サンプル調査(random/java)

　田中 健人

　山内 拓磨

サンプルとして配布されている探索しないランダムで制御されるAI(?笑)のプログラムを理解するためにサンプルのソースについてまとめる。

このサンプルAIはGameState.java、Point.java、RandomPlayer.javaの3つからソースから構成される。

※以降、ゲームマスターとは、AIプログラムと分けて公式のゲームプログラムのことを指す。

* GameState.java  
    
  GameStateクラスの仕様が記述してある。  
  GameStateクラスは以下の要素からなる。  
    
   <定数>
  + YMARGIN  
    デフォルトで100に設定されている。  
    ゲームの仕様上、ゴール地点後にも、ゴールの位置から-1した位置から、視野の距離だけ障害物がある可能性がある。  
    つまり、ステージのlength+視野-1まで障害物の情報を格納しないと、永遠にゴールできない可能性があり、視野はゲーム開始時に与えられるので、そのことを考慮したマージン。  
    仕様上は視野は100より大きい値が与えられる可能性があり、もし、視野が100以上あたえられ、ゴール直前でy軸方向に速度が100以上出ててた場合、YMARGINが100ではゴール直前で動けなる可能性がある。  
      
      
    <クラス>
  + PlayerStateクラス  
    敵のplayer、自分のplayer、2種類のインスタンスとして生成されている。
    - position,velocityを持つ(どちらも後述するPointクラスのインスタンス)
    - getPosition(),getVelocity()でそれらを取得できる
    - update()では受け取ったScannerから、position,velocityを値を更新する。  
      先頭から、x軸のposition、y軸のposition、x軸のvelocity、y軸のvelocityの値を受け取る。  
      ステップ毎にゲームマスターから与えられる入力から、GameStateのupdateメソッドにより、敵、味方それぞれのインスタンスに対して、position、velocityが更新される。

<メソッド>

* + GameState() : コンストラクタ

初期化時の入力をScannerを用いて各変数に格納し、初期化を行う

* + - remainingTime : 残り考慮時間
    - step : ステップ数　０で初期化
    - stepLimit : ステップ数制限
    - width,length : コースのサイズ
    - visionLimit : 視野の最大距離
    - obstacle : 障害物の情報　0で初期化　型はint

y方向に余裕を持たせてある(length+YMARGIN)

* + - playerState,oppnentState : 自分と相手の情報　PlayerState型
    - hasMoved : boolean型　falseで初期化

(例)ステージsample-course.smrjkyでゲームを行う場合

ゲームマスターから、初期情報として、

20000　100　15 　100　8

という入力がされる。

それぞれ、残り考慮時間、ステップ数制限、コースの横幅、コースの縦幅、障害物を目視できる距離となり、GameState内の各変数に格納される。

* + update()
    - ステップ毎に与えられる入力をもとにScannerを使って情報を更新
    - 変数lastPositionに1ステップ前の自分のAIのpositionを格納
    - 前ステップのpositionがnullでない(一番最初の更新である0ステップ目の時nullとなる)、かつ前ステップのpositionに現在のvelocityを加えた位置に障害物ない、かつ入力で与えられた現ステップの開始位置が前ステップのpositionに現在のvelocityを加えた位置と同じではない場合、hasMovedをfalseに

それ以外の場合はhasMovedをtrueにする  
↑サンプルのプログラムは確かにこの通りになっている。

だけど、2つめの条件が必要な意味が分からない

前のpositionに現在の速度を加えた位置に障害物があったら、必ず三つ目の条件(現ステップのpositionが前のpositionに現在のvelocityを加えた位置と同じではない)を満たすのに、、

* + - 相手の情報もScannerを使って更新
    - 視界内の障害物の情報を更新する
  + printCurrentState()
    - 現在の状況を出力する(実行するときに指定する、エラーファイルに出力される)  
      実際、RandomPlayerでは毎ステップこのメソッドにより、状況を出力するが、本番では時間が限られているので使用しなほうが浪費時間が多少なりとも減ると考えられる。  
      しかし、出力する場合と出力しない場合で、使用時間を計測したところ、実行した瞬間のPCの状況により使用時間が変動するため、正確に計測できなかった。  
      ステップによっては50～100近く同じ処理でも変動することを考えると、この程度誤差の範疇といえる。
  + isObstacle()
    - 引数で与えられた点に障害物が存在するかを判定

存在する場合にtrueを返す

引数はPointでもx,yでもいい

* + hasObstacleCollision()
    - おそらく動線を含めた障害物との衝突判定  
      positionと速度を与えたとき、そのpositionからその速度で障害物にぶつかるか、ぶつからないかを判定。  
      (もちろんこれまで目視した障害物の範囲で)
  + hasPriority()
    - 優先権を持つかどうかの判定

優先権を持つ場合にtrueを返す

↑正確にはこれは優先権を持つかの判定にはなっていない、

ルールドキュメントには

「優先権を持つのは，位置のy 座標がより小さいプレイヤである． y

座標が同一である場合，位置のx 座標がより小さいプレイヤが優先

権を持つ． ただし，動線が相手プレイヤの動作前の位置に達するか

通過する場合は，優先権が相手プレイヤに移る．

」

とあるが、この判定は「ただし」までの考慮しかされていない。  
そのため、次に示すhasOpponentCollision()メソッドから、このメソッドは呼び出されている。

* + hasOpponentCollision()
    - 相手の開始位置と衝突するかの判定(たぶん)

衝突する場合にtrue  
 ↑ルールブック上の優先権を持つかどうかの判定

* + isPlayable()
    - 与えられた加速度で行動可能であるかどうかの判定
    - 次ステップ開始時でのy方向の速度が負になる場合falseを返す(このゲームのルール上BEST愚か行為だから)
    - hasMovedがtrue、かつ障害物とも相手の開始位置とも衝突しない場合trueを返す

(可視化できている範囲内で障害物にぶつからず、かつ優先権も奪われない、かつhasMovedがtrue)

→hasMovedを見ている意味が分からない

もし、前回から今回に移動ができなかったら、どんな加速度でもPlay出来

ないなんてルールはないよね？

これじゃ、ぶつかったら一回休みのルールが存在するような判定。

* + getPlayerState(),getOpponetState()
    - それぞれの状態を取得する
* Point.java
  + 点を表現するためのクラス
  + xとyの値を持ち、getX()、setX(value)などで値の取得、代入が可能。
  + add(Point p)はpの持つx,yに自身が持つx,yの値をそれぞれ加算した値を持つ新たなPointのインスタンスを返す。
  + equals(Object obj)はobjが異なるクラスではなく、かつx,yが同じ値であるときにtrueを返す。それ以外はfalseを返す。
  + toString()はx,yの値を文字列にして返す。
  + hashCode()はObjectクラスのequals()でオブジェクト同士が同じかどうかを識別するための値を返すもの。２つのオブジェクトのhashCode()の値が同じであればequals()はtrueを返す。
* RandomPlayer.java
  + candidates
    - 候補手を格納するvector

型はvector<Point>

* + play()
    - まず、ルール上指定できる加速度の全組み合わせに対して、それが移動可能かを調べ、移動可能であるならば候補手に追加
    - 候補手がない場合(移動できないなら、せめてスピード落としておかないと、次も移動できなるなるからだと推測できる)
      * x=0でなければx方向への加速度を-1に

x=0であればx方向への加速度を1に

* + - * y=0でなければy方向への加速度を-1に

y=0であればy方向への加速度を1に

* + - * この組み合わせを候補手に追加
    - 候補手の中からランダムで加速度を選択

GameStateはそのまま汎用的に利用されることを期待されているが、チューニングできる余地はある。(より繰り返しを減らしたりできる箇所はある)

さらに、hasmoved値が、動いたかどうかの判定として用意したのであれば、そのようには機能していない。(移動してなくても、優先権により移動していない場合この値はtrueとなる。あるいは、移動していなくても、見えてない障害物が原因だった場合、この値はtrueとなる。)

そして、たとえ、このhasmoved値が正しく機能していたとしても、isPlayableの判定で用いられるのはおかしい。

障害物にぶつかり、移動しなかった場合、どんな加速度であってもisPlayableがfalseを返すことになるから。

最低でも、このことを、検証・修正しないと、このクラスを用いることはできない。

RandomPlayerは、とりあえず動ける方向へランダムに動いていく、というアルゴリズム

もし動けないならば単純な条件から加速度を設定する

先を見ずに場当たり的に、しかもランダムに動くため効率は当然悪い

さて、次にこのRandomPlayerサンプルより強いAI、MrRandTanaka1を作成する。

RandomPlayerは障害物、敵AIの動きはしっかり見てあるので、当然それ以外で優位にたつ。

動ける選択肢から、より良いものを選ぶように、もちろん他のサンプルと同様、基本的な

探索手法を用いれば強くなれるが、とりあえず、ちょっと強くするために、

y軸方向に加速度を+1する選択肢の中から、動けたら、そちらを採用するようにする。

(xはランダムのまま)

そのソースは下記リンク先(まあ、ほぼなにも書き換えてないけど。)

GameStateおよび、Pointも利用しているので、それらを認識できるように

コンパイルしてください。

<https://drive.google.com/open?id=1BjyONj8SHCVAExAumVduWwOuWj5JM4XU>

sampleのコースとして用意されている、11種類の多様なコースがあるので、それに対して

、5回ずつ試行してみた。その結果を次に示す。

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | sample-course | course01 | course02 | course03 | course04 | course05 | course06 | course07 | course08 | course09 | course10 |
| MrRandTanaka1勝利 | 5 | 5 | 0 | 0 | 5 | 0 | 5 | 5 | 5 | 0 | 5 |
| RandomPlayer勝利 | 0 | 0 | 0 | 0 | 0 | 5 | 0 | 0 | 0 | 0 | 0 |
| あいこ | 0 | 0 | 5 | 5 | 0 | 0 | 0 | 0 | 0 | 5 | 0 |

結果から、明らかに強くなったが、いくつか分析と反省を加える。

courseは非常にバラエティ豊かに用意されており、検証する際は、サンプルに用意されているこの11種類で充分分析可能だと分かった。

そして、コースによっては

特に、course06に関しては明らかに、左ポジションの方が有利となる。

そのため、今回は

MrRandTanaka1は全て左ポジションで検証したため、正確な検証とは言えなくなる恐れがる。

今後は、必ず、交互にポジションを入れ替え、1ステージに対して偶数回レースを行うほうがいい。(公式のルールは1ゲーム必ず2試合。)

それでも、明らかに強くなったことは実験結果から言える。(ので、次から気を付けます、みんなも気を付けてください)

course3.course4.course9は全てあいことなった。

これは、コースを見ればわかるように、y座標は均等で左右に長い障害物が存在した場合、x軸についてランダムな今回のアルゴリズムでは、x軸に行ったり来たりして、障害物を超えれずに、どちらもタイムアウトとなったためである。

これを解決する方法の1つとして、y軸で変化できなかった場合、障害物がない方向に、優先的にx軸を加速させればよいだろう。

また、couse5は全敗した。

これは、コースをみるとわかるように、ゴールライン丁度でないとゴールしにくい構造となっており、ひたすらy軸方向に加速するMrRandTanaka1はゴール直前で動けなくなるためである。

その点、サンプルのrandomはy軸を加速させることを優先しないため、純粋な乱数でゴールできる。

これを解決させるためには、y軸を加速することに優先度をおくのでなく、ゴールラインに近づくように優先度をおくとよいだろう。

最後に、もう1つ気づいたことで、MrRandTanaka1もsampleのRandも、障害物を明らかに視野にいれているのに、障害物などにぶつかる場合がある。

これは、正に、すでに述べたGameStateのhasmovedのよくない仕様のせいだと推測される。

hasmovedの判定をisplayableに入れてなければ、視野にいれた障害物にぶつかることは、スピードの出し過ぎで選択肢がなくなった場合しか起こり得ない。

やっぱり、この結果からも、GameStateクラスを利用する前に、厳密に検証したほうがよいだろう。