

マイクロマウス合宿 2022 「デバッグ手法 & 探索アルゴリズム」紹介

2022.6.26

Mice Busters
なおフィス@starknighthood

本講義について

- 目的
 - デバッグ手法の種類とログ解析を理解
 - マイクロマウス向け探索アルゴリズムの基礎を理解
- 目標
 - バグがあると気づいたとき、どの手順でデバッグすればよいか、方針を立てられるようになる
 - ゴールできる探索アルゴリズムの構成要素を理解する

デバッグ手法について

【モチベーション】

人が作り上げる以上、ロジックの中にバグや不具合が「必ず」存在します。
自分が作り込んだバグは、他人には容易に解決できません。
デバッグ手法に身につけ、自分で解決できるようになりましょう。

目次（デバッグ手法）

- マイクロマウスで使えるデバッグ手法
 - 一覧紹介
 - 各手法のユースケース
 - 比較
- ログの解析について
 - 実例、手順紹介
- ログの工夫ポイント

デバッグ手法の一覧

マイクロマウスで実現可能なデバッグ手法を5つに分類

| No. | 種類 | | 概要 | メリット | デメリット |
|-----|--------|-----|--|--|--|
| 1 | I/O | LED | 特定のロジックを通過した瞬間にLEDを光らせる | ・ CPUの処理リソースを使わない | ・ 表現力に乏しい ・ 手元がないと確認できない |
| | | ブザー | 特定のロジックを通過した瞬間にブザーを鳴らす | ・ 音階の種類だけ表現力がある ・ 離れた位置から確認ができる | ・ ある程度、連続して音を鳴らさないと聞こえない。人が知覚できるまでに時間がかかる |
| 2 | printf | | 実行時にUARTなどで文字列をコンソールに送り、文字列を確認する | ・ 文字列に計算結果など、具体的な情報を表せる | ・ UARTなど、マイコンの機能を使うため、処理コストがかかる ・ コンソールに接続しないと確認できない。リアルタイム性が低い |
| 3 | デバッガー | | 専用の端末を用いて、マイコンの内部状態を観測する | ・ メモリ配置情報、確保状態など、プログラムの実行中の詳細な状態を確認できる | ・ 専用の端末が必要 ・ プログラムを止めながらの確認になるため、連続した処理の確認に向き |
| 4 | ログ解析 | | マイコンの内部メモリに確認したいデータを格納し、あとでprintfで確認する | ・ 所定のデータを後で送信できるため、実行負荷などの制約がない | ・ ログに入れることができるデータ量はマイコンのRAMの量によって決まる |
| 5 | 動画解析 | | カメラを用いて撮影 (スマホで十分。スロー再生も有効) | ・ 他のデバッグ手法との組み合わせが可能 | ・ 数値化が難しい |

各デバッグ手法のユースケース

機体完成から、大会の競技中まで、各デバッグ手法の利用タイミングを整理

| 工程 | 詳細 | I/O | printf | デバッガ | ログ | 動画 | 備考 |
|----------|----------------|-----|--------|------|----|----|---|
| 完成後の動作確認 | I/O動作 | ○ | | ○ | | | 完成直後は デバッガー役に立つ |
| | 通信機能確認(UART) | | ○ | ○ | | | |
| | 通信機能確認(SPIなど) | | ○ | ○ | | | |
| 基本動作作成 | UI | ○ | ○ | ○ | | | 時系列情報の確認になる ためコンソールも必要 |
| | センシング | △ | ○ | | ○ | | |
| | 行動計画（物理量計算） | △ | | | ○ | | |
| | モーション（直進、ターン等） | △ | | | ○ | ○ | これらの繰り返しで完成 度を上げるため、確認方 法の高機能化は重要 |
| 競技用機能作成 | 探索アルゴリズム | △ | ○ | △ | ○ | ○ | |
| | 最短経路導出 | △ | ○ | ○ | ○ | | |
| | 最短経路走行 | △ | | | ○ | ○ | |
| 大会中 | 走行前準備 | ○ | | | | | |
| | 走行中 | ○ | | | △ | ○ | |

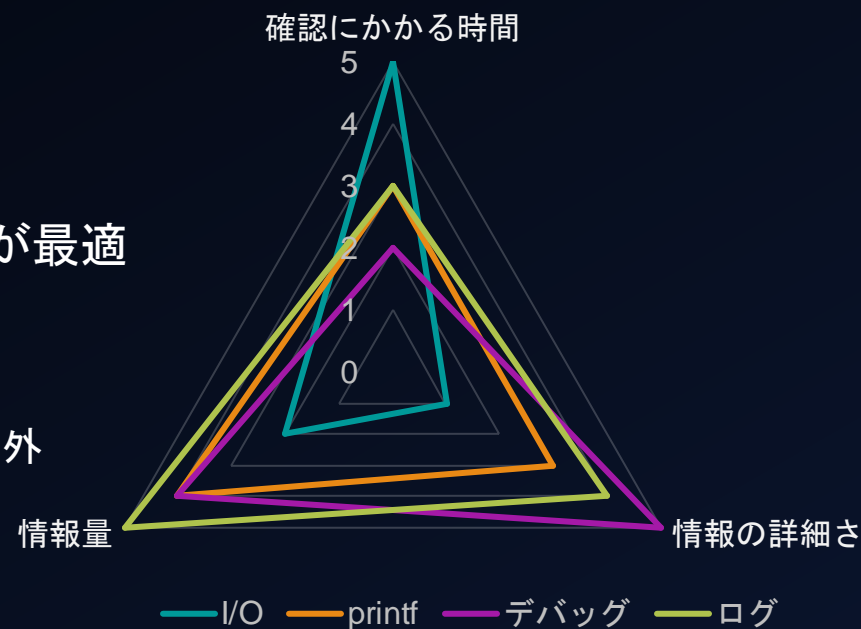
※探索アルゴリズムなどはPC上で実装・シミュレーションしてからマイコンに移植するのも有効

デバッグ手法の比較

- 見解
 - I/Oなら大会の競技中も有効
 - デバッガは詳細な情報が得られるが、時間がかかる
 - 時系列情報は量が多いため確認するには、ログ解析が最適

※ 比較結果は所感によるもの。

※ 「動画解析」は単純比較すべきでないと判断したため、除外



今回、「ログ解析」について詳しく紹介

ログ解析の概要

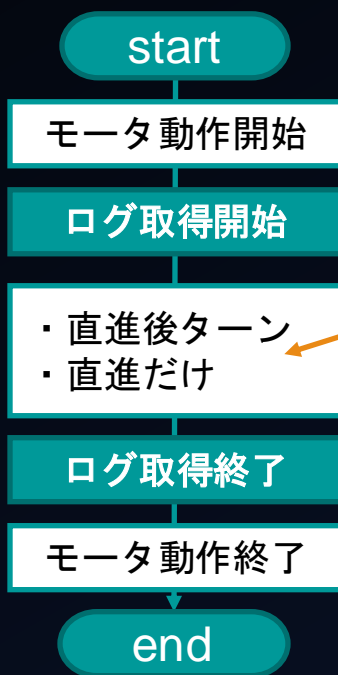
- マイコン内の記憶領域に時系列データを格納し、あとでデータを見る手法
 - 指令値： 速度、加速度、角速度、角加速度、Duty など、算出した指示結果
 - 状態： 実速度（左/右）、角度、角速度、バッテリー電圧 など、計測した機体の状態
 - センシング結果： 壁センサー（生値/距離変換後） など、計測した環境情報
 - 内部状態： 指示状態やモード（直進、ターン中など） など、指示中の状態
- 時系列データの溜め込む先
 - RAM： 書き込み速度を気にする必要がない。リセット後もデータは消える。
 - ROM/FLASH： 書き込み手続きが必要。時間がかかるが、リセット後もデータが残る。
 - 書き込めるデータの量： RAM < FLASH < ROM （だいたいこんな感じ）
- 確認方法
 - printfで解析環境にデータを移し、ツールなどで解析することになる。

ログ解析手順の紹介

手順は3工程

- ① ログデータ収録 . . . 動作中のデータをメモリに格納
- ② ログデータの出力 . . . 収録したデータを確認しやすい方法で出力
- ③ ログ情報の描画 . . . グラフ描画可能なツールにデータをコピーしグラフを描画

① 収録



テストシナリオ

収録先のデータ領域
float log_v[1000];
float log_v_enc_r[1000];
float log_v_enc_l[1000];
float log_w[1000];
int log_sen_r45[1000];
int log_sen_l45[1000];

② 出力

```
// csv形式で出力
for( int i=0;i<1000;i++){
    float sen_r45_dist = convert(log_sen_r45[i], R45);
    float sen_l45_dist = convert(log_sen_l45[i], L45);
    printf("%d, %0.2f, %0.2f, ", i, v, w, ~~省略~~);
}
```

生値と距離のように後で変換できるデータは後で変換することで、データ量の節約になる

```
// コンソール
0, 0, 0, 10~~~
1, 0.1, 0, 11,~~~
. . . .
```

teratermなどで出力

excelなどcsvを解釈できるソフトにコピーしてグラフ描画

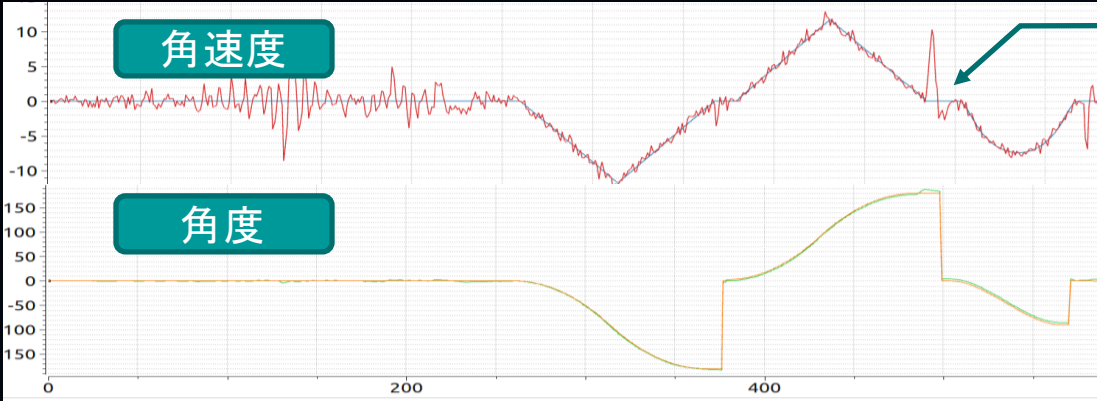
③ 描画

| excel | | | | |
|-------|------|------|----------|---------|
| index | v | w | l45_dist | l45_raw |
| 0 | 0.00 | 0.00 | 45.00 | 250 |
| 1 | 0.10 | 0.00 | 44.86 | 254 |

【実例紹介】 PlotJugglerの描画紹介



ログ解析の実例紹介



これについて考える

上図の不具合に対し、以下の手順で原因を探る

| No. | 手順 | 結果 | 備考 |
|-----|---------------------------|--|--------------------------------|
| 1 | 観測した事象 | ターンの直後に左に動こうとする | ・動画を撮ると確度高く確認できる |
| 2 | 仮説 | ターン完了時点で理想位置から右にずれているため、壁から離れようとする制御によって左に動こうとした | ・データからどのような現象が起きているのか推測する技術が必要 |
| 3 | 仕様の確認 | 「右にずれる」に関わる機能・仕様として ・壁との距離を見てターンの軌道を補正する機能がある | どの機能が悪さをしているのか、見当をつける |
| 4 | ・事実確認、原因の特定 ・次の調査対象の選定 | ・補正の計算式に誤りがあった ⇒ 補正式を直す（完） ・誤りがなかった ⇒ 5へ | 誤りかどうかは自分で設計した仕様と照合する |
| 5 | 前提機能の特定 | 光センサーについて、1~4を再度実施し、原因を絞り込む | 基本これの繰り返し。 |

ログ周りで工夫するポイント

ログ解析では以下の工夫できる点がある。

①収録（実質、メモリ量のやりくり、節約）

- ・収録周期を4周に1回にする（間引き）
- ・収録するデータを圧縮する。（上級者向け）
 - ・浮動小数点を 単精度⇒半精度 に変換しログに格納。出力時に元に戻す。
- ・時間とデータ総量のバランス（短くリッチに or 長くほどほどに）

②ログ送信

- ・printfの送信速度が遅いと、出力に時間がかかるため、なるべく早くする。（2Mbpsとか）
- ・受信ソフトをpythonなどで自前で用意できる場合、「送信開始/終了」の目印を送れば、都度、ログをファイルに書き込むなど、格納ロジックの起点にできる。（自動化）

③ログ描画

- ・ログ描画の専用ソフトを使う . . . matlab、python、plotjuggler、excel

※エクセルの場合、描画範囲の指定など、不器用なため、プログラマブルなものが良い。

ここまでのまとめ

- デバッグ手法
 - メリット・デメリットを把握し、シーンに応じて使い分けよう
- ログによるデバッグ
 - 溜める、出力する、描画する 3工程が必要
 - 自分に合った可視化・グラフ化ツールを検討しよう （まずはエクセルから）
 - 原因の特定には、 事象の観測、仮説を立てる、仕様確認、事実確認 を繰り返そう



質疑応答

探索アルゴリズム紹介

【モチベーション】

マイクロマウスは、ゴールできないと、記録に残すことすらできません。
加えて、限られた競技時間、限られた処理リソースなどの制約があることから、
「単にゴールできる」だけでなく、
「効率よく探索するアルゴリズム」を「軽い処理」で実現する必要があります。

目次（探索アルゴリズム）

- 探索アルゴリズム、経路導出アルゴリズム一覧
- 足立法の概要、構成要素
 - 壁情報の管理
 - 歩数マップ生成方法
 - 計算コストを加味した歩数マップ生成

迷路探索アルゴリズムについて

以下、マイクロマウスでよく聞く探索アルゴリズム、および、経路導出アルゴリズム

| No. | 種類 | 概要 | メリット | デメリット |
|-----|-----------------------|---|--|---------------------------------|
| 1 | 左/右手法 | 左(右)壁に沿って進む | ・ アルゴリズムが単純 | ・ ゴールがスタート地点と壁が重なってないとゴールできない |
| 2 | 拡張左手法 (トレモー法) | 探索済みの区画にもう1度来たときに、左ではなく、直進、右折を選択する | ・ 単純なアルゴリズムで完走できる | ・ いずれはゴールにたどり着くが、効率は考えられてない。 |
| 3 | 足立法 ※マイクロマウス独自の探索法 | ゴールまでの歩数を数え、その数が小さくなるように進む。 | ・ 認識結果と移動制御に間違いがなければ、絶対にゴールできる | ・ 32x32向けには工夫が必要 (※原因について後述) |
| 4 | ダイクストラ | 分岐点から収束点までの経路を重みで評価し最短経路を選ぶ | 最短経路導出問題向けの話なので割愛。(面倒) エッセンスが足立法に全部内包されてる。 探索アルゴリズムが理解できれば、最短経路も出せる。 ※ ただし、最短＝最速とは限らない。 | |
| 5 | A*(A-star) | ダイクストラ法の発展版 マイクロマウスの場合、コスト計算が違うだけで、足立法と本質的には同じ考え | | |

今回、マイクロマウスで用いられる「足立法」に必須な要素について解説。

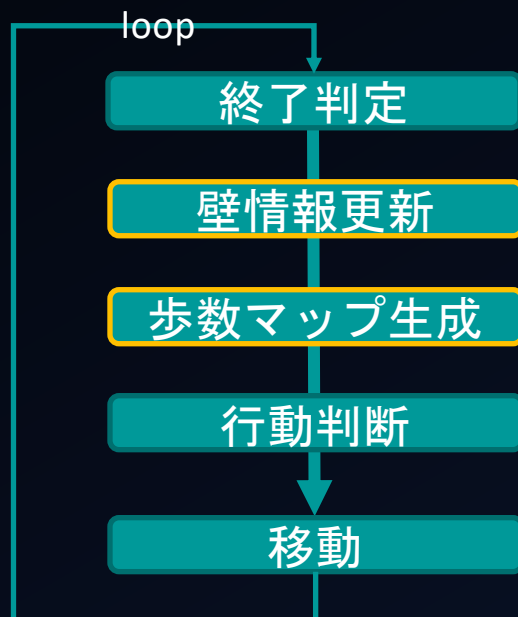
参考 : https://jsdkk.com/home/glossary/glossary-micromouse/#micromouse4_5
<https://www.rt-shop.jp/blog/archives/3692>

足立法の概要と構成要素

◆概要

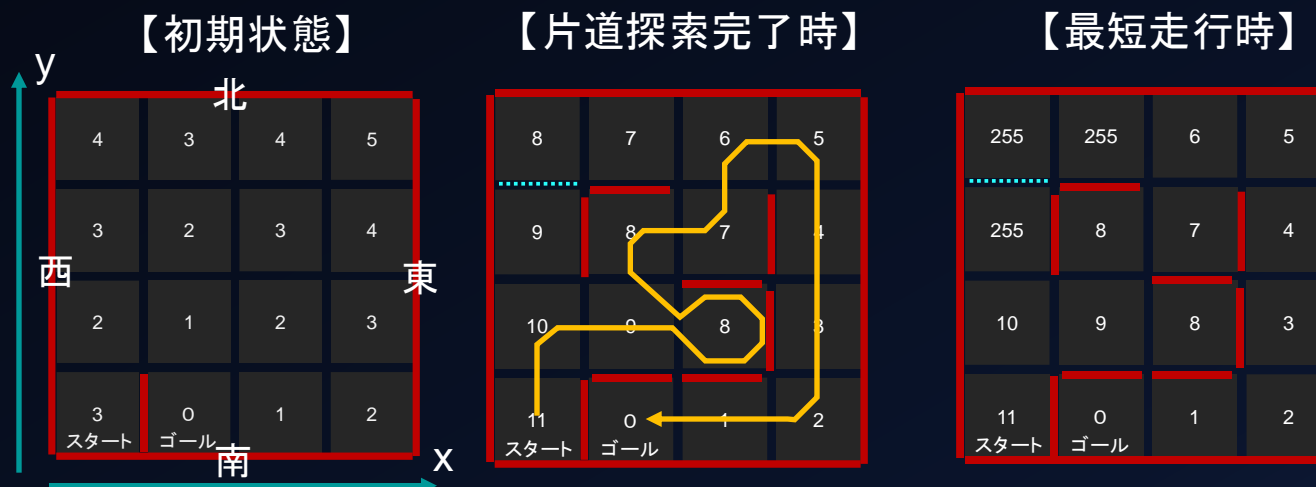
- ゴールまでの距離関係を表す「歩数マップ」を生成し、自分の現在位置から、ゴールまでの距離が短くなる区画に移動するアルゴリズム
- 移動先の区画で得られた「新たな壁情報」をもとに、都度、歩数マップを生成し直すことで、「近づきながら探索する」ことができる。

◆足立法の構成要素



◆歩数マップの例

..... 未探索かつ、確認していない壁




最短時は加速しながら走行するため、既知の区画を走るよう歩数マップを作る必要がある
⇒ 最短時は探索で得た迷路形状を踏まえ、歩数マップを生成する

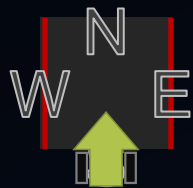
壁情報の管理

探索において、光センサーのセンシング結果を、適切に地図に格納する必要がある。

「各方向」の「壁の有無」と「探索済み」情報を表現するデータ構造例として、以下の2次元配列が用いられる

`unsigned char map[MAZE_SIZE][MAZE_SIZE];` // MAZE_SIZEはクラシックは16、ハーフは32

◆1マス(1byte)あたりの意味付け  はマウスの現在位置



$\text{map}[x1][y1] =$

| S | W | E | N | S | W | E | N |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |

= 0xf6 = 246

探索済み 壁の有無



$\text{map}[x2][y2] =$

| S | W | E | N | S | W | E | N |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 0 | 0 | 0 | 1 |

= 0xf1 = 241

- ・ 2進数、16進数を理解し、データ領域を有効活用することで、限られたRAMを節約する。

◆隣接した区画について



例：左図の上の区画にいるとき

$\text{map}[x][y]$ は南壁がある = $\text{map}[x][y-1]$ には北壁がある

⇒ 必ずしも区画に入らなくても、周囲に行くだけで判断できることがある

⇒ 区画を更新する際は隣接の「壁の有無」と「探索済み」を同時に更新する必要がある

壁情報の操作・解釈方法

Read/Update機能について紹介

Read 歩数マップ生成時

// ある座標のある向きの壁があるか

```
bool exist_wall(x, y, dir){  
    if (dir == North) {  
        return (map[x][y] & 0x01) == 0x01 ;  
    }  
    /* 他略 */  
}
```

| S | W | E | N | S | W | E | N |
|---|---|---|---|---|---|---|---|
| - | - | - | - | - | - | - | 1 |

// ある座標のある向きが探索済みか

```
bool is_stepped(x, y, dir){  
    if (dir == North) {  
        return (map[x][y] & 0x10) == 0x10 ;  
    }  
    /* 他略 */  
}
```

| S | W | E | N | S | W | E | N |
|---|---|---|---|---|---|---|---|
| - | - | - | 1 | - | - | - | - |

Update 壁情報を更新時

// ある座標のある向きの壁の有無と探索済み情報を更新

```
void update_wall_data(x, y, dir, existWall){  
    if (dir == North) {  
        map[x][y] |= 0x10; // 北探索済み  
        if (existWall) {  
            map[x][y] |= 0x01; // 北壁あり  
        } else {  
            map[x][y]  
                = (map[x][y] & 0xf0) | (map[x][y] & 0x0e);  
        }  
    }  
    /* 他略 */  
}
```

| S | W | E | N | S | W | E | N |
|---|---|---|---|---|---|---|---|
| - | - | - | 1 | - | - | - | 0 |

変更しない 北壁だけ0にする

歩数マップ生成方法 【概要編】

[0]全マス初期値255を入れ、
ゴール座標に0を入れる

| | | | |
|-----|-----|-----|-----|
| 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 |
| 255 | 0 | 255 | 255 |

スタート ゴール

[1]0のマスと隣接する壁がない
区画に1を入れる

| | | | |
|-----|-----|-----|-----|
| 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 |
| 255 | 1 | 255 | 255 |
| 255 | 0 | 1 | 255 |

スタート ゴール

[K] : k-1のマスと隣接する壁がない
区画にkを入れる

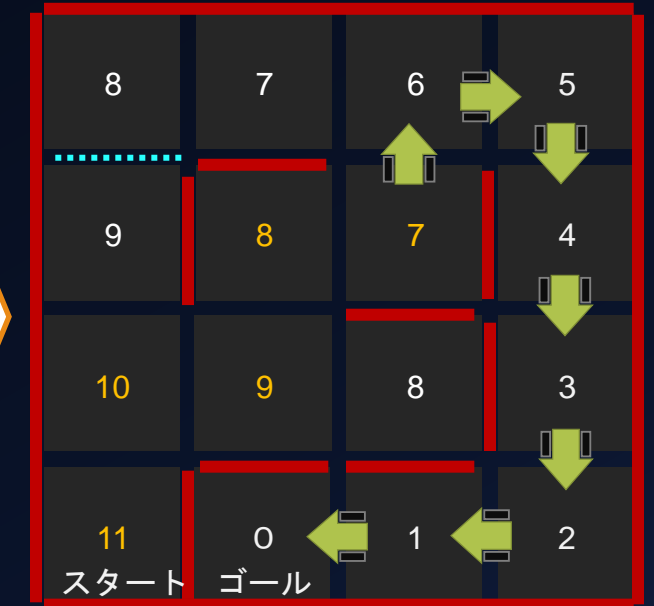
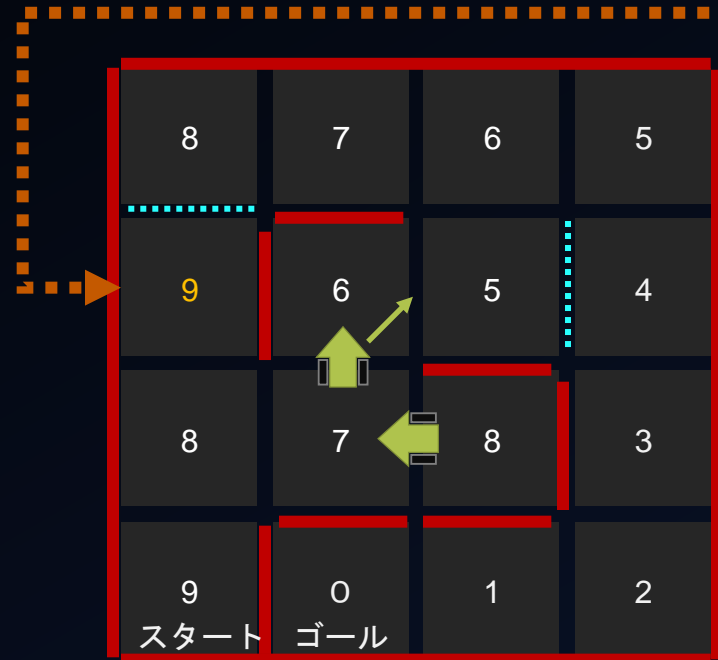
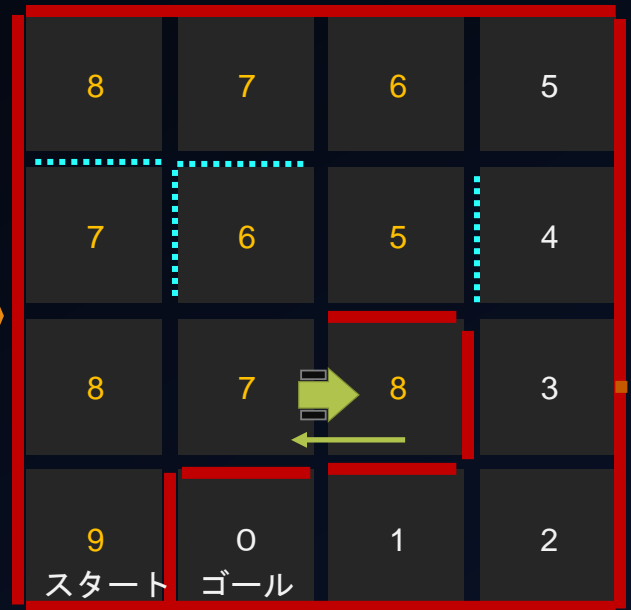
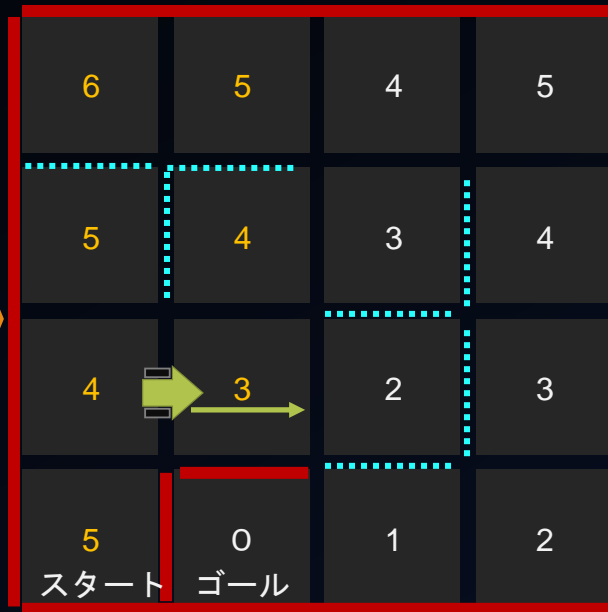
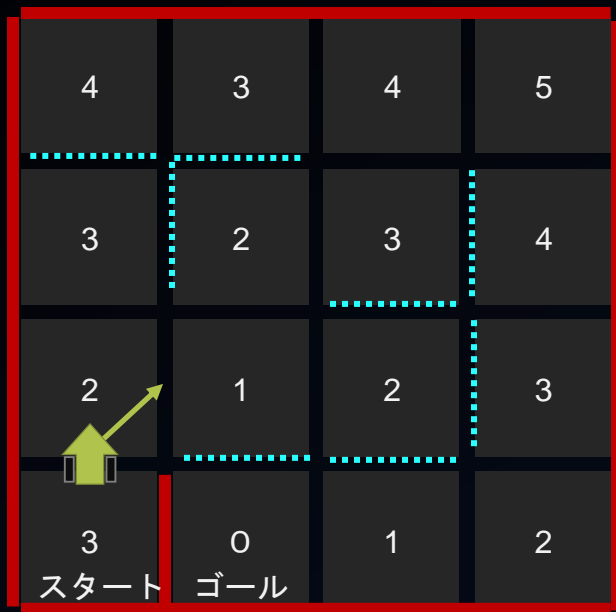
| | | | |
|---|---|---|---|
| 4 | 3 | 4 | 5 |
| 3 | 2 | 3 | 4 |
| 2 | 1 | 2 | 3 |
| 3 | 0 | 1 | 2 |

スタート ゴール

探索をしないと、実際の距離を加味せず
歩数マップを形成してしまう

壁情報をもとに歩数を生成すると、迷路の形状を踏まえたゴールからの距離「歩数マップ」が求まる

【参考】探索中の歩数マップ生成結果の推移



一旦、整理

- 歩数マップを作るには、「隣接する壁がない区画」を探し、更新元の歩数から+1した歩数値で上書きすればよい

⇒ 「壁の有無」判定ロジック (`exist_wall`) を駆使すればよい

Q1. 初期値として255を入れる理由は？

⇒ 上書きされない＝たどり着けない区画である と表現するため

⇒ クラシックの場合最大256マスあるため、256通り以上の表現力があればよい

Q2. 32x32など最大1024マスある場合は？

⇒ 1023以上を初期値として用いる

次は実装する際、気にすべき点を紹介

歩数マップ生成方法 【実装目線編】

Q1. 歩数マップを生成するには？ （再度、振り返り）

⇒ 隣接する更新されてない&壁がない区画を見つけ、歩数を加算する

Q2. 0はゴール座標で既知だが、右図では1が2か所ある

⇒ 「歩数が1」の座標を探す労力は？ ⇒ 16x16箇所検索して 2hit

⇒ 0~255歩まで生成する場合、最悪の検索回数は？

「16 x 16」箇所 x 256歩分 = 65,536 回 （！？）

Q3. マイクロマウス競技の迷路サイズ（32x32）のときの労力は？

⇒ 「32 x 32」箇所 x 1024歩分 = 1,048,576 回 （！？）

| | | | |
|-----|-----------|-----|-----|
| 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 |
| 255 | 1 | 255 | 255 |
| 255 | 0 スタート | 1 | 255 |

数十万、百万の規模の計算を行うと、他の計算の余裕がなくなることがある

⇒ 歩数マップ生成中も進むため、距離ズレを引き起こしたり、
スラローム実行時の前距離がなくなることがある。

◆対策案

更新した区画の座標を記憶することで、処理を短くできないか？

サンプルコード【100万回ループ版】

※事前に初期化し、ゴール座標に0を入れてから以下を実行

```
for (int dist = 0; dist < 1024; dist++) { //歩数が更新されなくなるまで実行
    //全マス検索して更新すべき座標を探す
    for (char i = 0; i < 32; i++) {
        for (char j = 0; j < 32; j++) {
            if (dist_map[i][j] == dist) { // 座標を見つけたら
                if (!exist_wall(i, j, North)) { // 各方角の壁の有無を見て、歩数を更新
                    if (dist_map[i][j + 1] > dist + 1) { // 更新した方が距離が短くなる場合
                        dist_map[i][j + 1] = dist + 1; //更新する
                    }
                }
            }
        }
    }
    // 他の方角は省略
}
```

←更新する区画がなくなったら打ち切ることができるが、
「絶対に1024回やらない」保証はない

←諸悪の根源

↑次の更新元にすべき座標はわかるが、「重複無く」「順序正しく」更新する必要がある
⇒ 歩数Kの座標すべてを更新した後でないと、K+1を更新し始められない。

⇒ 歩数が低い順に更新するロジックを作るには？

歩数マップ生成 【更新元座標メモ化方式】

0. 全区画に初期値を入れて、ゴール座標を0にする
1. ゴール座標を「更新元配列」に入れる
2. 更新元配列の先頭から、座標を取り出し、隣接する壁がない区画を更新する
※更新元配列から取り出した座標を削除する
3. 2で更新した座標を更新済み配列の最後尾に追加する
4. 2~3を繰り返し、更新元配列がゼロになったら終了

「配列の先頭から、座標を取り出す」

このようなデータ構造をQueue（キュー）配列といい、先に登録されたデータを優先して処理する際(※1)に用いられる。

※1 FIFO: First-In-First-Out

更新中のキュー配列の動き

(1, 0)を使って更新した結果

| | | | |
|-------------|----------|-----|-----|
| 255 | 255 | 255 | 255 |
| 255 | 255 | 255 | 255 |
| 255 | 1 | 255 | 255 |
| 255 スタート | 0 ゴール | 1 | 255 |

↓更新元配列

| | |
|-------|----------|
| (1,0) | ←削除済み |
| (1,1) | ←次の更新元座標 |
| (2,0) | |

(1, 1)を使って更新した結果

| | | | |
|-------------|----------|-----|-----|
| 255 | 255 | 255 | 255 |
| 255 | 2 | 255 | 255 |
| 2 | 1 | 2 | 255 |
| 255 スタート | 0 ゴール | 1 | 255 |

↓更新元配列

| | |
|-------|----------|
| (1,0) | ←削除済み |
| (1,1) | ←削除済み |
| (2,0) | ←次の更新元座標 |
| (1,2) | |
| (2,1) | |
| (0,1) | |

◆更新元配列の例（1つの配列に圧縮する工夫もあるが、16x16までに限られる）

```
unsigned char update_memo_list_x[]
```

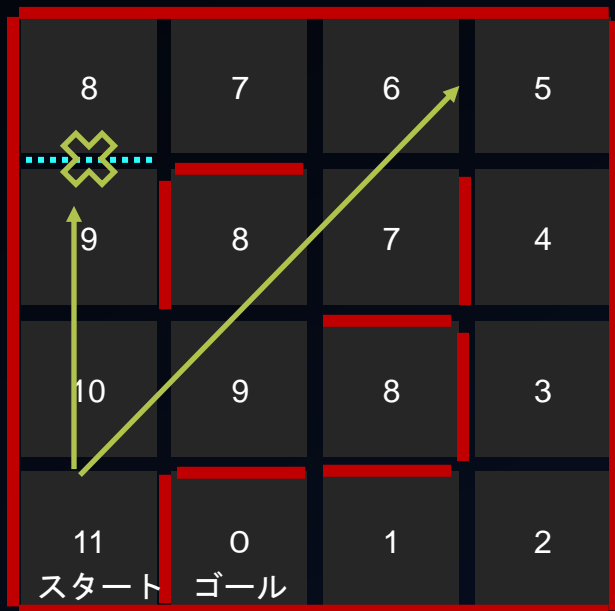
```
unsigned char update_memo_list_y[]
```

キュー配列を活用し、更新元座標を記憶することで、更新時の検索は必要なくなる。
結果、更新にかかる処理の回数は

—「16×16」箇所 × 256歩分 = 256 回（元は65,536）

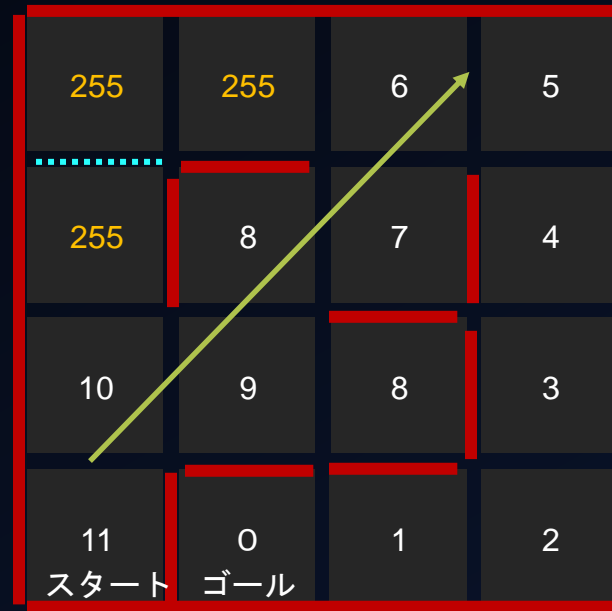
—「32×32」箇所 × 1024歩分 = 1,024 回（元は1,048,576）

【補足】最短走行するには？



歩数が小さくなる方向に動くと、
確認していない区画に入ってしまうことがある

見てないところに壁があるかもしれない



見てない区画は更新しなければよい

＝最短走行向けの歩数マップには、
「探索済み」かを加味すればよい

まとめ

- 「デバッグの手法」について
 - 5種類の方法を使い分けよう
 - 特に、ログ解析について
 - グラフ化ツールを駆使しよう
 - 原因の特定には、 事象の観測、仮説を立てる、仕様確認、事実確認 を繰り返そう
- 「探索アルゴリズム」について
 - 足立法・・・マイクロマウスでメジャーな探索手法
 - 目的地からの距離が短くなる区画へ移動するアルゴリズム
 - 壁情報の管理・・・ビットを駆使しながら効率よく格納しよう
 - 歩数マップ・・・目的地からの区画単位の距離を示した配列
 - 生成は効率よく行う必要がある（キュー配列）

最後に

足立法における、歩数マップは「1マス=1歩」とし、生成しましたが、本来、加減速や斜め走行を行うことから、必ずしも「最短=最速」とは限りません。

複数の経路を比較し、より早い経路を選択するなど、工夫すべきところが多くありますが、まずは、「安定して探索する」ことを目指してください。



ご清聴、ありがとうございました。