BDIエージェントを用いた
実世界での協調作業に伴う衝突回避について

奈良女子大学理学部情報科学科　4回生
新出研究室　隅田 麻由

概要

自律エージェントを用いたロボット制御は, 知能ロボットの実現に有用であると考えられる. また複数のロボットが1つの作業を行うという協調作業によって, 単一のロボットでは困難な問題の解決が可能となり, 応用の拡大が期待できる.

そこで我々はBDIエージェントを2台のロボットに搭載し, 実世界での協調作業における問題点の検証を行った. その結果いくつかの場面において, ロボット同士の衝突およびロボットと障害物との衝突が発生するという問題点が確認された. 本論文ではエージェントに実装した衝突回避のための機能を組み合わせることによりこの問題を解決し, その手法について述べる.

1　はじめに

現在, 自律的な知能ロボットの実現を目指した研究が多くなされている. 知能ロボットの応用としては, 掃除ロボット, 介護ロボット, 案内ロボットなど様々な応用が視野に入れられている. このような知能ロボットの実現にあたり, ロボット制御における自律エージェントの利用は有効であると考えられる. 自律エージェントには様々なものがあるが, その1つとしてBDIエージェントが挙げられる.

BDIエージェント [1] とは, 信念 (Belief), 願望 (Desire), 意図 (Intention) という3つの心的状態パラメータを用いて, 熟考しながら自律的に行動選択を行う合理的かつ自律的なエージェントである. BDIエージェントは成し遂げたい願望を持つと, その目標を達成するための手段 (プラン) を選択する. そしてそのプランを実行するための意図を形成する. 意図とは, 目標達成までの大まかな行動方針である. 形成された意図を持続的に保持することで, エージェントの自律的な行動が可能となる. またBDIエージェントは, 保持している意図と衝突を起こすような意図は形成しない. そのため矛盾のない一貫した行動をとることが可能である. したがって, 意図の持続性および一貫性はBDIエージェントの大きな特徴であると言える.

しかしBDIエージェントは, シミュレーション環境などのソフトウェアエージェントで使用されることが多く, 実世界におけるロボット制御への応用例は少ないのが現状である. そこで我々はBDIエージェントを搭載した2台のロボットを使用し, 実世界での協調作業における問題点の検証を行った. その結果, 2台のロボットがぞれぞれ自律動作を行う際に, 同じ地点へ動こうとする衝突現象を回避する必要があることが分かった.

我々は, この問題をロボット同士がコミュニケーションをとることで解決した. 本論文では, 我々が実現した衝突回避の方法について述べる. なお本研究は, 奈良女子大学理学部4回生の小山との共同研究である.

2　関連研究との比較

本研究の関連研究として, Developing Multi-Agent Lego Robotics [2] が挙げられる.

[2] で行われている実験では, 光センサを用いて格子状に貼られたテープを知覚しながら進むという方法を取っている. しかし実世界にはそのようなマーキングが施されている保証はないため, この方法は実世界でのロボット制御として適切であるとは言い難い. よって我々はコンパスセンサによって地球の磁場を検出し, ロボットが方向を取得しながら進むという方法を取ることにした.

また [2] では, 同じ機能を持つロボットを複数用いて実験を行っている. しかし人間世界で行われる作業について考えてみると, 人間は役割分担によってより複雑な作業を行っている. このことからロボットが行う協調作業においても, 異なる機能を持つロボットを複数用いる方がより現実的であると考えられる. またこれにより, ロボットの特性を生かした作業が可能になると考える.

そこで本研究では, 異なる種類のロボットを 2 台用いて実世界における協調作業を行い, 協調作業における問題点の検証およびその解決を行った.

3 基本設計

3.1 BDI エージェント

3.1.1 BDI アーキテクチャ

BDI アーキテクチャとは, 動的に変化する環境を知覚し, 願望を達成するためにプランを選択しながら動作する BDI エージェントの内部アーキテクチャである. BDI アーキテクチャはエージェントの持つ心的状態 (信念, 願望, 意図), プランライブラリ, イベントキュー, インタプリタなどで構成される.

インタプリタは信念と自らの願望をもとに, 実行すべきプランをプランライブラリから選択する. そして選択したプランを実行するための意図を形成する. この意図は実行すべき時が来るまで保持される. また現在持っている意図と矛盾した意図は持たないため, エージェントは一貫した行動を取ることが可能である.

インタプリタの主要部は, 一般的に次のようなループで実装される. [1]

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.
```

ここで, B, D, I はそれぞれ信念, 願望, 意図という 3 つの心的状態を表す. 初期条件が与えられると, 外界からのイベント (event-queue), 心的状態およびプランライブラリから option-generator によって, 実行可能な意図の候補となるプランを選択する. そして deliberate によって実際に実行すべき意図を選択し, execute により実行する. その後 get-new-external-events により外界の変化を受け取り, 達成された意図や願望, もしくは不可能となった意図や願望の除去を行う. これにより心的状態の更新が行われ, 整合性が保たれる. このループを繰り返すことにより, エージェントは実現したい願望を達成することができる.

3.1.2 Jason

本研究では BDI エージェントの実装に Jason を使用した. Jason とは BDI エージェント記述用言語である AgentSpeak[3] のインタプリタである.

Jason ではエージェントを定義するエージェントプログラムと, そのエージェントが置かれる環境を定義した環境プログラムを用意する. 環境プログラムは Java で実装し, これにより環境とエージェントの相互作用が可能となる (図 1).

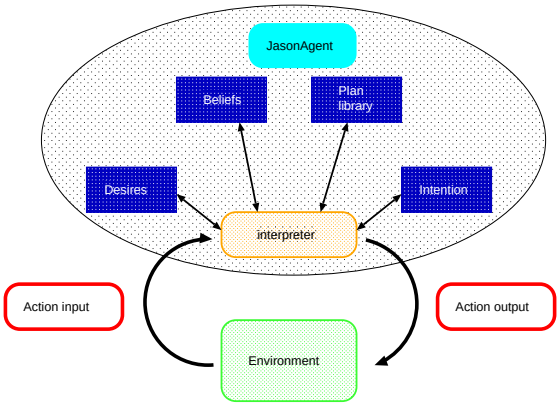


図 1: Jason におけるエージェントと環境との相互作用

3.2 LEGO MINDSTORMS NXT およびその制御

本研究では LEGO 社が販売している LEGO MINDSTORMS NXT(以下, NXT) というロボットを使用した. NXT は教育用に開発されたもので, 比較的制御しやすく安価なものであることから実験に適していると言える.

実験では 2 台のロボットを使用した. これらは別々の形状をしており, 備えている機能やセンサの種類および取り付け位置が異なっている. 詳しい形状および機能については 4.2 節で述べることにする.

NXT の制御は, 実験 [5] で用いられた制御方法を使用した. NXT が取るべき行動は, 実装されたエージェントによって Jason 内で決定され, その行動を示す命令がテキストとして Python プログラムに送られる. このプログラムは NXT を制御するためのものである. ここでは送られてきたテキストに対応する行動制御命令を選択し, その命令が Bluetooth アダプタによって NXT に送られる. これにより, エージェントが決定した行動を NXT が実世界で行うことができる. 本研究では当初 1 個の Bluetooth アダプタによる 2 台の NXT の制御を試みたが, この方法では速度低下の問題が発生した. そのため 2 個の Bluetooth を使用し, 2 台の NXT とそれぞれペアリングを行った.

またエージェント間の通信も同様に Jason 内で行われ, NXT 同士が直接通信を行うことはない. 通信によって得られた環境情報は, 知覚によって得られた情報と同様にエージェントの行動決定に用いられる.

4 協調作業

本研究で我々が協調作業として設定した例題は, 図 2 のような 4 × 5 マスのフィールド内に設置されている台を探索し, 適切な台の上に box と呼ばれる物体を置くというものである. 本章ではこの協調作業について述べる. なおフィールド内の座標は, 図 2 に示すように水平方向を x 軸, 垂直方向を y 軸とし, 左上の座標を (0, 0), 右下の座標を (4, 3) とする.

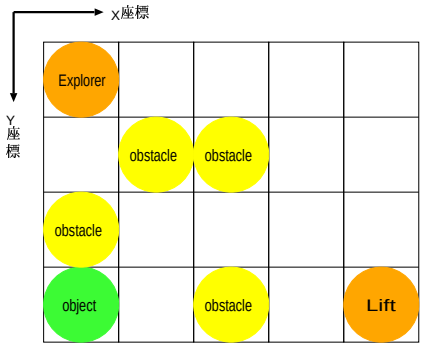


図 2: 実験に用いたフィールド

4.1 台の種類

フィールド内には 2 種類の台を設置する．1 つは box を置くことが出来ないもの (以下, obstacle), もう 1 つは box を置くことが出来るもの (以下, object) である．この実験では, フィールド内に必ず 4 個の obstacle および 1 個の object を置くものとする．ロボットは探索中に発見した台が object および obstacle のどちらであるかを判定し, object であると判定された台に box を乗せることを目標とする．ただしロボットの形状が違うことから, 台の判定方法はロボットによって異なる．それについては 4.2 節で述べることにする．

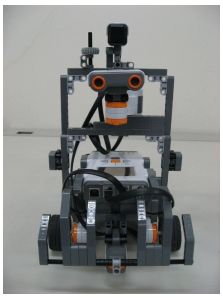
4.2 使用するロボット

実験では BDI エージェントを搭載した 2 台のロボットを使用した．これらについて以下に示す．

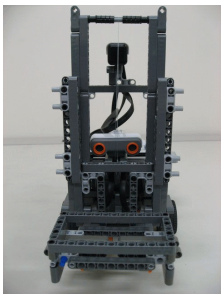
4.2.1 Explorer

Explorer は図 3(a) のような形状をしたロボットである．方向知覚用のコンパスセンサ, 障害物知覚用の超音波センサおよびタッチセンサを持つ．

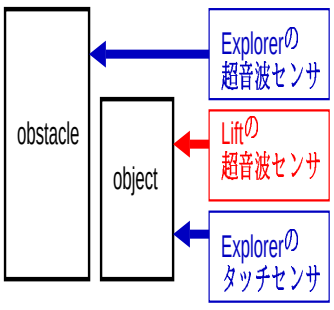
object および obstacle とロボットの持つセンサの高さ関係は, 図 3(c) に示す通りである．Explorer の超音波センサは obstacle よりも低い位置にあるため, これにより obstacle を知覚することができる．それに対して object と超音波センサの高さを比較すると, 超音波センサの方がより高い位置にあるために object を知覚することができない．しかし Explorer は下部にタッチセンサを持つため, これで台との衝突を検知することによって object を知覚することができる．したがって Explorer は, 自身の持つセンサによって台が obstacle と object のどちらであるかを判定する．



(a)Explorer



(b)Lift



(c) 台とロボットの高さ関係

図 3: ロボットの形状

4.2.2 Fork Lift

Fork Lift(以下, Lift) は図 3(b) のような形状をしたロボットである. Lift はコンパスセンサおよび超音波センサを持ち, それと併せて box を台の上に乗せるためのフォークリフトを持つ.

4.2.1 節で述べた Explorer とは異なり, Lift は object と obstacle の区別をすることができない. これは図 3(c) に示すように, Lift の持つ超音波センサがどちらの台よりも低い位置にあるためである. したがって Lift は台の知覚は可能であるが, 台の種類の判定は Explorer に委託する必要がある. Explorer が object であると判定した場合にのみ, フォークリフトを用いて box を台の上に乗せる作業を行う.

4.3 作業全体の流れ

作業全体の流れを図 4 に示す. ここでは主に search, check, up, return という 4 つの作業が行われる.

まず 2 台のロボットは, それぞれの探索領域内で台の探索 (以下, search) を行う. 探索領域は図 5 のように指定されており, 探索方法については [6] で述べられている. また作業開始の時点で, ロボットは台の位置を全く知らないものとする.

search の途中でロボットが台を発見した場合には, その種類と位置情報がエージェントの信念として追加される. ただしそのためには, 発見した台が obstacle であるか object であるかを判定する必要がある.

4.2.1 節で述べたように, Explorer は自身で object と obstacle との区別ができる. したがって Explorer は台を発見する度にその種類の判定を行いながら, object を発見するまで search を行う. そして object を発見した場合には, object の上に box を乗せる作業 (以下, up) を Lift に委託する.

一方, 台の種類を区別することが出来ない Lift は, 発見した台の種類の判定 (以下, check) を Explorer に委託する. このとき Explorer は search を一時中断し, check を優先的に行う. そしてその判定結果により, Lift が check 後に行う作業が決定される. つまり Lift は, 発見した台が obstacle であった場合には search を続行し, object であった場合には up を行うということになる.

そしてどちらかのロボットによって object が発見された場合には, ロボットはそれぞれのスタート位置に戻る作業 (以下, return) を行う. これは, Explorer の場合は Lift に up を委託した後に, Lift の場合は up が終了した後に行われる. ただしこの実験は object が 1 個だけであるという前提で行っている. したがって object が発見されたあとは, 未発見の台が存在したとしても search は行われない. return が完了した時点で, 協調作業は完了したものとする.

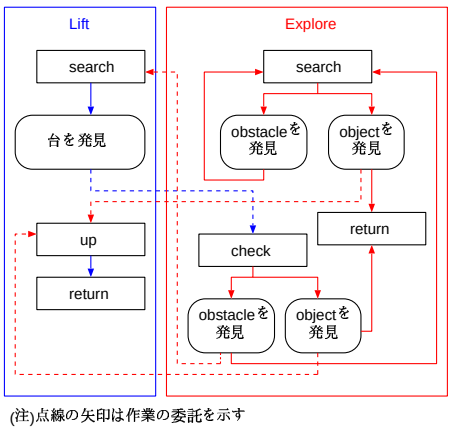


図 4: 全体の流れ

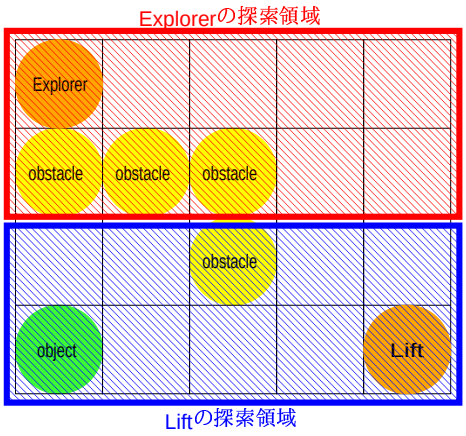


図 5: 探索領域

4.4 経路選択機能

協調作業の中には, 移動を必要とするものがいくつか存在する. 例えば Explorer が check を行う場合や, object の発見後に return を行う場合である. このような移動の際にロボットが自分で移動経路を決定するために, 本研究では経路選択機能を実装した.

この機能ではロボットの現在位置から目的地までの経路が選択され, 経路選択には A*アルゴリズム [4] を用いている. A*アルゴリズムとは, 求めた経路が最短であることを保証している経路探索アルゴリズムの 1 つである. したがってロボットは, 目的地までの最短経路に沿って移動することができる. またフィールド内にいる他のロボット, 既に発見している台, check 中の状態である台を含む経路は選択されない. check 中の状態とは, 発見された台の種類がまだ判定されていない状態, つまり判定待ちの状態を指す. ただし, 目的地にはロボットや台が存在しないものとして経路選択が行われる.

4.5 目的地選択機能

4.4 節で述べた経路選択機能では, 現在位置から目的地までの経路を選択できる. 現在位置は, 信念として保持している情報により簡単に取得できる. それに対して目的地は, 一意に決定する場合と複数の候補地が存在する場合がある.

本研究の協調作業では, Explorer が check を行う際に, 到達すべき場所の候補が複数存在する. なぜなら Explorer が check を行うためには, 判定すべき台に隣接する上下左右 4 マスのうちいずれかに移動する必要があるからである. よってこのような場合には, 目的地を 1 つ選択しなければならない.

そこで本研究では, 目的地の決定のために目的地選択機能を実装した. 目的地の選択後に行う移動をスムーズに行うために, この機能では選択した目的地までの経路を求めるように設計した. 目的地および経路の選択には, 次に示すアルゴリズムを用いている.

1. 判断の対象となる台の位置を S とおく
2. S に隣接する 4 つのマスを S_0, S_1, S_2, S_3 とおく
3. 最小経路の長さを示す変数 $MIN = 500$, その経路を示す変数 $P =$ NULL とおく
4. **for** $i = 0$ **to** 3 **do**
5. **if** S_i がフィールド外でない **and** S_i に台が存在しない **then**
6. 経路選択機能により現在位置から S_i までの経路 P_i を選択し, P_i の長さを L_i とおく
7. **if** $L_i > 0$ **then**
8. **if** S_i に他のロボットが存在する **then**
9. 重み付けとして L_i を 20 倍する
10. **end if**
11. **if** $L_i < MIN$ **then**
12. $MIN = L_i$, $P = P_i$
13. **end if**
14. **end if**

15. **end if**

16. **done**

17. このときの移動経路は P である

このアルゴリズムにより, ロボットの現在位置から最短距離で移動可能な目的地を選択することができる.

上記 7. では, $L_i > 0$ により S_i までの経路が存在しない場合を除いている. これは変数 MIN のとる値を 1 以上にするためである. もし MIN のとる値を 0 以上とした場合, S_0, S_1, S_2, S_3 の中に 1 つでも経路が存在しないマスが存在すれば, ロボットは必ずそのマスを目的地として選択してしまう. これでは適切な目的地が選択できないため, $MIN > 0$ とする必要がある.

また上記 9. で経路の長さに重み付けを行うことから, 台に隣接する 4 マスのうち Lift が存在しないマスを優先的に選択することができる. これにより 5.1 節で述べる行き詰まりの発生を最小限に抑えることが可能となる.

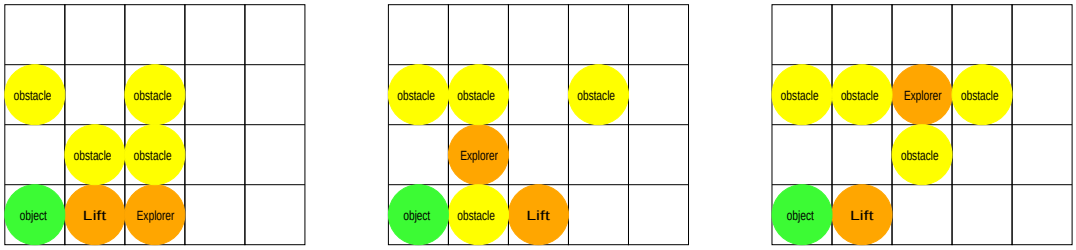
5 本研究における衝突回避

本章では, 著者が問題解決にあたった衝突回避について述べる.

5.1 衝突の発生

複数のロボットが共同で作業を行う場合, ロボット同士もしくはロボットと台との衝突発生が予想される. これによりロボットは正しい動作を行うことができず, 作業が中断されてしまう, もしくは安全な作業が難しくなるといった問題が発生する. したがって, このような衝突を回避する必要がある.

本研究で行った協調作業では, 次に示す (a)〜(c) の 3 種類の衝突が発生することが分かった.



(a) (b) (c)

図 6: 協調作業で生じる衝突

(a) 行き詰まりによる衝突

これは, ロボットが動けない状態に陥る行き詰まりによる衝突である.

図 6(a) は, Explorer が check を行おうとしている場面である. ここで判定の対象となっている台は, Lift が発見した (0, 3) に存在する台である. Explorer は check を行うために (2, 3) から (1, 3) へと移動しようとする. しかし (1, 3) には既に Lift が存在しており, また他に移動可能な経路が存在しないために, ここで衝突が発生する.

このような衝突を回避するためには, Explorer が Lift の位置を考慮しながら移動することが必要である. よって「1 マスに 1 台のロボットのみが存在する」という条件を設け, 他のロボットが存在するマスへの移動を中止することで, 衝突回避が可能となる. しかし図 6(a) のような状況では, ロボットが衝突回避を行ったとしてもこれ以上移動することができない. そのために作業が滞ってしまう. したがってこの衝突は, 行き詰まりの解消によって回避することが必要である.

行き詰まりの解消方法としては, 次の 2 点が考えられる.

1. 行き詰まりが生じない別の経路を選択する
2. Explorer が移動を開始する前に, Lift が適切な位置に移動する

上記 1. は, 4.5 節で述べた目的地選択機能により行うことができる. しかし他に移動可能な経路が存在しない場合には, この方法では行き詰まりを回避することができない. 上記 2. での “適切な位置” とは, Explorer の作業の邪魔にならない位置, つまり Explorer が選択した経路に含まれない位置である. Lift の移動が完了した後に Explorer が移動を開始すれば, このような行き詰まりは生じない. したがって上記 2. の方法により, 行き詰まりが解消されると考えられる.

(b) 同時移動による衝突

これは, 2 台のロボットが 1 つのマスに対して同時に移動することによって発生する衝突である.

図 6(b) は check が終了した後に, ロボットが search を続行しようとする場面である. ここでは Lift が発見した (1, 3) に存在する台が obstacle であると判定されたために, search を続行しようとしている. Explorer は自分の探索領域 (図 5 参照) に戻るために (2, 2) に進もうとする. 一方 Lift は現在位置である (2, 3) から search を続行し, (1, 3) にある obstacle を避けるために (2, 2) に進もうとする. これでは 2 台のロボットが同時に (2, 2) に進むことになり, 衝突が発生する.

これは, 相手が次に進もうとしているマスを把握していないことが原因である. しかし常に他のロボットの移動先を把握しておくというのは, 現実的な方法とは言い難い. そこで別の方法として, Lift が Explorer の移動を待つ, つまり Explorer の移動を優先的に行うという方法により衝突を回避できると考えられる.

(c)obstacle との衝突

これは, ロボットの移動中に発生する obstacle との衝突である.

図 6(c) は, Lift が発見した (0, 3) にある台の check を Explorer が委託されたところである. Explorer は現在位置である (2, 1) から目的地である (0, 2) までの最短経路を選択し, 移動を試みる. この時点で obstacle であると既に分かっている台が (0, 1), (1, 1), (2, 2) の 3 つであるとすると, Explorer は (3, 1), (3, 2), (3, 3), (2, 3), (1, 3) という経路を選択する. しかし実際には経路上に未発見の obstacle が (3, 1) に存在するため, この経路に沿って移動を行うと obstacle との衝突が発生する.

これは新たに台を発見した際に, 経路を再選択することで解決できると考えられる. そのためには発見した台の位置情報を信念として追加し, 経路選択機能を用いて再び経路選択を行う必要がある.

5.2 衝突回避方法とその実現

5.1 節で述べた衝突例のうち, (a) および (b) はロボット同士の衝突, (c) はロボットと台との衝突である. (c) の場合には前に述べたように, ロボットが経路を再選択する機能を持つことで解決できると考えられる.

一方 (a) および (b) の場合はロボットがそれぞれ自律動作を行うため, (c) に比べてやや複雑になる．ロボットは check や search などの作業を, ロボットを制御するエージェントが意図した通りに正しく行うことが出来る．しかし (a) および (b) に共通して言えることは, ロボットが他のロボットの作業や移動経路などを考慮せずに, 与えられた作業を行っているということである．そのため (a) のように Lift が Explorer の移動の邪魔になる位置で停止している, また (b) のように安全に移動できるかという確認なしに移動を開始するといったことが発生する．このことから衝突を回避するためには, ロボット同士がコミュニケーションをとり, 連携しながら作業を進める必要があると考えられる．

したがって衝突回避の実現方法として, ロボットが連携をとるためのコミュニケーション機能, そして経路を選択し直すための経路再選択機能を実装した．5.1 節 (a)〜(c) で提案した回避方法は, これらを組み合わせることで実現できる．追加した機能は以下の通りである．これらは全て Jason のプランとして実装している．

- コミュニケーション機能
  - 待機 (作業や移動の完了を待つ)
    - wait_finished
    - pause
  - 伝達 (作業完了を伝える)
    - stop_wait
  - 移動 (作業の邪魔にならない位置への移動を行う)
    - can_move
    - move_for_avoid
- 経路再選択機能
  - re_check
  - re_return

以下にこれらの機能について具体的に述べる．

5.2.1 wait_finished

wait_finished は, 他のロボットの作業完了や移動完了を待つための待機機能である．これは例えば, Explorer が Lift の移動が完了するのを待つ場合に使用される．

この機能により行われる待機は, 後述する stop_wait 機能により解除される．これは自分自身で待機を解除することも, また他のロボットが待機を解除するよう指示することも可能である．

これにより作業の完了をお互いに確認しながら作業を進めることが可能となる．ただし stop_wait が使用されなかった場合, エージェントは無限に待機し続けることになる．これでは作業が続行できないため, 最大の待機時間を 100000ms に設定している．この待機時間は変更可能である．

5.2.2 pause

pause は, 他のロボットが探索領域内へ戻るのを待つための待機機能である．ここでは, Explorer が探索領域内へ戻るのを Lift が待つために使用している．Explorer の現在位置が探索領域内かどうかを一定時間ごとに判定し, 移動が確認された後に Lift の作業が再開 (もしくは開始) される．

5.2.1 節の wait_finished とは異なり, pause は必ず自分自身で待機を解除する. この機能を Lift が使用することにより, Explorer の移動が優先される. またこの機能を使用した際に Explorer が既に探索領域内に存在すれば, Lift は待つことなく作業を行うことができる. そのためこの機能は, 衝突を防ぐための安全確認としても使用できる.

5.2.3 stop_wait

stop_wait は, 5.2.1 節で述べた wait_finished による待機を解除するための機能である. 例えば Lift の移動完了後に, 待機中の Explorer に対してこの機能を使用することで移動完了の合図として使用できる.

待機に関する方法としては, 一定時間停止した後に作業を再開するという方法も考えられる. しかしこの方法に比べて, wait_finished と stop_wait による待機方法は作業時間に応じた待機が可能であるため, 待機時間が長すぎる, もしくは短すぎるといった問題は起こらない. したがって相手を待たせるための一時的な機能として使用できる.

5.2.4 can_move

can_move は他のロボットが選択した移動経路上に自分が存在する場合に, 経路に含まれない位置 (以下, 回避場所) を選択するための機能である. 例えば Explorer が選択した移動経路上に Lift が存在する場合には, Lift が回避場所を選択する. そして 5.2.5 節で述べる move_for_avoid 機能によって Lift が回避場所へ先に移動を行うことで, Explorer の移動が円滑に行われると考える.

ロボットが移動のために選択した経路は, エージェントの信念として蓄えられている. これを利用し, ここでは次のアルゴリズムで回避場所が決定される.

1. 経路上に存在するロボットの現在位置を P とする
2. P に隣接する上下左右の 4 マスのうち, 「台および他のロボットが存在しない」, 「経路に含まれない」という 2 つの条件を共に満たすマスがあれば, そのマスを P として 4. へ
3. P に隣接する上下左右の 4 マスのうち, 「台および他のロボットが存在しない」という条件を満たすマスがあれば, そのマスを P として 2. へ
4. 回避場所が P に決定する

なお上記 2. および 3. で選択可能なマスが複数存在する場合には, 選択するマスの優先順位を Lift は $x+1$, $y-1$, $x-1$, $y+1$, Explorer は $y-1$, $x+1$, $x-1$, $y+1$ としている.

5.2.5 move_for_avoid

これは前節の can_move により決定した回避場所へ移動を行うための機能である. 現在位置から回避場所までの経路を選択し, 移動を行う. 移動後に stop_wait により移動の完了を知らせることが可能であり, 回避場所の選択と移動を 1 度に行うときに使用する.

5.2.6 re_check および re_return

これらはどちらも経路再選択を行うための機能である. 経路再選択の際には, 新たに発見した台の位置情報を信念として追加した上で, ロボットの現在位置, すなわち移動を一時中断した位置から目的地までの経路選択を行う.

経路選択による移動は check および return で必要となるが, return と check では経路選択後の作業内容が異なる. 経路再選択後に return もしくは check それぞれの作業を行うために, re_check および re_return という別名の機能を用意している. これらは, エージェントが現在持つ願望を全て削除したあとに re_check もしくは re_return を開始するものである. これにより何度でも経路再選択が可能であり, その後の作業も問題無く行うことができる.

6 実験

5.2 節で述べた衝突回避用の機能を追加した上で, 実際にロボットを用いた実験を行った. この章ではそれらの機能により, 設定した例題における衝突が解決されたことを示す.

6.1 初期設定

実験は以下のような初期条件のもとで行った.

- Explorer のスタート位置を (0, 0), 向きを $-y$ 方向に固定する
- Lift のスタート位置を (4, 3), 向きを $+y$ 方向に固定する
- object の位置は (0, 3) に固定する
- obstacle は探索領域が塞がれるような配置を除き, 様々な配置で試す
- ロボットは 1 マスに 1 台のみ存在する
- ロボットは 1 度に上下左右のうちいずれか 1 マスに移動する
- ロボットは現在向いている方向に対して, 現在位置から 1 つ先のマスを知覚することができる

6.2 実験結果

ここでは, 図 7 に示すように obstacle を (0, 2), (1, 1), (2, 1), (2, 3) の位置に設置した場合の結果を示す.

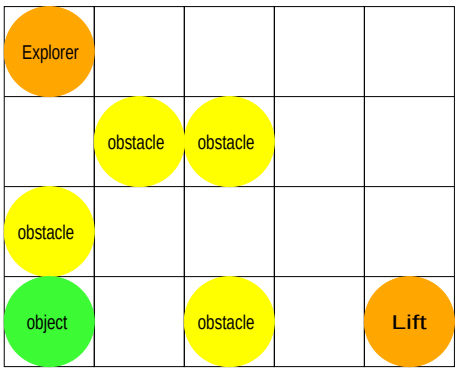


図 7: 配置図

実験の様子を図 8 に示す.

最初に 2 台のロボットは, それぞれ指定された探索範囲内で search を開始した (図 8(1)). その際に Lift が (2, 3) に存在する台を発見し, 探索中の Explorer に check を委託した (図 8(2)). このとき Explorer の現在位置は (0, 0) である. ここから目的地を (2, 2) と選択し, 移動経路を (0, 0), (1, 0), (2, 0), (2, 1), (2, 2) と選択した.

しかしその移動中, (2, 0) の位置で (2, 1) にある obstacle を新たに発見した (図 8(3)) ために, 経路再選択を行った. 再選択後の経路は, (2, 0), (3, 0), (3, 1), (3, 2), (2, 2) である. ここでは経路再選択が正しく行われ, obstacle との衝突が発生することなく目的地である (2, 2) への移動が完了した. またその後の作業である check も問題なく行われた.

そして Explorer は (2, 3) に存在する台を obstacle であると判定し, search を中断した位置である (0, 0) への移動を開始しようとした (図 8(4)). このとき Explorer の現在位置は (2, 2) であるため, 経路を (2, 2), (1, 2), (0, 2), (0, 1), (0, 0) と選択した. しかし先ほどと同様に, 途中で (0, 2) に存在する obstacle を発見したために経路の再選択を行った (図 8(5)). 再選択後の経路は, (1, 2), (2, 2), (3, 2), (3, 1), (3, 0), (2, 0), (1, 0), (0, 0) である.

一方 Lift は, check を依頼した (2, 3) にある台が obstacle であると分かったため, 現在位置である (3, 3) から search を続行する. しかしここでは Explorer との衝突の可能性が考えられる. そのため Lift は 5.2.2 節で述べた pause 機能により, Explorer の位置を確認してから search を続行する. 実験結果では図 8(5) および (6) より, Explorer が探索領域に戻るまで Lift が待機していたことが分かる. また Lift の待機中に Explroer が経路を再選択した場合でも, 衝突は起こらなかった.

続行した search において, Lift は再び (0, 3) に存在する台を発見し, Explorer に check を委託した (図 8(7)). ここでは Explorer の現在位置が (1, 0) であることから, 移動経路を (1, 0), (2, 0), (3, 0), (3.1), (3, 2), (2, 2), (1, 2), (1, 3) と選択した. 先ほど行った check とは異なり, この場合は Lift が Explorer の移動経路上に存在するために衝突の発生が予想される. そのため Explorer が移動を開始する前に, move_for_avoid 機能を使用した Lift の移動が必要である. ここでは Lift の回避場所を (4, 2) に決定した. Lift が移動する間, Explorer は wait_finished 機能により (1, 0) でしばらく待機を行った. そして Lift は stop_wait 機能により移動の完了を伝え, Explore は選択した移動経路に沿って (0, 2) へ移動することができた (図 8(8) および (9)). この結果より行き詰ま

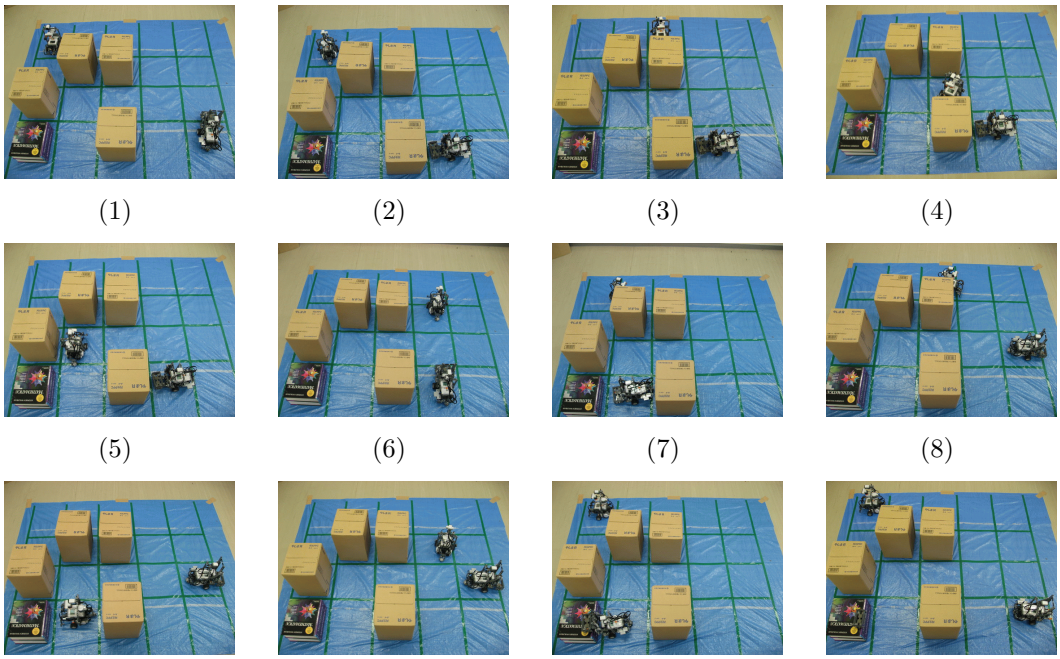


図 8: 実験結果

りが解消され, 衝突が発生することなく移動が行われたことが分かる. また行き詰まりが発生してからではなく, 事前に衝突回避を行うことで, より円滑に作業が進んでいると考えられる. なぜなら行き詰まりが発生してからでは, Explorer が来た経路を再び戻るという作業が必要になるからである.

その後 Explorer は (0, 3) に存在する台を object と判定したために, Lift に up を委託した. 今度は Explorer が Lift が行う作業の邪魔になるため, Explorer が先に移動しなければならない. ここでは Lift の移動経路が (4, 2), (3, 2), (2, 2), (1, 2), (1, 3) であることから, Explorer は回避場所を (3, 1) と決定した. 先ほどと同様に Explorer は移動完了を待機中の Lift に伝え, そのままスタート地点である (0, 0) まで return を行った (図 8(10) および (11)). そして Lift も同様に, (0, 2) で box を object 上に乗せたあと, スタート地点である (4, 3) まで return を行った. これで全ての作業が完了する (図 8(12)).

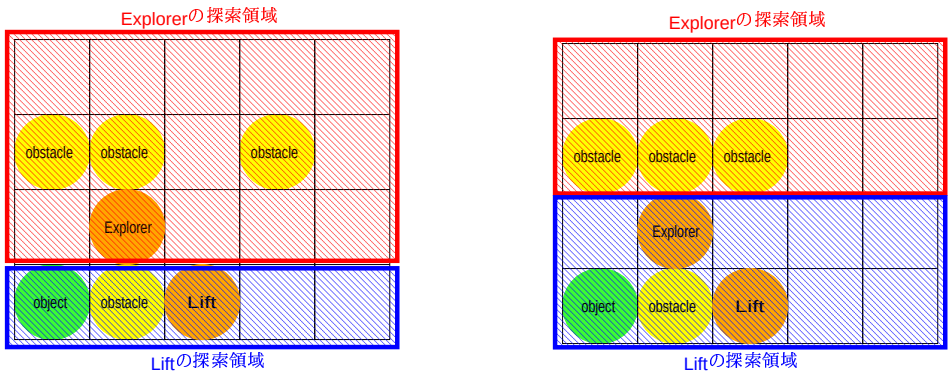
以上により, 衝突の可能性がある場面においてその回避が実現していることが分かる. また相手の移動経路から外れた位置に事前に移動するという回避方法により, 作業をより円滑に進めることが可能となったと考えられる.

7 考察

実験結果より, ロボットが連携をとりながら作業を進めることで, 衝突回避が可能となることが確認された. このことからロボット同士のコミュニケーションは重要であると考えられる. そして実装した機能を組み合わせることにより, 任意に設置した 4 つの obstacle に対して, 衝突が発生することなく作業が行われた.

しかし本論文で述べた手法は, 問題の初期設定によっては使用できない場合も存在する. 例えば 5.1 節 (b) で示した同時移動による衝突は, 5.2.2 節で示した pause 機能により Explorer が探索領域に戻るまで Lift が待機することによって解決した. しかし pause 機能では, 図 9(a) のように探索領域が指定された場合には衝突を回避することができない.

また図 9(b) のような obstacle の配置では, Lift は Explorer が探索領域に戻るまで待機する必要はない. ここで Lift が pause 機能を使用すると, Lift は Explorer が (3, 1) に移動するまで待機を行う. しかしこの場合は, Explorer が (3, 2) まで移動すれば衝突は発生しない. 更に Lift の待機時間も短くなる. したがってこの場合では, Explorer の回避場所は (3, 2) であることが望ましいが, 本研究の手法ではこのような最適な回避場所を求めることはできない.



(a) (b)

図 9: 本研究の手法において問題が生じる場合

このような pause 機能に伴う問題点の解決方法として, Lift が次に移動する経路を Explorer に与えることが考えられる. しかし探索を行っていない未知の領域では, 台の位置が明確でないた

め, そのような状況で選択された移動経路は曖昧なものとなる. また目的地の設定が難しくなり, どの地点までの経路が必要かという問題が発生する.

より柔軟な衝突回避を実現するには, エージェントが状況に応じて, どの位置に移動することで衝突回避が可能であるか, 更にどの位置に移動することで無駄のない回避が可能であるかということを考える必要がある. したがって, エージェントがどのように最適な回避場所を決定するかということが今後の課題となる.

また実験では, ロボットの移動誤差により台との衝突が発生し, 台の誤認識により作業が中断されてしまうという問題も発生した. よって今後は, 誤差の解消および誤認識の対処も必要である.

8 まとめ

本研究では BDI エージェントを搭載したロボットによる協調作業を行い, それに伴う衝突回避を実現した. 本研究の協調作業でロボットが行った作業は, search や check のように 1 台のロボットで可能なものばかりであった. しかし人間世界で行われる協調作業は, そのようなものばかりではない. したがって今後は使用するロボットの台数を増やし, 例えば複数のロボットで 1 つの荷物を運ぶような, より連携を必要とする作業を加えて実験を行いたいと考える. そして BDI エージェントによるロボット制御の実現に向け, 更なる技術向上を目指したい.

9 謝辞

本研究を遂行するにあたり, いつも親身に御指導をして下さった新出尚之准教授に深く感謝し, 厚く御礼申し上げます. また, 多くの助言を頂いた藤田恵先輩に感謝の意を表します. ありがとうございました.

参考文献

[1] Munindar P. Singh, Anand S. Rao, and Michael P. Georgeff, “Formal method in DAI”, Multiagent systems: a Modern Approach to Distributed Artificial Intelligence Research, pp.331-376, 1999.

[2] Anders Lemke, Johan Laidlaw, and Lars Zilmer-Pedersen, “Developing Multi-Agent Lego Robotics”, Technical University Of Denmark, 2007.

[3] Refael H. Bordini, Jomi Fred Hübner, and Michael Wooldridge, “Programming multi-agent systems in AgentSpeak using Jason”, WILEY, 2007.

[4] P. E. Hart, N. J. Nilsson, and B. Raphael, “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”, IEEE Transactions on Systems Science and Cybernetics SSC4 No.2, pp.100-107, 1968.

[5] 藤田恵, 片山寛子, 小島侑子, 新出尚之, “動的環境におけるでの BDI ベースロボットの再プランニングによる再行動決定法”, The 24th Annual Conference of the Japanese Society for Artificial Intelligence, 2010.

[6] 小山由, “動的環境における自律エージェントによる複数のロボットを用いた協調動作の実現”, 奈良女子大学理学部情報科学科 2010 年度卒業論文, 2011.