Robocodeの達人たちが明かす秘訣: 敵の動きを追跡する

敵に応じて最善の移動アルゴリズムを選択する

David McCoy
Writer
Independent

2002年 5月 01日

どの標的合わせシステムにも限界があります。どんな場合にも、予測するのが難しい移動パターンが存在するからです。この記事から、敵に合わせて最善の移動アルゴリズムを自分のロボットに選択させる方法を学んでください。

この記事では、過去のパフォーマンスに基づいて移動状態を選択する技法を紹介します。物事をできるだけシンプルにするために、実例には、データの長期的な保持は含めません (もっとも、その機能を追加するのに、それほど多くの労力はかかりません)。このヒントの範囲は、コードの概要を説明するものにとどめるつもりです。クラスやメソッドのさらに詳細な説明を知りたい場合は、ソース・コードのコメント文書を参照してください。ソース・コード全体は、参考文献から入手できます。

数学のユーティリティー・クラス

典型的なロボットは、いくつかの数学的なアルゴリズムに大きく依存しています。そこで、共通に利用するアルゴリズムについて、staticメソッドを持つユーティリティー・クラスを作成するのは良い習慣です。

この例の場合のユーティリティー・クラスは、`BotMath` クラスです。このクラスには、1つのメソッド`calculateDamage()` が含まれており、これは、弾丸のパワーに基づいて弾丸によるダメージを計算するメソッドです。

AdvancedRobotクラスを拡張してパブリッシャー/サブスクライバーのサポートを提供する

状態、状態マネージャー、およびその他のサポート・クラスをロボットに追加すると、さまざまなイベントにアクセスする必要が生じます。それらのイベントを状態マネージャーに渡して、そこから個々の状態にイベントを渡してもらうという方法もありますが、もっと効率的なのは、関係のあるイベントだけを、それを必要とするオブジェクトに直接渡すという方法です。

`ExtendedRobot` クラスでは、関心のあるイベントだけをオブジェクトが受信するように、オブジェクトを登録することができます。 `EventRegistry` クラスには、さまざまなイベントへの登録

に使用するための定数が含まれています。 `EventListener` 抽象クラスは、それらのイベントを受信するのに必要なメソッドを定義しています。

Robocodeの標準のイベントに加えて、 `ExtendedRobot` クラスは、状態マネージャーと状態の両方が必要とする3つのイベントを生成します。 `enable`、`disable`、および`execute` イベントは、登録されたすべての状態マネージャーに直接渡され、それらの状態マネージャーはそのイベントをアクティブな状態に渡す役割を果たします。 `enable()` メソッドは、各ラウンドの最初に呼び出されます。このメソッドの実装では、変数を初期化し、関係のあるRobocodeイベントに対して登録する必要があります。 `disable()` メソッドは、各ラウンドの最後に呼び出されます。このメソッドの実装では、リソースを解放し、クラスが現在登録しているすべてのRobocodeイベントに対して登録を解除する必要があります。 `execute()` メソッドは、各動作 (turn) ごとに呼び出され、状態マネージャーおよび状態が、そのような動作を前提としたアルゴリズムを実行できるようにします。これらのイベントを受信する必要のあるクラスは、 `CommandListener` インターフェースを実装しなければなりません。

状態を管理する

`StateManager` クラスは、最善の状態を選ぶ役割を果たします。各動作ごとに、そのクラスの`execute()` メソッドは、現在アクティブな状態の`isValid()` メソッドを呼び出します。その状態の`isValid()` メソッドが偽を返す場合は、新しい状態が選択されます (このプロセスは、敵の数に基づいて状態を切り替える場合などに利用できます)。

新しい状態を選択することに関係する大部分の作業は、 `StateManager` の`selectNewState()` メソッドによって処理されます。このメソッドは、候補に上がっている各状態から統計データを要求し、そのデータを評価して、最善の状態を選択します。 `StateManager` は、2段階のアプローチによって最善の状態を選択します。まず最初に、勝敗率が最高の状態を使用しようとします。適切な状態が見つからない場合、`StateManager` は、時間あたりのダメージの割合が最も低い状態を選択します。

状態

`State` 抽象クラスは、各状態の実装に必要なメソッドを定義します。それには、標準的な`enable()`、`disable()`、および`execute()` メソッドに加えて、状態マネージャーと通信するための特化されたメソッドが含まれます。 `getStatistics()` メソッドは、その状態の過去のパフォーマンスに関するデータの入った`Statistics` オブジェクトを返します (この例では、状態は単一の`Statistics` オブジェクトを管理しますが、敵ごとの統計を検索するための`HashMap` を使用するように容易に変更できます)。 `getName()` メソッドは、デバッグ・メッセージで使用する、その状態のわかりやすい名前を返します。 `isValid()` メソッドは、その状態をいつ使用するべきかを判別する役割をします。このメソッドは、状態の選択の際と、状態が選択された後の各動作の際の両方で呼び出されます。

各`State` クラスは、その状態がどれほど適切に機能しているかを追跡し、そのデータを`Statistics` オブジェクトに記録する働きをします。

この記事のソース・コードには、2つの簡単なサンプル状態を含めておきました。それは、`State` 抽象クラスのサンプル実装を提供する、 `TrackState` クラスと`CannonFodderState` クラスです。

統計

`Statistics` クラスは、状態がどれほど適切に機能しているかを追跡するための、簡単なコンテナー・クラスです。ダメージ率 (時間あたりのダメージ)、その状態が使用されている時に出会った数、負けの数、および勝ちの数を表す変数が含まれています。さらに、このデータを設定および取得するための3つのヘルパー・メソッドも含まれています。 `update()` メソッドは、`Statistics` オブジェクト内の変数を設定する簡単な手段を提供します。 `getDamageRatio()` および`getWinLossRatio()` メソッドは、状態の統計データを返します。

効果的であり得るはずの状態が、1回の悪いラウンドのために不適格と見なされてしまうことを防止するため、状態が最低でも3回の出会いを経過するまでは、統計データは有効と見なされません (この評価期間のあいだ、 `Statistics` オブジェクトは既定のデータ値を返します)。

すべてをまとめる

ロボットのメイン・クラスは、状態マネージャーと状態を設定する作業のほとんどを実行します。各ラウンドごとに、状態マネージャーを作成し、状態マネージャーに状態を追加し、状態マネージャーをコマンド・リスナーとして登録し、 `ExtendedRobotenable()` メソッドを呼び出してコマンド・リスナーを活動化する必要があります。さらに、ロボットのメイン・クラスでは、すべての動作ごとに、 `ExtendedRobotexecuteTurn()` メソッドを呼び出します (このメソッドは、すべての登録されたコマンド・リスナーの`execute()` メソッドを呼び出します)。各ラウンドの最後に、ロボットのメイン・クラスは、 `disable()` メソッドを呼び出して、コマンド・リスナーを正常にシャットダウンしなければなりません。

複数の状態のあるロボットの場合、典型的な`run()` メソッドは、次のような構成になります。

```
    public void run() {
        try {
            // Set up and enable the state manager
            StateManager navigation = new StateManager(this);
            navigation.addState(new CannonFodderState(this));
            navigation.addState(new TrackState(this));
            addCommandListener(navigation);
            enable();
            // Set turret to move independent of body
            setAdjustGunForRobotTurn(true);
            // Main bot execution loop
      while(true) {
                // Spin gun
                setTurnGunRightRadians(Math.PI);
                // Allow StateManager to do it's thing
       executeTurn();
                // Finish the turn
                execute();
      }
        } finally {
            // Disable the State Manager
            disable();
        }
    }
```

この技法は、これですべてです。この技法を実装するのが複雑な作業になることはわかっていますが、2つの大きな継続的な利点が提供されます。1つには、アーキテクチャーが非常に拡張性の高いものになります。状態を追加するだけで、機能を拡張できるからです。もう1つの点として、

いつでも最善の状態を選べるという利点により、ロボットがはるかに動的なものになります。したがって、負けにくくなることは言うまでもありません。

ダウンロード

内容	ファイル名	サイズ
Source code	j-movementstate.zip	13KB

著者について

David McCoy

David McCoyは、8か月前に同僚からRobocodeを紹介されて以来、すぐにそのとりこになりました。彼の競技用ロボットdroid.PrairieWolfは、Gladiatorial Leagueの混戦を呈する競技会で首位にランクされたことがあります。Davidの連絡先は、dsmccoy@velocitus.net です。