ロボットと迷路、そしてサブサンプション・アーキテクチャー

Java 言語で仮想ロボットをプログラミングする

Paul D. Reiners
Computer Programmer
IBM

2007年 12月 04日

ロボット・シミュレーターは本格的な研究ツールであると同時に、IBM のコンピューター・プログラマー Paul Reiners がこの記事で示すように、Java™ プログラミングを使った本格的な楽しみへの道でもあります。光を求めて迷路を進む仮想ロボットを、Java 言語で Simbad を使って作成する方法を学びましょう (Simbad は Java の 3D 技術をベースにするオープンソースのロボット・シミュレーターです)。そしてサブサンプション・アーキテクチャーによるロボット設計の概念を実現する方法を学びましょう。

はじめに

ロボティクスは、はるか昔に SF の世界を去り、産業自動化や医療、宇宙探査、その他のアプリケーションの進歩を推進しています。ソフトウェアによるロボット・シミュレーターは、ロボティクスのエンジニアの開発作業を単純化するだけではなく、研究者にとって AI (artificial intelligence: 人工知能) アルゴリズムや機械学習を研究するためのツールにもなります。そうした、研究に焦点を当てたシミュレーターの 1 つが、Java の 3D 技術の上に構築された、オープンソースの Simbad プロジェクトです (「参考文献」を参照)。この記事では、Simbad ツールキットを使って仮想ロボットをプログラミングし、またサブサンプション・アーキテクチャーによるロボット設計の概念を実現する方法を説明します。

この記事はまず、ロボティクスの概要を簡単に説明することから始め、そしてサブサンプション・アーキテクチャーの概念を説明します。次に、Simbad ツールキットを紹介し、Simbad の中にサブサンプション・アーキテクチャーを実装する方法を説明します。さらに、このアーキテクチャーを使って単純なロボットをプログラミングします。最後に、素晴らしい迷路の世界について説明し、2 つ目のロボットをプログラミングします。このロボットは Homer Simpson (訳注: Homer Simpson は米国で人気のアニメ、「シンプソンズ」の登場人物、「参考文献」を参照) とは異なり、迷路から抜け出すための道を見つけることができます。これらは物理的なロボットではありませんが、Simbad の仮想世界に「住む」ロボットです。

ロボットのプログラミング

ロボットという言葉には、一般的に受け入れられている定義はありません。この記事では、ロボットは下記の 3 つの部分を持つものと見なします。

- センサーの集合
- そのロボットの行動を定義するプログラム
- アクチュエーターとエフェクターの集合

従来のロボティクス

従来のロボティクス (つまり 1986年以前のロボティクス) では、ロボットは中枢となる「頭脳」を持ち、その頭脳が世界の「マップ」を構築して維持し、そのマップに基づいて行動計画を立てるものでした。まず、ロボットのセンサー (例えばタッチ・センサーや光センサー、超音波センサーなど) が、ロボットを取り巻く環境から情報を取得します。ロボットの頭脳は、ロボットのセンサーから収集したすべての情報を融合し、そのロボットの世界のマップを更新します。そしてロボットは行動方針を決定します。ロボットはアクチュエーターとエフェクターによって動作します。アクチュエーターは基本的にはモーターであり、エフェクターに接続されていて、エフェクターがロボットを取り巻く環境とのインターフェースを取ります。エフェクターの例としては車輪やアームがあります (アクチュエーターという言葉は、曖昧な使い方をされることがよくあり、アクチュエーターあるいはエフェクターのいずれかを意味することがあります)。

簡単に言えば、従来のロボットは、(場合によっては複数の) センサーからの入力を取得し、このセンサーの情報を融合して世界のマップを更新し、世界に対する現在のビューに基づいて行動計画を立てて、行動します。ただしこの方法には、問題があります。その 1 つは、非常に計算負荷が重いことです。他にも、外部の世界は常に変化しているため、世界のマップを最新の状態に維持することは困難であるという問題もあります。さらに言えば、外部世界のマップや記憶すらなくても、多くの生命体 (例えば昆虫など) は繁栄していることを考えれば、この生命体をエミュレートした方が適切ではないでしょうか。こうした問題から、BBR (behavior-based robotics: 行動規範型ロボティクス) と呼ばれる新しいスタイルのロボティクスが生まれました。BBR は今日のロボティクス研究において、おそらく最も支配的な概念です。

サブサンプション・アーキテクチャー

BBR はサブサンプション (subsumption) アーキテクチャーを使って実装することができます。サブサンプション・アーキテクチャーの発明者は、現在は MIT の AI 研究所の所長である Rodney A. Brooks です。彼は 1986年に画期的な彼の論文「Elephants Don't Play Chess」の中で、このアーキテクチャーを紹介しました (「参考文献」を参照)。行動規範型ロボットは、独立した単純な一連の行動 (behavior) から構成されています。行動は、その行動をトリガーするもの (通常はセンサーの読み取り値) と、取られる行動 (通常はエフェクターが関係します) によって定義され、階層として他の行動の上に積み重ねられます。2 つの行動が競合する場合には、中央のアービトレーターがどちらの行動を優先するかを決定します。ロボット全体としての行動は創発的 (emergent) であり、BBR の推進者によれば、部分の総和を超える可能性があります。上位レベルの創発的な行動は、下位レベルの行動を包摂 (subsume) します。ここでは実際にロボットを設計するのではなく、単に行動を追加して何が創発されるのかを調べてみましょう。

Simbad: ロボットのシミュレーション環境

レゴ マインドストーム (LEGO Mindstorms)

この記事はソフトウェアによるロボットの作成に焦点を当てていますが、物理的なロボットを作成してみたい場合には、レゴ マインドストームという素晴らしいロボティクスのキットがあります。

レゴ マインドストームを扱うレゴ本社には、「We will do for robotics what iPod did for music. (我々は iPod が音楽に対して行ったことをロボティクスに対して行う)」というスローガンが貼られています。レゴ社はマインドストームというロボティクスのキットの最初のバージョンを 1998年に発売しました。このキットは即座に大人気となり、レゴ社の予想を上回るほどの売れ行きでした。250 ドルという値段は少し高価に思えるかもしれませんが、これは iPod Classic の値段であり、そして今や誰もが iPod を持っているのです。

しかし iPod はマインドストームのように改造することはできません。マインドストームが最初に発売されると間もなく、ハードウェア・ハッカーは マインドストーム RCX のブリック (マインドストーム・ロボットの「頭脳」) を分解し、リバース・エンジニアリングを始めたのです。レゴ社はこれを予想しておらず、それを許すべきなのか、あるいは止めさせるための文書を送るべきなのか判断に迷いました。称賛すべきことにレゴ社の上層部は、マインドストームのハッカー達が好きなように改造することを許す、と決定したのです。

その結果、マインドストームのコミュニティーが活況を呈しています (「参考文献」を参照)。マインドストームには、NXT-G と呼ばれるドラッグ・アンド・ドロップ方式のグラフィカル・プログラミング言語しか付属していませんが、ソフトウェア・ハッカー達は、すぐに C や Java 言語など他の言語を マインドストームに移植し始めました。おかげで、マインドストーム キットの 50% は大人が使用しているものと推定されています。

Simbad を利用すると、ソフトウェアでロボットをシミュレートすることができます。このプロジェクトの Web サイトによると、「プログラマーは Simbad によって、独自のロボット・コントローラーの作成や、環境の変更ができ、また用意されているセンサーを使うことができます。Simbad は主に、自律ロボットや自律エージェントを使って、状況依存型 AI や機械学習、さらにはもっと一般的な AI アルゴリズムを学ぶための簡単なベースとなるものを求める研究者やプログラマーを対象にしています。」

Simbad は Louis Hugues と Nicolas Bredeche によって、Java 言語で作成されています。このプロジェクトは SourceForge.net がホストしており、GNU General Public License の下で自由に使用し、変更することができます。

技術的な詳細

Simbad の世界は、エージェント (ロボット) と、動かないオブジェクト (箱や壁、光など) を含むことができます。Simbad の世界での時間は、離散的なティックに分割されています。Simbad はエージェント間でのタイムシェアリングによってスケジューリングします。Simbad のエージェントは物理的なロボット同様、(距離、接触、光などの) センサーと、(通常は車輪による) アクチュエーターとを持っています。ロボットはティックごとに行動することができます。

エージェントは `performBehavior()` メソッドをオーバーライドして行動を決定します。`performBehavior()` では、ロボットはセンサーの値の 1 つを読み取り、並進速度と回転速度を設定します。`performBehavior()` は瞬時に行われるため、「1 メートル前進」などのコマンドを発行することはできません。この制約を回避するため、通常はそのロボットの状態を追跡する必要があります。また現在の状態をクロックの何ティック分維持しているのか追跡するために、タイマー変数を使う必要があるかもしれません。

Simbad の API

この記事の演習では、Simbad の API パッケージのうち下記の 2 つを主に扱います。

- **`simbad.sim`**: このパッケージのクラスは、ロボットと、そのロボットが住む世界の両方を表します。この中には (主なものとして) 下記が含まれます。
  - `Agent`: Agents はロボットです。
  - `Arch`: ロボットが周りを回れるアーチ、またはロボットが下をくぐれるアーチ。
  - `Box`: ロボットの世界の障害物として使用することができます。
  - `CameraSensor`: ロボットの世界をロボットの視点から見ることができます。
  - `EnvironmentDescription`: ロボットと (壁や箱などの) オブジェクトを追加する対象となる「世界」を表します。
  - `LampActuator`: ロボットに追加できるランプを表します。
  - `LightSensor`: 光の強さを感知します。
  - `RangeSensorBelt`: ロボットの周囲にある一連の範囲センサーを含んでいます。
  - `RobotFactory`: これを使ってロボットにセンサーを追加します。
  - `Wall`: ロボットに対する、別のタイプの障害物。
- **`simbad.gui`**: このパッケージのクラスはロボットの世界を表示し、またこれらのクラスを使うとロボットの世界をコントロールすることができます。この中には (主なものとして) 下記が含まれます。
  - `Simbad`: ロボットの世界とセンサー入力、そしてコントロールを表示するフレーム

Simbad でサブサンプション・アーキテクチャーを実装する

ルンバ (Roomba)

私がこの記事を書いている間、ルンバは私の足元のじゅうたんを掃除しています (そうしている間、知らないうちに子猫がルンバにそっと近づいています)。ルンバは、MIT の卒業生 3 人 (Rodney Brooks と Colin Angle、そして Helen Greiner) によって設立された会社、iRobot で開発されました。ルンバはサブサンプション・アーキテクチャーを使って構築されています。またオープンなインターフェースを持っているため、そのインターフェースを利用して、あらゆる種類の興味深い改造を行うことができます。Tod E. Kurt による著書『Hacking Roomba』は、興味深い改造方法を数多く紹介しています (「参考文献」を参照)。

Simbad でサブサンプション・アーキテチャーの実装を開始するためには、`BehaviorBasedAgent` と呼ばれる、`Agent` のサブクラスを定義します。`BehaviorBasedAgent` は `Behavior` の配列と、どの `Behavior` が他のどの `Behavior` を抑制するかを指定する、`boolean` のマトリックスを含んでいます。

```
private Behavior[] behaviors;
private boolean suppresses[][];
```

`BehaviorBasedAgent` は `Behavior` のスケジューラーとして動作します。リスト 1 は、(`currentBehaviorIndex` クラス変数を使って次の順番の行動を追跡することで) 行動を反復し、またそれらの行動の調停を行うコードを示しています。

リスト 1. 反復行動と調停行動のコード

```
protected void performBehavior() {
   boolean isActive[] = new boolean[behaviors.length];
   for (int i = 0; i < isActive.length; i++) {
      isActive[i] = behaviors[i].isActive();
```

```
   }
   boolean ranABehavior = false;
   while (!ranABehavior) {
      boolean runCurrentBehavior = isActive[currentBehaviorIndex];
      if (runCurrentBehavior) {
         for (int i = 0; i < suppresses.length; i++) {
            if (isActive[i] && suppresses[i][currentBehaviorIndex]) {
               runCurrentBehavior = false;

               break;
            }
         }
      }

      if (runCurrentBehavior) {
         if (currentBehaviorIndex < behaviors.length) {
            Velocities newVelocities = behaviors[currentBehaviorIndex].act();
            this.setTranslationalVelocity(newVelocities
                  .getTranslationalVelocity());
            this
                  .setRotationalVelocity(newVelocities
                        .getRotationalVelocity());
         }
         ranABehavior = true;
      }

      if (behaviors.length > 0) {
         currentBehaviorIndex = (currentBehaviorIndex + 1)
               % behaviors.length;
      }
   }
}
```

`performBehavior()` が `simbad.sim.Agent.performBehavior()` をオーバーライドしていることに注意してください。

`Behavior` には下記の 2 つの抽象メソッドがあります。

- **`isActive()`** は、センサーの現在の状態を基に、アクティブであるかどうかに応じてブール値を返します。(すべての Behavior は一連のセンサーを共有します。)
- **`act()`** は、並進速度と回転速度という速度のセットを (この順序で) 返します。これらの速度は、この行動がモーターに対して要求する動作を示します。

例: 光を求めて動き回るロボット

さて、下記 の 4 つの `Behavior` を持つサンプル・ロボットを作成しましょう (4 つの Behavior の優先順位は降順です)。これをリスト 2 からリスト 5 に示します。(この記事で使用しているソース・コードをダウンロードしてください。)

- `Avoidance`: 衝突の後で方向を変更するか、あるいは衝突を回避します。
- `LightSeeking`: 光に向かって移動します。
- `Wandering`: 時々、ランダムに方向を変更します。
- `StraightLine`: 直線的に移動します。

リスト 2. Avoidance クラス (Simbad の SingleAvoiderDemo.java デモ・コードをベースにしています)

```
public boolean isActive() {
   return getSensors().getBumpers().oneHasHit()
```

```
         || getSensors().getSonars().oneHasHit();
}

public Velocities act() {
   double translationalVelocity = 0.8;
   double rotationalVelocity = 0;
   RangeSensorBelt sonars = getSensors().getSonars();
   double rotationalVelocityFactor = Math.PI / 32;
   if (getSensors().getBumpers().oneHasHit()) {
      // if ran into something
      translationalVelocity = -0.1;
      rotationalVelocity = Math.PI / 8
            - (rotationalVelocityFactor * Math.random());
   } else if (sonars.oneHasHit()) {
      // reads the three front quadrants
      double left = sonars.getFrontLeftQuadrantMeasurement();
      double right = sonars.getFrontRightQuadrantMeasurement();
      double front = sonars.getFrontQuadrantMeasurement();
      // if obstacle near
      if ((front < 0.7) || (left < 0.7) || (right < 0.7)) {
         double maxRotationalVelocity = Math.PI / 4;
         if (left < right)
            rotationalVelocity = -maxRotationalVelocity
                  - (rotationalVelocityFactor * Math.random());
         else
            rotationalVelocity = maxRotationalVelocity
                  - (rotationalVelocityFactor * Math.random());
         translationalVelocity = 0;
      } else {
         rotationalVelocity = 0;
         translationalVelocity = 0.6;
      }
   }

   return new Velocities(translationalVelocity, rotationalVelocity);
}
```

リスト 3. LightSeeking クラス (Simbad の LightSearchDemo.java デモ・コードをベースにしています)

```
public boolean isActive() {
   float llum = getSensors().getLightSensorLeft().getAverageLuminance();
   float rlum = getSensors().getLightSensorRight().getAverageLuminance();
   double luminance = (llum + rlum) / 2.0;

   // Seek light if it's near.
   return luminance > LUMINANCE_SEEKING_MIN;
}

public Velocities act() {
   // turn towards light
   float llum = getSensors().getLightSensorLeft().getAverageLuminance();
   float rlum = getSensors().getLightSensorRight().getAverageLuminance();
   double translationalVelocity = 0.5 / (llum + rlum);
   double rotationalVelocity = (llum - rlum) * Math.PI / 4;

   return new Velocities(translationalVelocity, rotationalVelocity);
}
```

リスト 4. Wandering クラス

```
public boolean isActive() {
   return random.nextDouble() < WANDERING_PROBABILITY;
}

public Velocities act() {
   return new Velocities(0.8, random.nextDouble() * 2 * Math.PI);
}
```

リスト 5. StraightLine クラス

```
public boolean isActive() {
   return true;
}

public Velocities act() {
   return new Velocities(0.8, 0.0);
}
```

リスト 6. は、どの行動が、他のどの行動を抑制するかを指定しています。

リスト 6. 行動全体に関する抑制を指定する

```
private void initBehaviorBasedAgent(BehaviorBasedAgent behaviorBasedAgent) {
   Sensors sensors = behaviorBasedAgent.getSensors();
   Behavior[] behaviors = { new Avoidance(sensors),
         new LightSeeking(sensors), new Wandering(sensors),
         new StraightLine(sensors), };
   boolean subsumes[][] = { { false, true, true, true },
         { false, false, true, true }, { false, false, false, true },
         { false, false, false, false } };
   behaviorBasedAgent.initBehaviors(behaviors, subsumes);
}
```

この例の `Behavior` は (抑制に関して) 総合的な優先順位を持っていますが、必ずしもそうである必要はないことに注意してください。

演習問題として、下記を試してみてください。

- 社交的な行動を追加する: 友達の方には進み、敵からは離れるように移動します。
- 光を回避する行動を追加する。
- 一部のロボットから光を発するようにし、それらのロボット同士が引きつけ合うようにする。

迷路

> 「ついにやった。Tremaux のアルゴリズムを使えば、あの迷路が解けることはわかっていたのだ。」・・ Lisa Simpson (訳注: Lisa Simpson も米国で人気のアニメ、「シンプソンズ」の登場人物)

迷路を解決する既存のアルゴリズムはいくつかありますが、そのうち一般的な 2 つは、迷路の規模によらず一定量のメモリーを消費します。この 2 つは壁沿いに移動する (wall-following) アルゴリズムと、(英国の Exeter の Jon Pledge が 12 歳の時に発明した) Pledge アルゴリズムです。Tremaux のアルゴリズム (Lisa Simpson のお気に入りのアルゴリズム) は優れたアルゴリズムですが、この記事では単純にするために壁沿いに移動するアルゴリズムと Pledge アルゴリズムに絞ることにします。

迷路生成アルゴリズム

迷路を解決するアルゴリズムも多数ありますが、迷路を生成するアルゴリズムも多数あります。この記事で取り上げている迷路は、単純迷路 (perfect maze) と呼ばれるものです。単純迷路は、迷路の任意の地点から他の任意の地点まで、厳密に 1 つのルートしか持ちません (これによって、ループや「島」(他から切り離されて孤立した部分) を持つ迷路は除外されます)。単純迷路を生成する大部分のアルゴリズムは、周囲の壁のみを持ち、区分ごとに内側に「成長する」壁を持つ迷路から開始します。迷路を確実に完全なものにするためには、新しい区分を追加するごとに、ループまたは他から切り離されて孤立した部分を作成していないかどうかを確認する必要があります。

壁沿いに移動するアルゴリズム

壁沿いに移動する方法は単純な迷路アルゴリズムであり、皆さんも子供の頃に学んだかもしれません。このアルゴリズムを使って迷路を解くために必要なことは、左手を左の壁に (あるいは右手を右の壁に) 触れたまま、迷路を出るまで壁を伝い続けることだけです。迷路の入り口と出口が迷路の外枠にある場合には、このアルゴリズムが必ず有効であることは容易にわかります。しかし、もしゴールが「島」(迷路の中で、他の部分から切り離された部分) の中にある場合には、このアルゴリズムでは壁を乗り越えて島に飛び込むことはできないため、解を見つけることはできません。

Pledge アルゴリズム

Pledge アルゴリズムは、島と島の間を飛び越えることができるため、壁沿いに移動する方法よりも高度な方法であり、また大規模な迷路を解決することができます。Pledge アルゴリズムの基本的な考え方は、絶対方向 (例えば東西南北など) を選び、常にその方向に向かおうと努力することです。この方向を優先方向 (favored direction) と呼ぶことにします。壁に突き当たった場合には、右に方向転換し、左の壁沿いに移動し、これを方向が優先方向となり、なおかつ、(時計回りの方向転換をマイナス、反時計回りの方向転換をプラスとして) 方向転換の和の合計がゼロになるまで続けます。この条件を満足する地点に達したら再度優先方向に直進する、というように続けます。方向転換の和の合計がゼロであるという要件は、例えば大文字の G のような形のループ (どういう意味か、実際に紙に書いて試してみてください) など、ある種のループに入らないようにするために必要です。

Algernon: 迷路を解決するロボット

では、Algernon という名前の迷路 (maze) 解決ロボットを作成して、皆さんの友達を驚かせ (amaze) ましょう (訳注: 著者は maze と amaze をかけているようです)。

ロボットを設計する

壁沿いに移動するアルゴリズムあるいは Pledge のアルゴリズムを実装するためには、ロボットがいつ分岐点に達したか、そして実際に分岐点に達したら、どの方向の経路に進むのかを知る必要があります。

これを実現するための方法は複数あると思いますが、ここではこれを、ロボットの左側にソナー・センサーを取り付けることで実現します。このセンサーは、迷路に左向きの経路があることを発見すると知らせます。またロボットが進んでいる経路が終わる (つまりロボットが壁に「突き当たる」) ことを知るために、ロボットの前面にタッチ・センサーを取り付けます。

壁沿いに移動するアルゴリズムをプログラミングする

ここでは `algernon.subsumption` パッケージ (ソース・コードをダウンロードしてください) を使って Algernon をプログラミングします。Algernon はロボットとしては非常に簡単なものであり、単純に「手続き型」の方法でプログラムすることができます。しかし、サブサンプション・プログラミングを使うと、たとえこのような単純なロボットの場合でも、コードをずっと簡潔で理解しやすいモジュール構造にすることができます。

アルゴリズムの実装を単純にするために、壁は直角に配置されているものとします。従って、このロボットが行う方向転換は、すべて 90 度の左折または 90 度の右折です。

左手の壁沿いに移動するアルゴリズムを考える場合には、このアルゴリズムを下記の 4 つの行動に分解することができます。

- 直進する
- 壁に突き当たったら右に曲がる
- 左手に経路が見えたら左に曲がる
- ゴールに達したら停止する

この 4 つの行動の優先順位を決める必要があります。このリストでの正しい優先順位は降順です。すると、それぞれが `Behavior` を継承する、下記の 4 つのクラスが得られます。

- `GoStraight`
- `TurnRight`
- `TurnLeft`
- `ReachGoal`

リスト 7 は `GoStraight` のコードです。`TRANSLATIONAL_VELOCITY` は定数であり、0.4 に設定されています。

リスト 7. 直進のための Behavior コード

```
public boolean isActive() {
   return true;
}

public Velocities act() {
   double rotationalVelocity = 0.0;

   return new Velocities(TRANSLATIONAL_VELOCITY, rotationalVelocity);
}
```

リスト 8 は `TurnRight` のコードです。`getRotationCount()` は、使用している回転速度で 90 度回転するために必要なクロックのティック数を返します。

リスト 8. 右折のための Behavior コード

```
public boolean isActive() {
   if (turningRightCount > 0) {
      return true;
   }

   RangeSensorBelt bumpers = getSensors().getBumpers();
   // Check the front bumper.
```

```
   if (bumpers.hasHit(0)) {
      backingUpCount = 10;
      turningRightCount = getRotationCount();

      return true;
   } else {
      return false;
   }
}

public Velocities act() {
   if (backingUpCount > 0) {
      // We back up a bit (we just ran into a wall) before turning right.
      backingUpCount--;

      return new Velocities(-TRANSLATIONAL_VELOCITY, 0.0);
   } else {
      turningRightCount--;

      return new Velocities(0.0, -Math.PI / 2);
   }
}
```

Algernon が左に曲がるときには、最初に少し前進し、Algernon の後ろが (Algernon の左側で終わっている壁から) 離れるようにします。そしてその後で左に回転します。最後に、さらにもう少し前進し、壁が再度 Algernon の左側になるようにする必要があります。リスト 9 は `TurnLeft` のコードです。

リスト 9. 左折のための Behavior コード

```
public boolean isActive() {
   if (postGoingForwardCount > 0) {
      return true;
   }

   RangeSensorBelt sonars = getSensors().getSonars();
   // Check the sonar on the left.
   if (sonars.getMeasurement(1) > 1.0) {
      // There is a passageway to the left.
      preGoingForwardCount = 20;
      postGoingForwardCount = 40;
      turnLeftCount = getRotationCount();

      return true;
   } else {
      return false;
   }
}

public Velocities act() {
   if (preGoingForwardCount > 0) {
      preGoingForwardCount--;

      return new Velocities(TRANSLATIONAL_VELOCITY, 0.0);
   } else if (turnLeftCount > 0) {
      turnLeftCount--;

      return new Velocities(0.0, Math.PI / 2);
   } else {
      postGoingForwardCount--;

      return new Velocities(TRANSLATIONAL_VELOCITY, 0.0);
   }
}
```

リスト 10 は `ReachGoal` のコードです。

リスト 10. ゴールに到達するための Behavior コード

```
public boolean isActive() {
   RangeSensorBelt sonars = getSensors().getSonars();

   // Is there open space all around us? That is, are we out of the maze?
   double clearDistance = 1.2;
   return sonars.getMeasurement(0) > clearDistance
         && sonars.getMeasurement(1) > clearDistance
         && sonars.getMeasurement(3) > clearDistance
         && sonars.getMeasurement(2) > clearDistance;
}

public Velocities act() {
   // Stop
   return new Velocities(0.0, 0.0);
}
```

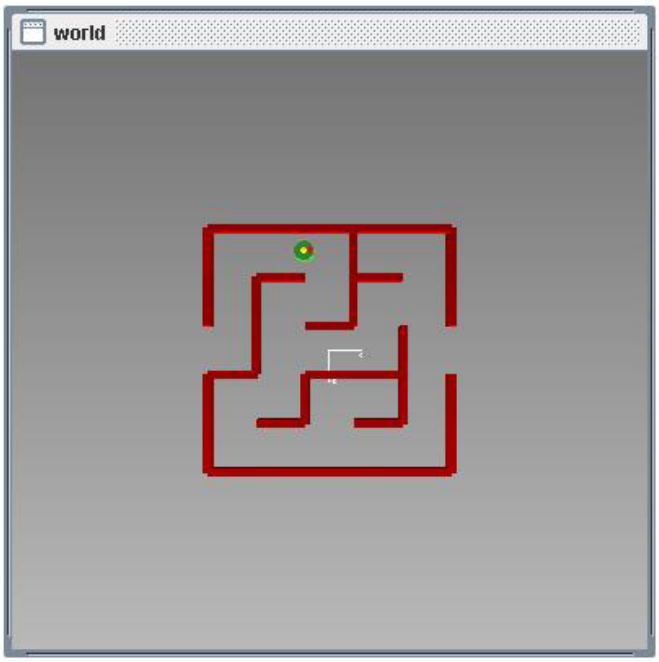
リスト 11 は Algernon のメインの行動のためのコードです。

リスト 11. Algernon の行動を制御するコード

```
private void initBehaviorBasedAgent(
      algernon.subsumption.BehaviorBasedAgent behaviorBasedAgent) {
   algernon.subsumption.Sensors sensors = behaviorBasedAgent.getSensors();
   algernon.subsumption.Behavior[] behaviors = { new ReachGoal(sensors),
         new TurnLeft(sensors), new TurnRight(sensors),
         new GoStraightAlways(sensors) };
   boolean subsumes[][] = { { false, true, true, true },
         { false, false, true, true }, { false, false, false, true },
         { false, false, false, false } };
   behaviorBasedAgent.initBehaviors(behaviors, subsumes);
}
```

図 1 は Algernon が迷路を進んでいる様子を示しています。

図 1. Algernon が迷路を進む



このロボットが、ロボットのコンポーネント部分は迷路に関して (さらには壁に関してさえ) 何も知らないにもかかわらず、迷路を解決できることに注意してください。Algernon は、迷路を抜け出す方法を計算するための中央頭脳は何も持っていません。これがサブサンプション・アーキテクチャーの成果です。つまり一連の単純で階層構造の行動から、複雑で、見るからに目的を持っているかのような行動が得られるのです。

まとめ

この記事では、シミュレートしたロボットをプログラミングしました。物理的なロボットのプログラミングは、もっと困難です。物理的な世界では、ありとあらゆる面倒な要素が入り込んできます。例えば、壁沿いに移動するアルゴリズムで動くロボットを左側の壁をたどりながら壁に並行に移動させるのは非常に容易でした。現実の世界では壁の表面が不完全なため、ロボットが急に方向を変えて左側の壁に衝突しないように、あるいは左側の壁から離れすぎないようにすること自体が非常に困難です。プログラミングを楽しむのであれば、必ずしもロボットを作るための作業を楽しむ必要はありません。興味深いロボットを作成するためには、おそらくプログラミングのスキルよりもメカニカルなスキルの方がむしろ必要です。

もし皆さんが、独自のロボットを作成してプログラミングしたい場合には、優れたロボティクスのキットとしてレゴ マインドストームがあります。マインドストームよりも安価なものとして、BEAM (Biological Electronic Aesthetics Mechanics) ロボットを作成する方法があります。BEAM は行動規範型ロボティクスの概念を一歩進め、完全にプログラミングをなくしています。全体としての行動は、ハードワイヤーによる、アナログの反射と応答の行動によって決まります。30 ドル、あるいはそれ以下で、BEAM キットを使って最初のロボットを作成することもできますし、Gareth Branwyn による『Absolute Beginner's Guide to Building Robots』(「参考文献」を参照) などの本に紹介されているさまざまな案を参考にロボットを作成することもできます。あるいはルンバを購入して、それを改造する方法もあります。

私は、ほんの少しの間ロボットをプログラミングし、また他の人のロボット・コードを見た後で、驚くようなことに気付きました。ごくわずかのコードだけで、非常にたくさんのことをロボットに行わせることができるのです。(ただし、その少量のコードを完全に適切なものにするためには、さまざまな定数を何度も変更して実験する必要があるかもしれません。) レゴ マインドストームのキットを利用すれば、午後の時間を使って説明書を読みながら、最初の簡単なロボットを作成することができます。

世の中ではアマチュアのロボット・サブカルチャーが大盛況です。ロボットの本やロボット・コンテスト、ロボットのビデオがあり、そしておそらく皆さんの地域にもロボット・クラブがあるはずです。それらを調べてみるのもよいかもしれません。

著者について

Paul D. Reiners

Paul Reiners は Sun 認定の Java プログラマーであり Java 開発者です。彼は、Automatous Monk や Twisted Life、Leipzig を含む、いくつかのオープンソース・プログラムの開発者です。彼は 1991年5月に University of Illinois at Urbana-Champaign で応用数学 (計算理論) で修士号を取得しています。これは、同じキャンパスで HAL 9000 が初めて電源オンされた時の約 9 カ月前です (電源オンの日は正確には 1992年1月12日です) 。彼はミネソタに住み、時間がある時には電子ベース・ギターを練習し、また従業員によるジャズ・バンドで演奏しています。