動的な環境に対応する BDI エージェントのためのプランナの設計と実装

奈良女子大学大学院人間文化研究科
情報科学専攻 I07-018
藤田 恵

提出年月日：2009 年 1 月 30 日
指導教員：新出 尚之

概要

BDI エージェントはプランを選んで意図を形成するので、動的な環境に適応するには、適切なプランを臨機応変に生成できるプランナが必要となる。本論文では BDI エージェントに適切な、階層的かつ部分的なプランニングや、動的な環境変化に応じた再プラニングを可能とするプランナの設計と実装について述べる。また、将来的には高速化を目指すため最終的には PS3 のセルプロセッサ上で実装を行い、個々の SPE に階層的なプラン生成を独立に実行させ、強化学習によるプラン選択の洗練を行う。

1　はじめに

自律的であり、合理的なエージェントとしてよく用いられる BDI エージェントとは、信念 (Belief)、願望 (Desire)、意図 (Intention) の 3 つの心的パラメータを用いて熟考を行い、達成したい目標に向けて大まかなプランを立てて、それぞれのプランでのサブ目標なる中間目標に対してそれを実現すべく意図を形成し、動的環境にあわせながらそれぞれの目標を達成し、最終的に達成したい目標を達成するというエージェントのことである。

BDI エージェントは意図の概念を明示的に持つので、行動の選択はエージェントの意図によって規定される。従ってエージェントはこれに専念する形で行動する。また必要に応じて意図を破棄することで一貫した行動と動的環境への対処を行っている。

意図はプランを選ぶことで形成されるので、BDI エージェントが適切な行動をするためには適切なプランの選択が必要である。環境は動的であるので、常にそのときの環境に合わせたプランを形成して、実世界で不備なく行動させる必要がある。しかし現実では行動の失敗ということも起こりえる。行動の失敗に対して途中で失敗することを考慮したルートをすべてあらかじめプランして用意しておくのは予測困難なことも多く、大変非効率である。

従来の BDI エージェントの実装では、プランは与えられているということが前提であり、環境が変わったとき、プランの再選択はするが再プランを行うことはなかった。従って、選択可能な代替プランがなくなってしまったら、元のプランの続きが行えるまで待つかあるいは立往生することになる。

このため、適切な行動を行うには再プランを行うことが必要である場合もある。この場合、適切なプランを作成するにはプランナが環境に臨機応変であることが重要である。

我々は BDI エージェントに適切な、動的環境に対応したプランを生成できるプランナの実現を目指しており、本論文ではその設計と実装方針について述べる。また、その設計方針に基づいた動的環境に対応するプランナの実装を進めており、現時点ではこれは後に述べるプランナ、SHOP2 のエンジンを用いている。本論文ではこの実装についても述べる。

プランナの設計方針としては、あらかじめ大まかなプランを作っておき、それぞれにはサブゴールと言われる、その大まかなプランの一つ一つに対するゴールが存在し、それを達成することを目標として、大まかなプランを先頭から順に階層的に分解する。そして、一つ一つの大まかなプランは現在の状態を初期条件としてそれを元に目標を達成しようと試みる前進探索プランナを適用して達成されていくようにする。このとき、エージェントに常に周囲を知覚させることにより、臨機応変に行動しながらも目標を達成させるという手法をとる。

ここで、サブゴールに対する達成手順 (サブプラン) を得る際には、プランの実行の進行に伴って階層的分解により、そのサブゴールの部分のプランだけを構成するので、仮に途中で最終的に達成したい目標が変更されて全体のプランを放棄することになっても、あらかじめ立てられていたプラン全体は大まかな設計のみであるので放棄による無駄な計算は小さく見積もることができる。

また、大まかなプランを立てるという設計により、仮に一連のプランの流れに従ってエージェントが行動していくとき、環境の変化に伴い途中であらかじめ立てた大まかなプランの一部をこなすことなく最終的に達成したい目標が達成できることになった際にも、途中の大まかなプランの内容は最初から計算されていないので省くだけでよく、計算時間の短縮が行えるということができる。

5.1 で述べるように階層的なプランナのアイデアはこれまでにもあるが、我々の目的は BDI エージェントの

行為設定法としてのプランナを構築することにより、目的に向かって一貫したかつ柔軟な行為決定を実現することである。

このアプローチは BDI エージェントの根本的設計である Bratman の意図の原理 [1](大まかなプランを作りそれを意図として階層的にサブゴールを達成するプランを順に埋めていく) を良く反映している。

例として旅行計画を立てる場合を考える (図 1)。目的地に移動して観光して帰宅するというプランを組むとき、それぞれは図のように分解できて、さらに移動手段として考えられる方法がいくつか存在する。このとき、その中から状況により一つ選択して最寄駅に移動する手段を埋めてサブゴールを達成するようにする。

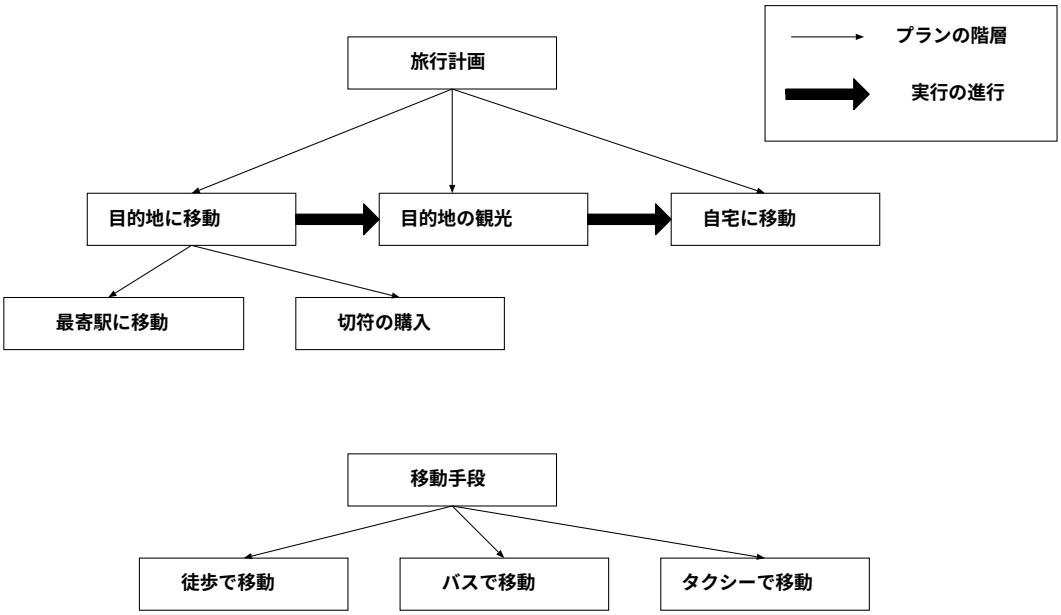

図 1　Bratman の意図の原理を現す簡易図

従来知られている著明なプランナには TLPLAN[2] や SHOP2[6] があるが、BDI エージェントが利用するプランナとしては都合の悪い点がある。それは、これらのプランナが静的な問題向けであり、周囲状況への認識がない状態で目標の最後までプランを組むというプラン作成を目的にして作られているので、BDI エージェントに適用した場合、一通りのプラン全体を実行して失敗してしまったら全体を再プランし直すという効率の悪いものになってしまうという点である。

従って、我々はこれを改良して、常に外部への認識を意識して部分的にプランを組む方針に変えることにより、上述の機能を具体化することにする。

2　BDI アーキテクチャ

先に述べた BDI エージェントのインタプリタは一般的に次のようなループ (図 2) で実装される [7]。

ここで、B は信念、D は願望、I は意図である。

initialize-state を実行することにより、初期状態が与えられ do ループの中に入る。このループの中では、event-queue を読んで、行動のリストである option を生成する。

次に deliberate を用いて最適な行動を option から取り出し、それを意図構造体 I に加える。そして意図 I が原始論理式であれば execute を用いて実行し行動をする。

```
BDI-initerpreter
initialize-state();
do
    option := option-generator(event-queue, B, D, I);
    selected-options := deliberate(options, B, D, I);
    execute(I);
    get-new-external-events();
    drop-successful-attitudes(B, D, I);
    drop-impossible-attitudes(B, D, I);
until quit.
```

図 2　BDI インタプリタ

この後、get-new-exeternal-events により外部の変化を受け取る。ここで成功した目標と成功した意図を捨てて願望と意図を修正し、次に不可能であった目標や実現不可能な意図を捨てて再び願望と意図を修正し整合性を保つ。

これを繰り返すことにより最終的に実現したい願望を達成するという仕組である。

最終的な願望を達成するまでの間、エージェントは多数の副願望を達成していかなければならない。しかし、エージェントは最初から詳細なプランを立てるのではなく最初は最終的な目標までの大まかなプランを立案し、実行の進行に伴って、大まかなプランの一つ一つをより詳細なプランで埋めていく方式をとる。

従って、この構造を具体的に作成する時、階層的プランニングが必要であり、さらに動的環境においては失敗が予想されるのは当然であるから、このことより再プランができることも必要である。

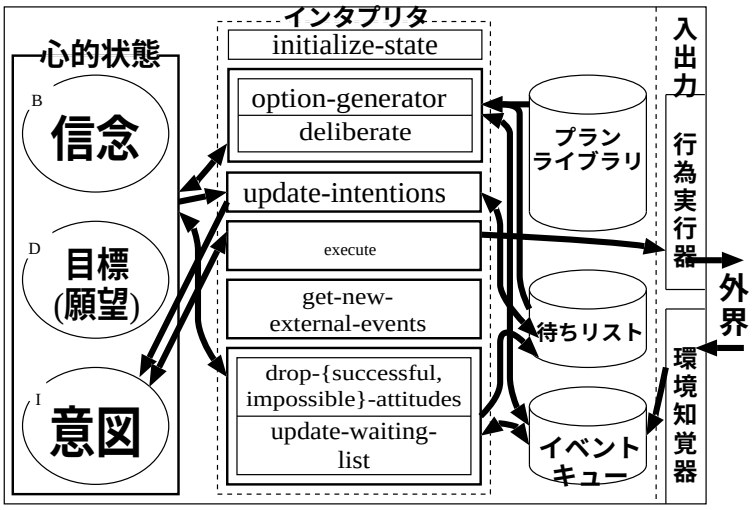

図 3　BDI アーキテクチャの概要図

3　プランナの設計方針と実装

3.1　我々のプランナの基本構造

我々のプランナの構造としては基本、SHOP2 によるプラン生成のようにプランを基本行為の段階まで一度に分解することはせず、SHOP2 でサブプランの列を生成する段階で停止させる。

そうすることにより、問題を小規模なものに分解でき、かなり大まかなプランを作成することができる。そしてその各々の大まかなプラン (サブプラン) をもう一度 SHOP2 に与えることにより、今度は精密に分解していくようにする。ここで、サブプランは前進探索で探索し、毎回周囲の状況を捉えるようにしたいので、サブプランの中身は分解後、基本行為とさらに小さいサブプランの列の形にする。そして基本行為の部分を実行するというようにする。

これを繰り返し、最終的にサブプランがすべて基本行為として実行されてゴールに行き着いたらそのサブプランは成功したことになり、エージェントは次のサブプランを同じ要領でこなしていくという方針をとる。

仮にサブプランの中身の分解した基本行為を実行して失敗した場合にはそこの時点を基準とした外部状況をもとにプランを作成し直すことにより再プランニングを行う。この再プランニングも SHOP2 のエンジンを用いて行う。

このようにサブプランをすべてこなして、最後に大元のプランの目標を達成する手法をとる。

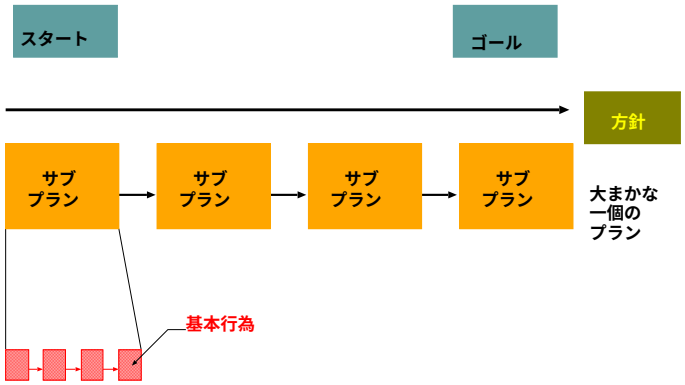

図 4　プランナの模型図

3.2　現在の実装について

我々のプランナは現在のところ、近年開発が盛んなものの一つである SHOP2 のエンジンを用い、これを呼び出す形で実装している。SHOP2 とは “Simple Hierarchical Ordered Planner 2” の略である。

3.2.1　SHOP2 とは

SHOP2 は階層的プランナであり、Common Lisp で記述されている。大元の問題をメソッドと言われる問題に依存した問題の分解方法を使って小さなプランに分解していく。これを繰り返し、最終的に基本行為 (それ以上分解できない最小単位の行為、SHOP2 では原始論理式で表記される) に分解しきったときにプランが完成する。

3.3 実装方針

初めに SHOP2 により大まかなプランを構築し、それ以上分解せずに止める。

その大まかなプランを構成する小さなプランそれぞれをサブプランと呼び、サブプランにはさらに細かい解法をつけ合わせ、これに名前をつけて保存しておく。

そして大まかなプランを実行していく際に、これらサブプランはこの名前で呼び出してさらに細かいプランを作成するようにした。もちろんこの分解を行う際にも SHOP2 のエンジンを使用している。

そしてこの細かいプランを前進探索を行い、外部知覚を行いながらエージェントは行動を行う。

ここで、サブプラン内のプランを実行して失敗した際にはサブプラン単位で再プランを行い、行動をさせて成功すれば次のサブプランに移るようにしている。

3.4 我々のプランナの内部構造

先に説明したように SHOP2 ではすべてのプランを階層的に分解して得るという方式でプランニングを行っているが、このプランナでは SHOP2 での問題の設定にあたる“ドメイン”の記述をするマクロ、解く問題を決定するマクロ、プランを作る関数を、SHOP2 の機能を呼び出す形で Common Lisp の関数およびマクロとして書き直し、この節の最初に述べたようにサブプランの名前とサブプランを解く解法をひとまとめにしたデータを用意できるようにした。

これにより、各サブプランを名前をひくことにより呼び出せるようになった。よって SHOP2 による階層的分解を途中で止めることができ、大まかなプランを立てられるようにした。

また、この名前をプランのスタックに積んでいくことにより、自動的にサブプランは順序どおり実行されるようにしている。そして、仮にサブプランが失敗に終わったときはその名前を消さずにもう一度スタックからポップするという形にして再プランを行っている。

さらに、解く問題を決定するマクロの中には複数の問題を解くという設定が必要なところがあり (4 節で述べる例題ではキウイを三羽捕まえるというところ)、これらは SHOP2 では半順序プランとして扱われるが、本プランナでは状況に応じ、最初に実行することが有利なサブプランから実行されるようにしたいので、ここの記述では別途評価関数を外部から与えこれによって順番を決めて、問題を一つ一つに分解してそれぞれを順にプランを作成する関数にかけてプランを作るということができるようにした。

尚、現在この順番を決める判定は例題内ではキウイとエージェント間の距離を基準として行っているが、実装はすべて機能別に作ってあるので問題によって書き換えれば汎用性はあるものである。

そして、名前とサブプランの解法によるデータからプランを立てるという関数についてであるが、これは、プランのスタックから取り出した名前と名前とサブプランの解法によるデータ内の名前を比較して一致すればそのサブプランに書いてある解法で SHOP2 によりプランを生成するという形でプランを生成している。

これらのマクロや関数を使って SHOP2 の機能を拡張して我々の設計したプランナを作成したわけであるが、実際これを動的な環境において動かすにはこれらのマクロや関数に必ずその動作時の状況を渡す必要がある。

これの実現には状態を記述すべき場所に状態を格納した大域変数を書いておき、マクロを再帰的に呼び出すことにより毎回の状態を評価できるようにして最新の状態をその変数に格納することにより、エージェントが最新の信念の状態を保持できるようにする。

3.5 実装上の問題点

設計として大まかなプランを通して一つの問題を細かくわけて解決するという手法をとっているが、後の例題で述べる通り、一連の動作の流れにまったく関係のない外部知覚 (捕まえる対象物が見えなくなって捕まえられない) などが起こった場合、全体としての再プランを行うことが必要とされるが、現段階の実装では際プランニングを起動する条件付けが困難なためまだ実装をおこなっていない。

実際、一連の動作の流れから予測できない状況は外部知覚やエージェントの立場を考えて考慮できる問題であっても、単にその場合だけで場合分けして再プランを考えても他にも同じような状況に至る場合はあるので、その問題が起こる特有の場合だけの原因を念入りに探る必要がある。

例えば例題においては、捕まえる対象物がいない場合というのはエージェントを起動した直後か、実際回りに捕まえる対象がいない時や、すべてを捕まえてしまった時などに起こる、これらのどの場合に再プランニングを起こすべきかは問題依存である。

3.6 BDI エージェントとプランナの関係

我々のプランナはエージェントから動的に起動されるものなので、実行するにはそれを起動する主体となるエージェントも必要となる。我々は動作確認のため、4 節で述べる例題に即したエージェントの簡易な実装を行った。その実行の過程は図 5 のようになる。このエージェントは問題依存のものであるが、このプランナを用いて行為を行う一般的なエージェントを構築した場合も図 5 と同等の実行課程をとるものと考えられる。その実装は今後の課題である。

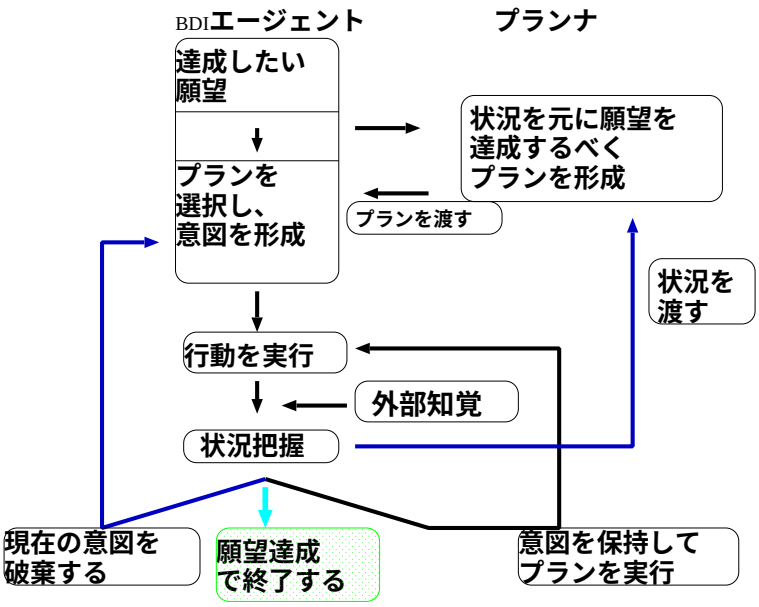

図 5　BDI エージェントとプランナの関係

4　例題を用いた実験

本節では、動的環境内を動くオブジェクトを複数捕まえることを目標の一部とする例題を用いて、我々のプランナの動作について説明する。

従来の BDI エージェントではもしオブジェクトを捕まえられなかった場合、そのオブジェクトを捕獲できるようになる時期を待つ状態になってしまう。しかし、再プランをして別のオブジェクトを先に捕える方が有利な場合が考えられる。今回の方式ではこれができ、別のオブジェクトを捕まえることができるようになった。このように、この例題は今回の実装の有用性を示している。

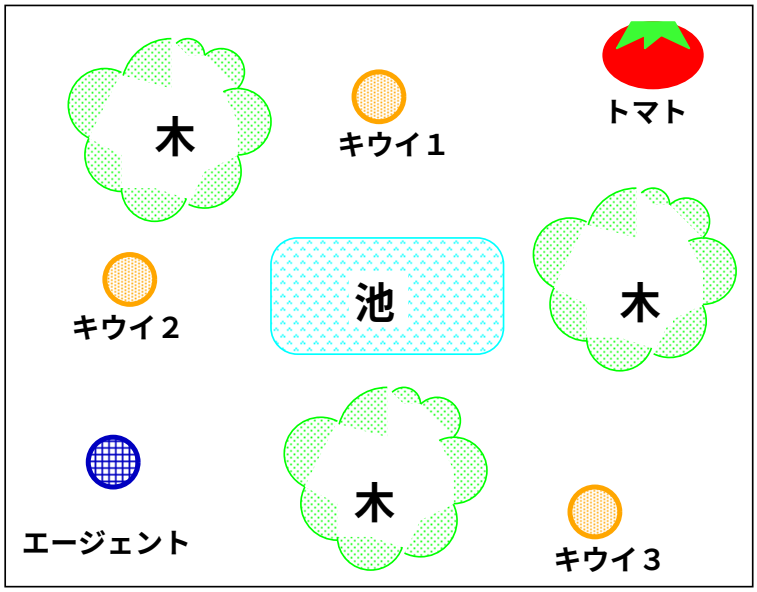

図 6　例題の想像図

4.1　例題の設定

この例題は SHOP2 のサンプルとして付属している問題を改変したものである。

ある空間内に木が３本立っていて、真ん中に池があり、キウイという鳥が三羽存在していて、農園にトマトがなっている。そして、エージェントは初期状態ではバンジョーを保持しているとする。

ここで、問題はエージェントが農園に行ってトマトを収穫することである。この目標を達成するには、空間内にいる動きまわるキウイが邪魔なので、まずキウイを全部捕獲せねばならない。さらにキウイを捕獲するには最初に手に持っているバンジョーを置かねばならない (図 6 を参照)。従って、エージェントにはバンジョーを置く、キウイを全部捕獲する、トマトを収穫する、の順に行動しなくてはならない。

キウイは次の 4 つの行動のいずれかをランダムにとる。

1. ただとどまっている
2. 動きまわっている
3. 池で水を飲んでいる

4. 木の穴に入っている

ただし状態 4 にあるキウイは捕えるのは不可とする。これに対してエージェントは、初期状態としてバンジョーを保持しているので、キウイを見つけたら、バンジョーを置いて手を開けることを第一のプランとする。そして、キウイを捕獲するために、キウイに接近、捕獲に至る。

ここでキウイが捕獲できるかどうかはランダム決定されるので、失敗した場合、再プランニングを行って別のキウイを先に捕獲できるようになっている。

また、キウイが穴に入ってしまった場合、5 ステップの間捕獲することはできない、従ってエージェントは別のキウイを捕獲するという手段をとるべく、この場合もまた再プランニングさせるようにする。尚、今回の実験ではキウイが穴に入ってしまうという状態をキウイの取りうる状態に入れることを行っていない。

4.2 例題の具体化

```
#S(KIWI-MAP :XSIZE 10 :YSIZE 12 :TREE ((2 2) (8 5) (5 9))
  :KIWI
  (#S(EACH-KIWI :NUM 1 :POS (6 2) :STAT WALK :HALT 0)
   #S(EACH-KIWI :NUM 2 :POS (2 5) :STAT STAY :HALT 1)
   #S(EACH-KIWI :NUM 3 :POS (8 9) :STAT WALK :HALT 0))
  :TOMATO ((8 8)) :AGENT (1 1))
```

図 7　例題の空間の表現

3.2 節で述べたプランナを実際に動かすために 3.6 節で述べた通りエージェントを実装した。このエージェントでは 4 節の状況を次のように具体的に実現した。

木の座標とキウイの座標を設定しておいて x 座標の幅を 10、y 座標の幅を 12 に設定した空間を作り (図 8)、その中にエージェントとキウイとトマトが存在していることを表記している。図 7 のように Common Lisp の構造体として表現されている。Common Lisp で実装したのは、我々が実装に用いた SHOP2 が Common Lisp で記述されているためである。

4.2.1 マップ上でのキウイの動作

図 7 中で STAT がキウイの状態を示しており、取りうる値は STAY と WALK と DRINK と HIDE であり、それぞれの行動に対して、STAY は 1 ステップ、WALK は 0 ステップ、DRINK は 3 ステップ、HIDE は 5 ステップ停止する。HALT はキウイの静止時間であり、この値は 1 ステップごとに 1 つ減る。HALT が 0 になるとキウイの状態はランダムに遷移する (WALK に 70%、現時点で HIDE は 0%、残り 2 つに 10% ずつの確率)。WALK 状態のキウイは 1 ステップにつき回りの 8 マスのどれかにランダムに動く。また、HIDE 状態のキウイは捕獲できない。他の状態のキウイは捕獲しようとすると 60% の確率で捕獲できる。

通常の人間のプランニングでは、基本行為は確率的に成功するものを前提としてプランニングを行い、実際にも成功の可能性が高いことが多い。例えば、交通機関を利用してどこかへ行くとき、各交通機関は正常に動くものと仮定し、そして突発的に交通機関の不通が起きたときのみ再プランをする。しかしここでは、予測できない失敗への対処がうまくできることを実験で確認する目的のため、キウイの捕獲が失敗する確率は高く設定した。

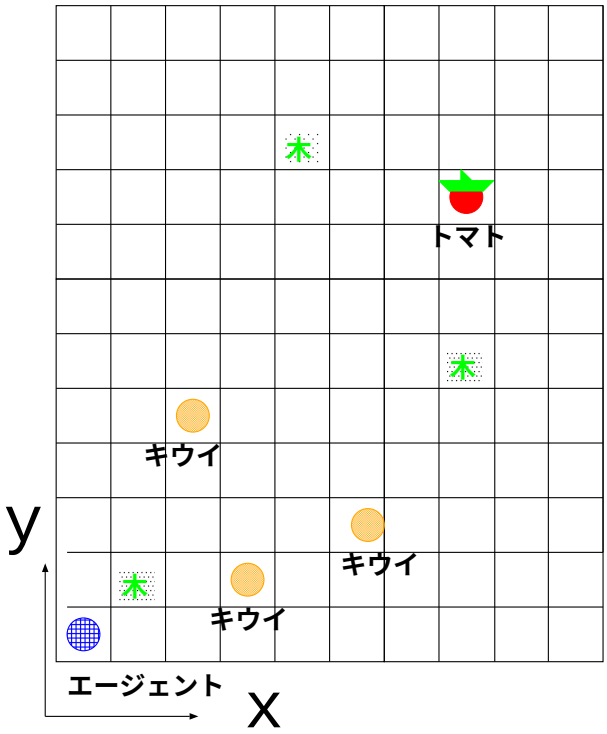

図 8　実装上でのマップ

4.2.2　エージェントの設計

図 7 の中での AGENT(1,1) がエージェントの位置を示している。

初期状態では (1,1) にいて、エージェントは自分自身から半径 3 マス内にキウイが存在するとキウイを発見することができるようになっている。キウイを見つけるとエージェントはキウイを捕獲するようにプランナにプランを作ってもらう。

尚、失敗すると失敗したという知覚を元に再プランを行うようになっている。

またキウイが見えなくなって見失ってしまった場合、エージェントはトマトを収穫するため農園に向かうようにしている。キウイがいないのなら邪魔者はいないのでキウイを捕らえるというプランを達成することなくトマトを収穫するという目的を果たせるためである。

このように、エージェントの実装は環境の知覚を行って信念 (状態) を更新し、プランナを呼び出して行為を決定して、実行 (環境への作用を行う) という形にしている。

4.3　結果

図 9、図 10 に見られるように大まかなプランを作成したように、エージェントはキウイを探して農園に向かい、発見の後バンジョーを置いた。それから先を図 11 に表示してある。

まず、キウイ 2 が一番エージェントに近かったのでキウイ 2 を捕まえるプランを作成してもらい、その通りに動いた。結果捕獲に失敗してしまい、その後キウイ 1 を捕まえるプランを再プランにより立てて見事に捕まえた。

```
Defining domain ...
Defining problem SEARCH-KIWI ...
------------------------------------------------------------------------
Problem SEARCH-KIWI with :WHICH = :FIRST, :VERBOSE = :PLANS
Totals: Plans Mincost Maxcost Expansions Inferences  CPU time  Real time
          1     1.0     1.0          3          0     0.044      0.054
Plans:
(((!GO-SEARCH AGENT)))   ; キウイを捕まえるために探すプラン


Defining domain ...
Defining problem PROBLEM1 ...
------------------------------------------------------------------------
Problem PROBLEM1 with :WHICH = :FIRST, :VERBOSE = :PLANS
Totals: Plans Mincost Maxcost Expansions Inferences  CPU time  Real time
          1     1.0     1.0          3          0     0.000      0.002
Plans:
(((!DROP BANJO)))       ; キウイを見つけたらバンジョーを置くプラン


Defining domain ...
Defining problem GET-KIWI ...
------------------------------------------------------------------------
Problem GET-KIWI with :WHICH = :FIRST, :VERBOSE = :PLANS
Totals: Plans Mincost Maxcost Expansions Inferences  CPU time  Real time
          1     2.0     2.0          5          0     0.000      0.003
Plans:
(((!NEAR-KIWI 1) (!GET-KIWI 1))) ; 三羽のキウイに接近して捕まえるためのプラン
                                 以下 1、2、3 と番号付けしてあるが、 サブプラン内ではエージェントに
                                  一番近いキウイが捕まえる対象となる
```

図 9　大まかなプラン 1

キウイ 1 を捕まえたので外部の状況からキウイ 1 の情報が消えていることがわかる、またエージェントの信念に (HAVE-KIWI 1) が加わっているので捕まえたことがわかる。

このほか、キウイを捕まえる過程においてキウイがエージェントから離れて見えなくなってしまうことが起こった。

この場合、エージェントは農園に向かいながらキウイを探索するという手法をとらせたところ、エージェントはそのままキウイを発見することなく農園に着いてしまい、キウイをすべて捕まえるというサブプランをすべてこなすことなくトマトをとるサブプランに移ることができた。この過程を図 12 に載せてある。

```
Defining problem GET-KIWI ...
------------------------------------------------------------------------------
Problem GET-KIWI with :WHICH = :FIRST, :VERBOSE = :PLANS
Totals: Plans Mincost Maxcost Expansions Inferences  CPU time  Real time
          1     2.0     2.0          5          0     0.004      0.003
Plans:
(((!NEAR-KIWI 2) (!GET-KIWI 2)))


Defining problem GET-KIWI ...
------------------------------------------------------------------------------
Problem GET-KIWI with :WHICH = :FIRST, :VERBOSE = :PLANS
Totals: Plans Mincost Maxcost Expansions Inferences  CPU time  Real time
          1     2.0     2.0          5          0     0.004      0.003
Plans:
(((!NEAR-KIWI 3) (!GET-KIWI 3)))


Defining domain ...
Defining problem GET-TOMATO ...
------------------------------------------------------------------------------
Problem GET-TOMATO with :WHICH = :FIRST, :VERBOSE = :PLANS
Totals: Plans Mincost Maxcost Expansions Inferences  CPU time  Real time
          1     1.0     1.0          3          0     0.004      0.002
Plans:
(((!GET TOMATO)))                    ; トマトを収穫するためのプラン
```

図 10　大まかなプラン 2

4.4　考察

実行した結果、エージェントは二通りの手段で目的を果たした。一つはエージェントが農園に向かい、キウイを発見、バンジョーを置く、キウイをすべて捕まえ、農園に到着してトマトを収穫というものであった。

もう一つはエージェントが農園に向かい、キウイを発見、バンジョーを置く、キウイを捕獲しつつ農園に向かい、運良く到着しトマトを収穫するという手段である。

前者はプランナが最初に与えた通りの全うなプランであり、それを一つ一つこなしたという形である。後者の方はプランどおりに行動しつつも環境の変化によりキウイがいなくなった隙に農園まで辿り着き、トマトを収穫したという形で、こちらの方が環境に柔軟に対応した結果であったと思われる。

```
; バンジョーを置いた後のエージェントの知覚による信念の状態
((FREE BANJO) (CAN-VIEW-KIWI 3) (CAN-VIEW-KIWI 2) (CAN-VIEW-KIWI 1))

Defining domain ...
Defining problem GET-KIWI ...
------------------------------------------------------------------------
Problem GET-KIWI with :WHICH = :FIRST, :VERBOSE = :PLANS
Totals: Plans Mincost Maxcost Expansions Inferences  CPU time  Real time
          1     2.0      2.0          5          8     0.004       0.005
Plans:
(((!CLOSER-KIWI 2) (!PICKUP-KIWI 2)))  ;2 が一番近かったので接近して捕まえるプランを立ててもらった

; 外部の状況を構造体にて示す
#S(KIWI-MAP :XSIZE 10 :YSIZE 12 :TREE ((2 2) (8 5) (5 9))
   :KIWI
   (#S(EACH-KIWI :NUM 1 :POS (4 2) :STAT DRINK :HALT 3)
    #S(EACH-KIWI :NUM 2 :POS (3 5) :STAT WALK :HALT 0)
    #S(EACH-KIWI :NUM 3 :POS (6 3) :STAT WALK :HALT 0))
   :TOMATO ((8 8)) :AGENT (2 3))

; 捕まえようとして失敗してしまったエージェントの信念の状態
((FAIL-GET-KIWI 2) (CAN-GET-KIWI 2) (FREE BANJO) (CAN-VIEW-KIWI 2)
 (CAN-VIEW-KIWI 1))

Defining domain ...
Defining problem GET-KIWI ...
------------------------------------------------------------------------
Problem GET-KIWI with :WHICH = :FIRST, :VERBOSE = :PLANS
Totals: Plans Mincost Maxcost Expansions Inferences  CPU time  Real time
          1     2.0      2.0          5          8     0.000       0.004
Plans:
(((!CLOSER-KIWI 1) (!PICKUP-KIWI 1))) ; 別のキウイの方が近かったので再プランを行った

#S(KIWI-MAP :XSIZE 10 :YSIZE 12 :TREE ((2 2) (8 5) (5 9))
   :KIWI
   (#S(EACH-KIWI :NUM 2 :POS (4 6) :STAT WALK :HALT 0)
    #S(EACH-KIWI :NUM 3 :POS (6 3) :STAT STAY :HALT 1))
   :TOMATO ((8 8)) :AGENT (3 1))
; 捕まえてキウイを捕獲した後の信念の状態
((CAN-VIEW-KIWI 3) (HAVE-KIWI 1) (FAIL-GET-KIWI 2) (CAN-GET-KIWI 2)
 (FREE BANJO))
```

図 11　捕獲失敗後の再プランの例

```
Defining domain ...
Defining problem GET-KIWI ...
------------------------------------------------------------------------
Problem GET-KIWI with :WHICH = :FIRST, :VERBOSE = :PLANS
Totals: Plans Mincost Maxcost Expansions Inferences  CPU time  Real time
          1     2.0     2.0           5          8     0.004       0.018
Plans:
(((!CLOSER-KIWI 3) (!PICKUP-KIWI 3))) ; キウイ 3 を捕まえるプラン

#S(KIWI-MAP :XSIZE 10 :YSIZE 12 :TREE ((2 2) (8 5) (5 9))
   :KIWI (#S(EACH-KIWI :NUM 2 :POS (1 6) :STAT DRINK :HALT 0)) :TOMATO ((8 8))
   :AGENT (5 4))
; 現在のエージェントの信念の状態、キウイ 2 が見えなくなってしまっているのがわかる
((HAVE-KIWI 3) (HAVE-KIWI 1) (FREE BANJO))

Defining domain ...
Defining problem SEARCH-KIWI ...
------------------------------------------------------------------------
Problem SEARCH-KIWI with :WHICH = :FIRST, :VERBOSE = :PLANS
Totals: Plans Mincost Maxcost Expansions Inferences  CPU time  Real time
          1     1.0     1.0           3         14     0.004       0.003
Plans:
(((!MOVE AGENT))) ; 農園に向かうプランを立てた

#S(KIWI-MAP :XSIZE 10 :YSIZE 12 :TREE ((2 2) (8 5) (5 9))
   :KIWI (#S(EACH-KIWI :NUM 2 :POS (1 5) :STAT WALK :HALT 0)) :TOMATO ((8 8))
   :AGENT (7 6))   ; トマトの位置とエージェントの位置が非常に近くなっているので農園に着いたことがわかる
このときのエージェントの信念の状態
((HAVE-KIWI 3) (HAVE-KIWI 1) (FREE BANJO))
```

図 12　すべてのキウイを捕まえる前に農園に辿り着いた例

5　関連研究と将来展望

5.1　関連研究

動的な環境に対応する階層的なプランナの例は他にもあり、代表的なものとしては [8] が挙げられる。[8] は動的環境での半順序プランニングを目指しているが、BDI エージェントへの結合は考えられていないため、意図によって一貫した動作を行う機能は持っていない。これに対し、我々の設計はプランナが構築したプランを BDI エージェントが意図として形成することで一貫的な動作を行わせやすい。

また、BDI エージェントでのプランニングに関する研究としては [4] や [3]、[5] がある。[4] や [3] は、BDI エージェントでのプランニングに関する形式化や限られた時間で熟考を中断するアルゴリズムは示しているが、現実的に動作するプランナを提示してはいない。また [5] は BDI エージェントへのプランナの統合であ

るが、この論文で扱われているのは propositional planning である。これに対し、我々がベースとしたプランナ SHOP2 は HTN(Hierarchical Task Network) に基づいており、階層的プランニングを行うことが可能である。

5.2 将来展望

現在の我々のプランナはエージェントに一つのプランを返し、エージェントの失敗と同時に再プランして、また、さらにエージェントにプランを与えてそれを繰り返すことにより、目標を達成するというものである。

しかしエージェントが失敗した時に改めて再プランをするのではなく予測がたてられているかの如く、その再プランの時点からの代替プランが既に存在していればエージェントはスムーズに動くことができる。我々は、実世界ではこのような先見的プランニング処理が高速にできるほうがエージェントにとって有望であると思われるので、この高速化をセルプロセッサによる独立した複数の CPU 上でのプランニングにより物理的に実現し、並列的なプランニングを実装する計画である。

さらに現実では、様々なプランを実行するとき、サブプランを取り出したときその中には共通した動作があると考えられ、こうした事を考慮してサブプランに対して評価を下すことが出来るようにしたい。この評価は強化学習 [9] を取り入れて行う予定である。

評価を行うことにより、そのサブプランで行った行動の価値が決まり、次に同じ行動をとるとき、その際の評価の高かったサブプランが優先的に実行されるようになる。従って行動を行えば行うほどに行動がパターン化して洗練されていく。

このとき、強化学習で既存プランを選んでいる間はエージェントに対して次のサブプランへのプランを立てる余力をもつことができるようになるので、行動に対する予測をたてることが可能になる。

また、プランの破棄に対して、途中でプランナが中断をしてプランを組み直すという機構も採り入れたい。[3] にあるように中断機構を採り入れると、その問題に対して解決する時間がどれほどかかるかわからないような時に無限に動作を行うことなく止められることが出来るので、動的環境のような環境が変わってしまって対処ができなくなってしまうような時に中断できることは必要な要素である。

参考文献

[1] Anand S, R. and Georgeff, M. P.: Modeling Rational Agents within a BDI-Architecture, *Proc, of International Conference on Principles of Knowlegde Representation and Reasoning* (1991).

[2] Bacchus, F. and Kabanza, F.: Using Temporal Logics to Express Search Control, *Artificial Intelligence*, Vol. 116 (2000).

[3] de Silva, L., Dekker, A. and Harland, J.: Planning with Time Limits in BDI Agent Programming Languages, *Proc. of 13th Australasian symposium on Theory of computing*, Vol. 240, pp. 131–139 (2007).

[4] de Silva, L. and Padgham, L.: Planning on demand in BDI systems, *ICAPS-05* (2005).

[5] Meneguzzi, F. R., Zorzo, A. F., Móra, M. D. C. and Luck, M.: Incorporating planning into BDI systems, *Proc. of SCPE 2007* (2007).

[6] Nau, D., Au, T.-C., Ilghami, O., Kuter, U., Murdock, J. W., Wu, D. and Yaman, F.: SHOP2: An HTN Planning System, *Journal of Artificial Intelligence Research*, Vol. 20, pp. 379–404 (2003).

[7] Singh, M. P., Rao, A. S. and Georgeff, M. P.: Formal method in DAI, *Multiagent systems: amodern approach to distributed artificial intelligence* (1999).
[8] 林久志, 長健太, 大須賀昭彦: 動的環境で動作するエージェントのための半順序 HTN プランニング, エージェント合同シンポジウム (JAWS2003) 講演論文集, pp. 252–259 (2003).
[9] 三上貞芳, 皆川雅章: 強化学習, 森北出版 (2000).