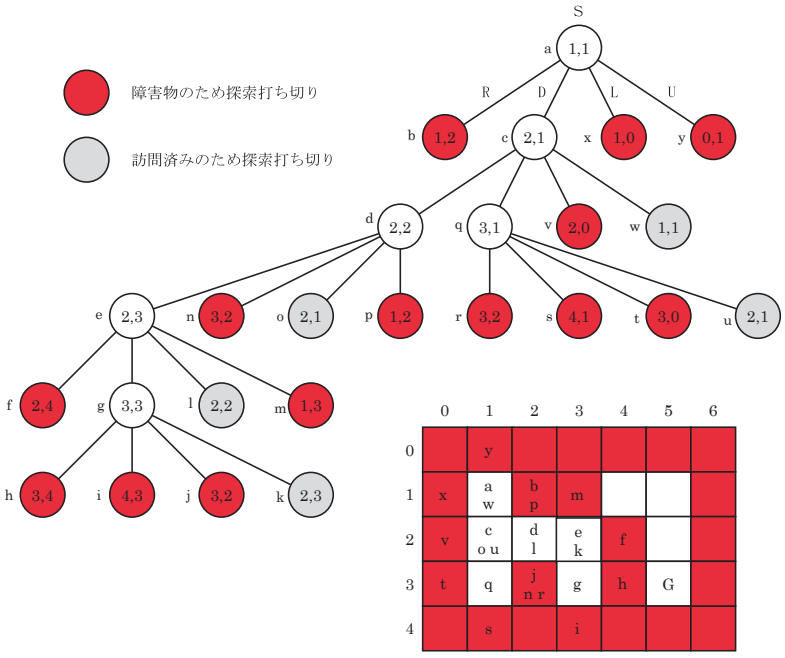


図 41: 到達不能な場合 (迷路 2) の深さ優先探索

演習 30 到達可能性を判定するプログラムを作成し、マップを表す入力データ m01.txt〜m09.txt に適用せよ[72]。なおいずれのマップもその幅 (w) と高さ (h) は 100 以下であり、スタート地点はマップの左上隅、ゴール地点はマップの右下隅とする。

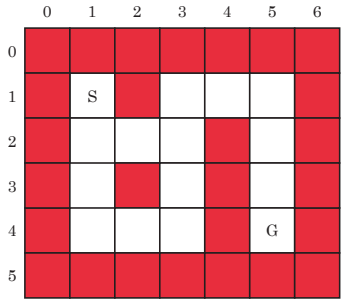


図 42: ループを含むマップ (迷路 3)

ヒント 入力データの読み込みは、プログラム例 27 において入力データから 2 次元配列 `map` を構築する部分を参照。マップの表示は不要なので、`HandyGraphic` を使う必要は無い。

11.2 最短経路長と最短経路

前節では、到達可能性の判定を行うだけであったが、最短経路を求めるには、図 43 に示すように、

- まず、スタート地点のマス目にラベル「1」をつける。
- ラベル「1」が付いたマス目に隣接するマス目に対して、障害物ではなく、かつ、まだラベルが付いていなければ、ラベル「2」を付ける。
- ラベル「2」が付いたマス目に隣接するマス目に対して、障害物ではなく、かつ、まだラベルが付いていなければ、ラベル「3」を付ける。
- ····

のように繰り返しラベル付けを行えばよい。最初にゴール G に達した時点で、G に付いたラベルが、最短経路長となる[73]。G が出てこず、ラベルが付けられなくなった場合は、S から G へ到達できないことを意味している。

図 44 にこのような探索の様子を示す。まずスタート地点 S であるマス目 (1,1) にラベル「1」を付ける (1A)。マス目 (1,1) に対して、例えば、右→下→左→上の順に、障害物ではなくまだ最短経路が求まっていない (ラベルが付いていない) かをチェックする。

右側 (1,2) は障害物なので何もしない (2A)。下側 (2,1) は今回初めて訪問する (障害物ではなくラベルが付いていない) のでラベル「2」[74]をつける (2B)。左側 (1,0) と上側 (0,1) は障害物なので何もしない (2C, 2D)。

[72] m01.txt〜m04.txt は迷路 1〜迷路 4 (図 38, 図 40, 図 42, 図 43 参照) を表す入力データである

[73] ここでは、経路上のマス目の個数を経路長と定義する。マス目に付けるラベルがスタート地点からの最短経路長になっている

[74] このマス目はラベル「1」の付いたマス目 (1,1) から進んできたマス目なので元のマス目 (1,1) のラベルに 1 足した値を個のマス目のラベルとする。

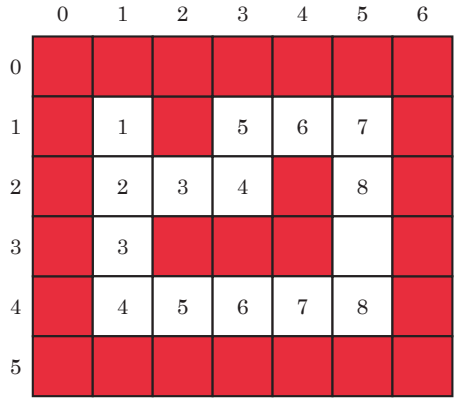


図 43: 最短経路 (迷路 4)

これでラベル「1」がついたマス目は全てチェック済みなので、次にラベル「2」が付いたマス目 (2,1) をチェックする。右側 (2,2) と下側 (3,1) は今回初めて訪問する (障害物ではなくラベルが付いていない) のでラベル「3」[75]付ける (3A, 3B)。左側 (2,0) は障害物なので何もしない (3C)。上側 (1,1) は既にラベル「1」が付いている (即ちスタート地点からの最短経路長として 1 が求まっている) ので何もしない (3D)。

これでラベル「2」がついたマス目は全てチェック済みなので、次にラベル「3」が付いたマス目をチェックする。ラベル「3」が付いたマス目は (2,2) と (3,1) の二つがあるが、先にラベル付けされたマス目 (2,2) からチェックする。右側 (2,3) は今回初めて訪問するのでラベル「4」を付ける (4A)。下側 (3,2) は障害物なので何もしない (4B)。左側 (2,1) は既にラベル付けされているので何もしない (4C)。上側 (1,2) は障害物なので何もしない (4D)。

次にラベル「3」が付いたもう一つのマス目 (3,1) をチェックする。右側 (3,2) は障害物なので何もしない (4E)。下側 (4,1) は今回初めて訪問するのでラベル「4」を付ける (4F)。左側 (3,0) は障害物なので何もしない (4G)。上側 (2,1) は既にラベルが付いているので何もしない (4H)。

以下同様に、5A → 5H, 6A → 6H, 7A → 7H, 8A → 8E の順に繰り返す。8E でゴール地点 G(4,5) にラベル「8」が付けられたので、S から G への最短経路長は 8 となる。

このような探索方法は**幅優先探索 (breadth first search)** と呼ばれる。

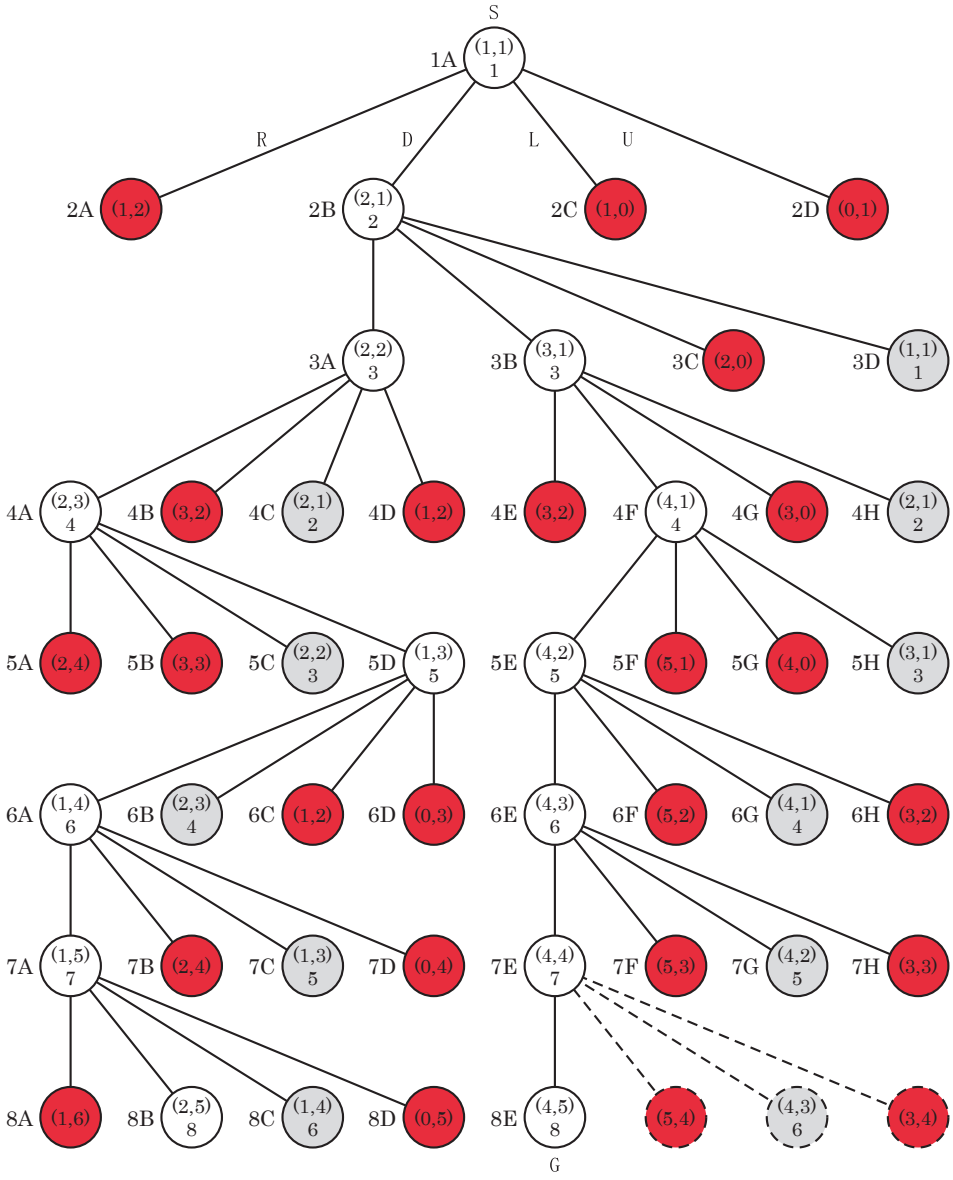
最短経路を求めるには、G から S に向かって、ラベルの値が一つずつ減少するようにマス目をたどれば良い。図 44 の例では、G=$(4,5)^{(8E)}$ ← $(4,4)^{(7E)}$ ← $(4,3)^{(6E)}$ ← $(4,2)^{(5E)}$ ← $(4,1)^{(4F)}$ ← $(3,1)^{(3B)}$ ← $(2,1)^{(2B)}$ ← S=$(1,1)^{(1A)}$ が最短経路となる[76]。

なお、幅優先探索において、新たなラベル付けが出来なくなるまで探索してもゴール地点にラベルが付かなければ、スタート地点 S からゴール地点 G に至る経路が存在しないことを表している。

幅優先探索は、**待ち行列 (キュー, queue)**[77]を用いることにより、容易に実現出来る。待ち行列では、データの追加は行列の最後尾に行われ、待ち行列からのデータの取り出しと処理は、行列の先頭から行われる (図 45 参照)。

待ち行列を用いた幅優先探索の概略を以下に示す。関数 bfs を呼び出した結果、ゴール地点のラベル (map[h][w]) が 0 なら到達不能であり、0 以外ならその値がスタート地点からゴール地点に至る最短経路長となる。

[75] これらのマス目は、ラベル「2」が付いたマス目 (2,1) から進んできたマス目なので 1 を足して「3」をラベルとする。
[76] 最短経路が複数存在する場合はこのアルゴリズムでその内の一つが求まる。
[77] FIFO (first-in-first-out) とも呼ばれる。



	0	1	2	3	4	5	6
0		2D		6D	7D	8D	
1	2C	1A 3D	2A 4D 6C	5D 7C	6A 8C	7A	8A
2	3C	2B 4C 4H	3A 5C	4A 6B	5A 7B	8B	
3	4G	3B 5H	4B 4E 6H	5B 7H			
4	5G	4F 6G	5E 7G	6E	7E	8E	
5		5F	6F	7F			

障害物のため探索打ち切り

訪問済みのため探索打ち切り

未訪問

図 44: 迷路 4 の幅優先探索

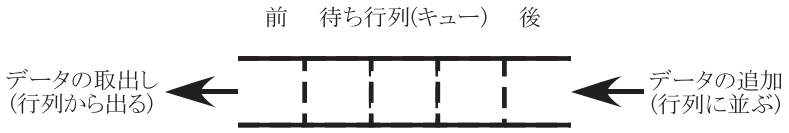


図 45: 待ち行列 (キュー)

```
void bfs( )
{   スタート地点 (1,1) に対し、map[1][1]=1 とし、(1,1) を待ち行列に入れる。
    待ち行列が空で無い間 {
        待ち行列の先頭よりデータを取り出し、そのデータを (i,j) とする。
        マス目 (i,j) に隣接する (右、下、左、上) 各マス目 (i',j') に対して {
            マス目 (i',j') が障害物で無くまだラベルが付いていない (map[i'][j']==0) なら
            map[i'][j'] = map[i][j]+1 とし、(i',j') を待ち行列に入れる。
            マス目 (i',j') がゴール地点なら戻る。
        }
    }
}
```

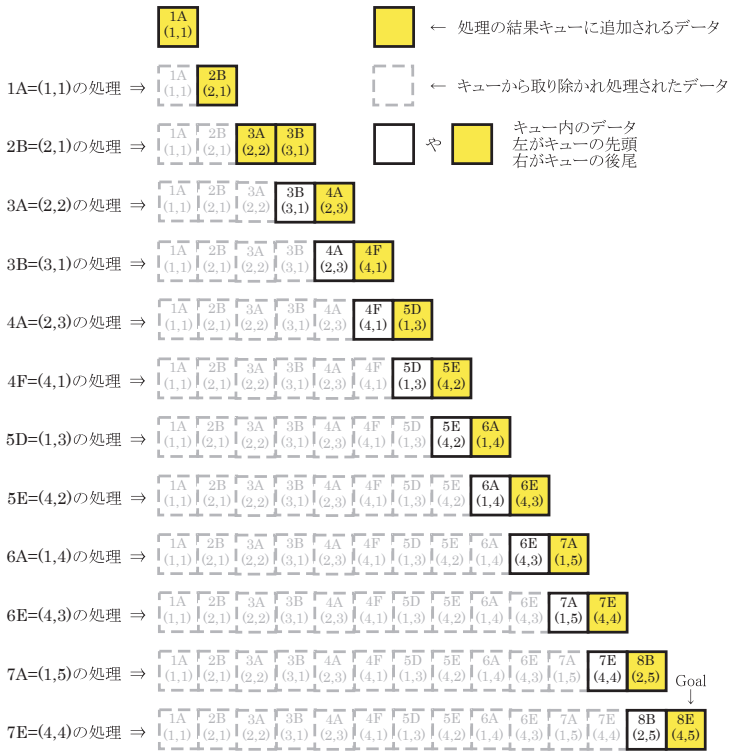


図 46: 幅優先探索時のキューのデータ

図 44 に示す例の場合、待ち行列に対するデータの追加と取り出しは次のように進んでいく (図 46 参照)。

1. (1,1) を待ち行列に追加。
2. 待ち行列よりデータを取り出す。取り出したデータは (1,1)。
3. (1,1) の隣接位置で障害物ではなくラベルの付いていないマス目 (2,1) を待ち行列に追加。

4. 待ち行列よりデータを取り出す。取り出したデータは (2,1)。
5. (2,1) の隣接位置で障害物ではなくラベルの付いていないマス目 (2,2) と (3,1) をこの順に待ち行列に追加。
6. 待ち行列よりデータを取り出す。取り出したデータは (2,2)。
7. (2,2) の隣接位置で障害物ではなくラベルの付いていないマス目 (2,3) を待ち行列に追加。
8. 待ち行列よりデータを取り出す。取り出したデータは (3,1)。
9. (3,1) の隣接位置で障害物ではなくラベルの付いていないマス目 (4,1) を待ち行列に追加。
10. 待ち行列よりデータを取り出す。取り出したデータは (2,3)。
11. (2,3) の隣接位置で障害物ではなくラベルの付いていないマス目 (1,3) を待ち行列に追加。
12. 待ち行列よりデータを取り出す。取り出したデータは (4,1)。
13. (4,1) の隣接位置で障害物ではなくラベルの付いていないマス目 (4,2) を待ち行列に追加。
14. ···

演習 31 最短経路とその長さを求めるプログラムを作成し、演習 30 で用いたマップを表す入力データ m01.txt〜m09.txt に適用せよ。

ヒント 待ち行列は、行列に入る可能性がある最大のデータ数が分からない場合は、線形リストを用いて実現することが多い。しかしながら、ここで取り扱っている問題の場合、迷路の大きさとして、高さと幅の最大値は 100 としているので、待ち行列に入るマス目の個数は $100 \times 100 = 10000$ 以下となる。このような場合は、配列を用いて以下のように簡単に実現できる[78]。

```
struct position {   // マス目の位置を示す構造体
    int raw;        // 行位置
    int column;     // 列位置
};
#define QSIZE 10000                 // 待ち行列の大きさ
struct position queue[QSIZE];       // 待ち行列
int front=0;                        // 待ち行列の先頭位置 (データを取り出す位置)
int rear=0;                         // 待ち行列の最後尾 (データを追加する位置)

void enqueue(struct position data)      // 待ち行列に data を追加。
{
    queue[rear++] = data;           // 最後尾に追加。
}

struct position dequeue(void)       // 待ち行列からデータを取り出し、そのデータの値を返す。
{
    return queue[front++];          // 先頭のデータを取り除いて返す。
}

int is_empty(void)                  // 待ち行列が空なら 1, 空でなければ 0 を返す。
{                                   // front と rear が一致していれば待ち行列は空。
    return (front == rear);         // 条件式の値は、条件成立ならば 1, 不成立なら 0。
}
```

演習 32 演習 31 において、結果を図 47 や図 48 のように表示するプログラムを作成せよ。[79]

[78] 以下の実現例では、待ち行列のオーバーフローや空の待ち行列からデータを取り出そうとしていないかなどのエラーチェックは行っていない。
[79] 挑戦課題です。

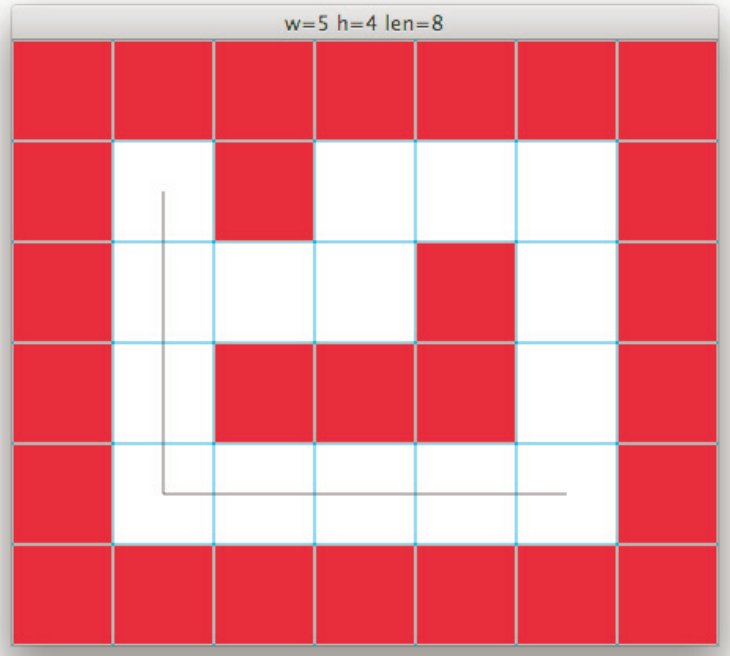


図 47: 最短経路の表示例

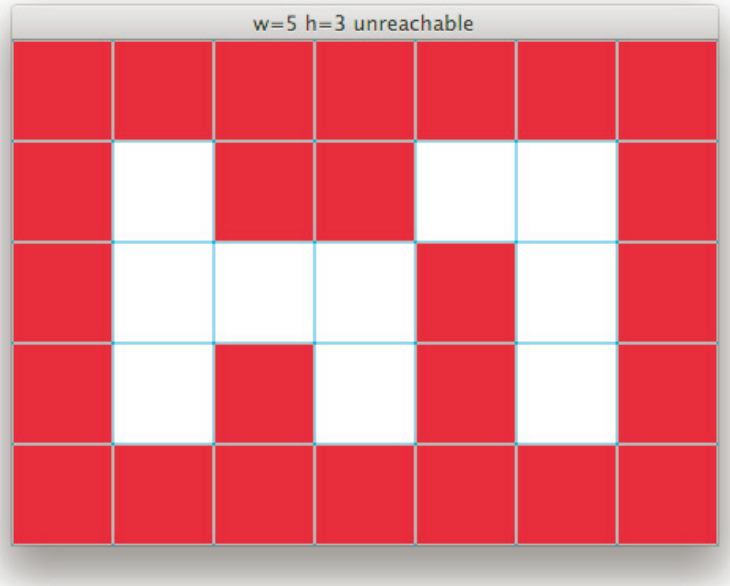


図 48: 到達不能の場合の表示例