ゲームAI作成に使えるいくつかのアルゴリズム

情報通信工学科

目次

- 経路探索
  - トレーシングによる障害物回避
  - ブレッドクラム経路探索
  - AStarアルゴリズム
- 有限状態機械
- ルールベースAI

経路探索

トレーシング

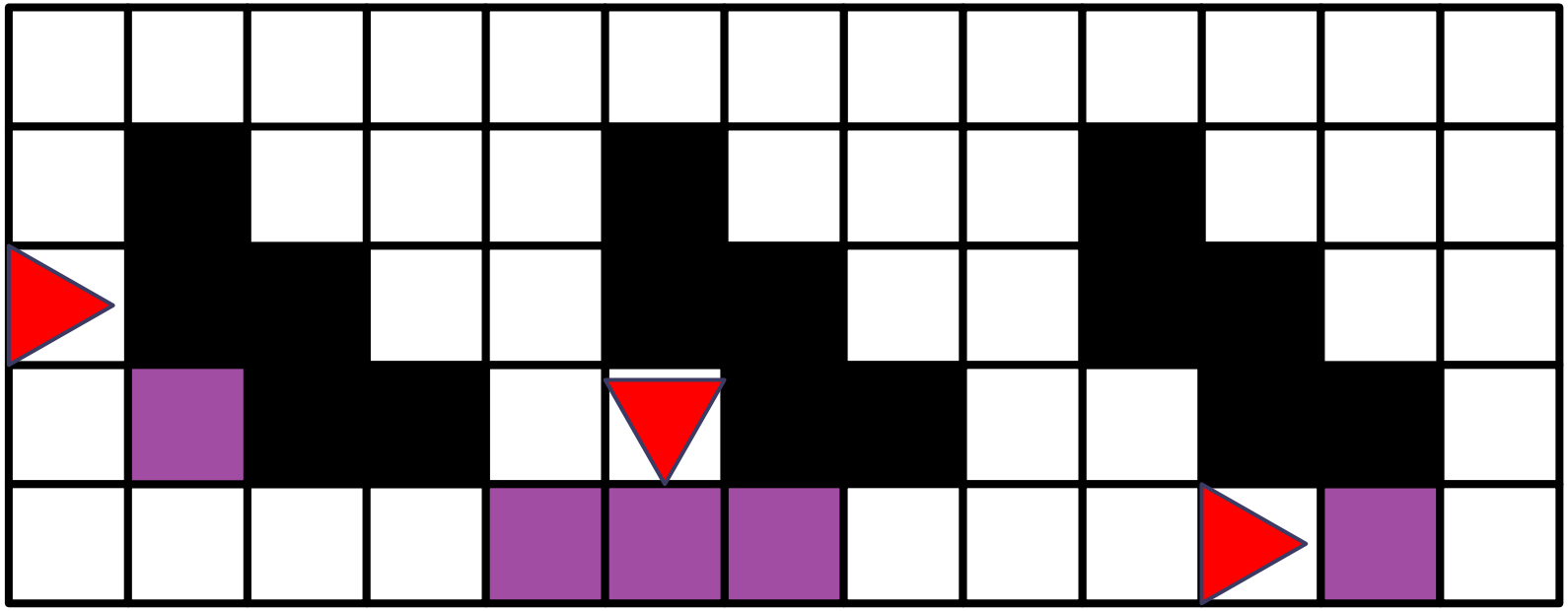
トレーシング

- 障害物に沿って移動することで障害物を回避する
- 障害物に衝突する前は経路探索モード
- 障害物に衝突した後はトレーシングモード
- ポイントはトレーシングから経路探索に戻るタイミング

トレーシング

トレーシング

- 進行方向を検索して移動可能かつ壁が隣にある部分に移動する。

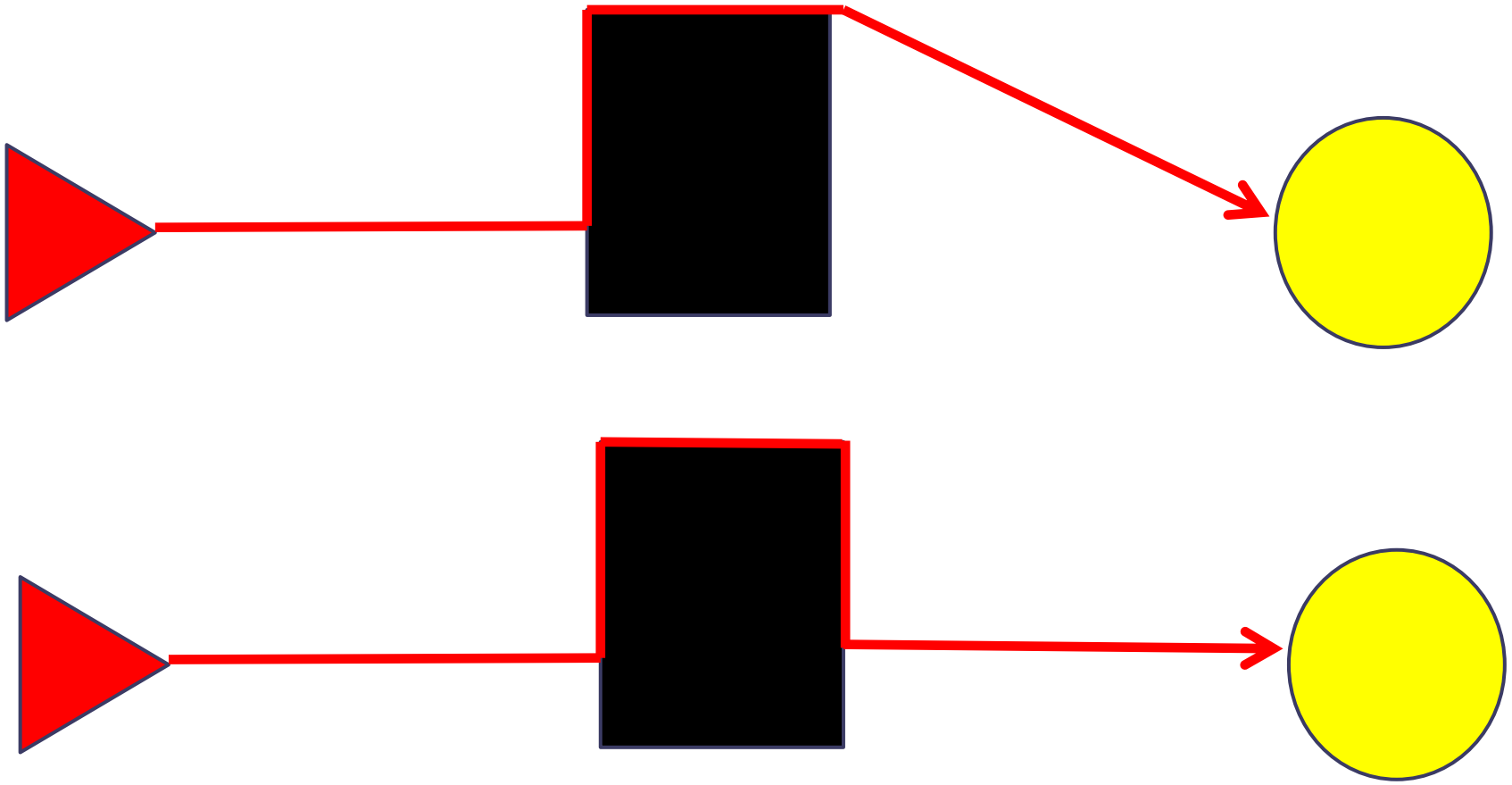


トレーシング

トレーシングをやめるタイミング

- その１：一定距離をトレーシングした後、経路探索を実施し、今いる地点から目的地への経路が発見できたのならトレーシングを終了し、経路を進む。
- その２：障害物と衝突した場所から目的地までを直線で結び、移動オブジェクトがその直線と交差した時トレーシングを終了し、経路探索を実施して経路を進む。

トレーシング

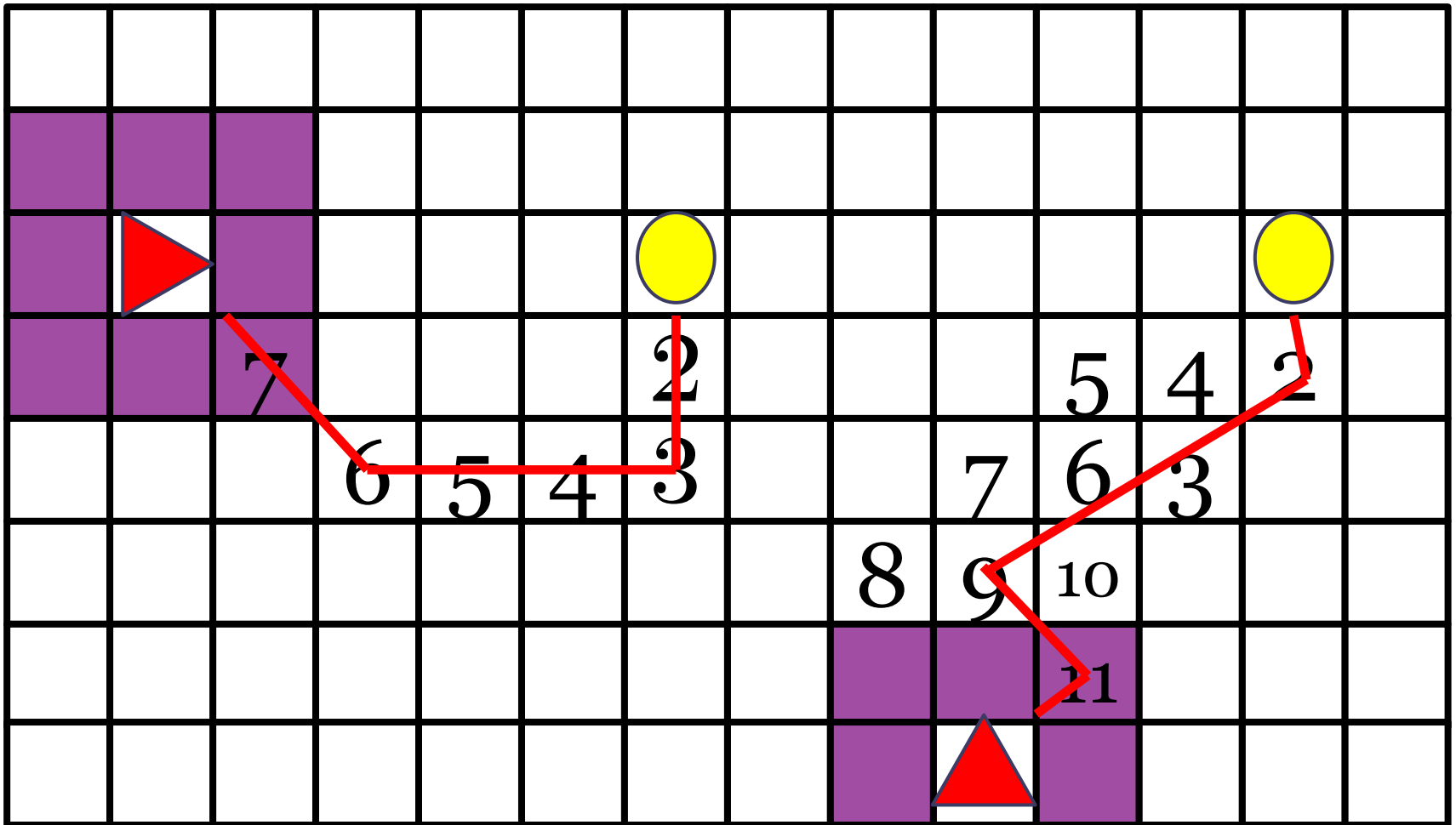


ブレッドクラム経路探索

ブレッドクラム経路探索

- プレイヤーを追いかけることで障害物などを回避する経路探索。プレイヤーの後を追いかけてるので必然的に通れる場所しか通らない
- ブレッドクラムとはパンくずのこと
- プレイヤーはブレッドクラムを落としながら移動してるのでそれを追いかける。
- もちろんブレッドクラムはプレイヤーには見えない
- ブレッドクラムに番号を振っておくことで古いブレッドクラムと新しいブレッドクラムを分ける
- 検知範囲内の新しいブレッドクラムを追うことで最短経路で移動できる。

ブレッドクラム経路探索



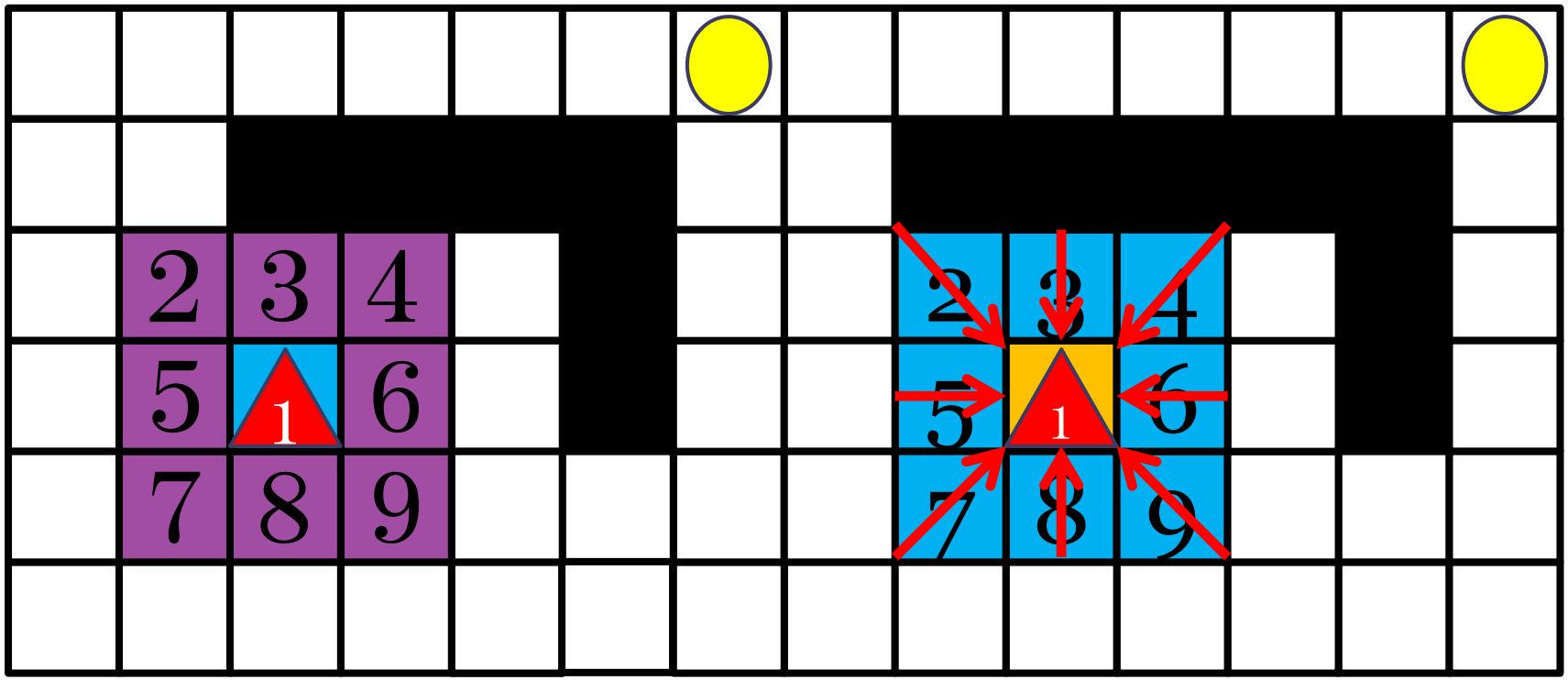
AStarアルゴリズム

- 様々な経路探索アルゴリズムの中でも最もよく使われているアルゴリズム
- ダイクストラ法の改良
- 単純な直線の最短経路から障害物を回避した最短経路、移動にコストがかかる場合の最短経路も全て見つけることが出来る
- つまり色々と応用が効く
- ただしなかなか計算量が多い

AStarアルゴリズム

- 開始ノードから周囲のノードに探索を拡大して経路を探索する
- 探査の必要があるノードのリストをオープンリストという
- 探査の終了したノードのリストをクローズドリストという
- オープンリストから探索するノードを選択して探索し、クローズドリストに移す作業をオープンリストがEmptyになるか目的地がオープンリストに追加されるまで続ける
- オープンされたノードは自分をオープンした親ノードを記録しておく

AStarアルゴリズム



オープンリスト={1}

オープンリスト={2,3,4,5,6,7,8,9}
クローズドリスト={1}
ノード2,3,4,5,6,7,8,9の親ノードは
ノード１

- 問題はオープンリストの中からどのノードを選ぶべきか

AStarアルゴリズム

- オープンリストからどのノードを選ぶべきか
- 優先されるノードの決定方法が必要
- スコアリングによってノードを評価することで実現できる
- スコア＝開始ノードからの合計コスト
  ＋ヒューリスティック値
- 開始ノードからの合計コスト=開始ノードからの距離
  +地形コスト
- 地形コストを導入することでダメージを受ける床を踏んで自爆して死ぬアホAIを回避できる
- 地形コストの算出を各ノードの基本コストにキャラクター特性を考えて増減させるとより個性的なAIになる
  - 浮いてるキャラ→地形コスト=基本コスト-N
  - 特定の地形でダメージを受けるキャラ
    →地形コスト=ダメージを受けるノードの基本コスト+N
    →地形コスト=基本コスト

AStarアルゴリズム

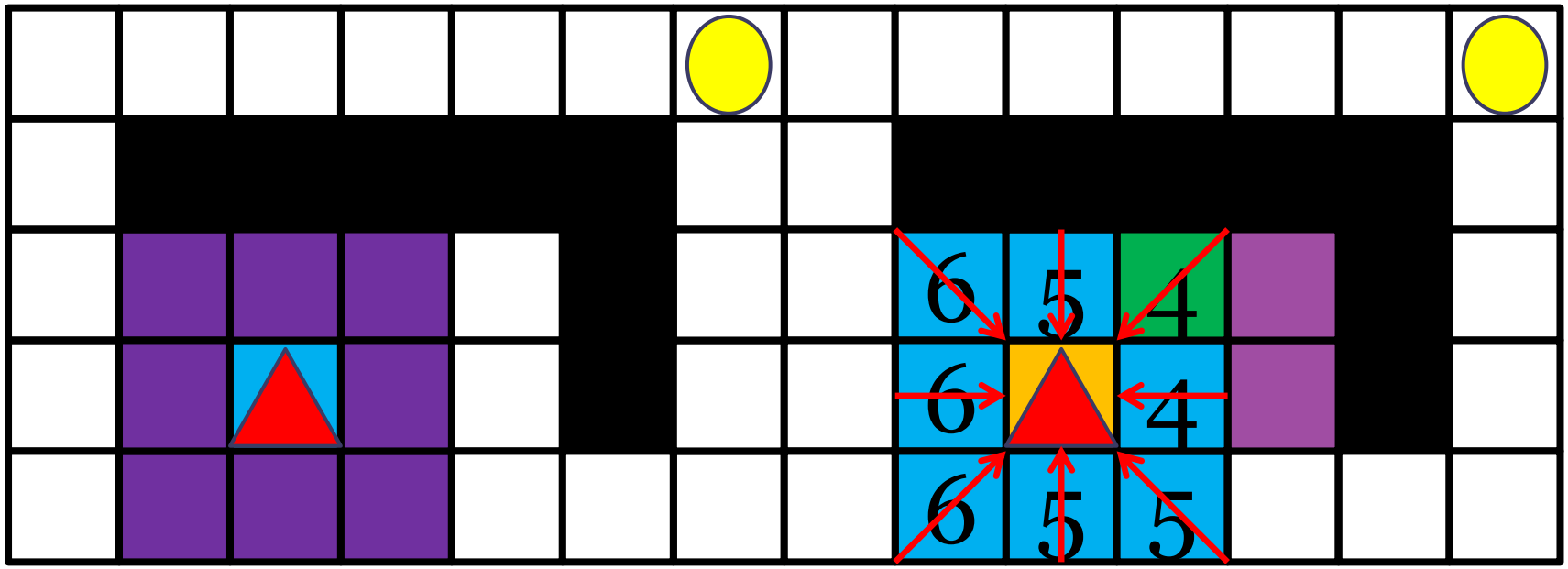
- スコアの一番低いノードをオープンリストから選んでそのノードの周囲を探索する
- ヒューリスティックとは推測のこと
- ノードから目的地までの距離を推測する
- ヒューリスティックを算出する方法を変えることで色々と応用できる
- 今回はノードから目的地までのマンハッタン距離をヒューリスティック値にする
- スコアが同じになった場合は自前の定義で選ぶ

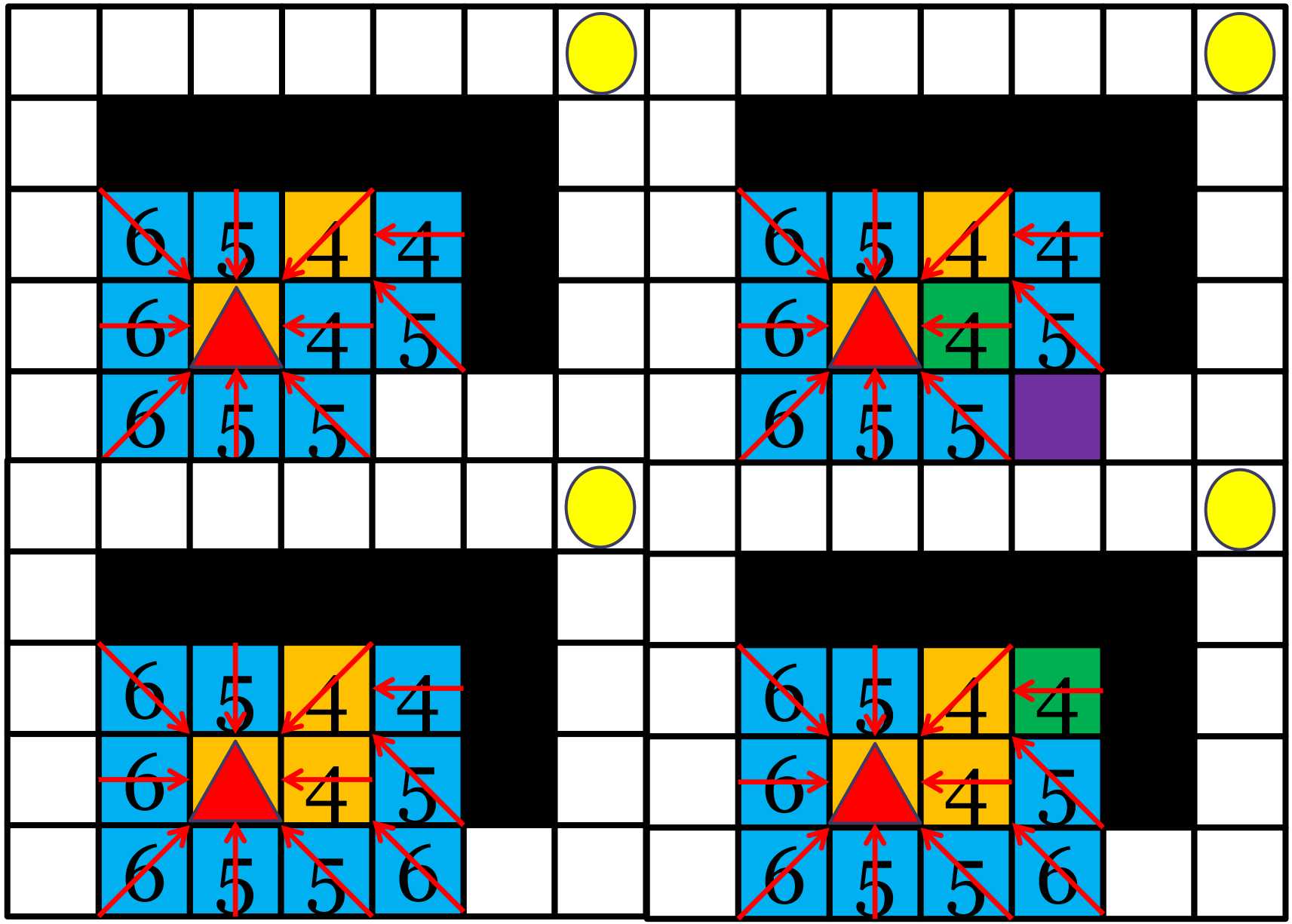
  例：ヒューリスティックが小さい方

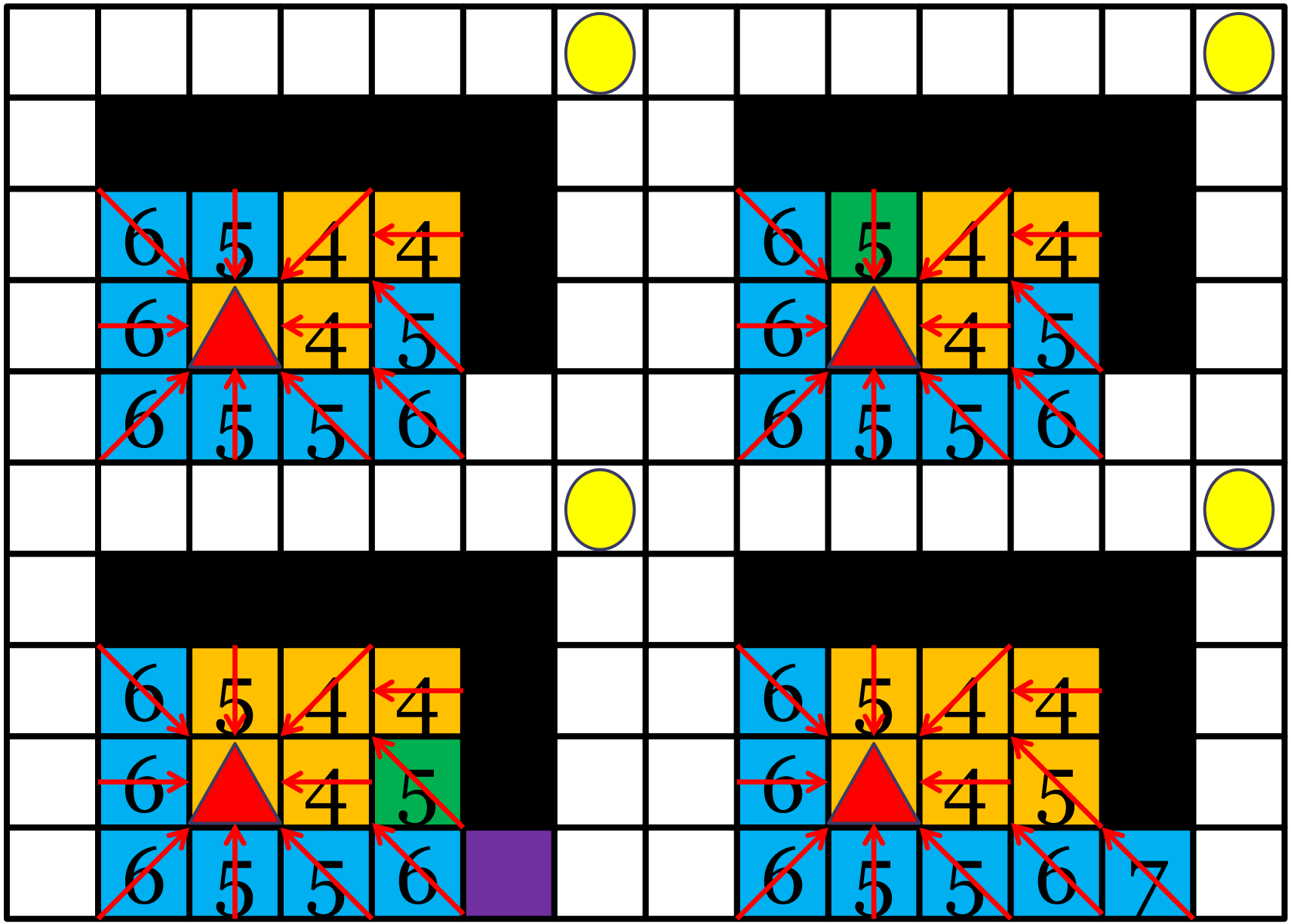
  　　リストのより先頭に居る方

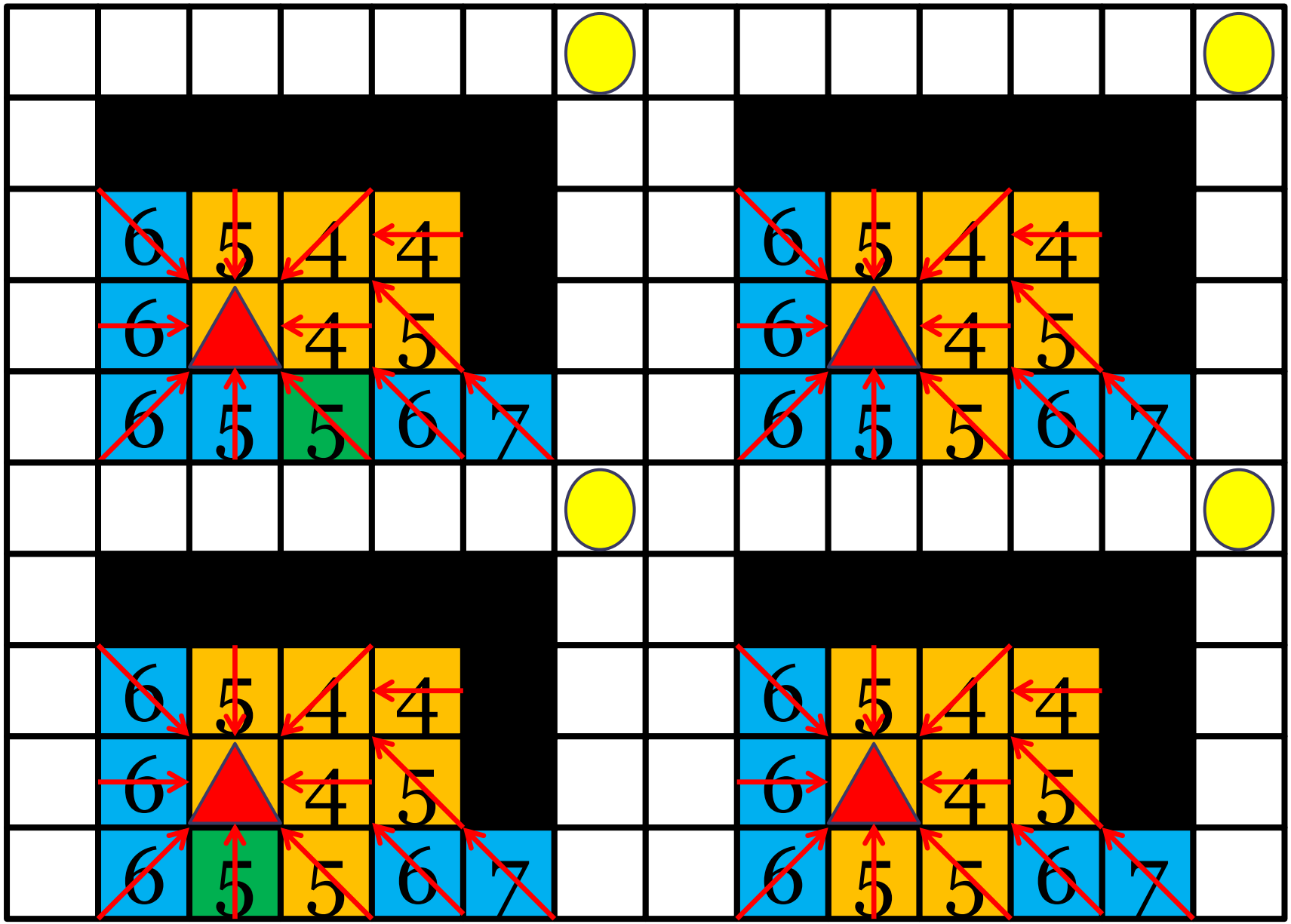
AStarアルゴリズム

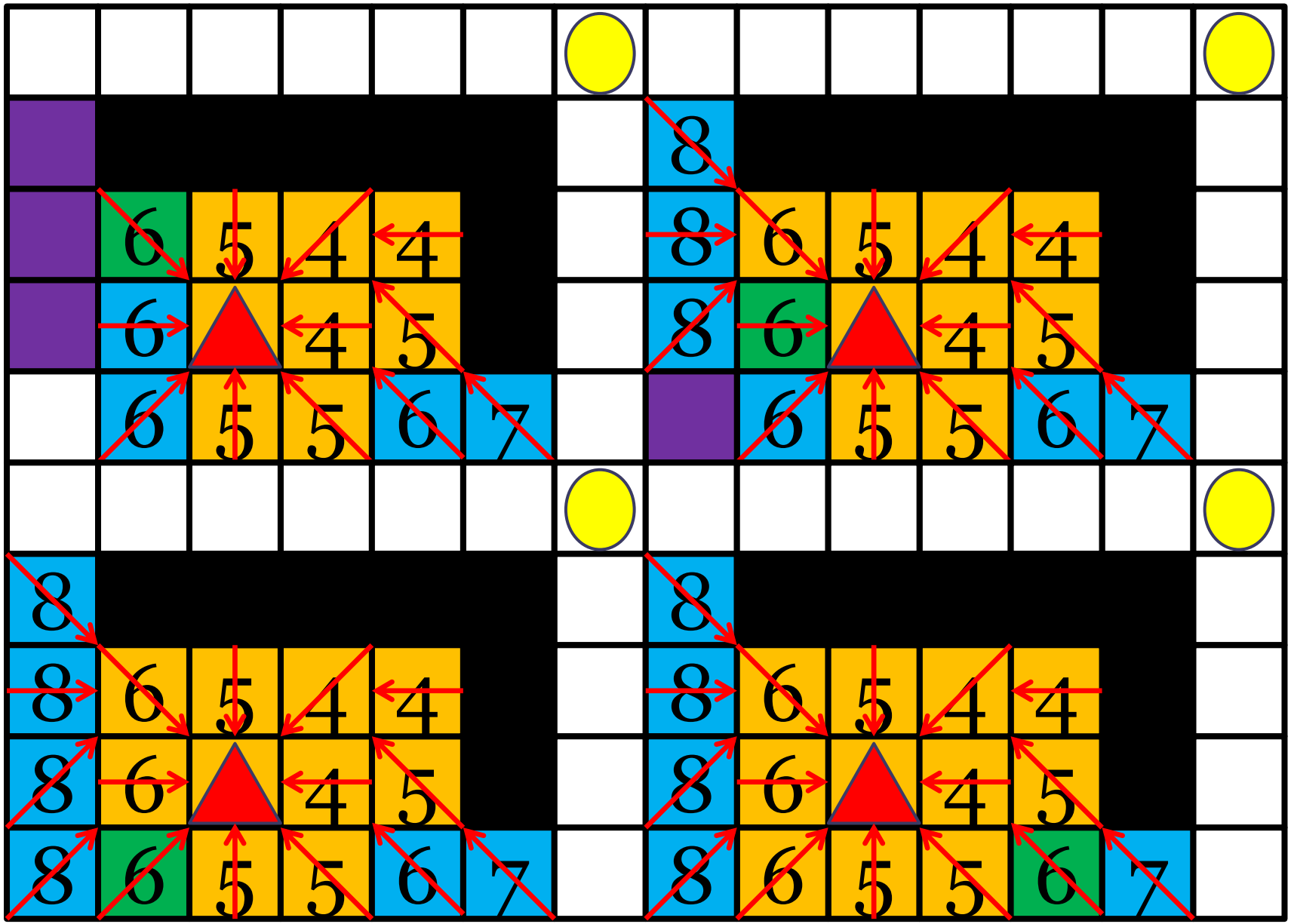
- 地形コストは無し
- 先ほどまでと違いマスに書いてあるのはスコアになっていることに注意
- 青と緑オープンリストに入ってるマス
- 緑はオープンリストから選ばれたマス
- 紫が探索範囲
- オレンジがクローズドリストに入っているマス
- 右上・左上・右下・左下の順で進む

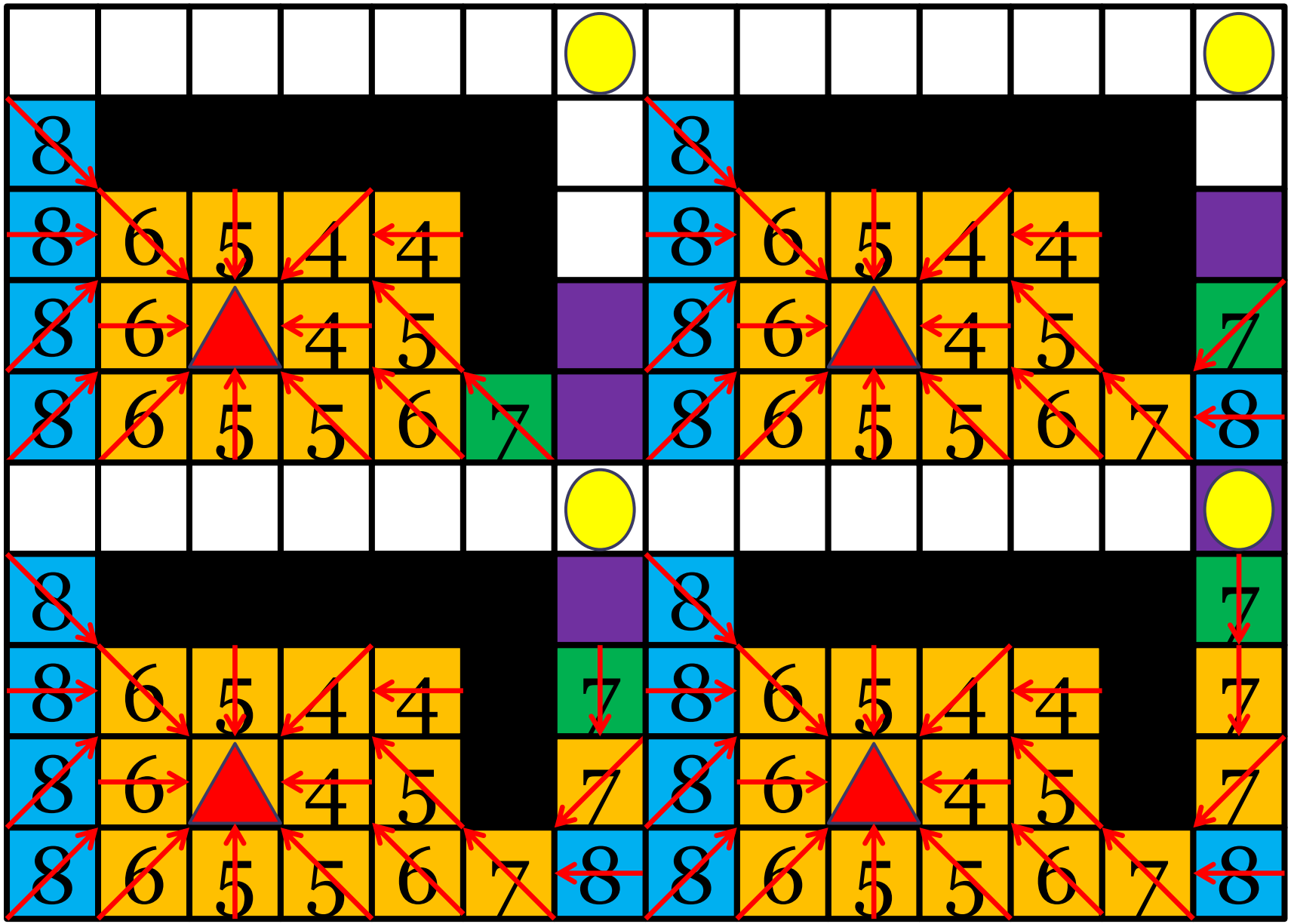


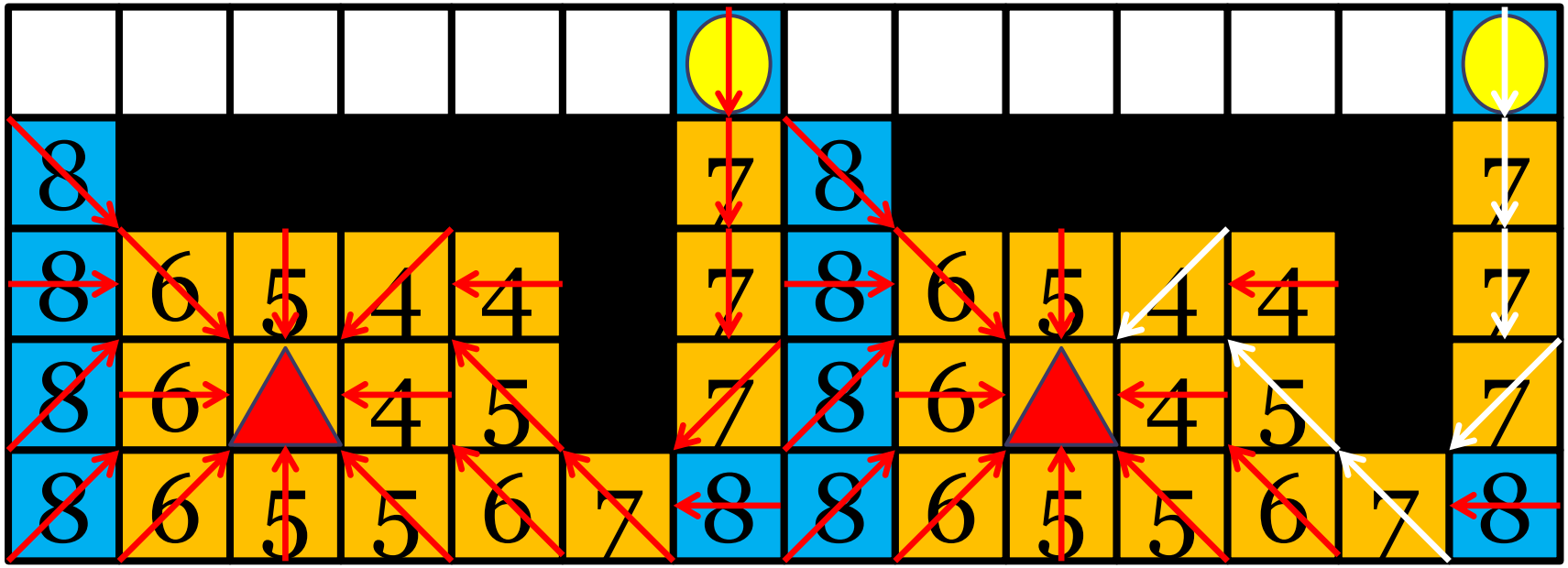

6 5 4 4
6 4 5
6 5 5
6 5 4 4
6 4 5
6 5 5
6 5 4 4
6 4 5
6 5 5 6
6 5 4 4
6 4 5
6 5 5 6


6 5 4 4
6 4 5
6 5 5 6
6 5 4 4
6 4 5
6 5 5 6
6 5 4 4
6 4 5
6 5 5 6
6 5 4 4
6 4 5
6 5 5 6 7


6 5 4 4
6 4 5
6 5 5 6 7
6 5 4 4
6 4 5
6 5 5 6 7
6 5 4 4
6 4 5
6 5 5 6 7
6 5 4 4
6 4 5
6 5 5 6 7


6 5 4 4
6 4 5
6 5 5 6 7
8
8 6 5 4 4
8 6 4 5
6 5 5 6 7
8
8 6 5 4 4
8 6 4 5
8 6 5 5 6 7
8
8 6 5 4 4
8 6 4 5
8 6 5 5 6 7

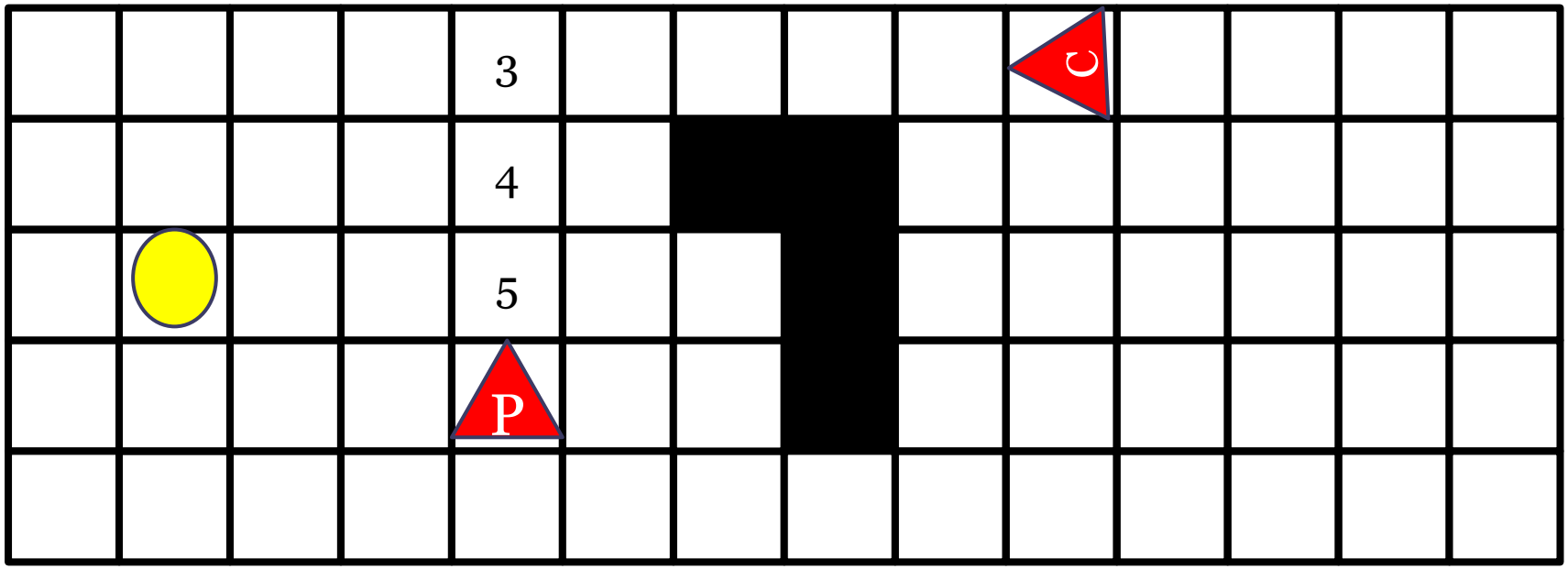

8
8 6 5 4 4
8 6 4 5
8 6 5 5 6 7
8
8 6 5 4 4
8 6 4 5 7
8 6 5 5 6 7 8
8
8 6 5 4 4
8 6 4 5
8 6 5 5 6 7
7
8
8 6 5 4 4
8 6 4 5
8 6 5 5 6 7 8
7
8
8 6 5 4 4 7
8 6 4 5 7
8 6 5 5 6 7 8
8
8 6 5 4 4 7
8 6 4 5 7
8 6 5 5 6 7 8



- 白い矢印が今回見つけた経路
- ７回の移動でたどり着いている
- これはいくつかある最短経路の内の１つ
- 同じスコアを適当に選んだので動きがあまり知的に見えない

AStarアルゴリズム

- 地形コストを動的に変化させると更に面白くなる
- 例えばプレイヤーの視線上のノードの基本コストを増加させるとプレイヤーからなるべく隠れて移動するAIになる
- マスの数字は地形コスト(何も書いてなければ１)
- PがプレイヤーCがAIキャラクター



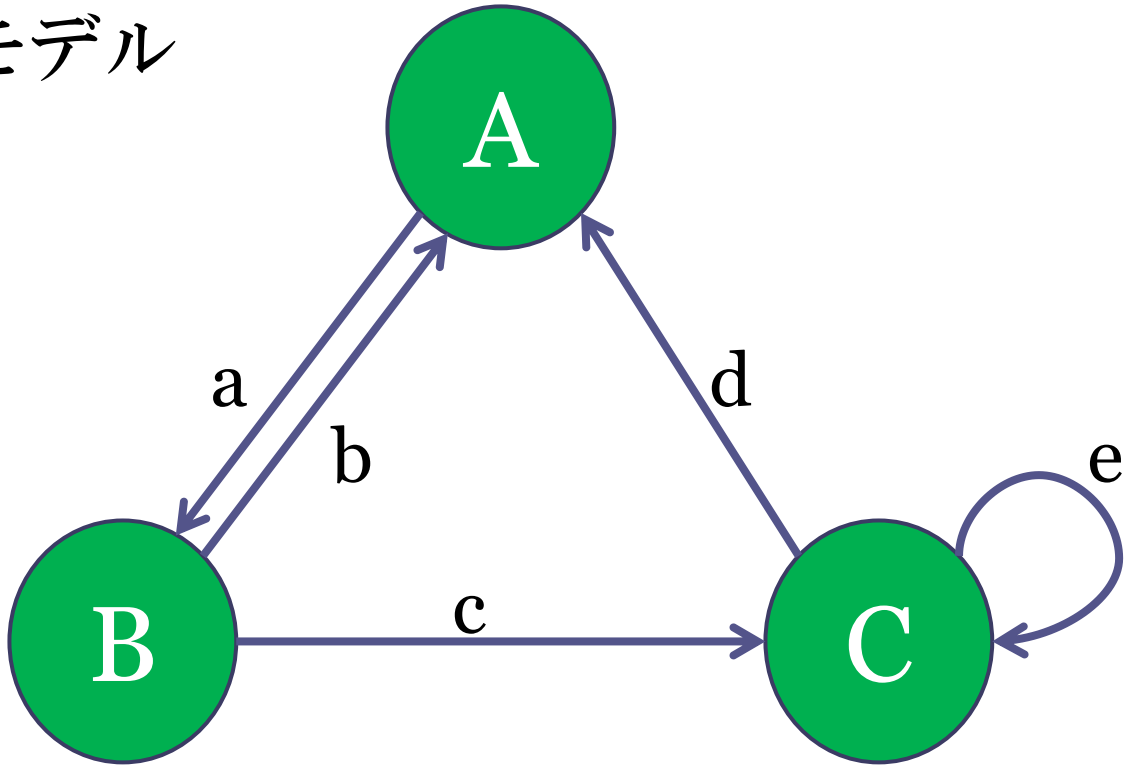
有限状態機械

有限状態機械

- いくつかの状態の遷移を表すグラフ
- 状態を変更するタイミングを判断する条件も一緒に表せる
- つまりAIの状態により行動を変更させる際に使う事ができる
- つか恐らくAI作ったら自然と使う
- 文章での説明が難しいので図で察して下さい

有限状態機械

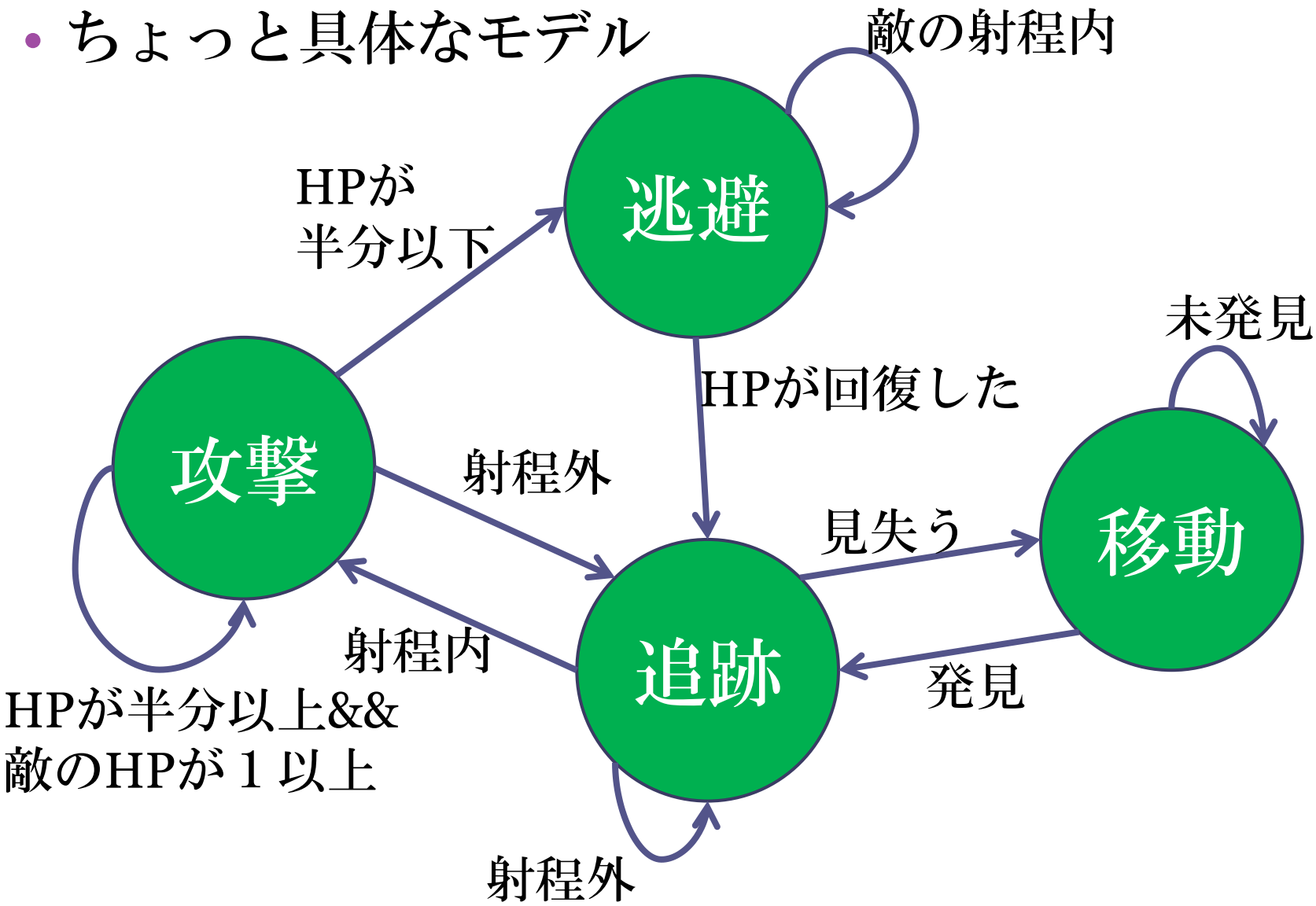
- 簡易モデル



- 円が状態{A,B,C}を表している
- {a,b,c,d,e}は遷移関数と言いこの関数が実行されると状態が遷移する

有限状態機械

- ちょっと具体なモデル



有限状態機械

```
#define MOVE 1
#define CHASE 2
#define ESCAPE 3
#define ATTACK 4

class AiObject
{
private:
    int state;
    int hp;

public:
    AiObject();
    ~AiObject();

    void Move();
    void Chase();
    void Escape();
    void Attack();

    int GetState();

};
```

- こんな感じにクラス作って

```
AiObject ai[10];

for (int i = 0; i < 10; i++){
    switch (ai[i].GetState())
    {
    case MOVE:
        ai[i].Move();
        break;
    case CHASE:
        ai[i].Chase();
        break;
    case ESCAPE:
        ai[i].Escape();
        break;
    case ATTACK:
        ai[i].Attack();
        break;
    default:
        break;
    }
}
```

- こんな感じに使う

ルールベースAI

ルールベースAI

- 事前に用意されたDBからプレイヤーの次の動作を予測する
- 予測された動作によってAIの動作を変えることでAIが知的に見える
- 予想される動作に重み付けて予想することでいくつかの予想の中で一番的中率の高いものを呼び出す
- ワーキングメモリとルールの要素で成り立つ
- ワーキングにはそれまでの動作とルールから得られた予想が格納される
- ルールにはif-then形式のルールが格納されている
- ルールを呼び出して動作を起こしたりワーキングメモリに新しい情報を追加できる

ルールベースAI

- ２つの過去の動作から３つ目の動作を予想しようとする例を示す
- 動作は[攻撃,魔法,防御]の３つ
- 考えられるルールは27種類
  - [攻撃,攻撃,攻撃] [攻撃,攻撃,魔法]~[防御,防御,防御]

ルールベースAI

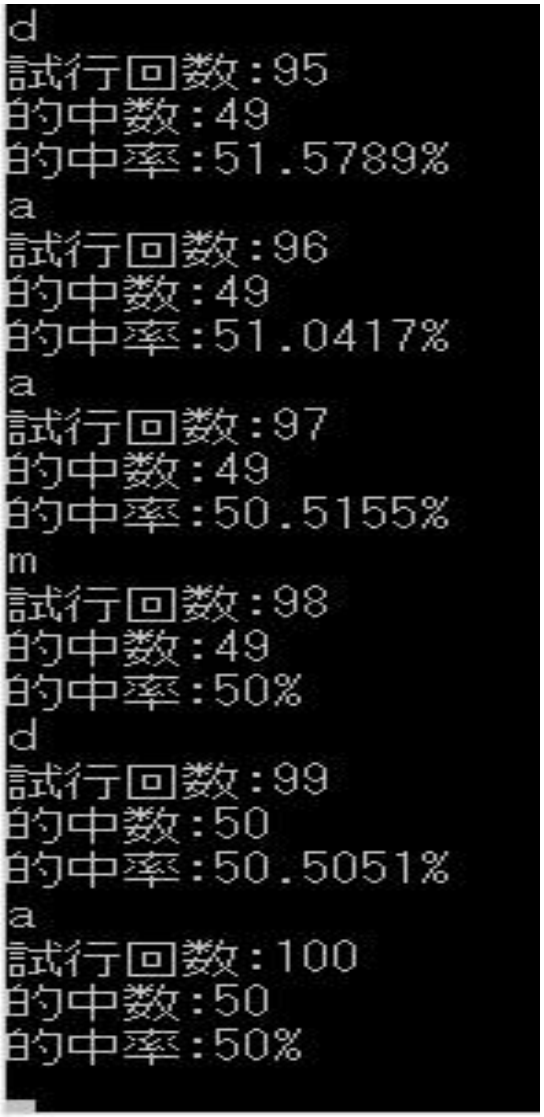
```
enum Action{ Attack, Magic, Defense,Unknown };

struct WorkingMemory
{
    Action actionA;//一つ前の行動
    Action actionB;//二つ前の行動
    Action actionC;//予想される行動
};

class Rule{
public:
    Rule();
    ~Rule();

    void SetAction(Action A, Action B, Action C);

    //SetActionでこいつらを初期化/////
    const Action ruleA;
    const Action ruleB;
    const Action rerutnAction;
    ////////////////////////////////////
    bool match;
    int weight;//ルールが的中したら増加

};
```

- こんな感じの構造体とクラスで実装できる
- 初期化時にはactionA、actionB、 actionCはUnknownにする

ルールベースAI

- 予想ができたらそれに対する動作が必要
- そのあたりもワーキングメモリに追加するとAIの挙動を制御できる
- ルールを外部のスクリプトなんかで置いておくと追加や調整が楽
- ちなみにこのアルゴリズムの的中率は65~80%位らしい
- ランダム予測だと30%位らしいので優秀

ルールベースAI



- 試しに動作させてみた結果
- 的中率50%
- 65%以上じゃないの(キレ気味
- 恐らく適当にボタン押してたので重み付けのパターンが活きてない
- それでもランダム予測よりは的中率が高い