Final Year Project Report

Full Unit - Interim Report

Measuring performance between AI Planning and CSP Solving techniques

Rohit Kakkar

A report submitted in part fulfilment of the degree of

BSc (Hons) in Computer Science Computer Science with Integrated Foundation Year

Supervisor: Santiago Franco Aixela

Department of Computer Science
Royal Holloway, University of London

April 12, 2024

Declaration

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

Word Count: 12046

Student Name: Rohit Kakkar

Date of Submission: 12/04/2024

Signature: 

Table of Contents

Abstract 4

1 Introduction 4

1.1 Project overview 4

1.2 Aims & objectives 5

2 Background Theory 7

2.1 Planning 7

2.2 Sudoku 8

2.3 Constraint satisfaction problems 9

2.4 Graph theory 10

2.5 Search Heuristics 11

3 Software engineering 12

3.1 Unified Modeling Language (UML) 12

3.2 Methodology 13

3.3 Testing 13

3.4 Version control system 14

4 Development 15

4.1 Planning: proof of concept 15

4.2 CSP: proof of concept 17

4.3 Classic Sudoku PDDL model 19

4.4 Final CSP implementation 21

4.5 PDDL: pathfinding 21

4.6 CSP: pathfinding 27

4.7 PDDL: parsing 30

4.8 Planning: Bespoke pathfinding planner 31

4.9 PDDL file generation 34
4.10 Maze generation and implementation 36
4.11 Maze implementation in CSP 38
4.12 New approach: Chunking and pattern database 39
4.13 Chunking and Landmark Heuristic 49
5 Review 51
5.1 Professional Issues 51
5.2 Timeline comparision 52
5.3 Key findings 53
5.4 Future steps 54
Bibliography 57
A Diary 58
A.1 Literature Review 60

Abstract

Planning is one of the core fundamentals of artificial intelligence and is essential for applying autonomous techniques to transition from an initial state to a desired goal state through a sequence of actions which are also known as planning problems [1]. Constraint Satisfaction Problems (CSPs) differ from planning in that they are more focused on finding a solution which meets given constraints without considering a sequence of actions or transitions between states. Unlike planning problems where the goal is to find a sequence of actions from initial to goal states, CSPs are more about solving for static configurations that meet all given constraints simultaneously. The performance of each method of finding solutions will be measured through a range of problems such as solving puzzles where we will consider measuring performance in Sudoku and pathfinding puzzles where we test performance in mazes. Off the shelf solvers and planners will be implemented as well as bespoke solutions to compare different propagation and heuristic techniques and their effectiveness in finding solutions.

Chapter 1: **Introduction**

Within this project, We will be assessing the performance of planning techniques compared to CSP solving techniques over two different problems, one being solving Sudoku and the second being a pathfinding problem using generated mazes to assess time taken and memory usage between each solving technique.

1.1 Project overview

Sudoku is an interesting problem due to its combinatorial complexity, "There are 6,670,903,752,021,072,936,960 possible solvable Sudoku grids that yield a unique result" [2]. This makes it quite a challenge for both AI and humans to solve as there are constantly new problems which test the boundaries of available AI solving techniques. I will be using the Planning Domain Definition Language (PDDL) to model the puzzle due to its effectiveness in modelling complex problems such as sudoku. I will be able to define both the initial state and the goal state using the provided methods along with predicates to help me define the constraints and structure of the puzzle. Furthermore to model Sudoku as a CSP, I will be using Satisfiability Modulo Theories (SMT) as a wide range of constraints and theories can be used in order to represent the constraints naturally as well its support with an off the shelf solver like z3. A 3x3 implementation will first be developed as a proof of concept, which will then be scaled up to a 9x9 version as the complexity drastically increases.

The Fast Downward planner will be chosen as this is a well regarded planner and has the option to use multiple different types of search such as A* which is known for its efficiency within pathfinding problems. This type of search is useful for sudoku as it can balance the shortest path and lowest cost solutions to find an efficient solution. On the other hand, LAMA with the landmark-based heuristic approach is better suited for the classic 9x9 sudoku problem as this type of search specialises in identifying the crucial steps in the puzzle-solving process. This versatility and the well-recognised as a great performing planner makes it an ideal choice to find a solution to sudoku.

Pathfinding in mazes, provides another great benchmark due to its combinatorial complexity [3] and the wide variety of algorithms that can be used to find solutions. The challenge would be to navigate through a maze, while optimising the shortest path, least cost or

avoiding obstacles. In order to develop the model for a classic maze problem, I will first be utilising PDDL to create the domain and problem files, and using off the shelf planners to find solutions. I will then scale it up by creating a bespoke planner in python and parsing the problem file to develop many algorithms such as A* to find solutions.

Time taken and memory usage will be measured to assess the performance of each algorithm as metrics in understanding efficiency and scalability. Time taken, will measure how long each algorithm takes from start to finish indicating speed and applicability to time-based real world problems. Memory usage, on the other hand, quantifies how much working memory is required by the algorithm during execution, indicating efficiency and feasibility on systems with constraints on memory capacity.

1.2 Aims & objectives

The primary aim of this project is to assess the effectiveness of the AI solving techniques, Planning and CSP solving over two different problems, one being Sudoku and the other being maze pathfinding providing insight on their effectiveness in terms of finding a solution, speed as well as efficiency.

Familiarising myself with PDDL was the first objective, which would be completed by working on a small 3x3 version of sudoku and having the online solver find a plan. This would help me learn the fundamentals of PDDL which included getting comfortable with the syntax as well as defining potential actions and goals. and find any mistakes early as a lot less code is required for the smaller implementation. Furthermore, I will be able to use this as a proof of concept before taking on the full puzzle as it is more manageable which should ensure an easier introduction to PDDL.

Utilising the 3x3 implementation, I will be increasing the complexity and work on modelling the classic 9x9 puzzle. This will be done by introducing new predicates to define the cell structure, and to identify which integers have been used for each constraint. This will help to more accurately model the 9x9 problem, while maintaining the new rules that come with the classic problem.

A popular SAT solver, z3 will be set up and utilised, which is known for its effectiveness with constraint satisfaction problems in order to benchmark the time taken to reach a solution for both a 3x3 implementation and the classic 9x9 puzzle which will be compared to the planning approaches.

To make up for the increase in complexity, the fast downward planner will be set up on my machine for the fourth objective as the standard online solver is limited in terms of options and times out in 10 seconds. This ensures the best chance for the implementation to work as it will run natively.

Metrics will be recorded from each solution and compared to analyse the effectiveness of each solving technique. The inbuilt time functions of each off the shelf solver and planner can be used to measure speed of each algorithm and the inbuilt memory function in Linux can be used to measure memory usage.

A simple pathfinding problem where a grid structure is used will be modelled in PDDL and slowly the complexity will increase by adding obstacles and increasing the grid size.

A bespoke planner will be developed in Python which will be able to parse the problem file, and apply an algorithm to find a solution along with a generator to generate problem files to

provide variety and test as many different conditions on each algorithm.

In Python, a CSP version of the same pathfinding problem will be implemented using a backtrack algorithm, leaving the option for different inference techniques and propagation.

Each implementation will be measured for both greedy and optimal solutions while measuring time taken and memory usage to provide insight into the strengths and weaknesses of each implementation.

With these objectives I aim to improve understanding on current AI solving techniques and analyse the benefits and drawbacks of each solution for their application in real-world solutions such as bus routes, and puzzle solving.

Chapter 2: **Background Theory**

2.1 Planning

A subsection of artificial intelligence, planning "seeks to build control algorithms that enable an agent to synthesize a course of action that will achieve its goals" [4]. through a set of given actions, the agent is tasked with traversing from the described initial state to a desired goal state. For example in a warehouse where robots are used to transport items, an AI planner would be tasked with finding the most efficient paths from pick up to drop off points, here the initial state would include the starting locations of the robots along with the pick up points for items along with item types. The goal would be successful pick up and drop off of items.

A planning problem can be split up into 3 parts, the first being the initial state, this is a description of the starting point of the problem. The second part being the goal state which is described as a desired state which can be partial depending on the requirements. Finally actions are defined to be used in transitioning between the two states. The resulting sequence is known as the plan or the solution.

The importance of planning can be seen in a range of environments, one such application, in manufacturing where staying competitive is a priority, AI planners can respond quickly to shifts in market demand or disruptions. This makes for increased efficiency and performance which can greatly benefit businesses where small changes can mean big implications if not dealt with adequately [5].

One form of search within planning is called heuristic search. Heuristics are informal judgemental rules or strategies which help prioritise which paths should be explored due to their likeliness of reaching a goal and which paths should be avoided due to their likeliness of not finding a plan [6]. Heuristic search uses these 'heuristics' to guide its search process through the state space. In AI planning, heuristics are significant as the aim is efficiency, especially in large search spaces and is done by constantly estimating the cost of the path at each given state.

Heuristics within AI planning can be placed into two categories, the first being domain-independent. Domain-independent heuristics do not rely on any specific information from the domain which makes them flexible and allow them to be applied to a diverse range of applications, the advantage is that there is no need for customisation and therefore can be used in dynamic environments. This is valuable in situations where the environments are always changing or extra information about the domain is not known. On the other hand we have domain-specific heuristics which are customised to specific problems which could be more complex and not as efficient to solve using domain-independent heuristics. The advantage is that the guiding process for the search is more efficient as more specific rules are introduced to aid the planner, which can mean more efficient solutions than domain-independent heuristics and the cost of flexibility and adaptability over different problems or dynamic environments.

Problem-solving can be visualised as a search within a state-space. This space is made up of states with each state being a representation of a problem to be solved. Operators are used as actions that can help with state traversal, creating a path towards the solution [7]. One state-space search algorithm is A* which is a common choice for pathfinding as it is known for its solution validity as well as having good performance [8]. State-space algorithms have their specific use cases which is where the entire state of the problem needs to be re-considered at each state, the advantage of these algorithms is their simplicity but as a result may be

inefficient in complex problems.

Planning Domain Definition Language (PDDL) was created with the aim of expressing planning domains and problems, while enabling definition of predicates, possible actions and their effects on the predicates [9]. PDDL was also developed as a way to standardise how planning problems should be modelled which allows for comparison of multiple different domains as they will have been modelled the same way, also increasing the collaboration within the planning community as this allowed for just one syntax to understand instead of several. Modularity was also a consideration as the design allows for scaling up existing work. PDDL was designed to work with multiple different planners and has a feature where the specific requirements of the domain can be expressed which can help with specific needs that a general planner may not be as effective with. Actions are defined with parameters, preconditions and the resulting effects which allows for the environment to be accurately represented at each state.

2.2 Sudoku

The word Sudoku originates from a shortened Japanese phrase 'suuji wa dokushin ni kagiru's which means 'the numbers must remain single'. This encapsulates the main rule of sudoku. The simple looking puzzle starts off as 9x9 structure of cells which are split up into 9 even 3x3 blocks. The problem is set up by placing integers from 1 to 9 in various cells while maintaining the rule that no integer can appear more than once in each row, column and block [10]. The goal is for the solver to fill in the rest of the cells using the pre-filled cells as clues to which integers are and are not valid in each cell. There will only be one solution to each sudoku puzzle as long as the puzzle has been set up in a valid way.

Sudoku can seem straightforward at the first glance, with simple rules and an easy to grasp concept, however, with a deeper dive, it becomes very interesting mathematically. Looking past just filling out numbers, sudoku is a great example of a combinatorial problem which is one that has a lot of possible combinations which makes them computationally a challenge to solve. As there are so many different problems and solutions along with its simple to grasp concepts, it is easy to see how sudoku became so popular, rivalling the crossword in newspapers.

6.671×10^{21} or 6,670,903,752,021,072,936,960 which is six sextillion, six hundred seventy quintillion, nine hundred three quadrillion, seven hundred fifty-two trillion, twenty-one billion, seventy-two million, nine hundred thirty-six thousand, nine hundred sixty. This is the number of possible solvable sudoku grids [10]. Making for a huge search space which presents significant computation challenges which raises it as a useful benchmarking tool for algorithms that are designed to efficiently navigate large state spaces.

Sudoku presents a unique problem-solving experience, testing strategy, logic and pattern recognition skills. The complexity is ideal for algorithms to be tested on before they are applied to similar real-world problems where strategy, logic, pattern recognition and a complex search space exist.

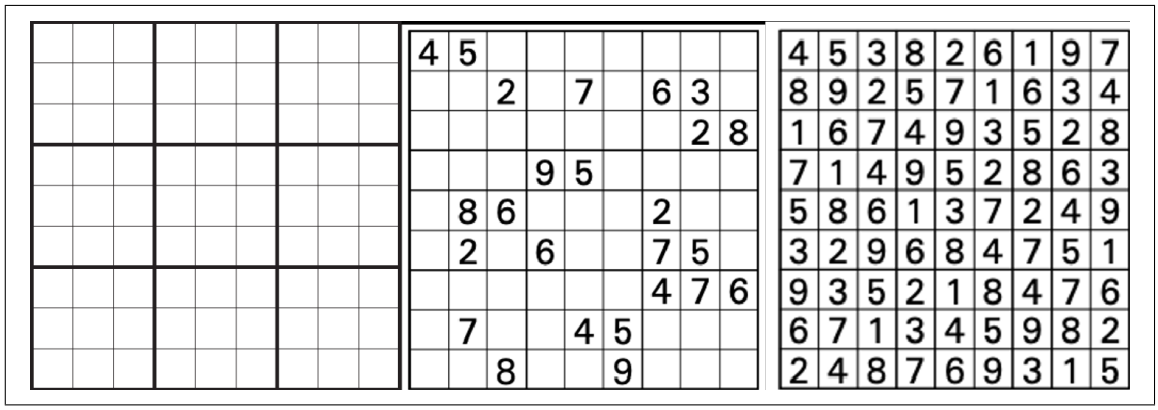


Figure 2.1: Sudoku.

2.3 Constraint satisfaction problems

Constraint satisfaction problems (CSPs) are defined by values from a finite given domain which are to be applied to variables while maintaining that all constraints or rules are satisfied [11]. CSPs are made up of 3 concepts, first being the variables which can be visualised as the unknowns which need to be solved, the second being domains which contain all the possible values that can be assigned to variables, and finally the constraints are rules that limit which values can be assigned to variables.

One example of a constraint satisfaction problem is sudoku. In this problem the variables can be seen as the cells that are required to be filled, the domain can be visualised as the possible integers that can be placed which are 1 to 9. The constraints in this problem are the rules where no integer can appear more than once in each row, column and block.

There are multiple techniques used to solve CSPs. One such technique is backtracking, this works by assigning a value to one variable at a time, once values have been assigned to all variables, we check if the constraints have been satisfied, if the answer is no, we backtrack to the last variable that was assigned a value and try a different value instead, this is repeated until all the constraints are satisfied [12].

A real-world application of CSPs can be seen with scheduling problems, for example, creating class schedules where each class needs to be assigned a time and location while avoiding any overlap while considering other factors such as availability and class capacity, this showcases the abilities of CSPs in a practical setting and also shows the diversity of problems that CSPs can be applied to.

2.4 Graph theory

Graphs are a "mathematical abstraction" [13], which are made up of a set of vertices which can be visualised as points on a map and a set of edges, an edge is a connection between two vertices. One example of this could be web pages on the internet as the vertices or nodes and hyperlinks between them being described as the edges between them. This definition of graphs is ambiguous as it can apply to many different environments, such as the one described prior or train stations with connecting tracks. A path is when the nodes can be arranged in a specific order. In this order, two nodes are connected to each other only if they come one after the other in the sequence, which means that each node in the path is connected only to the next node within the path.

An undirected graph is one where traversal is not limited to one direction, which means you can travel both ways between nodes, on the other hand a directed graph is one where each edge has a specific one way direction, an arrow is used to indicate the direction of travel for the specific edge and movement is limited to only this direction.

A unweighted graph is one where all the edges are equivalent in cost, this is indicated by a plain line, for example, a network of friends where an edge would mean two people or nodes are friends but require no weight or cost. In contrast to this A weighted graph is one where the relationship between nodes has a value or cost attached to it which will be indicated using a number near the targeted line. For example a weighted graph could include the time taken to reach a destination.

Vital to pathfinding algorithms, graph theory plays a big role in the navigation through networks and maps. One such algorithm is Dijkstra, some key points about Dijkstra is that it is used with weighted graphs and aims to find the shortest path. Nodes are inserted into a priority queue based on their distance from the currently observed node. It is important to note that Dijkstra fails if there are negative weights as the assumption that once the shortest path to a node has been discovered, it should not be able to get any shorter which is a problem in graphs with negative weights, a path with shorter cost can be introduced which would make for a shorter path which goes against the assumption, the algorithm does not go back to check previously finalised nodes which means that potentially shorter paths are missed due to the negative weights. Finally the runtime of Dijkstra depends on how the priority queue is implemented [13]. The applications of Dijkstra can be seen in a range of environments, one such environment where Dijkstra has had a significant impact is planning bus routes in Mumbai, improving the economy by reducing wait and travel times [14].

Another algorithm which is popular for use of pathfinding in game AI, A* works by constantly examining the most promising node that hasn't been explored fully yet, at the node, the algorithm will stop if a goal is reached or keep in mind the neighboring nodes to expand on later, this allows for a continuous flow towards the goal [15]. Some key points include, A* will always find a path from the starting node to the goal node as long as it is feasible, A* also utilises heuristics and is efficient for pathfinding in larger, more complex graphs.

2.5 Search Heuristics

Heuristics are, "methods for arriving at satisfactory solutions with modest amounts of computation", [16]. The definition signifies the role of heuristics in reducing the effort or cognitive load associated with tasks. Heuristics aim to serve simpler processes that can replace more complex algorithms, aiming for satisfactory, rather than optimal soultions with reduced computational cost. A real-world example is while grocery shopping, where the heuristic would be to shop primarily around the outer ring of the shop in order to maintain a balanced diet, which is where fresh fruit and veg, meats, dairy and bakery products are often located. In contrast to the middle aisles which tend to contain more processed and packaged foods [17].

Heuristics can also be applied to AI problem solving tasks, where they are domain-specific methods and techniques developed to improve the performance of algorithms. AI heuristics are tailored solutions, developed from the need to address the inefficiencies of exhaustive algorithms. The heuristics are designed to be specific to the challenges and characteristics of individual problem domains, emerging from the dissatisfaction with generic algorithms, which in return through further optimation, is able to provide more efficient paths to solutions [18].

One such heuristic used in planning is a landmark heuristic which involves the identification and utilisation of landmarks to direct the search towards the goal. Landmarks are critical points which are known for being a condition, action or event which must be satisfied along any path from initial state to goal. The first step of this heuristic is identifying landmarks, which is carried out by deriving crucial points. After landmark identification, landmarks must be ordered as some landmarks may be required to be reached first before the next. The heuristic values can then be informed by the unvisited landmarks and be guided towards them [19].

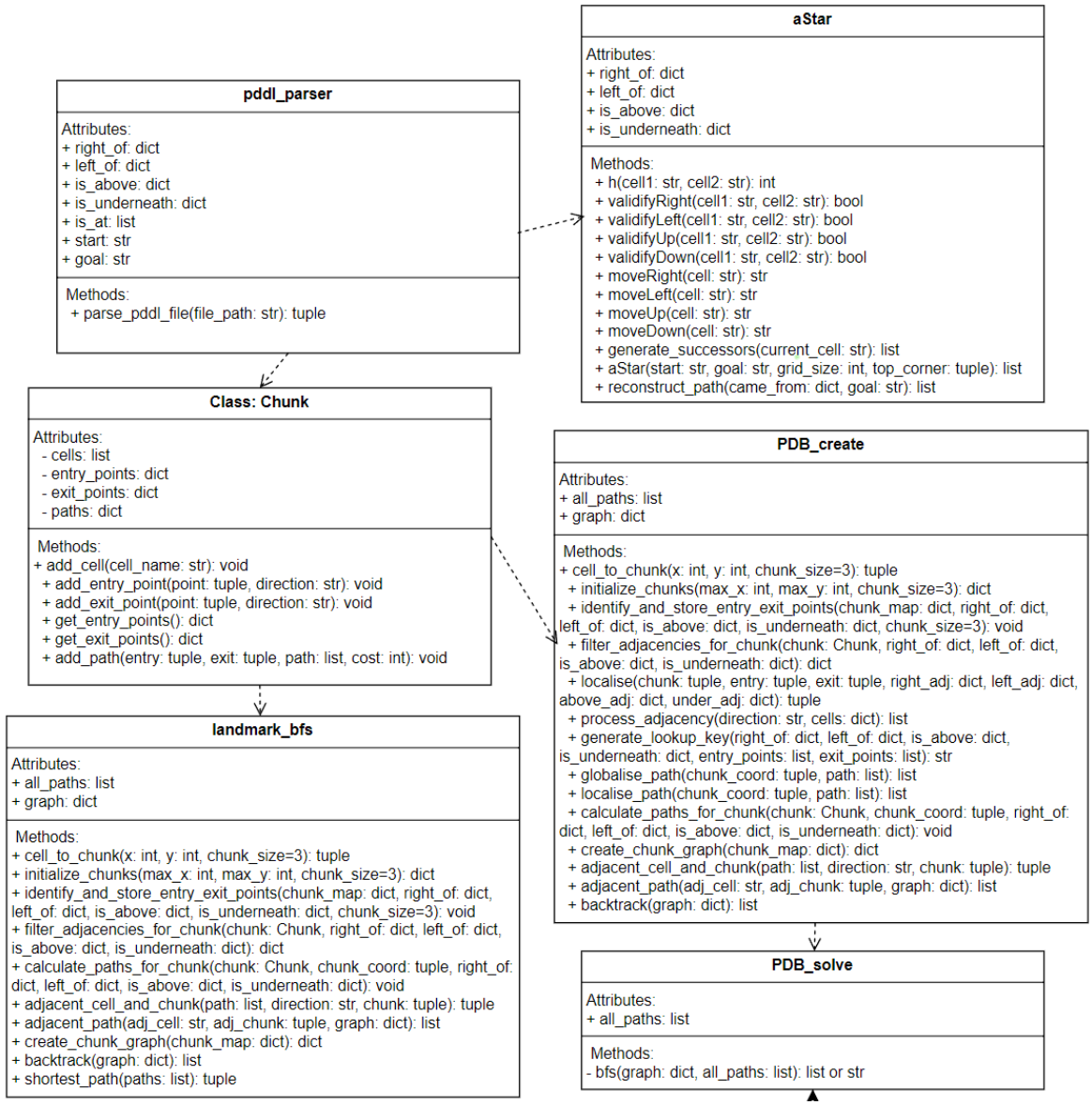
Another heuristic function is the pattern database, which is based on identifying patterns and storing solutions to those patterns in a lookup table. Pattern databases store optimal solutions for subsets of a problem, so once that subset is identified, instead of having to compute a solution, it can just be looked up in the database and reused saving on computation costs during search [20]. A pattern database can act like a pre-computed "cheat sheet" to guide the search. However the disadvantage of this is that there is pre-computation required which may not always be viable for the problem.

The manhattan distance is a simple heuristic function which estimates the distance between two states, based on the absolute differences in their coordinates. This can be used with a search algorithm such as A* to prioritise states that are estimated to be closer to the goal. The manhattan distance differs from the Euclidean distance as it focuses on the differences in horizontal plus the difference in vertical between the two points, making it ideal for movements which are limited to vertical and horizontal compared to Euclidean which takes a diagonal path to the goal.

Chapter 3: **Software engineering**

3.1 Unified Modeling Language (UML)

Unified Modelling Language is a visual language used to document projects. There are many different types of UML such as UML class diagrams, use case diagrams and activity diagrams depending on project. The project will be visualised using a UML package diagram, package diagrams are a subset of class diagrams, used to show the structure and organisation of various model elements. Each rectangle in this diagram, unless specified as a class, represents a package which contains a collection of classes, interfaces, or other packages. The arrows between signify the dependence of these packages.



The diagram starts with the parsing of a pddl file, which is the root of all solution techniques as the parsed adjacencies are utilised within every technique. After the pddl file has been parsed, it can be used to solve the maze using aStar, or split the maze up into chunks

which can open up more possibilities for divide and conquer techniques as well as generating pattern databases and creating landmarks through adjacent chunks to guide searches and apply precomputed solutions.

3.2 Methodology

The agile methodology has been chosen for this project due to the need for feasibility testing before working on the main PDDL implementation. There are also a number of unknowns within the project such as the learning curve of PDDL and SMT-LIB, this would allow for quick changes if any potential risks come up during the project. Furthermore, while generating obstacles in a grid as well as breaking the problem up into chunks, it is important to validate solutions on smaller subsets of the problem.

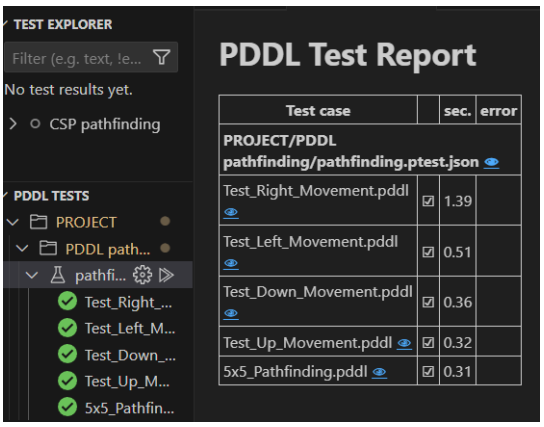
Agile development allows for the project to be broken down into more manageable sections which allows for smaller proof of concept models which can be analysed before working on more complex models that may require some changes. In a complex project that covers AI, the use of continuously reassessing can improve models and potential algorithms used. Furthermore the experimental nature of AI is positively benefitted from as it can easily accept changes in requirements as new discoveries are made during the project lifecycle.

Another benefit is the testing aspect of a project. Agile development allows for continuous testing which can find issues within an algorithm or implementation early on, preventing issues down the line after a lot more work has been completed.

3.3 Testing

Test Driven Development (TDD) has been a focus during the development of the project. From the beginning, we have prioritised writing unit tests prior to implementing code. This approach not only ensures that each piece of code is tested at the time of development, but has also allowed for the creation of a robust suite of tests that serves as a safeguard against future regressions.

During the development of pddl files, the built in unit test functionality of the PDDL extension in VSCode was utilised to test every action as it was implemented, allowing for the problem to be built up and bugs to be spotted early and more easily. As seen below, we can test multiple tests at the same time through the use of multiple problem files.



Moving onto the bespoke solutions, python's unittest library was used to create unit tests to extensively test every method as it was being implemented. During the implementation of pathfinding using CSP, we can see multiple test for each function, to ensure correctness and robustness from the ground up. These tests provided immediate feedback on the logic and functionality.

```
class TestIsAdjacent(unittest.TestCase):
    def test_adjacent_positions(self):
        self.assertTrue(is_adjacent((0, 0), (0, 1))) # Right
        self.assertTrue(is_adjacent((0, 2), (0, 1))) # Left
        self.assertTrue(is_adjacent((0, 0), (1, 0))) # Down
        self.assertTrue(is_adjacent((1, 0), (0, 0))) # Up
    def test_non_adjacent_positions(self):
        self.assertFalse(is_adjacent((0, 0), (2, 0))) # Far Down
        self.assertFalse(is_adjacent((0, 0), (0, 2))) # Far Right
        self.assertFalse(is_adjacent((1, 1), (2, 2))) # Diagonal
class TestVisitedFunction(unittest.TestCase):
    def test_not_visited_basic(self):
        visited = [(0, 1), (1, 1), (2, 1)]
        pos = (3, 1)
        self.assertTrue(not_visited(pos, visited),)
    def test_not_visited_already_visited(self):
        visited = [(0, 1), (1, 1), (2, 1)]
        pos = (1, 1)
        self.assertFalse(not_visited(pos, visited),)
    def test_not_visited_empty_visited(self):
        visited = []
        pos = (0, 0)
        self.assertTrue(not_visited(pos, visited),)
    def test_not_visited_border_position(self):
        visited = [(0, 0), (0, 1), (0, 2)]
        pos = (0, 3)
        self.assertTrue(not_visited(pos, visited),)
```

3.4 Version control system

GitLab was the choice of version control system for this project. Based on git, GitLab provides a lot of useful features such as the ability to track every change, along with reverting to prior states which allows me to easily manage each version of my project in case of any issues in the current implementation. GitLab provides functionality for creating branches for each feature of the project, which allows for each section to be worked on independently without any clashes from previous implementations or other files in the repository. Once each feature was completed and tested thoroughly, it can then be merged back into the main workflow.

Chapter 4: **Development**

4.1 Planning: proof of concept

The development stage began with understanding the syntax of PDDL. Modelling a more manageable problem before tackling a classic 9x9 problem could serve as a proof of concept which would help with the feasibility of the project, as well as providing a model that could be experimented on to gain familiarity without requiring many changes or time to test as the complexity is low.

```
(define (domain sudokuv)
(:requirements :strips :typing)
(:types
  digit cell
)
)
```

To begin defining the domain, 2 types are required, one for digits and one for cells, which allows for all the integers to be defined along with all of the required cells. For requirements the first one is ":strips" which allows me to define actions with preconditions and effects which is required for the 3x3 as the preconditions would be to check if a digit has been used yet or not and depending on that, the effect would be placing a digit. The ":typing" requirement allows me to use multiple types like digits and cells.

```
(:predicates
  (is-available ?d)
  (empty ?cell)
  (filled ?cell ?digit)
)
```

The predicates used for the 3x3 implementation include, "is-available" which takes the parameter ?d or digit and is a way that the available digits can be represented. "empty" takes in cells and represents which cells are required to be filled. Finally "filled" takes in both cells and digits to signify which cells are filled while describing the digits that are placed in them. This works as in the 3x3 problem, the integers only appear once so there is no need to worry about the rows, column and subgrids rules. The only rule that needs to be maintained is that there is only once instance of each digit.

```
(:action place-digit
  :parameters (?d - digit ?c - cell)
  :precondition (and (empty ?c) (is-available ?d))
  :effect (and (filled ?c ?d) (not (empty ?c)) (not(is-available ?d)))
)
```

Only one action is required, which is place-digit, this takes in the parameters digit, and cell. The precondtition, checks for cells that are empty and digits that have not been used yet. If both of these conditions are met, the effect is that the empty cell is filled with the available digit while also making sure that cell is now labelled as not empty and the digit is no longer available as it has been used once already.

The problem file can be split up into 3 sections. The first where all objects are defined.

```
(:objects
  one two three four five six seven eight nine - digit
  cell00 cell01 cell02 - cell
  cell10 cell11 cell12 - cell
  cell20 cell21 cell22 - cell
)
```

Here, all digits from one to nine have been defined as type digit, along with all the cells, the cells have their position in the object name so that they can easily be identified.

```
(:init
 (filled cell00 two) (not (is-available two))
 (filled cell01 four) (not (is-available four))
 (filled cell02 six) (not (is-available six))
 (filled cell10 eight) (not (is-available eight))

 (is-available one) (is-available three)
 (is-available five) (is-available seven)
 (is-available nine)

 (empty cell11) (empty cell12)
 (empty cell20) (empty cell21)
 (empty cell22)
 )
```

The initial state shows the sample problem I have used in which 4 cells are filled with digits, leaving 5 empty cells. The filled predicate has been used to show which cells are filled with their respective digits and alongside that I have made clear that the digit used in that cell is no longer available. Next the digits that are available have been defined using the is-available predicate, and finally the cells which have not been filled have been defined using the empty predicate.

```
(:goal
      (and
        (not (empty cell00))
        (not (empty cell01))
        (not (empty cell02))
        (not (empty cell10))
        (not (empty cell11))
        (not (empty cell12))
        (not (empty cell20))
        (not (empty cell21))
        (not (empty cell22))
      )
  )
```

Defining the goal state was a little challenging as at first, the attempt was to use the filled predicate while leaving the parameters unknown, however this returned an error with the unknown parameter, so after some experimenting, this was the approach that worked. The goal is now defined as each cell required to not be empty, or in other words, a backwards way to achieve all empty cells having digits placed in them to complete the puzzle.

```
Plan found:
0.00100: (place-digit one cell11)
0.00200: (place-digit three cell12)
0.00300: (place-digit five cell20)
0.00400: (place-digit seven cell21)
0.00500: (place-digit nine cell22)
```

The plan was generated using the online solver in 0.302 seconds, with the default configuration of the PDDL extension in VSCode. The result was a success, the place digit action placed the remaining digits in the empty cells. This problem served as a good introduction to PDDL, as problems were easy to detect using the built in validator and the structure of the syntax was straightforward.

4.2 CSP: proof of concept

The next step is to model the same problem but as a constraint satisfaction problem, in order to compare the performance of each implementation. The language of choice is SMT-LIB, which is typically used for constraint solving and solving SMT problems. The following code is used to set up the problem.

```
(set-option :produce-models true)
(declare-datatypes () ((Col x0 x1 x2)))
(declare-datatypes () ((Row y0 y1 y2)))
(declare-fun Board (Col Row) Int)
```

```
;;;;;; --- BEGIN-CONSTRAINTS-1 ---
; Sudoku constraints for distinct values in rows
(assert (distinct (Board x0 y0) (Board x0 y1) (Board x0 y2)
```

```
                    (Board x1 y0) (Board x1 y1) (Board x1 y2)
                    (Board x2 y0) (Board x2 y1) (Board x2 y2)))

  ; Sudoku constraints for distinct values in columns
  (assert (distinct (Board x0 y0) (Board x1 y0) (Board x2 y0)
                    (Board x0 y1) (Board x1 y1) (Board x2 y1)
                    (Board x0 y2) (Board x1 y2) (Board x2 y2)))
```

The first assert functions are used as to maintain the rules of sudoku, in which each row and column has to be distinct.

```
  ;;;;;; --- END-CONSTRAINTS-1 ---

  ;;;;; 3x3
  (assert (= (Board x0 y0) 6))
  (assert (= (Board x2 y1) 9))
  (assert (= (Board x0 y2) 2))
  (assert (= (Board x2 y2) 4))
```

Next, the assert statements above are used to define the problem, showing which positions on the board contain an integer.

```
  ;;;;;; --- BEGIN-CONSTRAINTS-DOMAIN ---

(assert (and (< (Board x0 y0) 10) (> (Board x0 y0) 0)))
(assert (and (< (Board x0 y1) 10) (> (Board x0 y1) 0)))
(assert (and (< (Board x0 y2) 10) (> (Board x0 y2) 0)))
(assert (and (< (Board x1 y0) 10) (> (Board x1 y0) 0)))
(assert (and (< (Board x1 y1) 10) (> (Board x1 y1) 0)))
(assert (and (< (Board x1 y2) 10) (> (Board x1 y2) 0)))
(assert (and (< (Board x2 y0) 10) (> (Board x2 y0) 0)))
(assert (and (< (Board x2 y1) 10) (> (Board x2 y1) 0)))
(assert (and (< (Board x2 y2) 10) (> (Board x2 y2) 0)))
```

For the domain, each cell on the board, can be assigned integers from 1 to 9 which is done with more assert functions.

```
; Compute solution
(check-sat)

(get-value ((Board x0 y0)))
(get-value ((Board x1 y0)))
(get-value ((Board x2 y0)))
(get-value ((Board x0 y1)))
(get-value ((Board x1 y1)))
(get-value ((Board x2 y1)))
(get-value ((Board x0 y2)))
(get-value ((Board x1 y2)))
(get-value ((Board x2 y2)))
```

Finally the solution is checked for satisfiability of all constraints and will then return all of the values of each cell on the board. The result was a success with the solution time taking 0.082 seconds.

4.3 Classic Sudoku PDDL model

In order to tackle this problem, some changes will be required as multiple instances of the same integer can occur in a complete solution. To deal with this, some new types and predicates were introduced.

```
(:types
  digit cell row column subgrid - objects
)

(:predicates
  (BelongRow ?r - row ?c - cell)
  (UsedRow ?r - row ?num - digit)
  (BelongColumn ?col - column ?c - cell)
  (UsedColumn ?col - column ?num - digit)
  (BelongSubgrid ?s - subgrid ?c - cell)
  (UsedSubgrid ?s - subgrid ?num - digit)
  (empty ?c - cell)
)
```

Three new objects were added, row, column and subgrid. The aim of these objects is to break the problem down into each row, column and subgrid in the problem so that each can be dealt with accordingly. New belong predicates were also introduced to help the planner in understanding the cell structure, allowing definitions as to which cells are associated with each row, column and subgrid. Furthermore to deal with the is-available predicate no longer being valid, three new predicates have been introduced to check if a digit has been used in a row, column and subgrid.

```
(:action place-digit
  :parameters (?num - digit ?c - cell ?r - row ?col - column ?s - subgrid)
  :precondition (and (empty ?c)
                     (BelongRow ?r ?c)
                     (BelongColumn ?col ?c)
                     (BelongSubgrid ?s ?c)
                     (not (UsedRow ?r ?num))
                     (not (UsedSubgrid ?s ?num))
                     (not (UsedColumn ?col ?num)))
  :effect (and (not (empty ?c))
               (UsedColumn ?col ?num)
               (UsedRow ?r ?num)
               (UsedSubgrid ?s ?num))
)
```

The place-digit action has required some changes too, increasing its complexity. Here, the place-digit action first checks if the cell is empty then checks for the cell structure and that the

digit has not been used in the row, column and subgrid yet. The effect, if these preconditions have been met is that the cell is now no longer empty and the digit is used within the specific location described by the row, column and subgrid number.

```
(BelongRow row0 cell00) ... (BelongRow row0 cell08)
...
(BelongRow row8 cell80) ... (BelongRow row8 cell88)

(BelongColumn column0 cell00) ... (BelongColumn column0 cell80)
 ...
(BelongColumn column8 cell08) ... (BelongColumn column8 cell88)

(BelongSubgrid subgrid0 cell00) ... (BelongSubgrid subgrid0 cell22)
 ...
(BelongSubgrid subgrid8 cell66) ... (BelongSubgrid subgrid8 cell88)
```

Above is how the predicates for BelongColumn, BelongSubgrid and BelongRow were used to define the structure of the cells and cells that are associated to each row, column and subgrid.

The first implementation contained a full 9x9 sudoku problem which resulted in the online solver timing out before a plan could be found. To mitigate this, the fast downward planner was set up and multiple algorithms from the planner were used to get a plan however each time, the planner timed out.

```
(UsedRow row0 six) (UsedColumn column1 six) (UsedSubgrid subgrid0 six)
(UsedRow row0 two) (UsedColumn column5 two) (UsedSubgrid subgrid1 two)
(UsedRow row0 seven) (UsedColumn column8 seven) (UsedSubgrid subgrid2 seven)

 (empty cell00)
 (empty cell02)
 (empty cell03)
 (empty cell04)
 (empty cell06)
 (empty cell07)
```

To get to the root of the problem, the model was adapted to run for just the first row and a row was added on after a successful plan was generated. Above, we can see three of the cells are populated with digits leaving 7 empty cells, the online solver found a plan for this problem with ease which meant that the model was not the problem, rather the increase in complexity between a 3x3 puzzle and a classic 9x9 puzzle.

This implementation worked all the way up to the 6th row, where the planner takes 11 seconds to find a plan if good progress is made early on, and for other problems in a 6x9 implementation, the planner times out again.

4.4 Final CSP implementation

To benchmark sudoku against a CSP solver, and get to the root of the problem, the 3x3 implementation in SMT-LIB mentioned earlier has been scaled up using the same assert statements, only on a full scale 9x9 problem. The only difference here is that the z3 solver was able to find a solution within 0.04 seconds.

4.5 PDDL: pathfinding

From here, development starts on the pathfinding problem, The same grid like structure that was introduced in the Sudoku domain will be implemented into this domain. A new pddl file is created to define the required predicates, and actions, with the aim being getting an agent from a defined point to a goal point.

```
(define (domain pathfinding)
(:requirements :strips :typing)
(:types
  position
)
(:predicates
  (at ?p - position)  ; Agent is at position p
  (cell ?p - position)  ; Indicates a cell at position p
)

; Actions here
(:action move-right
  :parameters (?from ?to - position)
  :precondition (and
      (at ?from)
      (cell ?to)
  )
  :effect (and
      (not (at ?from))
      (at ?to)
  )
 )
 )
```

Here we can see that I am using 2 predicates, 'at' and 'cell'. The 'at' predicate is used to signify where the current position of the agent is. The cell predicate is used to define all the cells in the grid that the agent would have to traverse to reach the goal. The first action 'move-right' has also been defined, the action takes in the parameters 'from' and 'to', which are used to describe which cell the agent is coming from and which cell the move would allow the agent to reach.

```
(define (problem problem-pathfinding)
(:domain pathfinding)
(:objects
   cell00 cell01 - position
)

(:init
      (at cell00)
      (right-of cell01 cell00)
)
(:goal
      (and
      (at cell01)
 )
  )
)
```

This is the first problem file created for the domain, the problem starts small to test the right action before we proceed to add more complexity to the problem. 2 objects are defined here, 'cell00' and 'cell01' which are the only 2 cells required to test a right movement. The initial state contains the current cell the agent is located at, along with one adjacency relationship which signifies that 'cell01' is to the right of 'cell00'. Finally the goal state is set to 'cell01'.

```
  (:predicates
    (at ?p - position)  ; Agent is at position p
    (cell ?p - position)  ; Indicates a cell at position p
    (right-of ?p1 ?p2 - position) ;Defines adjacency rules
  )

  ; Define actions here
  (:action move-right
    :parameters (?from ?to - position)
    :precondition (and
        (at ?from)
        (right-of ?to ?from)
    )
    :effect (and
        (not (at ?from))
        (at ?to)
    )
```

The domain file required updating to understand the adjacency relationship, so the predicate (right-of) was added to define a right adjacency relationship. The action was updated too, with the precondition that there is an adjacency relationship between the 'to' and 'from' cells. If there is a valid adjacency, the effect is that the agent is now at the cell to the right of where it previously was, and no longer where it previously was.

```
Test file:
{
  "defaultDomain": "pathfinding.pddl",
  "defaultOptions": "",
  "cases": [
    {
      "problem": "pathfinding-problem1.pddl"

    },
    {
      "problem": "pathfinding-problem2.pddl"
    }
  ]
}

Problem designed to fail:
(define (problem problem-pathfinding)
 (:domain pathfinding)
 (:objects
     cell00 cell01 - position
 )

 (:init
        (at cell00)
        (right-of cell00 cell01)
 )
 (:goal
        (and
        (at cell01)
  )
    )
)
```

Utilising the PDDL extension in VSCode, it was possible to create test cases using problem files, so another problem file that was designed to fail by reversing the cells in the right adjacency was created, so that there is no valid adjacency from where the agent is currently located.

```
  (:action move-left
  :parameters (?from ?to - position)
  :precondition (and
      (at ?from)
      (left-of ?to ?from)
  )
  :effect (and
      (not (at ?from))
      (at ?to)
  )
)
```

Next, the 'move-left' action is created with the same structure as 'move-right' action but with the difference being a new predicate called (left-of) which serves the same function as the (right-of) predicate but for cells to the left.

```
  (:action move-down
    :parameters (?from ?to - position)
    :precondition (and
        (at ?from)
        (is-underneath ?to ?from)
    )
    :effect (and
        (not (at ?from))
        (at ?to)
    )
  )

  (:action move-up
  :parameters (?from ?to - position)
  :precondition (and
      (at ?from)
      (is-above ?to ?from)
  )
  :effect (and
      (not (at ?from))
      (at ?to)
  )
)
```

The final 2 actions are implemented the same way, while leveraging new predicates to check if a cell is undereath another cell or if a cell is above another cell, validating the possible move for the action.

```
  (define (problem problem-pathfinding)
 (:domain pathfinding)
 (:objects
     cell00 cell01 cell02 - position
     cell10 - position
 )

 (:init
        (at cell10)
        (right-of cell01 cell00) (right-of cell02 cell01)
        (left-of cell00 cell01) (left-of cell01 cell02)
        (is-underneath cell10 cell00)
        (is-above cell00 cell10)
 )
 (:goal
        (and
        (at cell02)
  )
    )
)
```

Above is the final test file used which introduces all the different types of adjacencies and expects to test a move upwards. All of the actions were tested the same way.

```
  (define (problem problem-pathfinding)
(:domain pathfinding)
(:objects
   cell00 cell01 cell02 cell03 cell04 - position
   cell10 cell11 cell12 cell13 cell14 - position
   cell20 cell21 cell22 cell23 cell24 - position
   cell30 cell31 cell32 cell33 cell34 - position
   cell40 cell41 cell42 cell43 cell44 - position
)

(:init
  (at cell02)
  (right-of cell01 cell00) ... (right-of cell04 cell03)
  (right-of cell11 cell10) ... (right-of cell14 cell13)
  (right-of cell21 cell20) ... (right-of cell24 cell23)
  (right-of cell31 cell30) ... (right-of cell34 cell33)
  (right-of cell41 cell40) ... (right-of cell44 cell43)

  (left-of cell00 cell01) ... (left-of cell03 cell04)
  (left-of cell10 cell11) ... (left-of cell13 cell14)
  (left-of cell20 cell21) ... (left-of cell23 cell24)
  (left-of cell30 cell31) ... (left-of cell33 cell34)
  (left-of cell40 cell41) ... (left-of cell43 cell44)

  (is-underneath cell10 cell00) ... (is-underneath cell40 cell30)
  (is-underneath cell11 cell01) ... (is-underneath cell41 cell31)
  (is-underneath cell12 cell02) ... (is-underneath cell42 cell32)
  (is-underneath cell13 cell03) ... (is-underneath cell43 cell33)
```

```
  (is-underneath cell14 cell04) ... (is-underneath cell44 cell34)

  (is-above cell00 cell10) ... (is-above cell30 cell40)
  (is-above cell01 cell11) ... (is-above cell31 cell41)
  (is-above cell02 cell12) ... (is-above cell32 cell42)
  (is-above cell03 cell13) ... (is-above cell33 cell43)
  (is-above cell04 cell14) ... (is-above cell34 cell44)

 )
 (:goal
    (and
    (at cell44)
  )
  )
)
```

Finally a 5x5 grid is represented in the above problem file, with all the valid adjacencies and adjacencies inbetween represented with '...'. The objects are all the possible cells in a 5x5 grid, the initial state contains the start cell which is 'cell02', and all the relevant adjacencies. The goal state just requires for the agent to be located at the bottom right of the grid. The resulting path solved using the Fast Downward planner is below.

```
(move-down cell02 cell12)
(move-down cell12 cell22)
(move-down cell22 cell32)
(move-down cell32 cell42)
(move-right cell42 cell43)
(move-right cell43 cell44)
; cost = 6 (unit cost)
```

4.6 CSP: pathfinding

The CSP pathfinding approach taken, was inspired by Figure 6.5 in Artificial Intelligence: A Modern Approach [21], specifically the approach to backtracking to find a solution. In order to develop the bespoke solver, as pathfinding is not a typical problem modelled as a csp, the problem had to be broken down into its fundamentals.

```
path_length = 4

#variables
steps = [f"step_{i}" for i in range(1, path_length + 1)]

grid_positions = [(x, y) for x in range(5) for y in range(5)]
domains = {step: grid_positions for step in steps}

# Constraints
def is_adjacent(pos1, pos2):
    row_diff = abs(pos1[0] - pos2[0])
    return (row_diff == 1 and col_diff == 0) or
           (row_diff == 0 and col_diff == 1)
```

The first step was to initialise my domains and variables, for this I took the approach to set the steps in a path as the variables, and the avaible cells in the grid as the possible domains that could be assigned to a cell. The first constraint is also set which is to ensure that subsequent steps contain a cell which is adjacent to the previous so that the path makes logical sense.

```
def not_visited(pos, visited):
    return pos not in visited

def select_unassigned_variable(assignment, steps):
    for step in steps:
        if step not in assignment:
            return step
    return None

def is_consistent(step, pos, assignment, is_adjacent, visited):
    if not not_visited(pos, visited):
        return False

    if assignment:
        previous_step_number = int(step.split('_')[1]) - 1
        if previous_step_number > 0:
            previous_step = f"step_{previous_step_number}"
            if previous_step in assignment:
                last_step_pos = assignment[previous_step]
                if not is_adjacent(last_step_pos, pos):
                    return False
    return True
```

The not_visited function takes in a position and list visited, and checks if the position is in the list as we want to avoid going back to the same cell, and want to continue moving towards the goal. The select_unassigned_variable function takes in a list of assignments of cells to steps and checks whether the step has been assigned a cell yet. is_consistent is a function that checks if assigning a position to a step is consistent with the constraints and has not been visited yet.

```
def assign(step, pos, assignment, visited):
    assignment[step] = pos
    visited.append(pos)

def unassign(step, assignment, visited):
    if step in assignment:
        pos = assignment.pop(step)
        visited.remove(pos)

def goal_test(assignment, steps, goal_pos):
    if len(assignment) != len(steps) or assignment[steps[-1]] != goal_pos:
        return False

    visited = set()
    for i, step in enumerate(steps):
        if i == 0 and assignment[step] != start_pos:
            return False

        if i > 0:
            prev_step = steps[i - 1]
            if not is_adjacent(assignment[prev_step], assignment[step]) or
              assignment[step] in visited:
                return False

        visited.add(assignment[step])
    return True
```

Assign and unassign are used to assign cells to steps and unassign cells from steps whenever they are called. While the goal_test function checks length to make sure the path length is the same as the numeber of initialised steps, and that the last cell in the last step is a goal position. It then checks the start position, and then the adjacency constraint against each step. If all those checks pass, it returns True.

```
def backtrack(assignment, steps, domains, visited, goal_pos):
  if len(assignment)==len(steps) and goal_test(assignment, steps, goal_pos):
      return assignment

  step = select_unassigned_variable(assignment, steps)
  if step is None:
      return None

  for pos in domains[step]:
      if is_consistent(step, pos, assignment, is_adjacent, visited):
          assign(step, pos, assignment, visited)
          result = backtrack(assignment, steps, domains, visited, goal_pos)
          if result is not None:
              return result
          unassign(step, assignment, visited)
  return None
```

The backtrack functions is a recursive function that works by first checking if all variables are assigned and goal conditions are met, If the goal is not met, an unassigned step will be used to iterate over the domain. a consistency check will take place to check if the current cell in the domain is viable and if it is, then the cell is assigned to the step and the function will call itself. If a solution is found, the assignment which is the list of steps with valid cells will be returned. If all steps have been had attempts to be assigned by all cells and no solution is found, None is returned.

```
def iterative_deepening(start_pos, goal_pos, max_depth=20):
  for path_length in range(1, max_depth + 1):
      steps = [f"step_{i}" for i in range(1, path_length + 1)]
      domains = {step: grid_positions for step in steps}
      visited = [start_pos]
      assignment = {steps[0]: start_pos}

      solution = backtrack(assignment, steps, domains, visited, goal_pos)
      if solution:
          print(f"Solution found with path length {path_length}:", solution)
          return solution
  print("No solution found within the maximum path length.")
    return None
```

Finally, an iterative deepening technique is used to increase the max depth, or the number of steps the the path is given until a valid path is found. Iterative deepening takes in a start position, goal and the maximum depth which can be customised if a maze requires more steps to find a path. This approach ensures the shortest path. For each path length, the backtrack search is carried out again until there is a solution.

4.7 PDDL: parsing

Since the pddl file solver will be domain specific, we are only required to parse the problem file and retrieve, adjacencies as well as start and goal points.

```
def parse_pddl_file(file_path):
  right_of = {}
  left_of = {}
  is_above = {}
  is_underneath = {}
  is_at = []
  with open(file_path, 'r') as file:
      for line in file:
          # Right adjacency rules
          right_of_matches =
            re.findall(r'\s*\(right-of (\S+) (\S+)\)', line)
          if right_of_matches:
              for match in right_of_matches:
                  right_of[match[1]] = match[0]

          # Left adjacency rules
          left_of_matches =
            re.findall(rf'\s*\(left-of (\S+) (\S+)\)', line)
          if left_of_matches:
              for match in left_of_matches:
                  left_of[match[1]] = match[0]

          # Above adjacency rules
          is_above_matches =
              re.findall(rf'\s*\(is-above (\S+) (\S+)\)', line)
          if is_above_matches:
                for match in is_above_matches:
                   is_above[match[1]] = match[0]

           # Underneath adjacency rules
          is_underneath_matches =
            re.findall(rf'\s*\(is-underneath (\S+) (\S+)\)', line)
          if is_underneath_matches:
              for match in is_underneath_matches:
                  is_underneath[match[1]] = match[0]

          # (at X) matches
          is_at_matches =
            re.findall(rf'\s*\(at cellx(\d+)y(\d+)\)', line)
          if is_at_matches:
              for match in is_at_matches:
                x, y = match
                  cell_name = 'cellx' + x + 'y' + y
                  is_at.append(cell_name)

      start = is_at[0]
      goal = is_at[1]
  return right_of, left_of, is_above, is_underneath, start, goal
```

The pddl file is first opened to read in the function, and line by line it checks utilises regex to search for all the predicate types. For example, the first set of matches it it looking for is the right-of adjacencies which begin with '(right-of' and then it captures the next two items which are the relevant cells required to construct the dictionary. So line by line it checks for each adjacency while building up their respective dictionaries as well as another check for '(at' predicates which indicate where the agent starts and the required goal state is. As the goal state is specified after the initial state, we can assume the first match of '(at' will be where the agent starts and the second is the goal. Once the file has been completely read through, line by line, the dictionaries that hold the adjacencies as well as the start and the goal are returned.

4.8 Planning: Bespoke pathfinding planner

In order to solve the parsed problem file, I decided on using A* due to its simplicity and effectiveness coupled with the Manhattan distance as my pathfinding problem is limited to vertical and horizontal movements making the Manhattan distance a realistic estimate from the current cell to the goal.

```
  def h(cell1, cell2):
    x1, y1 = map(int, re.findall(r'(\d+)', cell1))
    x2, y2 = map(int, re.findall(r'(\d+)', cell2))
    return abs(x1 - x2) + abs(y1 - y2)
```

This is my heuristic function, it takes in 2 cells and returns the horizontal distance added onto the vertical distance.

```
  def validifyRight(cell1, cell2, right_of):
    return right_of.get(cell1) == cell2

def validifyLeft(cell1, cell2, left_of):
    return left_of.get(cell1) == cell2

def validifyUp(cell1, cell2, is_above):
    return is_above.get(cell1) == cell2

def validifyDown(cell1, cell2, is_underneath):
    return is_underneath.get(cell1) == cell2

def moveRight(cell):
    return right_of.get(cell)

def moveLeft(cell):
    return left_of.get(cell)

def moveUp(cell):
    return is_above.get(cell)

def moveDown(cell):
    return is_underneath.get(cell)
```

The above is how the actions are implemented, so just like with pddl, there is a precondition coupled with the effect. Each validify function takes in two cells along with the respective adjacency dictionary, then checks whether there is a valid adjacency for the cell pairing. If there is a valid adjacency, True is returned. The move functions take the cell that requires a respective movement with and returns the value of the key.

```
def generate_successors(current_cell, right_of, left_of,
                        is_above, is_underneath):

    successors = []

    right_cell = moveRight(current_cell)
    if right_cell and validifyRight(current_cell, right_cell, right_of):
        successors.append(right_cell)

    left_cell = moveLeft(current_cell)
    if left_cell and validifyLeft(current_cell, left_cell, left_of):
        successors.append(left_cell)

    up_cell = moveUp(current_cell)
    if up_cell and validifyUp(current_cell, up_cell, is_above):
        successors.append(up_cell)

    down_cell = moveDown(current_cell)
    if down_cell and validifyDown(current_cell, down_cell, is_underneath):
        successors.append(down_cell)

    return successors
```

A successor generator has been implemented which makes use of the precondition and move functions, by taking in a cell along with every adjacency relationship dictionary. The cell is then checked for all possible adjacencies and if there are any cells that are adjacent to it in the grid, a successors list is appended to and then finally returned once all successors have been appended.

```
  def aStar(right_of, left_of, is_above, is_underneath, start, goal,
            grid_size, top_corner):
    x_min, y_min = top_corner
    g_score = {f"cellx{x}y{y}": float('inf') for x
          in range(x_min, x_min + grid_size) for y
          in range(y_min, y_min + grid_size)}
    f_score = {f"cellx{x}y{y}": float('inf') for x
          in range(x_min, x_min + grid_size) for y
          in range(y_min, y_min + grid_size)}

    g_score[start] = 0
    f_score[start] = h(start, goal)

    open_set = PriorityQueue()
    # Priority, G-score, Node
    open_set.put((h(start, goal), h(start, goal), start))

    came_from = {}

    while not open_set.empty():
        current_cell = open_set.get()[2]
        if current_cell == goal:
            break

        for child_cell in generate_successors(
          current_cell, right_of, left_of, is_above, is_underneath):
            temp_g_score = g_score[current_cell] + 1
            temp_f_score = temp_g_score + h(child_cell, goal)

            if temp_f_score < f_score[child_cell]:
                g_score[child_cell] = temp_g_score
                f_score[child_cell] = temp_f_score
                open_set.put((temp_f_score, temp_g_score, child_cell))
                came_from[child_cell] = current_cell

    if current_cell == goal:
        return reconstruct_path(came_from, goal)
    else:
        return None

def reconstruct_path(came_from, goal):
    path = [goal]
    while goal in came_from:
        goal = came_from[goal]
        path.append(goal)
    path.reverse()
    return path
```

aStar takes in each adjacency relationship dictionary, along with the starting cell for the search, the goal cell, the size of the grid and the top-left corner coordinates of the grid for a future chunking idea. g_score and f_score are first initialised for each cell in the grid with a value of infinity. Then the start cell is has it's g score set to 0 and the f score, the estimated cost to the goal through the manhattan distance. open_set is a priority queue that will be used to prioritise paths with lower costs. came_from is a list which will keep track of the path

taken to reach each cell. A while loop is set so that while the set has entries, to continue exploring. With each iteration, the cell with the lowest f_score will be dequeued and if that cell is teh goal, the shortest path is complete and the goal along with came_from is passed into reconstruct_path. If the current cell is not the goal, the successor generator is called so that all of the current cells child cells can be used to estimate costs to a goal from. If the calucalted f score is lower than the existing f score, the scores are updated and the successor cell is added to the priority queue. came_from is updated to keep track of the path.

If the search is successful, reconstruct_path is called which traces back from the goal cell to the start cell using the dictionary, appending each cell to the path and then reversed to make logical sense then returned.

4.9 PDDL file generation

As the development of a bespoke planner concluded, the issue of creating pddl files manually came up as in order to test the planner extensively, having a way to generate pddl problem files would make the process more efficient. As well as being able to randomise start and goal cells to a range of different pathfinding efficiently.

```
import random

class PDDLProblemGenerator:
    def __init__(self, problem_name, grid_size):
        self.problem_name = problem_name
        self.grid_size = grid_size
        self.initial_position = None

    def generate_objects(self):
        objects = ''
        for i in range(self.grid_size):
            for j in range(self.grid_size):
                objects += f'cellx{i}y{j} - position\n'
        return objects

    def generate_initial_state(self):
        initial_state = ''
        # Randomly select a cell for the initial position
        i_start, j_start = random.randint(0, self.grid_size-1),
                           random.randint(0, self.grid_size-1)
        self.initial_position = (i_start, j_start)
        initial_state += f'(at cellx{j_start}y{i_start})\n'

        for i in range(self.grid_size):
            for j in range(1, self.grid_size):
                initial_state +=
                  f'(right-of cellx{j}y{i} cellx{j-1}y{i})\n'

        for i in range(self.grid_size):
            for j in range(self.grid_size - 1):
                initial_state +=
                  f'(left-of cellx{j}y{i} cellx{j+1}y{i})\n'
```

```
        for i in range(1, self.grid_size):
            for j in range(self.grid_size):
                initial_state +=
                  f'(is-underneath cellx{j}y{i} cellx{j}y{i-1})\n'

        for i in range(self.grid_size - 1):
            for j in range(self.grid_size):
                initial_state +=
                  f'(is-above cellx{j}y{i} cellx{j}y{i+1})\n'

        return initial_state

    def generate_goal_state(self):
        i_start, j_start = self.initial_position
        while True:
            i_goal, j_goal = random.randint(0, self.grid_size-1),
                             random.randint(0, self.grid_size-1)
            if (i_goal, j_goal) != (i_start, j_start):
                break
        goal_state = f'(at cellx{i_goal}y{j_goal})'
        return goal_state

    def generate_pddl(self, file_path):
        with open(file_path, 'w') as f:
            f.write(f'(define (problem {self.problem_name})\n')
            f.write('  (:domain pathfinding)\n')
            f.write('  (:objects\n')
            f.write(self.generate_objects())
            f.write('  )\n')
            f.write('  (:init\n')
            f.write(self.generate_initial_state())
            f.write('  )\n')
            f.write('  (:goal\n')
            f.write('    (and\n')
            f.write(self.generate_goal_state())
            f.write('    )\n')
            f.write('  )\n')
            f.write(')\n')

generator = PDDLProblemGenerator(
                  problem_name='problem-pathfinding', grid_size=5)
generator.generate_pddl('PDDL pathfinding/testaStar.pddl')
```

The class PDDLProblemGenerator generates problem instances. It takes the parameters, name, grid_size and an initial position. The method generate_objects, generates all of the cell objects in the grid by using the grid_size. generate_initial_state utilses the random library to produce 2 random integers within the confines of the grid, then converts that into a coordinate that matches the object format. After that, we have a few for loops to make up all of the adjacencies in the grid as there are no obstacles just yet so every cell is connected to every cell in the grid. generate_goal_state works the same way as the initial state, in that it generates 2 random integers to be used as the goal while comparing it to the start cell to make sure they are not the same, ensuring a path with length longer than 1. Finally the generate_pddl function combines everything and writes to a file in a structure that a pddl problem file would

follow.

4.10 Maze generation and implementation

In order to generate mazes, a library called pyamaze has been utilised due to its customisabilty and output of generated maze. Pyamaze is also able to display the generated maze visually which can help in debugging and validifying paths.

```
def generate_pddl(maze):
  pddl_str = "(define (problem maze-problem)\n    (:domain maze)\n"

  # Objects
  pddl_str += "    (:objects\n        "
  # Adjust for (y, x) format
  objects = [f"cellx{x-1}y{y-1}" for (y, x) in maze.keys()]
  pddl_str += " ".join(objects) + " - cell)\n"

  # Init with the start position
  pddl_str += "    (:init\n        (at cellx0y0)\n"

  # Sets for valid connections
  valid_right_of = set()
  valid_left_of = set()
  valid_is_above = set()
  valid_is_underneath = set()

  for (y, x), dirs in maze.items():
      x_adj, y_adj = x-1, y-1
      current_cell = f"cellx{x_adj}y{y_adj}"

      # East (right-of) connections
      if dirs.get('E'):
          target_cell = f"cellx{x_adj+1}y{y_adj}"
          if (y, x+1) in maze:
              valid_right_of.add((current_cell, target_cell))

      # South (is-underneath) connections
      if dirs.get('S'):
          target_cell = f"cellx{x_adj}y{y_adj+1}"
          if (y+1, x) in maze:
              valid_is_underneath.add((current_cell, target_cell))

      # West (left-of) connections
      if dirs.get('W'):
          target_cell = f"cellx{x_adj-1}y{y_adj}"
          if (y, x-1) in maze:
              valid_left_of.add((current_cell, target_cell))

      # North (is-above) connections
      if dirs.get('N'):
          target_cell = f"cellx{x_adj}y{y_adj-1}"
```

```
            if (y-1, x) in maze:
                valid_is_above.add((current_cell, target_cell))

    for right, left in valid_right_of:
        pddl_str += f"        (right-of {left} {right})\n"
        pddl_str += f"        (left-of {right} {left})\n"
    for above, underneath in valid_is_above:
        pddl_str += f"        (is-above {underneath} {above})\n"
        pddl_str += f"        (is-underneath {above} {underneath})\n"
    for underneath, above in valid_is_underneath:
        pddl_str += f"        (is-underneath {above} {underneath})\n"
        pddl_str += f"        (is-above {underneath} {above})\n"
    for left, right in valid_left_of:
        pddl_str += f"        (left-of {right} {left})\n"
        pddl_str += f"        (right-of {left} {right})\n"

    pddl_str += "    )\n"

    max_y, max_x = max(maze.keys())
    goal_cell = f"cellx{max_x-1}y{max_y-1}"
    pddl_str += f"    (:goal (\n        (at {goal_cell})\n    )\n)"

    pddl_str += ")\n"
    return pddl_str


def main():
    grid_size = 30
    m = maze(grid_size, grid_size)
    m.CreateMaze(loopPercent=21)
    # Set the agent's start position
    a = agent(m, 1, 1)

    # Set the maze's goal (end point) explicitly
    m.goal = (grid_size, grid_size)  # Bottom-right corner

    pddl_content = generate_pddl(m.maze_map)

    pddl_file_path = 'PDDL pathfinding/problem.pddl'

    with open(pddl_file_path, 'w') as file:
        file.write(pddl_content)

    m.run() # Used to visualise maze

if __name__ == "__main__":
    main()
```

The above code is an update on the pddl problem file generator, incorporating a maze aspect. Pyamaze allows us to set the size of the grid along with a loopPercent allowing for variation in maze complexity due to the number of paths that can reach the goal increasing. m.maze_map is a dictionary which contains valid connections in each direction. Objects follows the same pattern as it did in the original maze generation. The maze map is iterated through and for each cell in the map, the neighboring cells are checked to determine valid connections so

if there was a neighboring open cell east from the current cell, right of would gain another adjacency between those two cells. The two cells are then appended until the maze map has been completely iterated through, for each valid connection the pddl predicate is constructed. The goal is defined by the maximum x and y coordinates so in other words the bottom right of the maze. Once all predicates are constructed the complete representation is returned which is then written to a file in main.

4.11 Maze implementation in CSP

```
start_pos = (0, 0)
goal_pos = (4,4)

grid_size = 5
m = maze(grid_size, grid_size)
m.CreateMaze()
# Set the agent's start position
a = agent(m, 1, 1)

# Set the maze's goal (end point)
m.goal = (grid_size, grid_size)

mazeGrid = m.maze_map

solution = iterative_deepening(start_pos, goal_pos, grid_size, mazeGrid)
```

The technique for solving a maze pathfind using CSP remains mostly the same, above is the initialisation of the maze, which involves setting the size, start position and goal. Once the maze is initialised, it is passed through into the iterative deepening function.

```
  def is_move_allowed(maze_map, current, next):
    maze_current = (current[1] + 1, current[0] + 1)
    maze_next = (next[1] + 1, next[0] + 1)

    direction = (maze_next[0] - maze_current[0],
                 maze_next[1] - maze_current[1])

    if direction == (1, 0):
        return 'S' in maze_map[maze_current]
                  and maze_map[maze_current]['S'] == 1
    elif direction == (-1, 0):
        return 'N' in maze_map[maze_current] and
                      maze_map[maze_current]['N'] == 1
    elif direction == (0, 1):
        return 'E' in maze_map[maze_current] and
                      maze_map[maze_current]['E'] == 1
    elif direction == (0, -1):
        return 'W' in maze_map[maze_current] and
                      maze_map[maze_current]['W'] == 1

    return False
```

The structure of the CSP solver allows for a simple constraint addition to check for valid moves. Here there is a constraint called is_move_allowed, which takes in the maze_map along with the current cell and a next cell to be compared to. Since the maze_map starts it's coordinates at 1 and my solver used 0, the cells are converted into a format which can be compared to the maze map. The direction is based on the result of the difference in x and y. Utilising the offset, we can distinguish which direction the next cell is in. We check whether the direction is one that has the value 1 which would mean there is a valid cell connection and no wall. Walls are defined using 0 and open connections are defined with 1 in the maze map. If the move is one that is allowed, True is returned, if the move is not allowed, False is returned. Along with this, there is an extra check in consistency checking while and goal checking.

4.12 New approach: Chunking and pattern database

A different approach to solving the maze would be to split it up into chunks and solving the chunks then connecting them to find a final path.

```
class Chunk:
  def __init__(self):
      self.cells = []
      self.entry_points = {}
      self.exit_points = {}
      self.paths = {}

  def add_cell(self, cell_name):
      self.cells.append(cell_name)

  def add_entry_point(self, point, direction):
      self.entry_points[point] = direction

  def add_exit_point(self, point, direction):
      self.exit_points[point] = direction

  def get_entry_points(self):
      return self.entry_points

  def get_exit_points(self):
      return self.exit_points

  def add_path(self, entry, exit, path, cost):
      if entry not in self.paths:
          self.paths[entry] = {}
      self.paths[entry][exit] = {'path': path, 'cost': cost}
```

The class chunk is defined, with the constructor containing, cells which stores the cells located in each chunk. Entry points is a dictionary where the keys are entry points into the chunk and the values are the directions of entry into the chunk which are which adjacency of the cell was used to enter the chunk. Exit points are similar to entry points as its another dictionary which stores exit points and a corresponding direction. Paths is a nested dictionary which stores paths between entry and exit points within the chunk, along with their costs.

```
  def cell_to_chunk(x, y, chunk_size=3):
    return (x // chunk_size, y // chunk_size)

  def initialize_chunks(max_x, max_y, chunk_size=3):
    chunk_map = {}
    for x in range(0, max_x + 1, chunk_size):
        for y in range(0, max_y + 1, chunk_size):
            chunk_coord = cell_to_chunk(x, y, chunk_size)
            chunk_map[chunk_coord] = Chunk()
    return chunk_map
```

cell_to_chunk is a method which converts a given cell into its corresponding chunk. It performs integer division on the input coordinates x and y by the chunk_size to determine which chunk the cell belongs to, the result is a tuple representing the chunk coordinates (chunk_x, chunk_y). Initialise chunks, initialises a map of chunks for a given maximum x and y coordinates and a chunk_size. An empty dictionary called chunk_map is created to store chunks. Next it iterates over all possible starting coordinates of each chunk within the given maximum coordinates using nested for loops, for each starting coordinate within the range of max_x and max_y it utilises cell_to_chunk to calculate the corresponding chunk coordinates. A new chunk object is created for each chunk coordinate and is then stored in the chunk map dictionary with the chunk coordinate as the key. Finally the function returns a populated map of chunks containing all initialised chunks.

```
def identify_and_store_entry_exit_points(chunk_map, right_of, left_of,
                                         is_above, is_underneath, chunk_size=3):
  for chunk_coord, chunk in chunk_map.items():
      for cell in chunk.cells:
          col, row = map(int, re.findall(r'\d+', cell))
          adjacent_cells = [
              (right_of.get(cell), 'right'),
              (left_of.get(cell), 'left'),
              (is_above.get(cell), 'up'),
              (is_underneath.get(cell), 'down')
          ]

          for adjacent_cell, direction in adjacent_cells:
              if adjacent_cell:
                  adj_col, adj_row = map(
                      int, re.findall(r'\d+', adjacent_cell))
                  if cell_to_chunk(row, col, chunk_size) !=
                        cell_to_chunk(adj_row, adj_col, chunk_size):
                      if direction == 'right' or direction == 'down':
                          chunk.add_exit_point((col, row), direction)
                      else:
                          chunk.add_entry_point((col, row), direction)
```

The above function, takes in a map containing chunks along with all the adjacency relationship dictionaries parsed through using the pddl parser. Each chunk in the map is iterated over, while using regex to extract the column and row coordinates from each cell within the chunk. For each of these cells, the adjacencies are retrieved and the adjacent cell is stored along with the direction relative to the current cell. Adjacent cells and directions are iterated over to extract column and row coordinates, then it checks if the chunk containing the current

cell is different from the chunk containing the adjacent cell. This comparison ensures that entry and exit points are only identified if they lead out of the current chunk.

```
  def filter_adjacencies_for_chunk(chunk, right_of, left_of,
                                   is_above, is_underneath):
    filtered = {
        "right_of": {},
        "left_of": {},
        "is_above": {},
        "is_underneath": {}
    }
    for cell in chunk.cells:
        if cell in right_of and right_of[cell] in chunk.cells:
            filtered["right_of"][cell] = right_of[cell]
        if cell in left_of and left_of[cell] in chunk.cells:
            filtered["left_of"][cell] = left_of[cell]
        if cell in is_above and is_above[cell] in chunk.cells:
            filtered["is_above"][cell] = is_above[cell]
        if cell in is_underneath and is_underneath[cell] in chunk.cells:
            filtered["is_underneath"][cell] = is_underneath[cell]
    return filtered
```

filter_adjacencies_for_chunk is designed to filter through the adjacency relationships to find all the adjacencies that apply to a specific chunk to allow pathfinding within a chunk. The function takes in the chunk along with all adjacency relationships and iterates through each cell in the given chunk, storing adjacencies in filtered adjacencies respective of direction. Once all chunks have been iterated through, the final filtered dictionary is returned.

```
def localise_cell(cell, offset_x, offset_y):
    parts = cell.split('x')[1].split('y')
    cell_x, cell_y = int(parts[0]), int(parts[1])

    local_x = cell_x - offset_x
    local_y = cell_y - offset_y

    return f"cellx{local_x}y{local_y}"

def localise(chunk, entry, exit, right_adj, left_adj, above_adj, under_adj):
    x, y = chunk
    offset_x = x * 3
    offset_y = y * 3

    entry_x, entry_y = entry
    exit_x, exit_y = exit

    entry_x = entry_x - offset_x
    entry_y = entry_y - offset_y
    local_entry = (entry_x, entry_y)

    exit_x = exit_x - offset_x
    exit_y = exit_y - offset_y
    local_exit = (exit_x, exit_y)
```

```
    localized_right_adjacency = {localise_cell(key, offset_x, offset_y):
      localise_cell(value, offset_x, offset_y) for
      key, value in right_adj.items()}
    localized_left_adjacency = {localise_cell(key, offset_x, offset_y):
      localise_cell(value, offset_x, offset_y) for
      key, value in left_adj.items()}
    localized_above_adjacency = {localise_cell(key, offset_x, offset_y):
      localise_cell(value, offset_x, offset_y) for
      key, value in above_adj.items()}
    localized_under_adjacency = {localise_cell(key, offset_x, offset_y):
      localise_cell(value, offset_x, offset_y) for
      key, value in under_adj.items()}

    return localized_right_adjacency, localized_left_adjacency,
      localized_above_adjacency, localized_under_adjacency,
      local_exit, local_entry
```

As for each chunk, there would be a lot of paths being calculated, in order to save on computation costs, if the entry, exit points and cell relationships were the same, then the paths could be reused rather than having to be recalculated. In order to achieve this, all chunks would have to be localised so that they can be compared to each other. localise_cell is designed to take in a cell along with an offset and utilise the offset to compute local coordinates which are then returned in object form. Offset values are calculated in localise, by multipying the chunk coordinates by the chunk size which is set to 3 in my implementation. Next all of the entry and exit points are localised by subtracting the offset coordinate from the cell coordinate. Localised adjacency relationships are then produced by utilising the localise_cell function to localise every relationship in the dictionary.

```
def process_adjacency(direction, cells):
    sorted_cells = sorted(cells.items(), key=lambda item: item[0])
    return [f"{direction}-{key}-{value}" for key, value in sorted_cells]

def generate_lookup_key(right_of, left_of, is_above, is_underneath,
                        entry_points, exit_points):
    adjacencies_str_parts = []
    adjacencies_str_parts.extend(process_adjacency("right_of", right_of))
    adjacencies_str_parts.extend(process_adjacency("left_of", left_of))
    adjacencies_str_parts.extend(process_adjacency("above", is_above))
    adjacencies_str_parts.extend(process_adjacency("underneath",
                                                   is_underneath))

    adjacencies_str = ",".join(adjacencies_str_parts)

    entry_points_str = ",".join(sorted([f"entry-{pt[0]}-{pt[1]}" for
                                        pt in entry_points]))
    exit_points_str = ",".join(sorted([f"exit-{pt[0]}-{pt[1]}" for
                                       pt in exit_points]))

    lookup_key = f"{adjacencies_str}|{entry_points_str}|{exit_points_str}"
    return lookup_key
```

Process_adjacency, is designed to take a direction and a dictionary cells which represents adjacency relationships between cells. The cells dictionary is sorted based on keys then the

sorted cell relationships are iterated over and formatted as a string in the format "direction-key-value", which is then returned. The reason for this function is to convert into a string which can be appended onto entry and exit points for lookup keys. generate_lookup_key, passes in the adjacencies to be formatted into strings and joins together adjacency strings with entry and exit point strings. Which is how the lookup key is reached.

```
def globalise_path(chunk_coord, path):
    globalised_path = []
    x, y = chunk_coord
    offset_x = x * 3
    offset_y = y * 3
    for cell in path:
        parts = cell.split('x')[1].split('y')
        cell_x, cell_y = int(parts[0]), int(parts[1])

        global_x = cell_x + offset_x
        global_y = cell_y + offset_y

        globalised_cell = f"cellx{global_x}y{global_y}"
        globalised_path.append(globalised_cell)

    return globalised_path

def localise_path(chunk_coord, path):
    localised_path = []
    x, y = chunk_coord
    offset_x = x * 3
    offset_y = y * 3
    for cell in path:
        parts = cell.split('x')[1].split('y')
        cell_x, cell_y = int(parts[0]), int(parts[1])

        global_x = cell_x - offset_x
        global_y = cell_y - offset_y

        localised_cell = f"cellx{global_x}y{global_y}"
        localised_path.append(localised_cell)

    return localised_path
```

In the event of a sucessful lookup, we would need to globalise the path back to how it originally was before being localised and stored. The offset is calculated again by multiplying by chunk size. Next each cell in the path is iterated over and the cell integers are extracted to which the offsets are added back to and then reconverted back into object form to be appended onto the globalised path. localise_path works the same way but instead of adding the offset to globalise, instead the offset is subtracted from the cell coordinates to localise the path.

```
def calculate_paths_for_chunk(chunk, chunk_coord, right_of, left_of,
                              is_above, is_underneath):
    filtered_adjacencies = filter_adjacencies_for_chunk(chunk, right_of,
                                        left_of, is_above, is_underneath)
    entry_points = chunk.get_entry_points()
    exit_points = chunk.get_exit_points()
```

```
    path_db = {}

    new_x, new_y = chunk_coord
    top_left_x_new = new_x * 3
    top_left_y_new = new_y * 3
    for entry in entry_points:
        for exit in exit_points:
            lright_adj, lleft_adj, labove_adj, lunder_adj, lexit, lentry =
              localise(chunk_coord, entry, exit,
                filtered_adjacencies["right_of"],
                filtered_adjacencies["left_of"],
                filtered_adjacencies["is_above"],
                filtered_adjacencies["is_underneath"])
            lookup_key = generate_lookup_key(lright_adj, lleft_adj,
                                labove_adj, lunder_adj, [lentry], [lexit])
            existing_path = path_db.get(lookup_key, None)
            if existing_path is not None:
                new_path = globalise_path(chunk_coord, existing_path)
                cost = len(new_path)
                chunk.add_path(entry, exit, new_path, cost)
            else:
                start_cell = f"cellx{entry[0]}y{entry[1]}"
                goal_cell = f"cellx{exit[0]}y{exit[1]}"
                path = aStar(filtered_adjacencies["right_of"],
                            filtered_adjacencies["left_of"],
                            filtered_adjacencies["is_above"],
                            filtered_adjacencies["is_underneath"],
                            start_cell, goal_cell, 3,
                            (top_left_x_new, top_left_y_new))
                if path is not None:
                    new_path = localise_path(chunk_coord, path)
                    path_db[lookup_key] = new_path
                    cost = len(new_path)
                    chunk.add_path(entry, exit, path, cost)
```

calculate_paths_for_chunk is designed to calculate the paths from each entry to exit point within each chunk based on the filtered adjacencies for that chunk. First the adjacencies are filtered to only include the adjacencies within the given chunk. Next entry and exit points are extracted from the chunk. path_db is utilised to store the lookup keys and paths. Each combination of entry and exit points is iterated over and chunk specific adjacencies are retrieved using localise. The lookup key is also generated using the chunk specific localised adjacencies and localised entry and exit points. Next existing paths are searched for with the look up key, and if there is a path, it is globalised using the current chunk coordinates. Once globalised, the path is added onto the chunk paths. If the lookup key did not find any corresponding path, the old aStar function from the bespoke planner is used to find the path between the exit and entry points. If there is a resulting path, it is stored using the lookup key and then added onto the chunk specific paths.

```
def create_chunk_graph(chunk_map):
    graph = {}
    for chunk_coord, chunk in chunk_map.items():
        graph[chunk_coord] = {}
        for entry, exits in chunk.paths.items():
```

```
            graph[chunk_coord][entry] = {}
            for exit, details in exits.items():
                path = details['path']
                cost = details['cost']
                direction = chunk.entry_points.get(entry)
                graph[chunk_coord][entry][exit] =
                      {'path': path, 'cost': cost, 'direction': direction}
    return graph
```

The purpose of create_chunk_graph is to construct a graph representing the connections between chunks and their paths, with each chunk's coordinates as keys and dictionaries representing paths from entry to exit points as values, including the associated path, cost, and direction of entry. This is so it can be utilised during the search later when chunks are being connected so all the required information is located in the graph.

```
  def adjacent_cell_and_chunk(path, direction, chunk):
    cell = path[0]
    match = re.match(r'cellx(\d+)y(\d+)', cell)

    cell_x, cell_y = map(int, match.groups())
    chunk_x, chunk_y = chunk

    if direction == 'up':
        return f"cellx{cell_x}y{cell_y - 1}", (chunk_x, chunk_y - 1)
    elif direction == 'down':
        return f"cellx{cell_x}y{cell_y + 1}", (chunk_x, chunk_y + 1)
    elif direction == 'left':
        return f"cellx{cell_x - 1}y{cell_y}", (chunk_x - 1, chunk_y)
    elif direction == 'right':
        return f"cellx{cell_x + 1}y{cell_y}", (chunk_x + 1, chunk_y)
    else:
        return None, None


def adjacent_path(adj_cell, adj_chunk, graph):
    match = re.match(r'cellx(\d+)y(\d+)', adj_cell)
    paths = []
    x, y = map(int, match.groups())
    cell = (x, y)
    chunk_data = graph.get(adj_chunk, {})

    for entry, exits in chunk_data.items():
        for exit, details in exits.items():
            if exit == cell:
                adj_info = adjacent_cell_and_chunk(details['path'],
                           details['direction'], adj_chunk)
                paths.append((details['path'], adj_info,
                              details['direction']))

    return paths
```

During the search, these 2 functions were created to assist with finding the adjacent cell and chunk for a path, utiling direction and current chunk. And adjacent_path which assists

in providing information about the adajacent cell and its corresponding chunk coordinates based on the given adjacent cell and chunk. It iterates over the paths in the chunk graph to find the one where the exit matches the given adjacent cell. For each matching pair, it uses the adjacent_cell_and_chunk function to get the adjacent cell and chunk coordinates and appends them to the paths list. Finally, a list of tuples containing the path, adjacent cell and chunk coordinates is returned.

```
def backtrack(graph):
  max_chunk_coord = max(graph.keys(), key=lambda x: (x[0], x[1]))
  x, y = max_chunk_coord
  x = x // 2 + 1
  y = y // 2 + 1
  start = (x, y)

  start_paths = []
  queue = deque(start_paths)
  all_paths = []
  complete_paths = []

  for entry, exits in graph[start].items():
      for exit, details in exits.items():
          current_path_info = []
          direction = details['direction']
          path = details['path']
          reversed_path = path[::-1]
          direction = details['direction']
          start_paths.append((start, reversed_path, direction))
          queue = deque(start_paths)

  while queue:
      chunk, path, direction = queue.popleft()
      current_step = path[-1]
      current_chunk = chunk

      # Check if the current step is the goal
      if current_chunk == (0, 0):
          final_cell, final_chunk = adjacent_cell_and_chunk(path[::-1],
                                    direction, chunk)

          if final_cell:
              adjacent = adjacent_path(final_cell, current_chunk, graph)
              if adjacent:
                  for adj_path, (new_adj_cell, new_adj_chunk),
                                adj_direction in adjacent:
                      path = path + adj_path[::-1]

          all_paths.append(path)
          continue

      adj_cell, chunknew = adjacent_cell_and_chunk(path[::-1],
                                                   direction, chunk)
      if chunk == (start):
          chunk = chunknew
      adj_chunk = chunk
```

```
        if adj_cell is None or adj_chunk is None:
            continue

        adjacent = adjacent_path(adj_cell, adj_chunk, graph)

        for adj_path, adj_info, adj_direction in adjacent:
            adj_paths = [adj_path for adj_path, _, _ in adjacent]
            adj_infos = [adj_info for _, adj_info, _ in adjacent]
            adj_direction = [adj_direction for _, _, adj_direction in
                             adjacent]
            new_adj_cell, current_chunk = adj_infos[0]
            for adj_path, (new_adj_cell, new_adj_chunk), adj_direction in
                            adjacent:
                new_path = path + adj_path[::-1]
                queue.append((new_adj_chunk, new_path, adj_direction))
    for path in all_paths:
        if path:
            complete_paths.append(path)

    return complete_paths
```

This is the main backtrack function which works by selecting a chunk towards the middle of the maze and computing all possible paths through a breadth first search from the chosen chunk towards the top left chunk. Effectively replicating how pattern databases for rubik's cubes are created, by backtracking from the solution and precomputing paths so that if that pattern is encountered, a solution can be found more efficiently. Once all paths are found, they are returned in complete_paths, which is then stored in a pickle file ready to be deserialised and utilised for finding solutions.

```
  def bfs(graph, all_paths):
    max_chunk_coord = max(graph.keys(), key=lambda x: (x[0], x[1]))
    start = max_chunk_coord

    start_paths = []
    queue = deque(start_paths)
    all_paths = []

    for entry, exits in graph[start].items():
        for exit, details in exits.items():
            current_path_info = []
            direction = details['direction']
            path = details['path']
            reversed_path = path[::-1]
            direction = details['direction']
            start_paths.append((start, reversed_path, direction))
            queue = deque(start_paths)

    while queue:
        chunk, path, direction = queue.popleft()
        current_step = path[-1]
        current_chunk = chunk

        target = target_path(all_paths, current_step)
```

```
        if target is not None:
            return path + target

        # Check if the current step is the goal
        if current_chunk == (0, 0):
            final_cell, final_chunk = adjacent_cell_and_chunk(path[::-1],
                                                              direction, chunk)

            if final_cell:
                adjacent = adjacent_path(final_cell, current_chunk, graph)
                if adjacent:
                    for adj_path, (new_adj_cell, new_adj_chunk),
                                    adj_direction in adjacent:
                        path = path + adj_path[::-1]

            all_paths.append(path)
            continue

        adj_cell, chunknew = adjacent_cell_and_chunk(path[::-1],
                                             direction, chunk)
        if chunk == (start):
            chunk = chunknew
        adj_chunk = chunk
        if adj_cell is None or adj_chunk is None:
            continue

        adjacent = adjacent_path(adj_cell, adj_chunk, graph)

        for adj_path, adj_info, adj_direction in adjacent:
            adj_paths = [adj_path for adj_path, _, _ in adjacent]
            adj_infos = [adj_info for _, adj_info, _ in adjacent]
            adj_direction = [adj_direction for _, _, adj_direction in
                             adjacent]
            new_adj_cell, current_chunk = adj_infos[0]
            for adj_path, (new_adj_cell, new_adj_chunk),
                               adj_direction in adjacent:
                new_path = path + adj_path[::-1]
                queue.append((new_adj_chunk, new_path, adj_direction))

    return "Path not found"

def target_path(paths, target):
    # Search for complete paths that contain the target anywhere within them
    target_paths = [path for path in paths if
                    target in path and path[-1] == "cellx0y0"]

    if target_paths:
        # extract the part of the path from the target onwards
        shortest_path_containing_target = min(target_paths, key=len)
        target_index = shortest_path_containing_target.index(target)

        # return the part of the path from the target onwards
        path_from_target = shortest_path_containing_target[target_index:]
        return path_from_target
```

```
    else:
        return None

filename = 'PDDL_parser/paths.pkl'

with open(filename, 'rb') as file:
    all_paths = pickle.load(file)

solution = bfs(graph, all_paths)

print(solution)
```

The above is the code used to find the path from the top left of the maze to the bottom right. The same breadth first search approach is utilised again, however this time around, if the end of one of the found paths is the start of a path in the database, both paths will be connected and then returned, which saves the algorithm having to calculate a lot more paths which can be intensive on resources.

4.13 Chunking and Landmark Heuristic

```
def Identify_Landmarks(graph):
    landmarks = set()
    final_landmarks = set()
    paths_to_landmark = []
    adjacent_paths = []
    path_to_goal = []
    precomputed_paths = []

    for entry, exits in graph[(0, 0)].items():
        for exit, details in exits.items():
            path = details['path']
            reversed_path = path[::-1]
            exit_points = path[-1]
            landmarks.add(exit_points)
            path_to_goal.append(path)

    for entry, exits in graph[(0, 1)].items():
        for exit, details in exits.items():
            path = details['path']
            reversed_path = path[::-1]
            exit_points = path[0]
            paths_to_landmark.append(path)

    for entry, exits in graph[(1, 0)].items():
        for exit, details in exits.items():
            path = details['path']
            reversed_path = path[::-1]
            exit_points = path[0]
            paths_to_landmark.append(path)

    for path in paths_to_landmark:
```

```
        first_cell = path[0]  # Get the first cell of the path
        for cell in landmarks:
            if is_adjacent(first_cell, cell):
                adjacent_paths.append(path)

    path_groups = defaultdict(list)
    for path in adjacent_paths:
        start, end = path[0], path[-1]
        path_groups[(start, end)].append(path)

    shortest_paths = []
    for start_end, group_paths in path_groups.items():
        shortest = min(group_paths, key=len)
        shortest_paths.append(shortest)

    for path in shortest_paths:
        for paths in path_to_goal:
            if is_adjacent(path[0], paths[-1]):
                precomputed_paths.append(paths+path)

    for path in precomputed_paths:
        final_landmarks.add(path[-1])

    return final_landmarks, precomputed_paths
```

Another possible approach through chunking is using landmark heuristics and this approach above is influenced by LAMA, however instead of utilising A* to connect the chunks and paths, the same breadth first search from the pattern database creation file is utilised here, however it is adjusted slightly and directed rather than undirected using the landmark as a heuristic. In order to find landmarks within the maze, we take the goal cell, so in this example the goal is the top left of the maze or (0, 0) and we get the exit points of the chunk that contains the goal and the cell that is adjacent while providing a logical path into the exit point. These adjacent to the chunk, cells are utilised as the landmark and are used in a manhattan heuristic function to guide the algorithm towards the viable paths towards the goal.

Chapter 5: **Review**

5.1 Professional Issues

The impact on society is an important consideration in the realm of computing technologies and their creation. The development and application of AI based projects inherently, encounters a variety of professional issues which are not limited to academic issues, but extend towards society and integration into the real world.

Amazon is leading the change from having people fulfilling orders in their e-commerce fulfilment centres to utilising robots, in efforts to cut down on operational costs as "according to the German centre for Digital Technology and Management, having humans pick the items accounts for 55% of logistics companies' warehousing costs" [22]. The increased use of robotics, has in return reduced operating costs to 20% [22], due to the reduced time for each item in the warehouse to be picked and routed along with not having to pay wages to the employees who have been replaced. This sparks a societal issue where during the development of these robots, the main consideration was to cut costs and provide faster service to keep up with the growing demand, for which the side affect is a hit to the job market, especially jobs that are susceptible to automation, to which "around 47% of total US employment is in the high risk category" [23]. The impact of unemployment can be disastorous for a society as it can lead to a cycle of layoffs due to decrease in consumer spending affecting businesses. To account for this issue, the development of this project has not been based around a sector where there is possibily for jobs to be replaced, rather more of an analysis of the current AI solving techniques to provide a deeper understanding of their strengths and limitations. On the other hand, he indirect threat of layoffs still remains due to the nature of the types of problems being solved and their applicability into the real-world.

The issue of safety and reliably is also a concern within this project, as within the field of AI, if these algorithms were to be implemented without extended testing, there are risks of inaccurate solutions or failures in critical environments. The nature of both Sudoku and maze-solving/pathfinding problems offer a controlled environment to be able to study the limitations as well as the reliability and safety concerns associated with AI algorithms compared to if similar AI solving techniques are are be applied to real-world scenarios such as autonomous driving as a pathfinding problem, the risks are much higher. Testing the algorithms within controlled safe environments not only assists with identifying potential flaws but acts as a pillar to understanding the reliability of the presented AI solving techniques. The importance of extensive validation and testing before deployment in applications with potential safety implications is emphasised through the project as the problems are built up from smaller more manageable problems where flaws are easier to catch, however since there are more variables involved in real-world scenarios, it is of utmost importance to extensively test the techniques in controlled environments with minimal risk.

Plagiarism holds significant importance as a professional issue within the creation and application of computer technologies. During the research of different AI solving techniques to solve Sudoku puzzles and maze pathfinding problems, it was necessary to explore the current techniques that are being used along with frameworks and libraries. The handling of these resources in an ethical manner is of importance while citing correctly to acknowledge the original developers of their code and techniques. This is in efforts to honor the techniques utilised but also forward the culture of transparency and academic integrity within the computing community. The issue of Plagiarism still stands as not all resources are available in a citable form, such as an article. Regardless, Plagiarism remains a constant consideration as during

the development of the project, it is important to keep track of every research point and provide credit to resources where I gained the necessary knowledge in order to proceed with the project. Failure to cite correctly can result in more issues such as intellectual property disputes, and risking the integrity of the project.

Management of the project in terms of costing of time and resources has been a significant issue from the start of the project. The original goal of the project was to compare constraint satisfaction problem solving techniques to traditional planning techniques, using Sudoku as the domain while aiming to build upon them to reach more efficient solutions. This goal however was found not to be feasible due to limitations in traditional planning approaches. It was then decided upon with my supervisor, to alter the path of the project to focus on a problem that is more suited towards planning so that we are able to have a comparison of how the nature of a problem requires different approaches to solving it. Furthermore within the first timeline, the time required to research and apply a solution to Sudoku using planning, extended further than the expectation due to an incorrect approach of tackling the full problem instead of slowly building up and finding flaws early. After the revision of the project, and the slow build up, development strategy, the progress of the project improved with the only limitation being time constraints in implementing more heuristic functions and different CSP techniques.

5.2 Timeline comparision

In the original timeline there were quite a few alterations made as seen in the diary in Appendix A.

The initial phases focused heavily on understanding and applying PDDL, which was consistent with the original plan. However, as the project developed, it became clear that certain parts needed to be revisited, particularly the feasibility of using PDDL for the classic 9x9 Sudoku problem and the introduction of incremental development to find the source of the problem.

Another aspect was overestimating the amount of time required for researching the theory to gain enough understanding to start development as, working on the simple 3x3 puzzle resulted in a more productive development phase and understanding of PDDL which theory alone was not able to achieve.

The addition of a python script was another change required as it was quick to identify the lack of unit testing in PDDL and as a result, there had to be a different method to validify the models created before being able to analyse them. This was a necessary step as if there was a lack of testing, particularly with large outputs in the console, any human error while reading the output could cause implications further down the project.

For the second set of weeks, after the revision to the path of the project, the timeline was managed well, as instead of going for full solutions and spending too much time debugging, solutions were build up step by step and tested utilising TDD techniques. This ensured ample time to reach aims and objectives.

However one problem encountered was the output of the generated maze from pyamaze, as it was not mentioned in the documentation that the adjacencies were given in the form (y, x) which set me off track by a week, effectively reducing the remaining time, that could have been spent exploring Hmax, LMcut, IPDB, and LAMA based approaches.

5.3 Key findings

5.3.1 Comparison of 3x3 implementation

Both the fast downward planner and the z3 solver solved the problem very quickly with the fast downward planner taking 0.193 seconds and the z3 solver taking 0.082 seconds which makes the z3 solver 53.5% faster for this problem. While both times are fast, this difference in speed could entail significant benefits in environments where speed is key.

5.3.2 Comparison of 9x9 implementation

The fast downward planner unfortunately was not able to find a plan before timing out which is due to the complexity increase being too high for current algorithms used in planners such as fast downward. The closest implementation to a 9x9 that the planner was able to solve are certain configurations of a 6x9 with a 11 second time to solution using the fast downward planner. In comparison to this, the z3 solver was able to find a valid solution to the problem and at a fast pace too, taking 0.04 seconds to find a solution which makes it a staggering 27400% speed increase over a more complex problem.

Why is this? Sudoku is fundamentally a CSP which requires logical deduction and has seen huge improvements in solving using backtracking algorithms. Current off the shelf planners use algorithms which do not make much use of backtracking. SMT solvers however do make use of backtracking algorithms and we can see the impact in the search times above.

5.3.3 Table of Findings

Table 5.1: Performance Comparison of Different Methods

Method	Worst Case	Average Case
MazeCSP	67.2% 186.43s	66.38% 183.42s
aStar	67.4% 0.07s	66.64% 0.066s
pdb	66.6% 4.43s	66.48% 4.402s
landmark_bfs	66.8% 33.67s	66.7% 32.513s

5.3.4 Discussion of results

To reach the table 5.1, a 30x30 randomised maze was used to test performance. This maze was generated using pyamaze at a loopPercent value of 21%. The maze was then parsed into each of these solving techniques and solved 100 times to produce the results. For each solution, the % of memory usage was recorded along with how long it took to reach a solution.

For MazeCSP, this is the CSP based solver, The technique was only tested 10 times due to the significant time required to find a solution. The reason for the long run time is the iterative deepening aspect of the solution, designed to provide an optimal path rather than greedy. However specifying a path length for a randomised maze would bring more time based issues,

so that algorithm ranks last in terms of time taken, however is not last for memory usage, in fact on average, the MazeCSP ranks best algorithm in terms of the RAM usage percentage. In problems where memory is the priority out of memory and speed, using a CSP may not be a bad idea.

aStar has been the best performing algorithm, scoring an average time taken of 0.066s which is significantly faster than any other algorithm that has been tested in this study. In terms of memory usage, all of the algorithms are too close in memory usage to be able to make any conclusive findings, this may be because of a flawed memory function within psutil which is the library used to get the memory usage. The ability of aStar to balance the shortest path with the lowest cost, due to its heuristic nature, enables it to navigate through the 30x30 randomised maze with remarkable speed.

Now onto the chunk based approaches, the first of which being the pattern database, which still is able to solve the maze on an average of 4.402 seconds it significantly outpaces the landmark based approach which goes to show the value of precomputed patterns and pattern usage to solve pathfinding problems. Its average memory ranks second best which may be due to the precomputed paths saving on memory usage. Overall this is a moderate approach in both time taken and memory usage, offering a reasonable compromise for situations where neither time nor memory can be prioritised to the exclusion of the other. However it must be noted that there is a reliance on precomputed costs and pattern identification.

The last approach measured is the landmark breadth first search approach, which performed the worst out of the chunking techniques, however this could have been because of poor choices or landmarks, or search type, as it was an approach influenced by LAMA without using A*. The landmark approach also on average used the most memory, making it less ideal for resource-constrained environments.

In conclusion, The aStar algorithm would be my pick for best performer all round, this may be due to its simplicity, and nature, performing really well for pathfinding problems with horizontal and vertical movements. The chunk approaches may have performed worse, due to the unrequired computational overhead from splitting a maze into chunks along with the paths in between. Another thing to note is that the chunk based algorithms sometimes could not find a path, compared to aStar which always found a path and MazeCSP which dependent on the maximum depth also found a path.

5.4 Future steps

Delving into the refinement of the algorithms themselves to see how time and memory could be improved, which could involve a mixture of different heuristics for aStar, alternative data structures for pattern databases,, or more efficient state management techniques for MazeCSP and the landmark based breadth first search. Further tweaking with parameters and new strategies would be useful to reduce computational overload.

Measuring the scalability of each algorithm as some of the algorithms performed very close to aStar for short pathfinding problems, however when the path grew longer, the algorithms drifted further apart in terms of performance. Being able to understand what aspects affect scalability, may provide insight in further refinement of the algorithms. This would involve testing with larget sizes and more intricate structures to understand how performance metrics change. Robustness to change to parameters could also be revealed through the scalability analysis.

Combining current algorithms together to try an make up for limitations, like combining A star with the landmark heuristic to measure the performance of LAMA, could provide further insight into how each algorithm affects a search, which would allow for picking algorithms based on a more narrow requirement. Experimenting with hybrid approaches may uncover synergies that leverage the unique advantages of each method.

Further testing must be done on the memory usage for each algorithm as for the current method, using psutil, the results are too close to make decisions for based on memory alone. In order to gain a clearer picture of memory usage, exploring more sophisticated tools, or methodologies for measuring the memory usage of algorithms. This could be done in combination with different metrics such as CPU usage.

Exploring how incorporating machine learning techniques can be integrated with the algorithms to improve performance, as they could be used to predict optimal parameters based on the characteristics of the problem. Machine learning models, through their ability to learn from data, offers potenitally enhanced decision making within the search process. This integration could lead to more intelligent, self optimising algorithms that adjust their parameters in real time.

Exploring further into the heuristics used within the Fast Downward planner, such as LM-Cut, Hmax, and LAMA, could have provided extra knowledge into how they affect search algorithms and potenitally influence the overall efficiency and effectiveness of my implementations solving complex puzzles. Each of these heuristics represents a different approach to estimating the cost to reach the goal from a given state, by conducting a deep dive into how LMCut, Hmax, and LAMA impact the search process, insights could be gained into their respective strengths and weaknesses in various problem contexts.

Experimenting with different chunk sizes to measure the impact on performance, could provide further insight into the optimal number of chunks for a given sized maze.

In conclusion, this project has provided me great knowledge in the realm of AI solving techniques and what it takes to find new approaches, to tackle complex problems. Through the design, implementation and evaluation of various algorithms, this project has shed light on the complexity of AI planning and constraint satisfaction problems. One key take away from this project is the understanding that no single algorithm offers a one size fits all solution, as the pathfinding approach did not work at all for the Sudoku problem and the constraint satisfaction problem approach performed the worst in a pathfinding domain.

Bibliography

[1] H. Geffner, "Perspectives on artificial intelligence planning," in *AAAI/IAAI*, 2002.

[2] T. Conroy, "Will we ever run out of sudoku puzzles?" Encyclopedia Britannica, 2016, accessed 4 October 2023. [Online]. Available: https://www.britannica.com/story/will-we-ever-run-out-of-sudoku-puzzles

[3] J. W. Kim and S. K. Kim, "Fitness switching genetic algorithm for solving combinatorial optimization problems with rare feasible solutions," *The Journal of Supercomputing*, vol. 72, pp. 3549–3571, 2016.

[4] D. S. Weld, "Recent advances in ai planning," *AI magazine*, vol. 20, no. 2, pp. 93–93, 1999.

[5] D. E. Wilkins, "Can ai planners solve practical problems?" *Computational intelligence*, vol. 6, no. 4, pp. 232–246, 1990.

[6] D. B. Lenat, "The nature of heuristics," *Artificial intelligence*, vol. 19, no. 2, pp. 189–249, 1982.

[7] W. Zhang, *State-space search: Algorithms, complexity, extensions, and applications*. Springer Science & Business Media, 1999.

[8] D. Foead, A. Ghifari, M. B. Kusuma, N. Hanafiah, and E. Gunawan, "A systematic literature review of a* pathfinding," *Procedia Computer Science*, vol. 179, pp. 507–514, 2021.

[9] C. Aeronautiques, A. Howe, C. Knoblock, I. D. McDermott, A. Ram, M. Veloso, D. Weld, D. W. SRI, A. Barrett, D. Christianson *et al.*, "Pddl— the planning domain definition language," *Technical Report, Tech. Rep.*, 1998.

[10] B. Felgenhauer and F. Jarvis, "Mathematics of sudoku i," *Mathematical Spectrum*, vol. 39, no. 1, pp. 15–22, 2006.

[11] S. C. Brailsford, C. N. Potts, and B. M. Smith, "Constraint satisfaction problems: Algorithms and applications," *European journal of operational research*, vol. 119, no. 3, pp. 557–581, 1999.

[12] V. Kumar, "Algorithms for constraint-satisfaction problems: A survey," *AI magazine*, vol. 13, no. 1, pp. 32–32, 1992.

[13] A. Sadavare and R. Kulkarni, "A review of application of graph theory for network," *International Journal of Computer Science and Information Technologies*, vol. 3, no. 6, pp. 5296–5300, 2012.

[14] A. Kejriwal and A. Temrikar, "Graph theory and dijkstra's algorithm: A solution for mumbai's best buses," *The International Journal of Engineering and Science (IJES)*, vol. 8, no. 10, pp. 40–47, 2019.

[15] X. Cui and H. Shi, "A*-based pathfinding in modern computer games," *International Journal of Computer Science and Network Security*, vol. 11, no. 1, pp. 125–130, 2011.

[16] H. A. Simon, "Heuristics in judgment and decision making," *Annual Review of Psychology*, vol. 41, pp. 1–19, 1990.

[17] A. E. Sloan, "Shopping the perimeter of the store," *Food Technology*, January 2013.

[18] M. H. Romanycia and F. J. Pelletier, "What is a heuristic?" *Computational Intelligence*, vol. 1, pp. 47–58, 1985, received February 5, 1985; Revision accepted March 29, 1985.

[19] O. Kim and M. Sridharan, "A landmark-like heuristic for planning," *arXiv preprint arXiv:2403.07510*, March 2024.

[20] A. Felner, R. E. Korf, and S. Hanan, "Additive pattern database heuristics," *Journal of Artificial Intelligence Research*, vol. 22, pp. 279–318, 2004.

[21] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2010.

[22] R. Bogue, "Growth in e-commerce boosts innovation in the warehouse robot market," *Industrial Robot: An International Journal*, vol. 43, no. 6, 2016.

[23] C. B. Frey and M. A. Osborne, "The future of employment: How susceptible are jobs to computerisation?" *Technological Forecasting & Social Change*, vol. 114, pp. 254–280, 2017.

Appendix A: **Diary**

Week 4: October 20, 2023

Installed PDDL extension for VSCode, ran a couple examples in the github provided to see outputs and familiarise myself with running them.

Week 5: October 26, 2023

Researched use of test cases in PDDL and tried figuring out if it was feasible to use them in my project.

Week 6: October 30, 2023

Researched Constraint Satisfaction Problems through Artificial Intelligence Principles and Techniques 5. CSP, Part I: Basics and Naive Search

Week 7: November 2, 2023

Ran Z3 solver on an altered file for Sudoku trying a 3x3 for proof of concept and had to figure out how to use z3. Got it to run successfully.

Week 8: November 11, 2023

Ran Z3 solver on classic 9x9 grid, noted it takes some time to find a solution.

Week 9: November 13, 2023

Tried multiple different approaches of using the fast-downward planner, finally got it working

Week 9: November 14, 2023

Tried a different implementation of the 3x3 PDDL model, finally got it working

Week: 11: November 30, 2023

Decided on splitting up the code to see the breaking point as running the full implementation timed out the planner resulting in no plans

tests on 1 row and 3 rows are successful with 1 row taking 0.35 seconds on average to solve and the 3 rows implementation taking 0.55 seconds on average by the online solver.

testing 5x9 shows the first significant leap in time going from 1 second to 4.2 seconds on average to find a plan

Week 11: December 1, 2023

6x9 found as the breaking point

Week 12: December 4, 2023

Decided on moving onto problems more suited to modelling which do not require backtracking as this is something that off the shelf planners lack.

Week 12: December 7, 2023

Wanted to improve my testing methodology, decided to create a python script that can count

how many times a digit appears in each row, column and subgrid

added functionality to read files, and return valid if the plan is valid

Term 2: Week 2 January 18, 2024

Decided on a switch in project direction to pathfinding using the experience gained through modelling sudoku

January 19, 2024

Planning on modelling a 4x4 grid to model a pathfinding problem going from A to B for an agent, will be done in pddl and csp to compare the implementations.

January 21, 2024

Researched A* and how it could be implemented into my planning solution, along with backtracking with forward propagation for the csp solution as this shouldn't be too complex and can provide insight into the strenghts and weaknesses.

Term 2: Week 5 February 12, 2024

Connected fast downward planner to pddl vscode extension allowing for problem files to be used as test cases

February 17, 2024

All actions for PDDL pathfinding model complete, start creating problem files.

Term 2: Week 6 February 22, 2024

Sample problem files along with tests complete, start researching different implementations of csp version of the pathfinding problem.

February 25, 2024

Development for constraint satisfaction version of the pathfinding problem, Create variables, create domain, create constraints

Term 2: Week 7 February 26, 2024

Start thinking about how to develop in python using test driven development

February 29, 2024

Experiment with unittest

March 2, 2024

Research A*, start development using python

Develop manhattan distance heuristic as currently vertical and horizontal movements are allowed so the estimate is more realistic

Develop tests for the heuristic function

Term 2: Week 8 March 6, 2024

Research CSP methods for backtracking, constraints, Keep track of AIMA fig 6.5

Successor_generator

March 8, 2024

Look into parsing files

regex?

Term 2: Week 9 March 14, 2024

Noticed flaw in problem definition, double digits not accounted for

March 17, 2024

Parser needs accounting for new grid structure

Term 2: Week 10 March 22, 2024

Utilise adjacencies to get the max grid size

Term 2: Week 11 March 26, 2024

Test backtrack in csp

March 29, 2024

Maze generation possibilty, pyamaze

Term 2: Week 12 April 2, 2024

csp: new constraint for cell walls, update consistent and goal

April 6, 2024

Look into new heuristic possibilities/solving techniques

Term 2: Week 13 April 9, 2024

Chunking the maze, ipdb? hmax? lmcut? lama?

April 11, 2024

Pickle for storing paths, landmarks?

A.1 Literature Review

[1] Geffner's work encompasses foundational concepts and advanced strategies in AI planning, which serves as a theoretical base for my own developed implementations and understanding of these techniques. The work is a great starting point for researching planning and can be looked back upon to refine pathfinding techniques.

[2] Conroyś article provides great insight into how Sudoku is a Combinatorial problem and defining combinatorial problems. The article provides a more in-depth look at the complexity and the mathematical underpinnings of Sudoku, which is useful to the project as, during the development of a Sudoku solver, the workings of the problem are better understood.

[3] As my project deals with combinatorial optimisation, Kimś work on solving combinatorial problems can be beneficial when deciding on different techniques to compare and measure. Furthermore the efficiency and feasibility of solving complex problems with AI techniques is positively impacted by analysing the case studies and solutions.

[4] Danielś "Recent advances in AI" planning is very relevant to my project as I can gain an understanding of how AI planning has evolved from the past and the reasons for that. Allowing me to learn from these foundational concepts from 1999 as many modern methods are built upon these earlier advancements.

[5] As my project is not purely theoretical and involves practical application, Davidś work, discusses early challenges and successes in applying AI planners to real-world problems, offering a comparison to the capabilities of current plnners and solvers. The historical perspective on the effectiveness of these planners in the practrical scenarios can serve as a benchmark while also highlighting how these planners have improved over time.