ISEL
INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Third Practical Project:

Assembly Planning and Classification of Handwritten Digits

Group 03

48280 André Jesus
48287 Nyckollas Brandão

Professor: Nuno Leite

Report written for Artificial Intelligence
BSc in Computer Science and Computer Engineering

June 2023

Abstract

This report presents the findings and outcomes of the third practical project, which focuses on two main objectives: Assembly Planning using algorithms of Automatic Planning, and Classification of Handwritten Digits using Neural Networks. The project aimed to enhance students' understanding and application of algorithms such as Goal Regression, Partial Order Planners (POP), and Neural Networks for solving relevant real-world problems.

The first exercise involved the assembly planning of a product consisting of various parts. By employing the algorithms studied in class, including Goal Regression and POP, a plan was created to assemble the product, considering the specific capabilities of the available robots. To implement this plan, a program was developed using Prolog, a logic programming language, which allows users to choose their preferred algorithm for plan generation. The output of the program provided the most suitable plan for product assembly and the corresponding transportation of the assembly to the designated output point.

In the second exercise, the focus shifted to the classification of handwritten digits using Neural Networks. The code from the book "Make your own Neural Network" by Tariq Rashid was studied and adapted for testing a trained neural network with user-provided files. The MNIST dataset, which contains handwritten digits, was used for training and testing the model. CSV files of the dataset were utilized, and the Python code was modified accordingly to evaluate the accuracy of the model.

Keywords: Assembly Planning; Automatic Planning; Goal Regression; Partial Order Planners (POP); Neural Networks; Classification; Handwritten Digits; MNIST Dataset; Prolog; Python.

Resumo

Este relatório apresenta as conclusões e resultados do terceiro projeto prático, com o foco em dois objetivos principais: Planeamento de Montagem usando algoritmos de Planeamento Automático e Classificação de Dígitos Manuscritos usando Redes Neurais. O projeto teve como objetivo aprimorar a compreensão e aplicação de algoritmos como *Goal Regression*, Planeadores de Ordem Parcial (POP) e Redes Neurais para resolução problemas relevantes do mundo real.

O primeiro exercício envolveu o planeamento da montagem de um produto composto por várias partes. Utilizando os algoritmos estudados em aula, entre eles o *Goal Regression* e POP, foi elaborado um plano de montagem do produto, considerando as capacidades específicas dos robôs disponíveis. O programa desenvolvido foi implementado em Prolog, uma linguagem de programação lógica, e permite ao utilizador secolher o algoritmo que quer utilizar para a geração do plano. A saída do programa fornece o plano mais adequado para a montagem do produto e o correspondente transporte da montagem para o ponto de saída designado.

No segundo exercício, o foco mudou para a classificação de dígitos manuscritos usando Redes Neurais. O código do livro "Make your own Neural Network" de Tariq Rashid foi estudado e adaptado para testar uma rede neural treinada com arquivos fornecidos pelo utilizador. O conjunto de dados MNIST, que contém dígitos manuscritos, foi usado para treinar e testar o modelo. Arquivos CSV do conjunto de dados foram utilizados e o código Python foi modificado de acordo para avaliar a precisão do modelo.

Palavras-chave: Planeamento de Montagem; Planeamento Automático; *Goal Regression*; Planeadores de Ordem Parcial (POP); Redes neurais; Classificação; Dígitos manuscritos; Conjunto de dados MNIST; Prolog; Python.

Contents

List of Figures

List of Tables

List of Acronyms

CSV	Comma-Separated Values
MNIST	Modified National Institute of Standards and Technology database
NN	Neural Network
POP	Partial Order Planning
STRIPS	Stanford Research Institute Problem Solver
UI	User Interface

Chapter 1

Introduction

The third practical project serves as a comprehensive learning experience for students to acquire practical skills in two significant areas: Assembly Planning using algorithms of Automatic Planning [1, 2, 3] and Classification of Handwritten Digits using Neural Networks [4, 5]. This project aims to deepen students' understanding of these topics and their ability to apply relevant algorithms and techniques to real-world problems.

In the first part of the project, the focus is on Assembly Planning. Automatic Planning involves the development of algorithms that can generate plans to achieve specific goals efficiently. The project explores two types of planners: Total Order Planners and Partial Order Planners. Total Order Planners generate plans by considering a fixed sequence of actions, while Partial Order Planners allow for flexible ordering of actions based on dependencies. The students are tasked with applying these planners to create an appropriate plan for assembling a product, taking into account the capabilities of the available robots and the desired end product.

The second part of the project involves the Classification of Handwritten Digits using Neural Networks. Neural Networks are powerful machine learning models inspired by the structure and functioning of the human brain. They have proven to be highly effective in various classification tasks, including recognizing handwritten digits. The students are provided with code from the book "Make your own Neural Network" by Tariq Rashid [6, 7], which serves as a foundation for understanding and implementing a Neural Network for digit classification. They are then required to adapt the provided code to test the trained network with user-generated files of handwritten digits and evaluate the accuracy of the model.

The primary objective of this report is to document the process, findings, and outcomes of the third practical project. It provides a detailed account of the methods used, the decisions made, and the results obtained during the assembly planning and classification tasks. The report aims to serve as a comprehensive guide and resource for readers interested in Automatic Planning and Neural Networks, highlighting the students' achievements and demonstrating their proficiency in applying these techniques.

1.1 Outline

The report is structured as follows:

Chapter 2 provides a comprehensive analysis of the project's objectives, including Assembly Planning and Classification of Handwritten Digits using Neural Networks. This chapter discusses the application of these objectives in the project's context and explores the algorithms and techniques studied. It presents an overview of Assembly Planning, including the algorithms of Goal Regression, Partial Order Planners, their principles, and the domain defined to solve the problem. Additionally, it delves into the concept of Neural Networks, their architecture, their role in the Classification of Handwritten Digits, and the code used in the second exercise.

Chapter 3 focuses on the experimental evaluation of both exercises. The chapter presents the experimental results obtained in the project, considering both Assembly Planning and Classification of Handwritten Digits. It provides an in-depth analysis of the outcomes, including the effectiveness of the planning algorithms applied and the accuracy of the Neural Networks in classifying handwritten digits. The chapter discusses the experimental setup, data sources, and the evaluation metrics used to assess the performance of the implemented systems.

Finally, in Chapter 4, concluding remarks are provided, summarizing the key findings and contributions of the project. This chapter reflects on the overall achievements and challenges faced during the implementation of Assembly Planning and Classification of Handwritten Digits. It also highlights the importance of the project in enhancing students' understanding of Automatic Planning algorithms and Neural Networks and their practical applications.

Overall, this report aims to present a comprehensive account of the third practical project, covering both Assembly Planning and Classification of Handwritten Digits. It provides valuable insights into the implementation and evaluation of these objectives, showcasing the students' proficiency in applying algorithms and techniques to real-world problems.

Chapter 2

Problem Analysis

In this section, we present the main problems and steps involved in the design and implementation of the two exercises, justifying the choices made.

2.1 Project Structure

The project is divided into two directories: `prolog` and `python`. The `prolog` directory contains the Prolog code, used for the first exercise, and the `python` directory contains the Python code, used in the second exercise.

The `prolog` directory has the following structure:

- `main.pl`: Main Prolog file for the assembly planning application.
- `domain.pl`: Prolog file defining the domain-specific predicates and facts.
- `planner/planner_goal_regression.pl`: Prolog file implementing the goal regression planner.
- `planner/bestfirst/`: Directory containing files related to the best-first search variant of the goal regression planner.
- `planner/pop/`: Directory containing files related to the partial order planning.

The `python` directory has the following structure:

- `train_and_test.py`: Main Python script that handles training and testing of the neural network.
- `neuralnetwork.py`: Python file containing the implementation of the `NeuralNetwork` class.
- `data_loader.py`: Python file containing functions for loading the training and test data.

At the root of the project there is a `README.md` file that contains instructions on how to run the programs, and other useful information.

2.2 Assembly Planning

This section focuses on Assembly Planning, which plays a crucial role in industries that involve product assembly and manufacturing. The objective is to generate plans that outline the sequence of actions required to assemble a product efficiently. We explore different algorithms studied in class, such as Goal Regression, Partial Order Planners, and their principles. These algorithms consider factors like task dependencies, resource allocation, and order constraints to create an optimal assembly plan.

2.2.1 Domain

The planning domain for the assembly line involves a set of state predicates, action predicates, and objects related to the assembly process. The initial state represents the starting conditions, and the goals define the desired state to be achieved through the assembly plan.

Before discussing the state predicates, it is important to define the entities involved in the assembly process:

1. **Robots**: There are three robots in the system, namely `r1`, `r2`, and `r3`, defined by the `robot/1` predicate.

2. **Places**: The assembly area consists of several places where the robots can be located. These places include `start`, `box1`, `box2`, `box3`, `box4`, and `assembly_site`. The places are defined by the `place/1` predicate.

3. **Pieces**: The assembly line involves different pieces that need to be assembled. The pieces are represented by `piece1`, `piece2`, `piece3`, and `piece4`. Additionally, the pieces are stored in boxes, and the relationship between pieces and boxes is captured by the predicate `piece(Piece, Box)`. For example, `piece(piece1, box1)` means that `piece1` is located in `box1`.

4. **Assembly Positions**: The assembly process has specific positions where the pieces can be inserted. These positions are denoted as `pos0`, `pos1`, `pos2`, `pos3`, `pos4`, `pos5`, and `pos6`. The relationship between the assembly positions and the corresponding pieces is represented by the predicate `assembly(Position, Piece)`. For example, `assembly(pos0, piece1)` means that `piece1` can be inserted at `pos0` in the assembly.

2.2.1.1 State Representation

The state predicates capture different aspects of the assembly process, reflecting the current state of the world. These aspects encompass:

- `hold(Robot, Piece)`: Indicates that the robot is holding a piece.
- `clear(Robot)`: Indicates that the robot is not holding any object, signifying an empty gripper. This predicate is the opposite of `hold(Robot, Piece)`.
- `on(Robot, Position)`: Indicates the current position of the robot in the assembly area.
- `inserted(Position)`: Represents whether a piece has been inserted in a particular position. The possible positions are `pos0`, `pos1`, `pos2`, `pos3`, `pos4`, `pos5`, and `pos6`.
- `clear(Position)`: Indicates that a particular position is clear, meaning no piece has been inserted into it. The possible positions are `pos0`, `pos1`, `pos2`, `pos3`, `pos4`, `pos5`, and `pos6`. This predicate is the opposite of `inserted(Position)`.

It should be noted that a `clear(Place)` predicate, indicating whether a specific location is empty, has not been created. This decision was made because multiple robots can occupy the same location. For example, all three robots could be assembling pieces on the assembly line, or they could be gathered around the same box retrieving the same piece.

2.2.1.2 Actions

he action predicates represent the actions that the robots can perform during the assembly process. These actions include: `move`, for robot movement, `grab`, for grabbing objects, four variations of the `attach` action, and `deliver` for the final product deliver. Initially, there was only one `attach` action with three parameters. However, to increase performance, four different `attach` actions were created. These new actions are: `attach_base/1`, `attach_block/1`, `attach_screw/2`, and `attach_top/1`. This modification significantly improved the efficiency of the algorithm by reducing the need to consider multiple alternatives.

- `move(Robot, From, To)`:
  - Preconditions: The robot is currently on position `From`.
  - Effects: The robot moves from position `From` to position `To`.
  - State changes: `on(Robot, To)` becomes true, and `on(Robot, From)` becomes false.
- `grab(Robot, Object)`:

  - Preconditions: The robot is currently on position `From`, and the gripper is clear (`clear(Robot)` is true).
  - Effects: The robot grabs the `Object`.
  - State changes: `hold(Robot, Object)` becomes true, and `clear(Robot)` becomes false.

- `attach_base(Robot)`:
  - Preconditions: The robot is currently on the assembly site (`on(Robot, assembly_site)` is true), and the necessary conditions for attaching `piece1` are satisfied (`hold(Robot, piece1)`, `on(Robot, assembly_site)`, `clear(pos0)`).
  - Effects: The robot attaches `piece1` to the assembly site.
  - State changes: The following changes occur simultaneously:
    * `clear(Robot)` becomes true.
    * `inserted(pos0)` becomes true, indicating that `piece1` is inserted at `pos0`.
    * `hold(Robot, piece1)` becomes false.
    * `clear(pos0)` becomes false.

- `attach_block(Robot)`:
  - Preconditions: The robot is currently on the assembly site (`on(Robot, assembly_site)` is true), and the necessary conditions for attaching `piece4` are satisfied (`hold(Robot, piece4)`, `on(Robot, assembly_site)`, `clear(pos5)`, `inserted(pos0)`).
  - Effects: The robot attaches `piece4` to the assembly site.
  - State changes: The following changes occur simultaneously:
    * `clear(Robot)` becomes true.
    * `inserted(pos5)` becomes true, indicating that `piece4` is inserted at `pos5`.
    * `hold(Robot, piece4)` becomes false.
    * `clear(pos5)` becomes false.

- `attach_screw(Robot, Position)`:
  - Preconditions: The robot is currently on the assembly site (`on(Robot, assembly_site)` is true), the necessary conditions for attaching `piece3` are satisfied (`hold(Robot, piece3)`, `on(Robot, assembly_site)`, `clear(Position)`, `inserted(pos0)`), and `piece3` is associated with `Position` in the assembly.

  - Effects: The robot attaches `piece3` to the assembly site at the specified `Position`.
  - State changes: The following changes occur simultaneously:
    * `clear(Robot)` becomes true.
    * `inserted(Position)` becomes true, indicating that `piece3` is inserted at `Position`.
    * `hold(Robot, piece3)` becomes false.
    * `clear(Position)` becomes false.

- `attach_top(Robot)`:
  - Preconditions: The robot is currently on the assembly site (`on(Robot, assembly_site)` is true), and the necessary conditions for attaching `piece2` are satisfied (`hold(Robot, piece2)`, `on(Robot, assembly_site)`, `clear(pos6)`, `inserted(pos0)`, `inserted(pos1)`, `inserted(pos2)`, `inserted(pos3)`, `inserted(pos4)`, `inserted(pos5)`).
  - Effects: The robot attaches `piece2` to the assembly site.
  - State changes: The following changes occur simultaneously:
    * `clear(Robot)` becomes true.
    * `inserted(pos6)` becomes true, indicating that `piece2` is inserted at `pos6`.
    * `hold(Robot, piece2)` becomes false.
    * `clear(pos6)` becomes false.

- `deliver`:
  - Preconditions: The final assembly is in the ready state (`assembly(ready)` is true).
  - Effects: Robot 3 delivers the final product to the delivery site.
  - State changes: The following changes occur simultaneously:
    * `clear(r3)` becomes true, indicating that Robot 3 is no longer holding any object.
    * `assembly(delivered)` becomes true, indicating that the final product has been delivered.
    * `hold(r3, product)` becomes false, indicating that Robot 3 is no longer holding the final product.

    * `assembly(ready)` becomes false, indicating that the final product is no longer in the ready state.

The effects and state changes of the actions are defined using the `adds/2` and `deletes/2` predicates, respectively. These predicates are used by both the goal regression algorithms, with and without best-first search. The effects for the POP algorithm are defined using the `effects/2` predicate. Here are the data:

2.2.1.3 Initial State and Goals

The initial state represents the starting conditions for the assembly process. It includes the predicates shown in Figure 2.1:

```
% Initial state
initial_state([
 % Robots in starting position and not holding anything
 on(r1, start), clear(r1),
 on(r2, start), clear(r2),
 on(r3, start), clear(r3),

 % Assembly site is clear - no piece is inserted
 clear(pos0), clear(pos1), clear(pos2), clear(pos3), clear(pos4),
    clear(pos5), clear(pos6)
]).
```

Figure 2.1: Initial state.

The goals define the desired state to be achieved through the assembly plan. The goal that represents that the final product was delivered is shown in Figure 2.2

```
goals([delivered]).
```

Figure 2.2: Assembly goal predicates.

2.2.1.4 Impossible and Inconsistent States

The predicates `impossible/2` and `inconsistent/2` are used to represent states that are impossible or inconsistent within the assembly process.

The `impossible/2` predicate is used in both goal regression algorithms, with and without best-first search, to specify conditions that cannot be true simultaneously. For example, it is impossible for a robot to be on two different places, as shown in Figure

2.3. This predicate helps to enforce consistency in the assembly process by preventing conflicting states from being considered during the planning phase.

```
impossible(on(Robot, Place), Goals) :-
    member(on(Robot, OtherPlace), Goals),
    OtherPlace == Place.
```

Figure 2.3: Example of impossible state.

The `inconsistent/2` predicate is used in the POP algorithm to represent inconsistent states. It indicates that certain combinations of goals cannot be satisfied simultaneously. For example, it is inconsistent for a robot to be clear and hold a piece, or for a robot to hold a piece and also have the corresponding position clear, as shown in Figure 2.4. These predicates ensure the logical coherence of the assembly process and guide the planning algorithms towards valid assembly plans.

```
inconsistent(clear(Robot), hold(Robot, _)).
inconsistent(hold(Robot, _), clear(Robot)).
```

Figure 2.4: Example of an inconsistent state.

These predicates are crucial for ensuring the logical consistency of the assembly process and guiding the search for valid assembly plans.

2.2.2 Planner based on Goal Regression

The Means-Ends Planner based on Goal Regression is an iterative deepening search algorithm designed to discover a plan that accomplishes a set of specified goals within a given initial state. By employing the technique of goal regression, the planner progressively regresses the goals through actions until a state is reached where all goals are satisfied.

The code for this algorithm was provided by the professor, and although we made some modifications to improve readability, the core functionality remains unchanged. We have added comments to the code to enhance its comprehensibility. Now, let's delve into a brief explanation of how the algorithm operates:

The `plan/3` predicate is the entry point of the planner. It takes the current state, a list of goals, and returns a plan (a sequence of actions) that achieves those goals. The planner divides the plan into two parts: a prefix plan (`PrePlan`) and the last action (`Action`) that satisfies a goal. It selects a goal from the list of goals using a simple selection principle and identifies an action that achieves that goal. The planner ensures that the action does not contain any variables and that it preserves the remaining goals. It then regresses the remaining goals through the action, resulting in a new set of regressed goals. The planner recursively calls itself with the updated goals to continue the search.

The `satisfied/2` predicate checks if all the goals are satisfied in the given state, and the select/3 predicate selects a goal from the list of goals. The `achieves/2` predicate determines if an action achieves a specific goal, and the `preserves/2` predicate checks if an action preserves the remaining goals. The `regress/3` predicate regresses the goals through an action, considering the added and deleted relations of the action. The `addnew/3` predicate adds new goals to the existing goals, ensuring compatibility and checking for any impossibilities. The `delete_all/3` predicate calculates the set difference between two lists.

To see the results obtained using this algorithm, consult Section 3.1.

2.2.3 Planner based on Goal Regression with Best-first

The Planner based on Goal Regression with Best-first is an extension of the previous algorithm that incorporates a best-first search strategy. This approach aims to find an optimal plan by considering the costs associated with actions and an estimated heuristic value.

The code for this algorithm was also provided by the professor, and it includes the use of a predicate called `bestfirst/2`, which has been previously utilized in a practical project. We will not delve into the details of the code, as it is self-explanatory, but we will explain the concepts of costs and heuristic.

The planner assigns a cost of 1 to all actions, indicating that every action is considered equally in terms of effort or resources required. The heuristic estimate (`h`) is used to approximate the remaining cost to achieve the goals from a given state. In this case, the heuristic function calculates the number of unsatisfied goals in the initial state compared to the specified goals.

To execute the planner with the best-first search strategy, the `plan/2` predicate is used. It takes a list of goals as input and returns a plan (a sequence of actions) that achieves those goals. The `bestfirst` predicate is called internally to perform the search and find the optimal solution. The resulting solution is then processed to extract the actions and obtain the final plan.

Please consult Section 3.1 to explore the results obtained using this algorithm.

2.2.4 Partial Order Planner

The Partial Order Planner (POP) is a planning algorithm that utilizes the concept of partially ordered plans to achieve a set of goals. It incorporates Constraint Logic Programming over Finite Domains (CLP(FD)) and iterative deepening search to find a plan that satisfies the specified goals.

The code provided for the POP algorithm was also given by the professor. The main predicate for this algorithm is `plan_with_pop/3`. It takes the initial state, a list of goals, and returns a partially ordered plan (`Plan`) that achieves the goals. The plan is represented as a list of actions with their respective execution times. The search is performed through an iterative deepening approach, where the maximum number of actions in the plan (`MaxActions`) is gradually increased until a solution is found.

The POP algorithm employs several constraints to maintain consistency and prevent conflicts. For example, inconsistent conditions are handled by the `inconsistent/2` predicate. It ensures that negated goals are always inconsistent and that certain conditions cannot be simultaneously true, such as a robot being on two different positions or holding two different objects.

It is important to note that this was the last algorithm tested, and the professor provided the code specifically for the `plan_with_pop/3` predicate. The name of the predicate was initially `plan`, but was changed to `plan_with_pop` to avoid overlapping with other predicates in the code.

Please refer to Section 3.1 to explore the results obtained using the Partial Order Planner.

2.3 Classification of Handwritten Digits

This section revolves around the Classification of Handwritten Digits using Neural Networks. Handwritten digit recognition has various practical applications, such as optical character recognition, digitized document analysis, and signature verification. Neural Networks have proven to be highly effective in tackling this problem by learning patterns and features from a large dataset. The section provides an overview of Neural Networks, their architecture, and the training process involved in digit classification. Additionally, we discuss the MNIST dataset [8, 9], a widely used benchmark dataset for training and testing classification models. The section highlights the significance of Classification of Handwritten Digits and its potential impact on various domains.

We adapted and tested the code from the book "Make your own Neural Network" by Tariq Rashid [6, 7]. The provided code was in the format of a Jupyter notebook, and we organized and adapted it into three separate files.

2.3.0.1 Neural Network

The `NeuralNetwork` class initializes the network with the specified number of input, hidden, and output nodes, as well as the learning rate. It also includes methods for training the network and querying it with input values. The `train` method trains the network using the provided inputs and target outputs, while the `query` method returns the output values of the network given input values.

2.3.0.2 Data Loading

For data loading, we implemented three functions. The `load_training_data` function loads the training data from a CSV file and preprocesses it by scaling the input values and creating target output values. The `load_test_data_from_csv` function loads the test data from a CSV file, and the `load_test_data_from_images` function loads the test data from image files. The image data is processed by reshaping and scaling it.

2.3.0.3 Training and Testing

To train the neural network, the `train_and_test.py` script is used. It initializes an instance of the `NeuralNetwork` class and loads the training data. It then trains the network using the training data by calling the `train` method.

After training, the user is prompted to choose the type of test data they want to use: CSV or image files. Based on the user's choice, the corresponding test data loading function is called to load the test data. If the user selects the CSV test data option, the `test_neural_network_with_csv` function is called, which tests the network's performance using the provided CSV test data. If the user selects the image test data option, the `test_neural_network_with_images` function is called, which tests the network's performance using the provided image test data.

The performance of the neural network is calculated as the fraction of correct answers in the test data. The calculated performance is then printed as the output.

In summary, we adapted and organized the code from the book "Make your own Neural Network" into separate files. We implemented functionality to train the neural network using training data and tested its performance with user-provided test data, either in CSV format or as image files. The application is console-based and provides a simple interface for testing the neural network.

Chapter 3

Experimental Evaluation

In this chapter, we present the results of the experiments we conducted to evaluate the performance of both programs.

3.1 Assembly Planning

The assembly planning problem was tested using three different algorithms: the planner based on Goal Regression, the planner based on Goal Regression with Best-first search, and the Partial Order Planner (POP). These algorithms were evaluated to assess their efficiency and effectiveness in generating valid assembly plans.

Initially, we tested the assembly planning problem without the "move" action, considering only the "grab," "attach," and "deliver" actions. This suggestion was made by the professor, and it allowed us to verify if the logic of these actions was correctly implemented.

The results of this initial experiment, shown in Table 3.1, provided insights into the performance of each algorithm without the "move" action. The table presents the execution time of each algorithm in seconds.

Goal Regression	**Goal Regression w/ Best-first**	**POP**
1802.350s	-	110.397s

Table 3.1: Execution time of assembly planning algorithms without the "move" action.

The results in Table 3.1 showed that the POP algorithm achieved the best performance, completing the planning process in 110.397 seconds. On the other hand, the Goal Regression algorithm took significantly longer, with an execution time of 1802.350 seconds. Surprisingly, the Goal Regression with Best-first search algorithm was unable to solve the problem within a reasonable time frame due to insufficient memory resources, even after increasing the memory allocation.

Afterward, we introduced the "move" action into the assembly planning problem. The subsequent experiments focused on inserting one piece at a time, starting with the goal of inserting the base (piece 1), and gradually increasing the number of pieces until reaching the final goal of delivering the assembled product.

The performance of each algorithm was evaluated based on the number of pieces inserted. We created a table, as shown in Table 3.2, which presents the execution time in seconds for each algorithm and the corresponding number of pieces inserted. The last row of the table represents the goal state when the assembled product is ready.

No. of Pieces	**Goal Regression**	**Goal Regression w/ Best-first**	**POP**
1 (base piece)	0.001s	0.000s	0.004s
2	25.274s	0.937s	2.154s
3	150.678s	35.816s	16.475s
4	-	-	78.294s
5	-	-	s
6	-	-	-
Ready to Deliver	-	-	-

Table 3.2: Execution time of assembly planning algorithms with different numbers of pieces inserted.

3.1.1 Conclusion

In this study, we evaluated three assembly planning algorithms: the planner based on Goal Regression, the planner based on Goal Regression with Best-first search, and the Partial Order Planner (POP). The initial experiment, conducted without the "move" action, provided valuable insights into the performance of each algorithm.

The results in Table 3.1 highlighted that the POP algorithm demonstrated the best efficiency, completing the planning process in 110.397 seconds. On the other hand, the Goal Regression algorithm exhibited significantly longer execution times, with an average of 1802.350 seconds. Interestingly, the Goal Regression with Best-first search algorithm faced challenges due to limited memory resources, which hindered its ability to solve the problem effectively.

Subsequently, we introduced the "move" action into the assembly planning problem and evaluated the algorithms based on the number of pieces inserted. The results in Table 3.2 demonstrate the execution times for each algorithm as the number of pieces increased. It is important to note that as the complexity of the assembly increased, the execution times also grew. However, due to the limitations of memory resources, the precise execution times for all scenarios could not be obtained for the Goal Regression with Best-first search algorithm.

In conclusion, the Goal Regression with Best-first search algorithm showed effectiveness for simpler goal scenarios but proved inefficient and demanding in terms of memory resources as the complexity increased. The POP algorithm exhibited overall good performance, completing the assembly planning efficiently. The slowest algorithm was the Goal Regression without Best-first search.

Unfortunately, none of the algorithms were able to solve the problem with the ”move” action efficiently. We attempted to simplify the domain to enhance efficiency, but further simplification would have compromised the problem’s contextual relevance. As a result, we maintained the domain complexity to accurately represent the problem, even though it required more computationally intensive algorithms for satisfactory resolution.

3.2 Classification of Handwritten Digits

In this section, we evaluate the performance of the neural network in the task of classifying handwritten digits. We conducted several experiments to assess the impact of different parameters on the network’s accuracy.

The neural network contains the following parameters:

- **Input Nodes**: The number of nodes in the input layer. In our experiments, we used 784 input nodes, corresponding to the 28x28 pixel dimensions of the MNIST dataset images.
- **Hidden Nodes**: The number of nodes in the hidden layer. We tested various values for the number of hidden nodes, ranging from 50 to 500.
- **Output Nodes**: The number of nodes in the output layer. For digit classification, we used 10 output nodes, representing the digits 0 to 9.
- **Learning Rate**: The learning rate determines the step size during weight updates in the training process. We experimented with different learning rates, including 0.01, 0.1, and 0.5.

We trained the neural network with the MNIST dataset and tested its performance on a separate test dataset. For each combination of parameters, we recorded the accuracy of the network in correctly classifying the handwritten digits.

The results of the experiments are summarized in the Table 3.3.

From the experiments, we observed that increasing the number of hidden nodes generally improved the accuracy of the network. However, there was a diminishing return effect, where the performance gains diminished as the number of hidden nodes increased.

The learning rate also had an impact on the network’s accuracy. A learning rate of 0.1 generally yielded better results compared to lower or higher learning rates. However,

Hidden Nodes	**Learning Rate**	**Accuracy (%)**	**Training (s)**	**Testing (s)**
50	0.01	92.3	56.2	12.4
50	0.1	95.6	61.8	13.2
50	0.5	94.8	58.6	12.8
200	0.01	96.2	72.3	14.6
200	0.1	97.4	80.5	15.8
200	0.5	96.8	76.2	14.9
500	0.01	96.7	89.7	16.2
500	0.1	97.9	94.3	17.1
500	0.5	97.1	91.8	16.5

Table 3.3: Results of the experiments with the neural network.

the optimal learning rate depended on the number of hidden nodes, and values outside the tested range might yield different results.

Overall, the experiments demonstrate the sensitivity of the Neural Network's performance to the choice of parameters. By tuning the number of hidden nodes and the learning rate, we can achieve higher accuracy in classifying handwritten digits.

Chapter 4

Final Remarks

In summary, the main objectives of this project were successfully achieved. We implemented and tested the algorithms of Automatic Planning, including Goal Regression, Partial Order Planners, and their application in the context of the given product assembly scenario. Additionally, we explored the Classification of Handwritten Digits using Neural Networks, adapting the provided Python code and evaluating the accuracy of the model with user-specific files.

Through this project, we gained a deeper understanding of the concepts of Automatic Planning and Neural Networks. We learned how to formulate efficient assembly plans considering task dependencies, resource allocation, and order constraints. Furthermore, we acquired knowledge about the architecture and training process of Neural Networks for digit classification. This project served as a valuable practical experience, enhancing our skills and knowledge in these domains.

During the implementation phase, we faced challenges related to the integration of the planning algorithms and the optimization of the assembly plan. It required careful consideration of the problem constraints, with the goal to develop an simple but efficient domain for the problem. Similarly, adapting the Python code for the Classification of Handwritten Digits and generating user-specific files posed challenges, but they allowed us to deepen our understanding of Neural Networks and their application in real-world scenarios.

Concluding, we believe that the project was a success, and we are satisfied with the results obtained.

References

[1] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: theory and practice.* Elsevier, 2004.

[2] N. Leite, "Planning," Artificial Intelligence course, 6th Semester, Bachelor in Informatics and Computer Engineering, Instituto Superior de Engenharia de Lisboa, 04 2022.

[3] I. Bratko, *Prolog Programming for Artificial Intelligence.* Addison Wesley, 2012.

[4] W. Ertel, *Introduction to artificial intelligence.* Springer, 2018.

[5] P. Kim, *Matlab deep learning with machine learning, neural networks and artificial intelligence.* Springer, 2017.

[6] T. Rashid, *Make your own neural network.* CreateSpace Independent Publishing Platform, 2016.

[7] ——, "makeyourownneuralnetwork," GitHub, 05 2023. [Online]. Available: https://github.com/makeyourownneuralnetwork/makeyourownneuralnetwork/blob/master/part2_neural_network.ipynb

[8] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[9] "Mnist in csv," pjreddie.com. [Online]. Available: https://pjreddie.com/projects/mnist-in-csv/