
University of Glasgow | School of Computing Science

Honours Individual Project Dissertation

# Development of Intelligent Agents with the BDI Framework

**Antonius Clarence Fionaldi**

April 1, 2022

# Contents

## Abstract

The BDI architecture is one of most popular approaches to developing intelligent agents where the (B)eliefs represent what the agent knows, the (D)esires what the agent wants to achieve, and the (I)ntentions those desires the agent has chosen to act upon. BDI agents can perceive the environments, deliberate to determine, for example, what plan to select under current beliefs, and act to interact with the environment. In this thesis, I will employ the BDI framework to develop two case studies and implement them in one of mature implementations

This paper will provide background, explanation, and technologies used to demonstrate and implement the said architecture to prove its potential viability.


.

# Introduction

## Aim

The aim for this project is to demonstrate and prove the capability and viability of BDI framework in developing autonomous agents. The motivation in this research is putting an autonomous/intelligent agent into practice. This is done by developing and putting a example intelligence agent in practice by the use of Jason platform. Therefore, the aim of this study/research is to develop two examples of intelligent agents with the implementation in Jason.

## Intelligent agents

"Intelligent agents are software entities that carry out some set of operations on behalf of a user or another program with some degree of independence or autonomy, and in so doing, employ some knowledge or representation of the user's goals or desires" (Franklin et al., 1996)

**Intelligent agents** are autonomous entities that perceive its environment and can interact with such. They can receive information (percepts) through their sensor and perform actions through their actuator (Bordini et al., 2007). The sensors and actuators provided are the core elements/definition of an intelligent agent. They are, therefore, reactive, as they continuously operate on the environment and can act from their acquired observations (Russel, 2002).

Belief-Desire-Intention paradigm is a popular framework to design intelligent agents (Bordini et al., 2007).

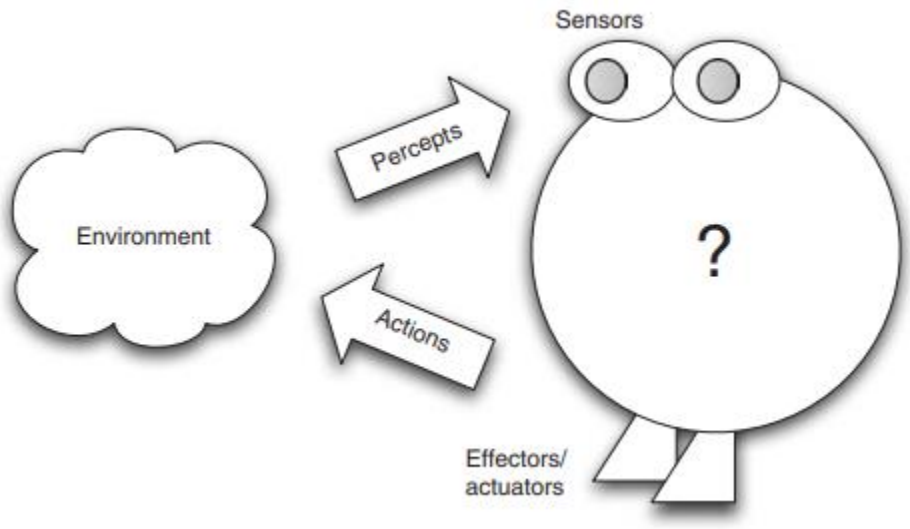


*Figure 1 - Agent and environment framework (Bordini et al., 2007)*

Belief is the current knowledge or perceptions of an agent. It represents the current information of data / beliefs. For example: if the agent found an apple and picked it up, it can be represented in the believe set as 'have(apple)'

Desire represents the goal that the agent wishes to complete, while intentions are the certain set of actions that the agent has chosen to do, to achieve the desire.

Event is the pre-defined update of belief/goals that is triggered by the environment or the agent's belief base. Example: If the agent found an apple, it would eat it.

## BDI

I chose BDI agent as it is a practical and simple implementation of an Artificial Intelligence system that can be used in a wide variety of systems. For example, a game master was made using BDI for the game "Neverwinter Nights", after evaluating player using it they found that it improves the game's experience, replayability, and overall interest (Luong et al., 2017)

Other examples are also seen in other fields such as, NPC in video games, robots in an industry, etc. One of the main ideas of BDI is its straightforwardness and simplicity ensures reliability and error-free implementation (Rivera- Villicana et al., 2018). Complex AI implementation could result in fatal consequences when there are bugs in the code. Industries, therefore, are very reluctant in implementing AI in its usage due to the risk it imposes (Yudkowsky, 2008). The simplicity of BDI programming ensures a bug-free code with sufficient testing (Bordini et al., 2007). Therefore, the examples developed in this paper will be using the BDI framework.

## Jason

'AgentSpeak' is a programming language designed for BDI architecture and multi-agents system. A particular implementation of AgentSpeak is 'Jason'. Jason differs from other AgentSpeak implementations by focusing directly and solely on the communication between the environment and the agents, making it ideal in making multi-agent systems (systems with multiple agents instead of one). It directly and imitatively operates on the direct semantics of 'Agent-Speak', making it easy to use and learn throughout (Bordini et al., 2007).

## Conclusion

I employ BDI framework and developed two concrete examples and implement them in Jason.

The first example is called '**Dillema**'. It involves a moving car facing a life-threatening dilemma. The car (agent) will attempt to make a decision that serves the safety and interest of the driver while avoiding pedestrians. The code I have made is a simple implementation of this. This code is designed to be an understanding of how BDI works and a simple demonstration.

The second example is called '**Travelling Salesman**'. Travelling Salesman is a Non Polynomial – Hard (NP-Hard) problem that requires brute-forcing the entire paths and comparing all the permutations to solve the shortest distance between the nodes. The example will be an implementation of this problem, as the agent will travel across all the possible paths and calculate the closest distance from all the permutations. This code is designed to be a more advanced/complex implementation of BDI.

# Background

BDI agent is a practical and simple implementation of an Artificial Intelligence agent system that can be used in a wide variety of systems. BDI handles and revolves with its belief-base and the subsequent plan-base. The actions done as a result of the agent's plan base to interact with its environment. BDI has been previously used in multiple situations, primarily in software. This framework has been used in the gaming industry. For example, a game master was made using BDI for the game "Neverwinter Nights", after evaluating player using it they found that it improves the game's experience, 'replayability', and overall interest (Luong et al., 2017)

Other examples are also seen in other fields such as, NPC in video games, robots in an industry, and other AI implementations. One of the main ideas of BDI is its straightforwardness and simplicity ensures reliability and error-free implementation (Rivera- Villicana et al., 2018). Complex AI implementation could result in fatal consequences when there are bugs in the code, this has been suggested by Yudkowsky in 2008. Industries, therefore, are very reluctant in implementing AI in its usage due to the risk it imposes (Yudkowsky, 2008). Other previous research has also highlighted the simplicity of BDI programming ensuring a bug-free code with sufficient testing (Bordini et al., 2007).

Belief-Desire-Intention paradigm is a popular framework to design intelligent agents (Bordini et al., 2007). The framework uses the notions of belief, desire, and intention to interact with the environment as perceived by the agent with the goal of accomplishing the agent's goal.

The general flow of BDI system would be:

- The agent has a certain desire that it wishes to achieve.
- The agent would have a believe base that it continuously have updates from the environment.
- Events can trigger due to the newly acquired beliefs, leading the agent to perform a certain set of actions that will fulfill an intention.
- Intentions are done to fulfill the desires of the agent.

The process would operate continuously an agent will keep interacting and perceiving its environment (Bordini et al., 2007).

For example, this is a simple idea of a BDI agent's process:

- A boy wants to eat an apple (Desire)
- The boy perceives an apple falling from a tree (Context/Environment)
- He acquired the belief that there is an apple for the taking (Belief)
- Therefore, he has the intention to eat it (Event -> Intention)
- He approaches the apple (Action)
- He picks it up, and eats it (Desire achieved)

Designing a BDI agent requires its set rules, beliefs, intentions, events, and plan rules to be specified and given formal semantics. All the information here are based on the CAN syntax (Winikoff et al., 2002).

BDI can be formally described by using a 3-element tuple:

$$\langle B, \Pi, \Lambda \rangle$$

$B$ : Initial belief base

$\Pi$ : Plan library

$\Lambda$ : Action plan library

And three types of predicates:

$!\,e$ : event predicate

$b$ : belief predicate

$act$ : action predicate

These predicates are used to act on terms, usually denoted as **t** to create atoms:

$!\,e(\boldsymbol{t})$ : event atom

$b(\boldsymbol{t})$ : belief atom

$act(\boldsymbol{t})$ : action atom

*B* is the set of beliefs of the agent. It represents facts/knowledge that the agent currently believes in relation to its environment. The belief predicate *b* is used to act on terms to create belief atoms, $b\{\boldsymbol{t_1}, \boldsymbol{t_2}, \dots, \boldsymbol{t_n}\}$. These belief atoms can be modified for negation i.e., **not** *b(**t**)* , ~*b(**t**)*. Which is a belief that it is **not** *b(**t**)*. Belief formulas (ϕ) can be made which set the rules of the creation/definition of a belief.

Π is the plan library of the agent. It contains the operational procedure for the agent to execute when given the right circumstances. The plan collection is formally defined as:

$$!e(\boldsymbol{t_1}) : \varphi(\boldsymbol{t_2}) \leftarrow \mathrm{P}(\boldsymbol{t_3})$$

where $e(\boldsymbol{t_1})$ as the triggering event, $\varphi(\boldsymbol{t_2})$ as the context condition, and $\mathrm{P}(\boldsymbol{t_3})$ as the plan body

In general terms, when $e(\boldsymbol{t_1})$ is triggered and condition $\varphi(\boldsymbol{t_2})$ fulfilled, then $\mathrm{P}(\boldsymbol{t_3})$ will be executed by the agent.

The agent does actions through its plan body, it serves to interact with the environment yet could change the belief base. The modifications/checks to the belief base can be:

(i) +*b(**t**)* , adding belief to the belief base
(ii) -*b(**t**)* , removing belief from the belief base
(iii) ?*b(**t**)* , conditional check for *b(**t**).* i.e., testing if *b(**t**)* returns true

Similarly, event goals can be expressed by:

(i) !*e(**t**)* , adding a desire/goal

(ii) -!*e(**t**)* , removing a desire/goal

The plan body therefore can be defined with the following syntax:

$$P ::= act \mid ?\phi \mid +b \mid -b \mid !e.$$

*act* = action

*?φ* = test goal

*+b* = belief addition

*-b* = belief removal

*!e* = event goal

| means '**or**' so the plan body can be easily understood as consisting of action **and/or** test goal **and/or** belief addition **and/or** belief removal **and/or** event goal

act denotes actions that will be conducted by the agent, regarding its environment.

*?φ* extracts *φ* from the belief base if it exists. The belief base might change during execution. The context condition of the plan extracts from the belief base before the execution, hence: the agent might work with an un-updated belief base.

*+b* adds an additional belief to the belief base. i.e., $B \cup \{b\}$

*-b* removes a belief from the belief base. i.e., $B \setminus \{b\}$

*!e* commands the agent to respond to this particular event goal

! denotes a goal, this is so to distinguish a triggering event (*e*) from an event goal (*!e*)

Ordering/sequencing between multiple agent programs uses the symbol ';' (semi-colon). The syntax *P1;P2* means executing P1 then P2, i.e., P1 followed by P2. The reverse also means the same, P2;P1 meaning the agent execute P2 then P1. Certain cases require the agent to do multiple commands concurrently.

Concurrency is done using the symbol ‖ . This however does not necessarily mean that both programs are done in exact simultaneous manner. It could mean that the two concurrent program ran in an 'interleaved concurrency' meaning one step in one program, and the next with the other program, and so forth.

From previous explanation, it implies that the desire/goal of the agent can only be achieved by enacting actions/intentions triggered by the environment. However, this may not necessarily be the case. An agent may want to have its desire satisfied when certain state is achieved by the agent. Example:

**eat(apple)** for the former, compared with **not(hungry)** for the latter

This type of objective is simply explained by having its declarative aspects met instead of an action taken to fulfill the goal. By the example, the agent knows that its goal is to be not hungry instead of eating an apple, as eating other fruits may fulfill the hungriness of the agent and the agent might still be hungry even after eating the fruit yet have its goal incorrectly fulfilled. The declarative syntax for this is:

$$goal(\phi_s, P, \phi_f)$$

$\phi_s$ : successful condition

$P$ : procedural program

$\phi_f$ : failure condition

$\phi_s$ is successful condition will be fulfilled when certain conditions are met, a.k.a. the $P$, procedural program.

$P$ are the procedural programs where it attempts to fulfill the condition for $\phi_s$ to be successfully achieved.

$\phi_f$ is the failure condition, indicating that the procedural programs have failed to fulfill the successful condition.

The procedural program will continuously attempt/retry to fulfill the conditions throughout execution until the successful condition or the failure condition are met. This addition of successful end goal and failed situation gives the agent the idea of the desired outcome of the goal, while also preventing the failure of the goal. The agent can not just assume that the desire is achieved after the goal is achieved, it will continuously attempt to fulfill the achieved condition even when it is already achieved.

In addition to the main agent programs, auxiliary agent programs are also employed to help with the functioning of the agent.

$$\text{nil} \mid \text{e} : (|\phi_1 : P_1, \ldots, \phi_n : P_n|) \mid P_1 \rhd P_2 \mid \text{goal}(\phi_s, \text{P}, \phi_f)$$

$\text{nil}$ : terminates the program

$\text{e} : (|\phi_1 : P_1, \ldots, \phi_n : P_n|)$ : when the event *e* is triggered, it executes a body of plans if the requirements (context condition) are met.

$P_1 \rhd P_2$ : executes the second program $P_2$ when the first program $P_1$ failed.

The action plan library $\Lambda$ is a collection of actions, denoted by the syntax:

$$\text{a}(\mathbf{x}) : \psi(\mathbf{x}) \leftarrow \varphi^-(\mathbf{x}); \varphi^+(\mathbf{x})$$

$\psi(\mathbf{x})$ : precondition

$\varphi^-(\mathbf{x})$ : deletion of belief atoms

$\varphi^+(\mathbf{x})$ : addition of belief atoms

Simply stated, when $\psi(\mathbf{x})$ is fulfilled then an existing belief is replaced by another belief ($\varphi^-(\mathbf{x}); \varphi^+(\mathbf{x})$)

# Design

## Dilemma

Two examples are developed to demonstrate the usefulness of BDI.

It is a simple car AI which will make difficult ethical decision when presented with an incoming pedestrian. The agent will have multiple decisions when faced with such dilemma. It can choose to sacrifice the car by hitting the side obstacles (such as trees or ponds) to save the crossing pedestrian yet sacrificing the driver. Otherwise, it could decide to hit the pedestrian to save the life of the driver. Depending on the circumstances and personality of the driver, the car will have to decide the best course of action.

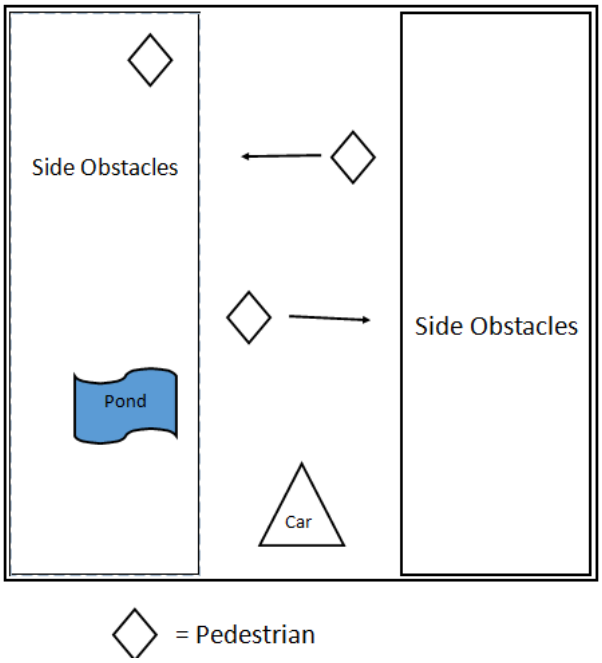


*Figure 2 - Diagram of the car environment*

Now, I will formalize this scenario in BDI syntax.

Firstly, if the car detects a crossing pedestrian from the environment, it will check if the car is able to brake, if so then it will stop

In relation with, $!e(\boldsymbol{t_1}) : \varphi(\boldsymbol{t_2}) \leftarrow P(\boldsymbol{t_3})$

**!stop: pedestrian --> check_brake;**

stop = $!e(\boldsymbol{t_1})$, triggering event

pedestrian = $\varphi(\boldsymbol{t_2})$, context condition

check_brake = $P(\boldsymbol{t_3})$, plan body

If it is able to brake, then will check the distance from the pedestrian to the car:

**!check_brake: can_brake --> calculate_distance;**

The distance determines the pace of braking, to ensure a smooth stop.

**!calculate_distance: far --> slowly_brake;check_pedestrian_status;**

**!calculate_distance: mid --> casually_brake;check_pedestrian_status;**

**!calculate_distance: close --> instantly_brake;check_pedestrian_status;**

Afterwards, it will continuously check if the pedestrian is still in front of the car until it passes

**!check_pedestrian_status: pedestrian --> check_pedestrian_status;**

**!check_pedestrian_status: no_pedestrian --> go;**

However, if it is unable to brake then the program will check the surrounding environment for a suitable area to crash into.

**!check_brake: cannot_brake --> check_surroundings;**

If there are safe spots that the car can crash into, then it will crash there.

**!check_surroundings: safe_area --> crash_into_safe_area;**

Else, it will check the fragility of the pedestrian if the agent is unable to find any safe spot to crash into.

**!check_surroundings: unsafe_area --> check_pedestrian_fragility;**

If the pedestrian is strong then it will crash the car into the pedestrian, knowing it will most likely survive.

**!check_pedestrian_fragility: strong --> crash_into_pedestrian;**

Otherwise check the personality of the driver.

**!check_pedestrian_fragility: fragile --> check_driver_personality;**

If the driver is selfless then it will crash into the sidewalk knowing it would likely to injure the driver.

**!check_driver_personality: selfless --> crash_into_sidewalk;**

Otherwise, it the driver is selfish then it will crash into the pedestrian.

**!check_driver_personality: selfish --> crash_into_pedestrian;**

The program is by no means a practical one to be used in real scenarios. The program only serves as a simulation and an idea of what BDI agent could do in a possibly real-life scenario. It only serves as a demonstration and simple/general implementation of BDI that is easily learned and understood.

## Travelling Salesman

The second example is a much more complex implementation of BDI that shows the extent of BDI capability.

This is an autonomous agent that calculates the distance between several cities by traveling and compares the overall shortest distance from all possible routes. For example, when one wants to visit Paris, Berlin, and Moscow, from London; the fastest route would be heading to Paris, and then Berlin, and finally Moscow instead of Moscow, Paris, and Berlin.

i.e., London -> Paris -> Berlin -> Moscow

will be the faster/shorter distance than,

London -> Moscow -> Paris -> Berlin

It is the Travelling Salesman Problem (TSP), where the only way to solve this is by brute-forcing the permutations, i.e comparing all permutations. Simplifying it has been a challenge for researchers as it is an NP-Hard problem. The implementation below is not an attempt to solve the NP-Hard complexity, yet only to demonstrate the generality and flexibility of BDI.

The environment consists of nodes (cities) and paths (route) between the nodes. The agent will traverse through all possible routes in the map. Initially, all nodes are '**available**' (except the host node) and all paths are labelled as '**unvisited'**. The nodes that have been traversed by the agent will be considered as '**unavailable**', and then paths that has been traversed by the agent will be labeled as '**visited'**.

Note: the paths are bi-directional and does not concern origin/destination cities

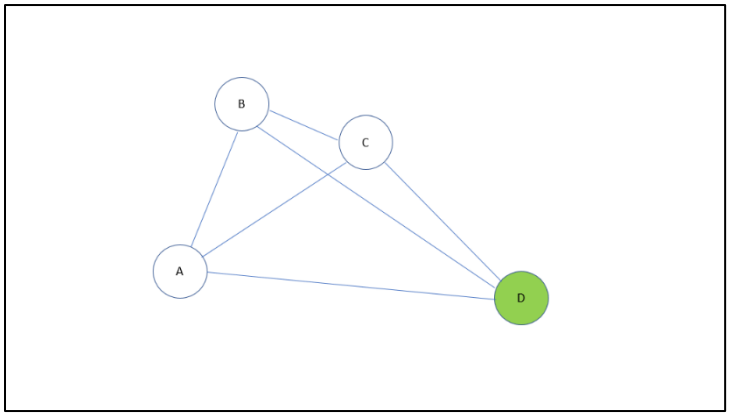


*Figure 3.0 – Example, 4 cities namely A,B,C,D with D as the host. Green indicates the current location of the Agent.*

The pseudocode for my implementation is as follows:

```
generate_permutation: available_paths --> check_paths;

//does not include paths where it originates or path to starting node
check_paths: available_nodes && unvisited_paths --> goto_path; check_paths;
check_paths: available_nodes && all_paths_visited --> unvisit_connected_paths; go_back; visit_previous_path;check_paths;

check_paths: unavailable_nodes --> go_back; visit_previous_path; compare_permutation;

go_back: no_path_back --> finish;
go_back: path_back --> go_previous_node;

compare_permutation: shortest_distance --> finish;
compare_permutation: not_shortest_distance --> check_paths;
```

The code explanation is crucial in understanding the next example.

Code explanation:

**available_paths** : checks if there are any unvisited paths (paths that are not labeled as '**visited**')

**check_paths** : checks if there are available nodes, if so check if there are unvisited paths, will do a different set of actions for each circumstances. Does not include the host node or paths to the host node.

**available_nodes** : checks if there are any available nodes (cities), meaning cities that are declared as '**available**' by the agent

**unvisited_paths** : checks if there are any paths that are not '**visited**', meaning it is not labeled as '**visited**' by the agent

**goto_path** : the agent will move into that node and will declare that path as '**visited**' and that node as '**unavailable'**

**go_back** : will go back to the node where it originally came from, freeing the current node (making it '**available**' and freeing the path (making it '**unvisited**')

**unvisit_connected_paths** : unvisit all paths from this node, meaning it will declare all paths from/to this node as '**unvisited**'. But since it is only used in check_paths, and check_paths does not include paths to/from the host node, there will be no cases in this program well the path to the host node will be set as '**unvisited'** by this action.

**visit_previous_path** : unvisit the previous path the agent just went from, meaning it will declare the previous path from '**visited**' to '**unvisited**'

Example situation with 4 nodes (cities):

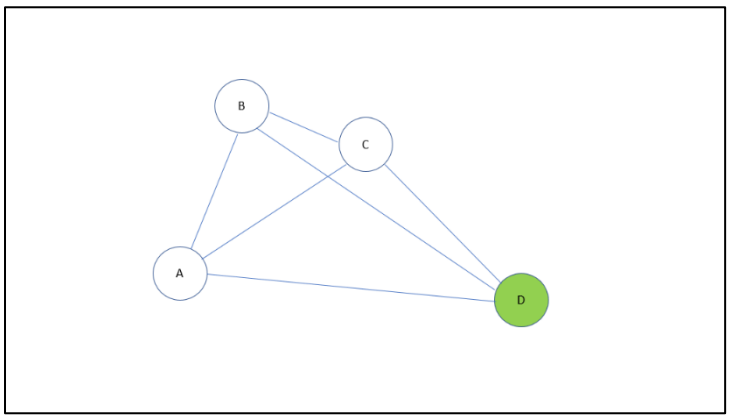


*Figure 3.1 - Starting situation, the agent is currently in host node D*

**!generate_permutation: available_paths --> check_paths;**

will check if there's available path from the starting node (in this case D)

It will then proceed with:

**!check_paths: available_nodes && unvisited_paths --> goto_path; check_paths;**

Since all paths connected are unvisited and the respective nodes available, it will go to any of the connected nodes. For example, the node A

So **D <-> A**

It will then repeat the same event however, it will not include paths where it originates from or path to the host node.

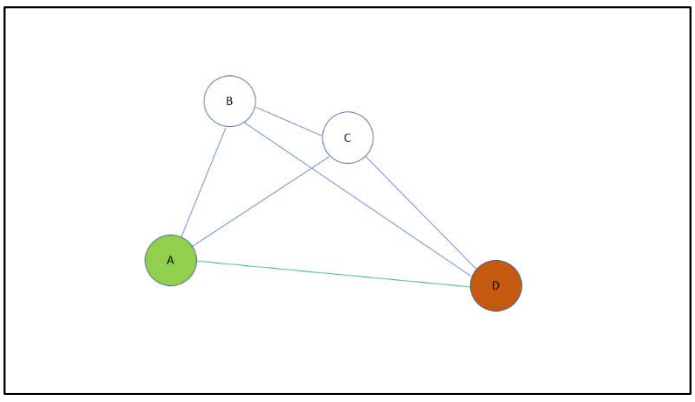


*Figure 3.2 - After D -> A, the agent is currently in node A, with the path between A and D (A <-> D) labelled as visited.*

The **A <-> D** path is not considered

It will choose another connected and unvisited path, and if the corresponding node's available.

In this case **A <-> C**

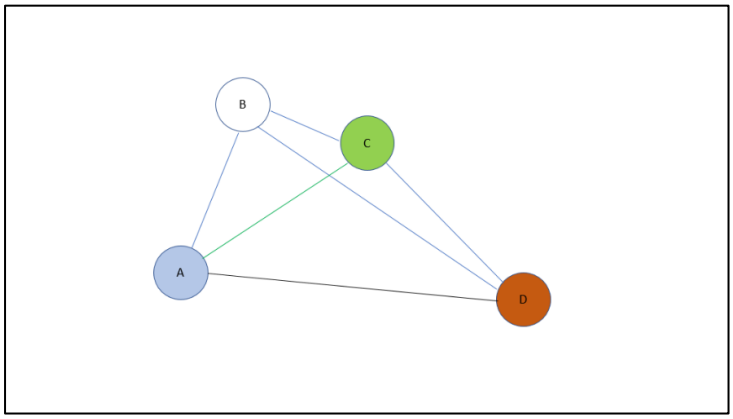


*Figure 3.3 - A -> C*

Then finally C **<-> B**

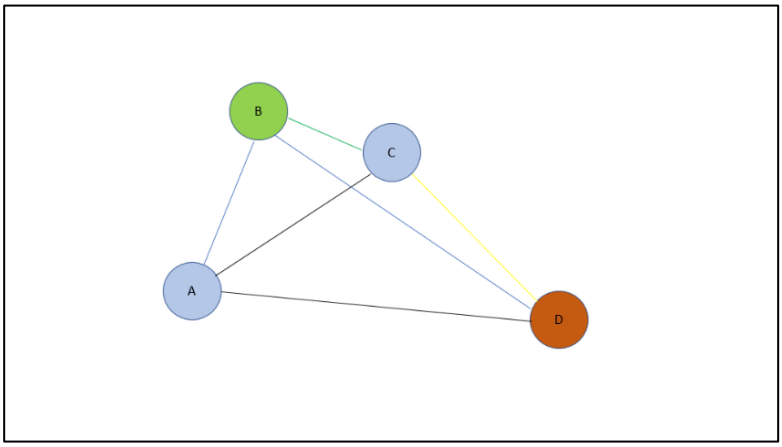


*Figure 3.4 - C -> B*

The path from C -> D is highlighted yellow as it goes to the starting node, and it is not considered.

Since there are no more available nodes.

**!check_paths: unavailable_nodes --> go_back; visit_previous_path; compare_permutation;**

(**visit_previous_path** means declaring the previous path it came from as visited, despite being unvisited by go_back, in this case the path **C <-> B** is declared as visited despite it should be unvisited by **go_back**)

Then there are two possible outcomes:

**!compare_permutation: shortest_distance --> finish;**

**!compare_permutation: not_shortest_distance --> go_back; check_paths;**

It will check if it is the shortest distance overall, if not then it will continuously search for it.

In this case it is assumed that the shortest distance is not found, then the second line of code, is run instead.

It will go back the path (**B <-> C**), freeing the node as available and labeling the path as “visited” (highlighted as red)

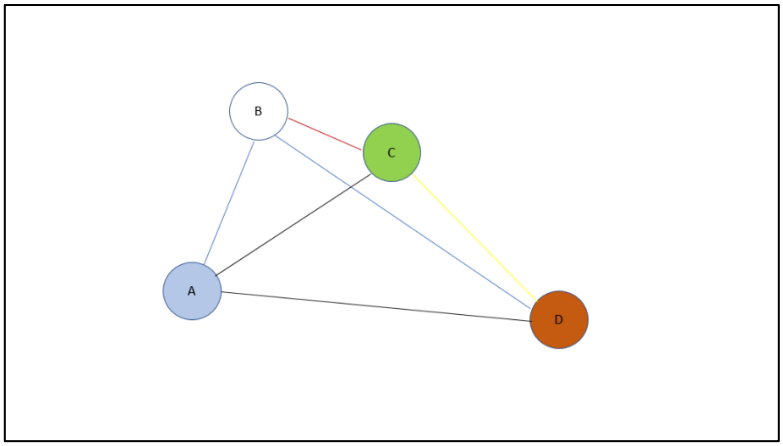


*Figure 3.5 - Back from B to C*

All connected paths are either visited (**C <-> B**), originates from (**A <-> C**), or heads to the starting node (**C <-> D**).

It will unvisit any connected visited paths, and visit the originating path (in this case **A <-> C**) before heading back to A

**!check_paths: available_nodes && all_paths_visited --> unvisit_connected_paths; go_back; visit_previous_path; check_paths;**

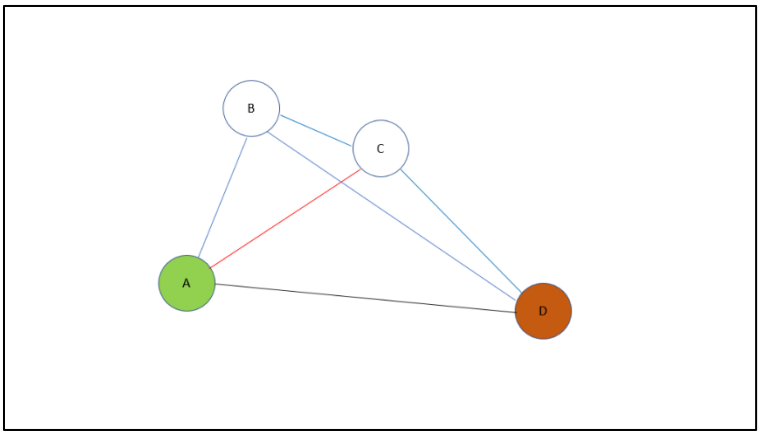


*Figure 3.6 - A <- C*

Since there is an available node and an unvisited path (**A <-> B**) it will head there.

**!check_paths: available_nodes && unvisited_paths --> goto_path; check_paths;**

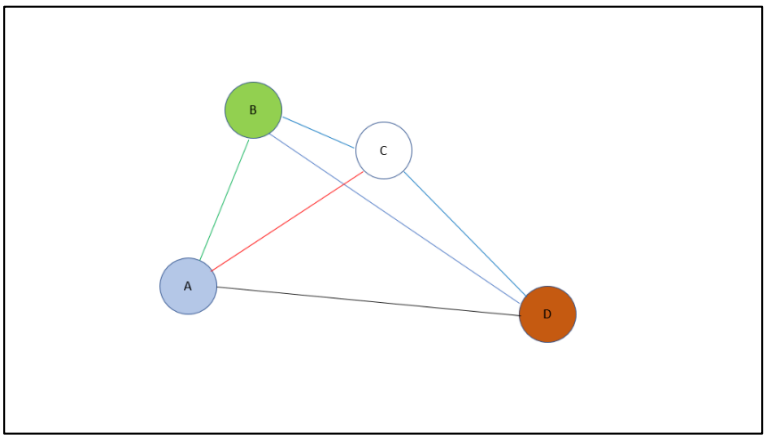


*Figure 3.7 - A -> B*

The process repeats, and now it goes from **B -> C**

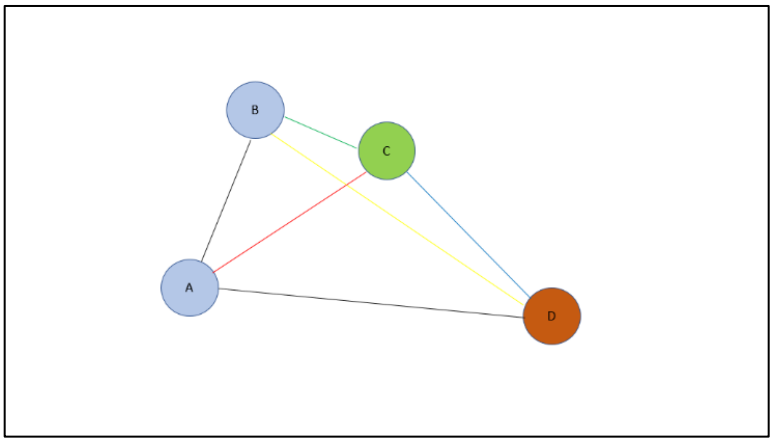


*Figure 3.8 - B -> C*

The agent will check if there are unavailable nodes:

**!check_paths: unavailable_nodes --> go_back; visit_previous_path; compare_permutation;**

Since all nodes are unavailable it will go to the previous node, visit the path to that node, and compare the permutations

**!compare_permutation: shortest_distance --> finish;**

**!compare_permutation: not_shortest_distance --> go_back; check_paths;**

The agent compares if this route is the shortest distance overall, assuming this is not then the agent will visit **B <-> C** and goes back to **B.**

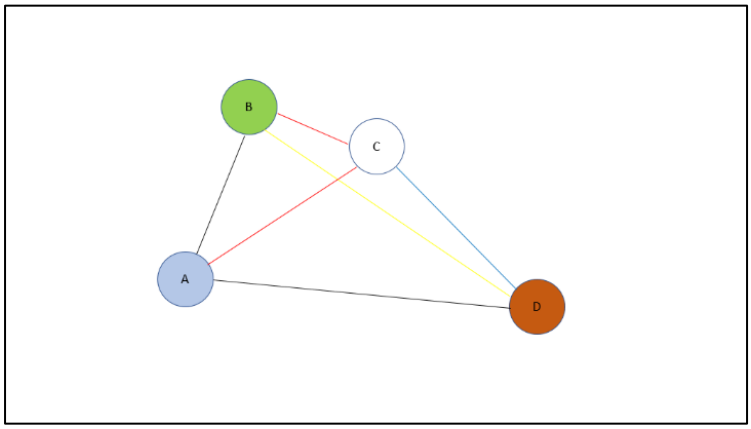


*Figure 3.9 - B <- C*

Since now all the paths are visited (it does not consider path to the host node (**B <-> D**) or the original path it came from (**A <-> B**)), and there is still an available node (**C**).

it unvisits **B <-> C** and goes back to **A**

**!check_paths: available_nodes && all_paths_visited --> unvisit_connected_paths; go_back; visit_previous_path; check_paths;**

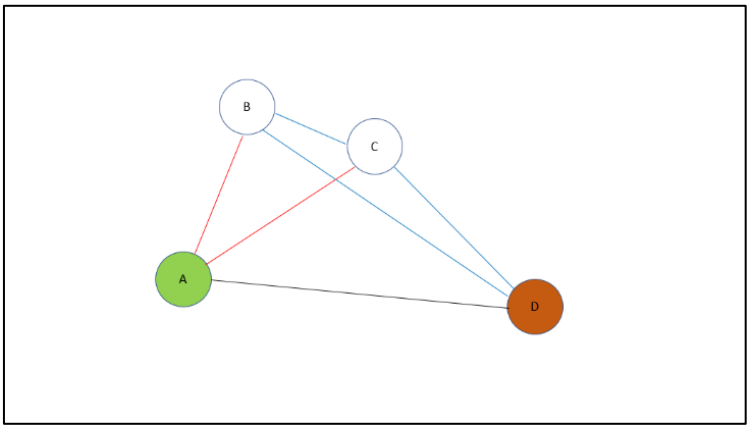


*Figure 3.10 - A <- B*

The process repeats, it unvisits **A <-> B** and **A <-> C** before visiting **D <-> A** and heading back to **D**

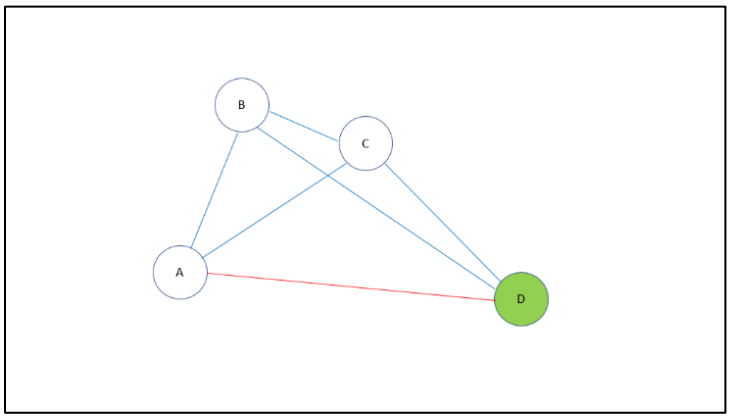


*Figure 3.11 - D <- A, back to the starting position.*

Therefore, all permutations deriving from **D <-> A** has been checked:

**D -> A -> C -> B**

**D -> A -> B -> C**

As now the **D <-> A** path is labeled “visited” the BDI Agent will repeat the same processes until all permutations are considered.

All the permutations will be stored and compared to output the permutation/combination of paths that has the least distance.

# Implementation

Implementing both example cases use Jason.

Both examples are based on ‘Cleaning Robots’ by Rafael Bordini and Jomi Hübner (Bordini, R. et al.).

Dillema

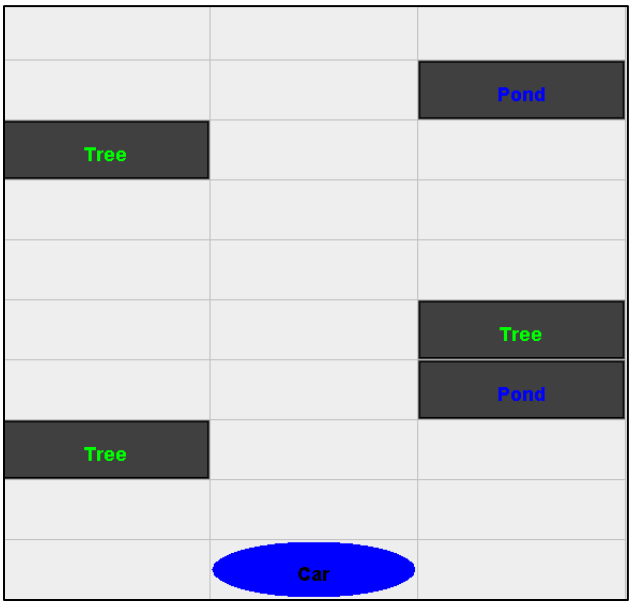


*Figure 4.1 - Dilema*

Travelling Salesman

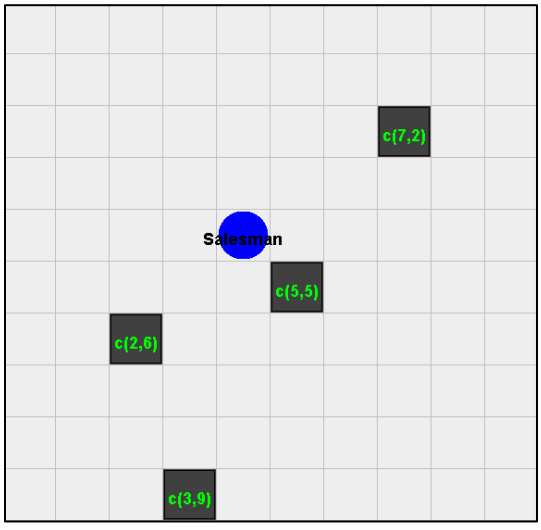


*Figure 4.2 - Travelling Salesman*

The program consists of the agent (programmed in Jason) and the environment (programmed in Java) which the agent interacts with. For every action, the agent does, a corresponding function in the environment implement the action.

## Example 1: Dillema

The program is designed with a more java-emphasis since the reader is more likely more familiar with java and this program is designed to be more of an understandable/familiarization code.

A Jason file with its environment and agent is set-up, the agent is a moving car in this program.

```
MAS dillema {

    environment: DillemaEnv

    agents: car;
}
```

*Figure 5.1 - Environment and Agent*

### Agent

The agent has a desire to move `!move.`

It will continuously do so while there are no pedestrians

```
+!move : not pedestrian
    <- move;
    !move.
```

It there the agent detects a pedestrian from the environment, it will have the intention to stop.

```
+!move: pedestrian
    <- !stop.
```

When there is the intention to stop, it will check the brakes.

```
+!stop <- !check_brake.
```

If the car is able to brake, then it will remove previous existing believe base regarding the pedestrian, and wait/stop until the pedestrian passes.

```
+!check_brake : can_brake & close
    <-
    !reset_beliefs;
    wait;
    !move.

+!check_brake : can_brake & mid
    <-
    !reset_beliefs;
    !move.

+!check_brake : can_brake & far
    <-
    !reset_beliefs;
    !move.
```

If it is **unable** to brake then it will check the surrounding environment for a safe place to crash into. If it found one, then it will crash into the safe object/obstacle.

```
+!check_brake : not can_brake & far
    <- check_surroundings_far.

+!check_brake : not can_brake & mid
    <- check_surroundings_mid.

+!check_brake : not can_brake & close
    <- check_surroundings_close.
```

If the car is unable to find a safe place to crash, then it will check the fragility of the pedestrian.

```
+unsafe <- !check_fragility.
```

Then the decision depends on the circumstances of the pedestrian and the personality of the driver.

If the pedestrian is fragile then it will check the personality of the driver

```
+!check_fragility : fragile
    <- !check_personality.
```

If the driver is selfless then it will crash into the side of the road.

```
+!check_personality : selfless
    <- hit_any_direction.
```

Otherwise if the driver is selfish, it will crash into the civilian.

```
+!check_personality : selfish
    <- hit_pedestrian.
```

If the pedestrian is strong (not fragile) and far ahead it will crash into the pedestrian, otherwise the same case apply as above with the agent checking the personality of the driver.

```
+!check_fragility : strong & far
    <- hit_pedestrian.

+!check_fragility : strong & not far
    <- !check_personality.
```

## Environment

```
public void init(String[] args) {
    model = new DillemaModel();
    view  = new DillemaView(model);
    model.setView(view);
    updatePercepts();
}
```

*Figure 5.2 - Initialization of the main model class and the view model.*

The environment consists of the model that is the main class of the environment, and view which serves to visualize the environment.

```
class DillemaModel extends GridWorldModel {
    boolean going_to_crash = false;
    boolean safe = false;

    Random random = new Random(System.currentTimeMillis());

    private DillemaModel() {
        super(3, GSize, 50);


        try {
            setAgPos(0, 1, 9);


        } catch (Exception e) {
            e.printStackTrace();
        }


        add(HARD, 0,1);
        add(SOFT, 2,0);

    }
```

*Figure 5.3 - DillemaModel*

The model creates a private function that establishes the location of the car **setAgpos(0,1,9)**

and the size of the model **super(3, GSize, 50)**

it also initializes a hard object at position **0,1** and a soft object at position **2,0**

The model contains multiple functions a.k.a. actions for each cycle of the program.

i. **cycle()** : cycle/running of the simulation when the agent faces no apparent threat.
ii. **_wait()** : the agent waits until the pedestrian goes through.
iii. **checkSurroundings(String distance)** : scan the environment for a suitable object that the agent (car) can crashes into, if so, crashes so.
iv. **hitSide()**: crashes the car into the side.

**void cycle()**

this action/function moves every object that is not the agent forward by one to simulate a moving vehicle.

It also randomly generates a pedestrian in front of the car at a random distance to simulate a moving pedestrian.

The pedestrian can be an old lady or a body builder to imply a weak or strong pedestrian.

```
void cycle() {

    for(int y=9; y>=0;y--){
            if (model.hasObject(HARD, 0 , y)){
                remove(HARD, 0, y);
                if(y!=GSize-1)set(HARD,0,y+1);
            }
            else if (model.hasObject(SOFT, 0 , y)){
                remove(SOFT, 0, y);
                if(y!=GSize-1)set(SOFT,0,y+1);
            }
            if (model.hasObject(HARD, 2 , y)){
                remove(HARD, 2, y);
                if(y!=GSize-1)set(HARD,2,y+1);
            }
            else if (model.hasObject(SOFT, 2 , y)){
                remove(SOFT, 2, y);
                if(y!=GSize-1)set(SOFT,2,y+1);
            }
            if (model.hasObject(OLDLADY, 1 , y)){
                remove(OLDLADY, 1, y);
                boolean pass = random.nextInt(100) < 15;
                if(y==GSize-1);
                else if (!pass)set(OLDLADY,1,y+1);
            }
            if (model.hasObject(BODYBUILDER, 1 , y)){
                remove(BODYBUILDER, 1, y);
                boolean pass = random.nextInt(100) < 15;
                if(y==GSize-1);
                else if (!pass)set(BODYBUILDER,1,y+1);
            }
    }
```

*Figure 5.4 - cycle*

```
if (random.nextBoolean() && random.nextBoolean()) {
    if (random.nextBoolean()) add(HARD, 0,0);
    else add(SOFT, 0,0);
}
if (random.nextBoolean() && random.nextBoolean()) {
    if (random.nextBoolean()) add(HARD, 2,0);
    else add(SOFT, 2,0);
}

if (random.nextInt(100) < 5){
    add(OLDLADY,1,random.nextInt(9));
}
else if (random.nextInt(100) < 5){
    add(BODYBUILDER,1,random.nextInt(9));
}
}
```

*Figure 5.5 - cycle*

In addition to pedestrians, it randomly generates a hard/soft object on the sidewalk.

### void _wait()

```
void _wait(){

    for(int y=9; y>=0;y--){
            if (model.hasObject(OLDLADY, 1 , y)){
                if(random.nextBoolean() && random.nextBoolean())remove(OLDLADY, 1, y);
            }
            else if (model.hasObject(BODYBUILDER, 1 , y)){
                if(random.nextBoolean() && random.nextBoolean())remove(BODYBUILDER, 1, y);
            }
    }


}
```

*Figure 5.6 - wait*

The action simulates a car waiting, it randomly attempts to remove the existing pedestrians to simulate a crossed pedestrian.

### void checkSurroundings(String distance)

```
void checkSurroundings(String distance){
    going_to_crash = true;
    int y=8;
    switch(distance){
        case "close":
            y = 8;
            break;
        case "mid":
            y = 7;
            break;
        case "far":
            y = 6;
            break;
    }

    for(;y<9;y++){
        if(model.hasObject(SOFT, 0 , y)){
            setAgPos(0, 0, y);
            safe = true;
            break;
        }
        else if(model.hasObject(SOFT, 2 , y)){
            setAgPos(0, 2, y);
            safe = true;
            break;
        }
    }

}
```

*Figure 5.7 - checkSurroundings*

Declare the variable ‘going_to_crash’ as ‘true’ indicating that the car is going to crash.

Attempts to crash into the farthest soft/safe object within the car’s distance.

If able so, it will indicate that it is safe.

The variables are important for the perception update.

### void hitSide()

Crashes the car into the left side of the road.

```
void hitSide(){

    setAgPos(0, 0, 8);

}
```

*Figure 5.8 hitSide*

The java environment does these actions by the order of the agent.

```
@Override
public boolean executeAction(String ag, Structure action) {
    logger.info(ag+" doing: "+ action);
    try {
        if (action.equals(move)){
            model.cycle();
        }
        else if (action.equals(wait)){
            model._wait();
        }
        else if (action.equals(csc)){
            model.checkSurroundings("close");
        }
        else if (action.equals(csm)){
            model.checkSurroundings("mid");
        }
        else if (action.equals(csf)){
            model.checkSurroundings("far");
        }
        else if (action.equals(hp)){
            model.cycle();
            model.cycle();
            model.cycle();
        }
        else if (action.equals(ha)){
            model.hitSide();
        }
```

*Figure 5.9 executeAction*

If the car decides to crash into the pedestrian ‘hp’, it simply runs the cycle 3 times as it is guaranteed it will crash into the pedestrian.

After the action is done, it will update the percepts

```
updatePercepts();
```

```
void updatePercepts() {
    clearPercepts();

    Location location = model.getAgPos(0);
```

Firstly, it will clear existing environment wise-percepts to the agent, it then takes in the locations of the grid within car's view where there could be pedestrians.

```
if (model.going_to_crash == true){
    if(model.safe == false){
        addPercept(u); //unsafe
    }
}
```

If the java environment knows that the car is going to crash and there are no safe spots to crash into, it will add 'unsafe' belief into the agent.

```
if (model.hasObject(OLDLADY, location)) {
    addPercept(p); // pedestrian
    addPercept(c); // close
    addPercept(fr);// fragile
}
else{
    location.y=7;
    if (model.hasObject(OLDLADY, location)) {
        addPercept(p);  //pedestrian
        addPercept(m);  //mid
        addPercept(fr); //fragile
    }
    else{
        location.y=6;
        if (model.hasObject(OLDLADY, location)) {
            addPercept(p);  //pedestrian
            addPercept(f);  //mid
            addPercept(fr); //fragile
        }
    }
}
```

The car will check ahead if there is an old lady. If so, it will update the agent's belief with the pedestrian, distance to pedestrian (close, mid, fragile), and vurnerability of pedestrian (fragile).

The same case would apply for the next type of pedestrian (body builder).

```
location.y=8;
if (model.hasObject(BODYBUILDER, location)) {
    addPercept(p); // pedestrian
    addPercept(c); // close
    addPercept(s); // strong
}
else{
    location.y=7;
    if (model.hasObject(BODYBUILDER, location)) {
        addPercept(p); // pedestrian
        addPercept(m); // mid
        addPercept(s); // strong
    }
    else{
        location.y=6;
        if (model.hasObject(BODYBUILDER, location)) {
            addPercept(p); // pedestrian
            addPercept(f); // mid
            addPercept(s); // strong
        }
    }
}
```

In this case, body builder represents a strong type of pedestrian.

## Example 2: Travelling Salesman

The program is almost entirely made in BDI, with the java environment playing a minimal role only in storing and comparing the permutations made by the agent

Firstly, as the same with the previous example, the environment and the agent are set-up.

```
MAS travelling {

    environment: TravellingEnv

    agents: traveller;
}
```

*Figure 6.1 - travelling*

### Agent

The agent has the intention to start `!start.`

`!start` sets up the percepts that the agent needs to know.

```
+!start
    <- set_up;
        !set(cities);
        !set(paths);
        !check(nodes).
```

`set_up` action receives all the location of the cities from the java environment

`!set(cities)` converts the source of belief of all the cities from the environment, into the agent's source as they are not deleted by each cycle.

```
+!set(cities) : city(X,Y,Z)[source(percept)]
    <- -city(X,Y,Z)[source(percept)];
       +city(X,Y,Z);
       !set(cities).
+!set(cities).
```

`!set(paths)` establishes the paths between all of the cities.

```
+!set(paths) : city(X,Y,_) & city(I,J,_) & path_not_exist(X,Y,I,J) & city_not_same(X,Y,I,J)
    <- +path(X,Y,I,J,unvisited);
       !set(paths).
+!set(paths).
```

The conditions are as follows for a path to be established in both cities:

(i) `city(X,Y,_) & city(I,J,_)` two distinct cities must exist
(ii) `path_not_exist(X,Y,I,J)` the paths must not exist yet

(iii) `city_not_same(X,Y,I,J)` It must not be the same city

The rule for `path_not_exist(X,Y,I,J)` is as follows:

```
path_not_exist(X,Y,I,J) :-  not path(I,J,X,Y,_) & not path(X,Y,I,J,_).
```

The rule for `city_not_same(X,Y,I,J)` is as follows:

```
city_not_same(X,Y,I,J) :- (X = I & Y \== J) | (X \== I & Y = J) | (X \== I & Y \== J).
```

Breaking down the rules :

- `(X = I & Y \== J)` if both cities have the same x coordinate then the y coordinate must not be the same
- `(X \== I & Y = J)` vice versa, both cities must not have the same x coordinate if the y coordinate is the same
- `(X \== I & Y \== J)` both cities x coordinates and y coordinates must **not** be the same

Since the paths are bi-directional and does not concern the origin/destination cities, many programs have the reverse function where location of both cities might be reversed in the path. i.e.,

`path(3,5,7,9,unvisited)` Is considered the same with `path(7,9,3,5,unvisited)` and there are 2 separate programs that handle for both different case.

All cities starts of with being available except the host city which will always be unavailable.

The program starts by checking the nodes i.e., cities

```
+!check(nodes) : city(_,_,available)
    <-  ?current_location(Ac,Bc);
        !check_visited(Ac,Bc).
```

If there are still available cities then the program will check the availability (visitedness) of all paths from the current city.

```
+!check_visited(Ac,Bc) : path(Ac,Bc,A,B,unvisited) & path_valid(Ac,Bc,A,B)
    <-  -path(Ac,Bc,A,B,unvisited);
        +path(Ac,Bc,A,B,visited);
        !travel(Ac,Bc,A,B).
```

The program to travel is as follows:

```
+!travel(As,Bs,A,B)
    <-  -city(A,B,available);
        +city(A,B,unavailable);
        +from(A,B,As,Bs);
        !go_to(A,B).

+!go_to(A,B): current_location(A,B) <- !check(nodes).
+!go_to(A,B)
    <- go_to(A,B);
        -current_location(_,_);
        !go_to(A,B).
```

It will declare the target city as unavailable and adds the current path as the path where it originates from.

If there are unvisited paths between the current city, and the path is valid (the city is available, and is not the path from the previous city that the agent (salesman) came from, it will mark the path as visited and travel into that city.

The reverse program holds the same :

```
+!check_visited(Ac,Bc) : path(A,B,Ac,Bc,unvisited) & path_valid(Ac,Bc,A,B)
    <-  -path(A,B,Ac,Bc,unvisited);
        +path(A,B,Ac,Bc,visited);
        !travel(Ac,Bc,A,B).
```

If all the valid paths are visited then it will unvisited all connected paths (except the path between the current city and the previous city, and the path into the host city) and travel back into the previous city. Afterwards it will check for available nodes (cities).

```
+!check_visited(Ac,Bc) : from(Ac,Bc,A,B)
    <- !unvisit_connected_paths(Ac,Bc);
       !go_back(Ac,Bc,A,B);
       !check(nodes).

+!unvisit_connected_paths(Ac,Bc) : path(Ac,Bc,A,B,visited) & path_valid(Ac,Bc,A,B)
    <- -path(Ac,Bc,A,B,visited);
       +path(Ac,Bc,A,B,unvisited);
       !unvisit_connected_paths(Ac,Bc).
```

The reverse program also holds for the above program :

```
+!unvisit_connected_paths(Ac,Bc) : path(A,B,Ac,Bc,visited) & path_valid(Ac,Bc,A,B)
    <- -path(A,B,Ac,Bc,visited);
       +path(A,B,Ac,Bc,unvisited);
       !unvisit_connected_paths(Ac,Bc).
```

If there are no more available cities, it will go back into the previous city and check the nodes again

```
+!check(nodes)
    <- ?current_location(Ac,Bc);
       ?from(Ac,Bc,A,B);
       !go_back(Ac,Bc,A,B);
       !check(nodes).
```

`!go_back` sets the current city as available and goes into the previous city

```
+!go_back(As,Bs,A,B)
    <- -city(As,Bs,unavailable);
       +city(As,Bs,available);
       -from(As,Bs,A,B);
       !go_to(A,B).
```

If there are no more valid paths between the city, and no previous cities, then it must holds true that the agent has finished all permutations and are currently in the host city with no possible path combinations left. The program will then finish and the permutations are calculated.

```
+!check_visited(Ac,Bc)
    <- +finish.
```

## Environment

The environment only contains 2 actions available for the agent

```
void goTo(int x , int y){
    Location t = getAgPos(0); //traveller

    if (t.x < x)       t.x++;
    else if (t.x > x)  t.x--;
    if (t.y < y)       t.y++;
    else if (t.y > y)  t.y--;

    setAgPos(0, t);
}
```

*Figure 6.2 - goTo*

goTo action moves the agent by one step into a certain direction.
setUp tells the environment to set-up the cities for the agent to traverse.

```
void setUp(){
    start = true;
}
```

*Figure 6.3 - setUp*

One example used here is having 4 cities set up.

```
if(TravellingModel.start == true){

    Literal c = Literal.parseLiteral("city(5,5,unavailable)");
    addPercept(c);
    c = Literal.parseLiteral("city(2,6,available)");
    addPercept(c);
    c = Literal.parseLiteral("city(3,9,available)");
    addPercept(c);
    c = Literal.parseLiteral("city(7,2,available)");
    addPercept(c);


    TravellingModel.start = false;
}
```

*Figure 6.4 – 4 different cities with different coordinates are set up, with city(5,5) as the host city.*

# Evaluation and Conclusion

BDI has been an interesting topic and programming paradigm for me to learn. Its simplicity and straightforwardness makes it a suitable and ideal paradigm to program basic Artificial Intelligences (AI).

While there are more advanced AI using more sophisticated technologies/techniques, BDI provides a way of developing and creating AI using standard programming methods. Its simplicity makes it practical and fast to develop/create, ideal for simple software AI that does simple objectives. Its simplicity also practically guarantees a bug-free project as debugging the code requires minimal time within my experiences so far with BDI.

One problem in programming with Jason is that it can easily be done to code the entire project in the java environment. It is entirely possible to create all the examples made entirely within the environment without any involvement from the agent. This would make it a standard java code and not a BDI code. For the first example, I intentionally make the program java heavy, as it is a very conceptual/human-friendly example that serves to educate and provide an easy explanation/implementation of BDI. It follows a very basic BDI syntax/implementation and most of the code is done in the java environment as it is assumed that the reader is much more familiar with Java than Jason. This example serves as an introductionary example.

The second example however, follow a much more sophisticated/complex BDI implementation that is done almost entirely in jason. This is to prove that the java environment actually serves only as the environment or a helper if necessary.

Jason is also not well-developed, there are limited libraries that I can use and even more limited documentation that hinders my progress/research here. The official documentation only lists the name of the functions without any explanation. Sources are primarily from the official website and a single book called ‘Programming Multi-Agent Systems in AgentSpeak Using Jason (Bordini et al., 2007). There are not much contributors or users of BDI, even more so those who code in Jason. There is only a single visualization library called the ‘GridWorld’ that all Jason projects use to visualize their code. There are not much examples of BDI so I have to make due of the current existing onces as a source of learning.

However, despite some shortbacks, BDI Agent framework and AgentSpeak is a very simple and straight-forward to learn. People could easily learn this concept and apply it at least conceptually to real life scenarios. It is perfect for demonstrating what at a minimum an AI can do.

Overall, there are much more room for development in Jason.

# Bibliography

Bordini, R. and Hübner, J. (no date) Cleaning Robots (Version 1.0) [Source code]

Bordini, R.H., Hübner, J.F. and Wooldridge, M., 2007. *Programming multi-agent systems in AgentSpeak using Jason*. John Wiley & Sons.

Franklin, S. and Graesser, A., 1996, August. Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents. In *International workshop on agent theories, architectures, and languages* (pp. 21-35). Springer, Berlin, Heidelberg.

Xu, M., 2020. *Extending BDI agents with robust program execution, adaptive plan library, and efficient intention progression* (Doctoral dissertation, University of Bristol). Sun, R., 2001. *Duality of the mind: A bottom-up approach toward cognition*. Psychology Press.

Rivera-Villicana, J., Zambetta, F., Harland, J. and Berry, M., 2018, October. Informing a BDI player model for an interactive narrative. In *Proceedings of the 2018 Annual Symposium on Computer-Human Interaction in Play* (pp. 417-428).

Russell, S. and Norvig, P., 2002. Artificial intelligence: a modern approach.

Luong, B., Thangarajah, J. and Zambetta, F., 2017. A BDI game master agent for computer role-playing games. *Computers in Entertainment (CIE)*, *15*(1), pp.1-16.

Yudkowsky, E., 2008. Artificial intelligence as a positive and negative factor in global risk. *Global catastrophic risks*, *1*(303), p.184.

Winikoff, M., Padgham, L., Harland, J. and Thangarajah, J., 2002. Declarative & procedural goals in intelligent agent systems. *KR*, *2002*, pp.470-481.