Universitat Politècnica de Catalunya
Universitat de Barcelona
Universitat Rovira i Virgili

Master in Artificial Intelligence

Planning and Approximate Reasoning

The Coffee Problem

Authors:
Emanuel Sanchez Aimar
Johannes Heidecke

November 2016

Contents

List of Figures

Listings

1 Introduction

1.1 Problem Description

This deliverable uses STRIPS (Stanford Research Institute Problem Solver) to solve the following linear planning problem:

> *There is a squared building composed by 36 offices, which are located in a matrix of 6 rows and 6 columns. From each office it is possible to move (horizontally or vertically) to the adjacent offices. The building has some coffee machines in some offices that can make 1, 2 or 3 cups of coffee at one time.*
>
> *The people working at the offices may ask for coffee and a robot called "Clooney" is in charge of serving the coffees required. Each office may ask for 1, 2 or 3 coffees but not more. The petitions of coffee are done all at early morning (just when work starts) so that the robot can plan the service procedure. Each petition has to be served in a single service.*
>
> *The goal is to serve all the drinks to all the offices in an efficient way (minimizing the travel inside the building, in order to not disturb the people working).*

1.2 Report Overview

2 Problem Analysis

To analyze this problem thoroughly, we describe the used predicates (subsection 2.1), the possible initial and goal states (subsection 2.2), the operators (subsection 2.3), the search space (subsection 2.4) and special cases (subsection 2.5).

2.1 Predicates

The used predicates are the following:

- `Robot-location(o)`: The robot is in office o. We chose a coordinate based representation of offices where each o is defined by its coordinates $[x, y]$.
- `Robot-free`: The robot does not have any coffee loaded.
- `Robot-loaded(n)`: The robot has n cups of coffee loaded with $n \in \{1, 2, 3\}$.
- `Petition(o, n)`: n cups of coffee are required in office o with $n \in \{1, 2, 3\}$.
- `Served(o)`: Office o has been served.
- `Machine(o, n)`: There is a coffee machine in office o that produces exactly n cups of coffee with $n \in \{1, 2, 3\}$.
- `Steps(x)`: The robot has moved x steps with $x \in \mathbb{N}_0$.

2.2 Intitial and Goal States

The **initial state** of any problem is composed as follows:

- **Required predicates**: *Robot-free*, *Steps(0)*, *Robot-location(o)* where *o* is a valid office location.
- **Optional predicates**: One or more *Petition(o,n)* with at most one petition for each office.
- **Conditionally required predicates**: For each $n^* \in \{1, 2, 3\}$ that occurs in a *Petition*(o, n^*), there has to be at least one *Machine(o, n)* with fitting $n = n^*$ in order to guarantee that the problem is solvable.

The **final state** of any problem is composed as follows:

- **Optional predicates**: A *Robot-location(o)* can be specified.
- **Conditionally required predicates**: For each o^* that occurs in a *Petition*(o^*, n), there must be exactly one *Served*(o^*) predicate in the final state.

2.3 Operators

This problem contains three operators that can be applied to reach the final state starting from the initial state. These operators are: $Move(o_1, o_2)$, $Make(o, n)$ and $Serve(o, n)$.

Move:

The operator $Move(o_1, o_2)$ allows the robot to move between offices and is set up in the following way:

- **Preconditions**: Robot-location(o_1), Steps(x).
- **Add**: Robot-location(o_2), Steps($x + distance(o_1, o2)$).
- **Delete**: Robot-location(o_1), Steps(x).

Make:

The operator $Make(o, n)$ allows the robot to make coffee in an office with a machine and is set up in the following way:

- **Preconditions**: Robot-location(o), Robot-free, Machine(o, n).
- **Add**: Robot-loaded(n).
- **Delete**: Robot-free.

Serve:

The operator $Serve(o, n)$ allows the robot to serve coffee and is set up in the following way:

- **Preconditions**: Robot-location(o), Robot-loaded(n), Petition(o, n).
- **Add**: Served(o), Robot-free.
- **Delete**: Robot-loaded(n), Petition(o, n).

2.4 Search Space

The search space of a problem is the subset of the state space that can be reached by applying the operators, starting at the initial state. The nodes in the search space correspond to states of the world, while the edges correspond to state transitions through operators.

To compute the size of the search space we have to consider how many states we can reach from a given initial state by only applying our operators. In order to compute the size of the search space we take the following considerations into account:

- The predicate Robot-location(o) is always present exactly one time in each state, since it is required in the initial state and no operator adds this predicate without deleting a different instantiation of it and vice versa. There are 36 different values for o (one for every office in the world.
- Robot-free and Robot-loaded(n) cannot be part of a state at the same time (see Make and Serve operators). Robot-loaded(n) can take three different values: $1, 2$ and 3. This leads to 4 different combinations: Robot-free, Robot-loaded(1), Robot-loaded(2) and Robot-loaded(3).
- The Petition(o, n) predicates are taken from the initial state and can then be removed via the Serve operator. Since each Petition predicate can either still be present or replaced with a Served predicate, the number of possible states is $2^{|P|}$ where $|P|$ is the number of petitions in the initial state.
- The predicate Steps(x) is only present once, but x can take a arbitrary integer value larger than 0.

The Steps(x) predicate makes the search space infinitely large. Since our goal is to both reach the final state while minimizing x, we will consider the search space ignoring x and then later twist our algorithm in a way to explore the search space with consideration to keeping x small.

Ignoring the Steps(x) predicate, our search space is of size $36 \cdot 4 \cdot 2^{|P|}$ with $|P|$ being the number of petitions in the initial state. The size of the search space grows exponentially with the number of petitions.

2.5 Special Cases

Some combinations of initial states and final states are not able to be solved. This is the case when the final state is not part of the search space. An example case for this scenario is a Petition(o, n_p) predicate that does not have a matching Machine(o, n_m) with $n_p = n_m$.

3 Planning Algorithm

3.1 STRIPS Algorithm

The implementation is based on the STRIPS algorithm which works in the following way:

The STRIPS is provided with an initial state, a final state (containing the goals of the problem) and a set of operators. We use a simplified version of STRIPS where the goal state cannot state predicates that need to be false - it can only specify a conjunction of predicates that need to be true in the end.

The STRIPS algorithm uses a stack that can contain single predicates, lists of predicates and operators to calculate a plan of operators that reach the final state starting from the initial state.

A pseudo-code example of the STRIPS algorithm can be found in listing 1.

Listing 1: The STRIPS algorithm

```
State initialState, finalState
State currentState = initialState;
List<Operator> operators;
Stack stack;
List<Operator> plan;

stack.push(finalState.predicateList);
for(predicate : finalState.predicateList):
    stack.push(predicate);

while(!stack.isEmpty()):
    element = stack.pop();

    case element instanceof Operator:
        currentState.apply(element);
        plan.add(element);

    case element instanceof List<Predicate>:
        for(predicate : element):
            if predicate not true in currentState:
                stack.push(predicate);

    case element instanceof Predicate:
        if element.isFullyInstantiated():
            if element not true in currentState:
                operator = findOperatorToResolve(element);
                stack.push(operator);
                stack.push(operator.preconditions);
                for(predicate : operator.preconditions):
                    stack.push(predicate);
            else:
                // Look for constant c to instantiate element.
                // Set c in entire stack.
                instantiate(element);

return plan;
```

3.2 Heuristics

In our specific implementation of the STRIPS algorithm to solve the coffee problem, we use two different heuristics to reduce the number of steps x that are needed to achieve the final state. These heuristics determine in which order the predicates belonging to a list of predicates get pushed onto the stack, and which constant is selected out of several options when instantiating the predicates.

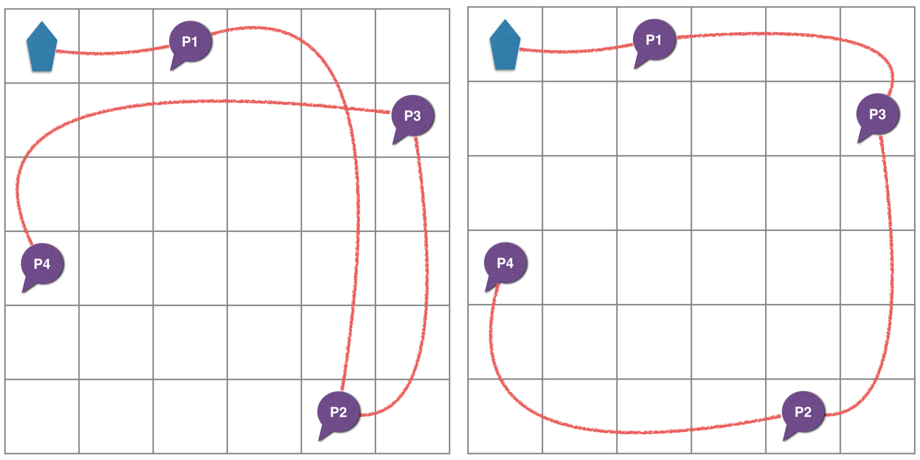
3.2.1 The Served-Heuristic

The Served-Heuristics determines in which order the robot processes the different petitions to serve the coffee. If the petitions are processed in a random order or simply based on their position in the description of the world, the robot is likely to make a lot of unnecessary steps. The optimal solution here belongs to the class of the Traveling Salesman Problem and is too complex to exactly compute for a large number of petitions.

In our implementation we resorted to the nearest-neighbor heuristic (NN-heuristic) to get a good solution for this problem with relatively little computational effort. The steps of this algorithm are the following:

1. Start at the beginning position of the robot.
2. Find shortest distance between current position and an unvisited Served predicate S^*.
3. Save S^* to sequence and set current position to S^*.
4. Mark S^* as visited.
5. If all Served predicates are visited, terminate
6. Go to step 2.

The difference between visiting petitions in their order (from P_1 to P_4) or choosing a path based on the NN-heuristic can be seen in figure 1. While the the path based on the petition number requires 21 steps, the NN-heuristic only requires 17 steps. Empirical results of the NN-heuristic can be found in section 6.



(a) 21 steps with petition number order (b) 17 steps with NN-heuristic

Figure 1: Petition Number Order vs. NN-Heuristic

3.2.2 The Machine-Heuristic

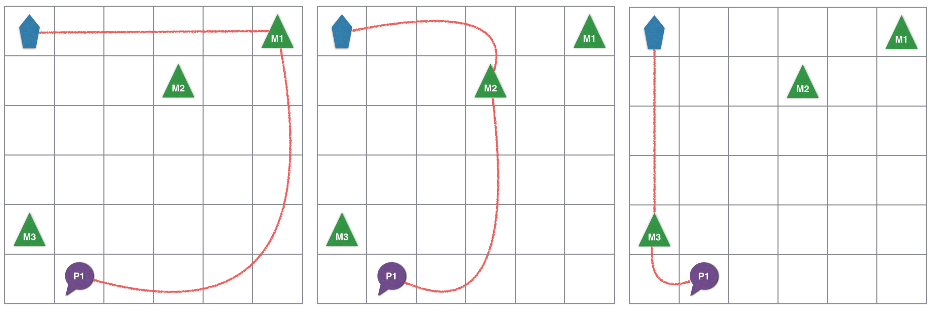
When planning to serve coffee for a specific petition, the planner has to find a suitable machine that can brew the fitting number of cups. In some worlds there might be more than one possible Machine predicate with a fitting n. The right choice of machine is focus of our second heuristic.

Without using a heuristic, STRIPS instantiates with an arbitrary compatible machine, which can lead to big detours (as shown in example (a) of figure 2). A first improvement of the number of steps can be achieved by choosing the machine that is closest to the current position of the robot, as in example (b) of figure 2.

In our heuristic we don't only consider the distance between current position and the machine, but also the distance from that machine to the next petition. For the robot position r, the machine position m and the petition location p, the distance is calculated as:

$$heurDist(m) = dist(r, m) + dist(m, p)$$

We then choose the machine m with the smallest $heurDist(m)$.



(a) First machine (b) Nearest machine (c) Best total distance

Figure 2: Comparison of Machine Selection

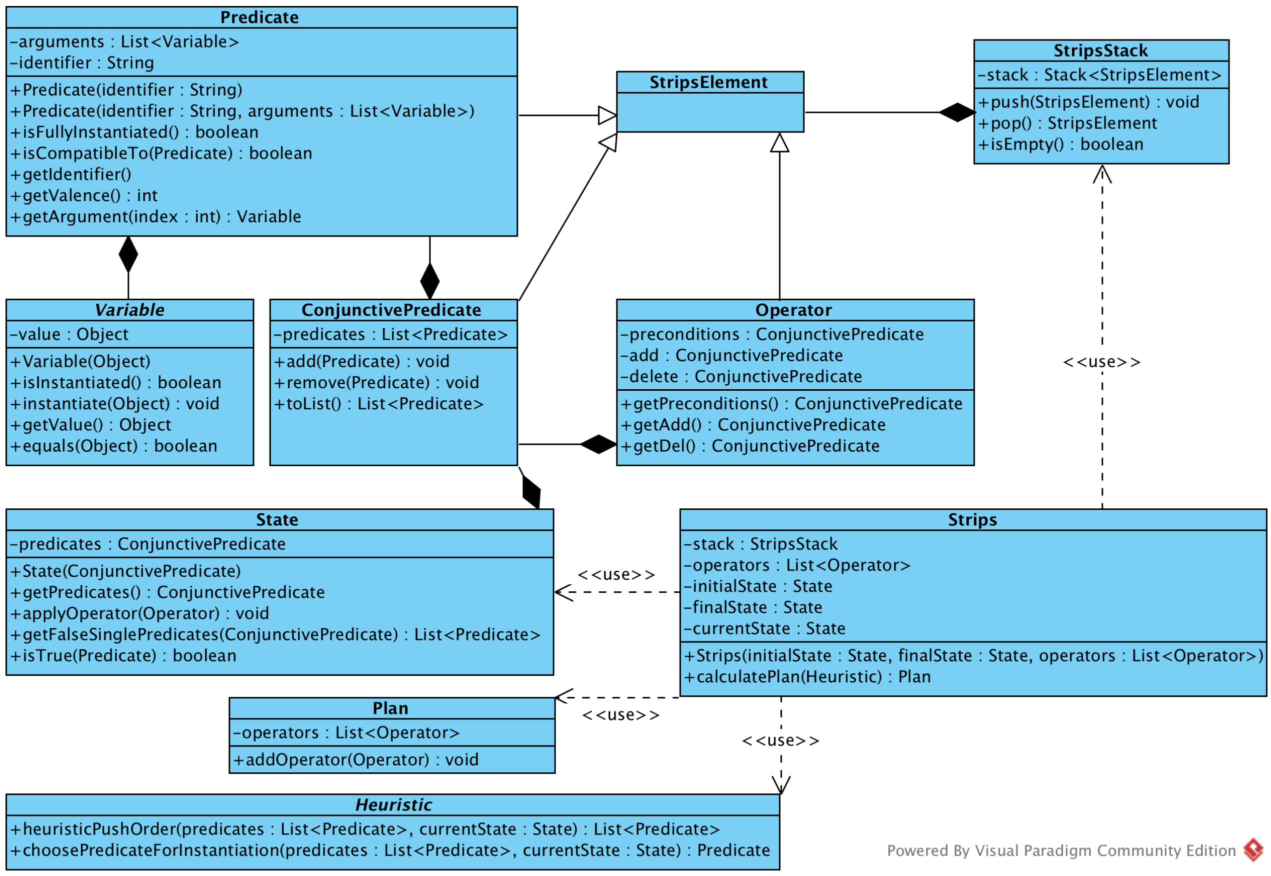
Empirical results of this heuristic can be

4 Implementation

4.1 General Structure

We implemented the STRIPS algorithm in a way that it can generically be used for different problems. Based on this structure we implemented the specific problem and a domain specific heuristic.

Figure 4 visualizes the following explanations in a UML class diagram. A larger image of this diagram is attached to this report (see appendix A).



Predicate
-arguments : List<Variable>
-identifier : String
+Predicate(identifier : String)
+Predicate(identifier : String, arguments : List<Variable>)
+isFullyInstantiated() : boolean
+isCompatibleTo(Predicate) : boolean
+getIdentifier()
+getValence() : int
+getArgument(index : int) : Variable
StripsElement
StripsStack
-stack : Stack<StripsElement>
+push(StripsElement) : void
+pop() : StripsElement
+isEmpty() : boolean
Variable
-value : Object
+Variable(Object)
+isInstantiated() : boolean
+instantiate(Object) : void
+getValue() : Object
+equals(Object) : boolean
ConjunctivePredicate
-predicates : List<Predicate>
+add(Predicate) : void
+remove(Predicate) : void
+toList() : List<Predicate>
Operator
-preconditions : ConjunctivePredicate
-add : ConjunctivePredicate
-delete : ConjunctivePredicate
+getPreconditions() : ConjunctivePredicate
+getAdd() : ConjunctivePredicate
+getDel() : ConjunctivePredicate
<<use>>
State
-predicates : ConjunctivePredicate
+State(ConjunctivePredicate)
+getPredicates() : ConjunctivePredicate
+applyOperator(Operator) : void
+getFalseSinglePredicates(ConjunctivePredicate) : List<Predicate>
+isTrue(Predicate) : boolean
<<use>>
Strips
-stack : StripsStack
-operators : List<Operator>
-initialState : State
-finalState : State
-currentState : State
+Strips(initialState : State, finalState : State, operators : List<Operator>)
+calculatePlan(Heuristic) : Plan
Plan
-operators : List<Operator>
+addOperator(Operator) : void
<<use>>
<<use>>
Heuristic
+heuristicPushOrder(predicates : List<Predicate>, currentState : State) : List<Predicate>
+choosePredicateForInstantiation(predicates : List<Predicate>, currentState : State) : Predicate
Powered By Visual Paradigm Community Edition

Figure 3: UML Class Diagram of STRIPS classes

The implementation is centered around the `Strips` class. Objects of this class use a `StripsStack` to solve a specific problem, given *initialState* and *finalState* of class `State` and a list of `Operator` objects. The method *calculatePlan* takes a realization of the abstract class `Heuristic` and calculates a `Plan` object based on the STRIPS algorithm and the domain specific heuristics specified in the *heuristic*.

The stack provided by `StripsStack` can be filled with objects of the class `StripsElement`, which is the superclass for `Predicate`, `Operator`, and `ConjunctivePredicate`.

`Predicate` objects have an unique *identifier* and a list of `Variable` objects that represent the arguments of the predicate. `Variable` objects can take another `Object` as their *value* or have a *value* of `null` if the argument is not instantiated.

Objects of `ConjunctivePredicate` represent a list of single `Predicate` objects. They are used in other classes such as `State` or `Operator` to handle conjunction lists of predicates.

The `Operator` class represents operators in the STRIPS algorithm with their lists of preconditions and the postconditions split up in both a `ConjunctivePredicate` *add* and *delete*.

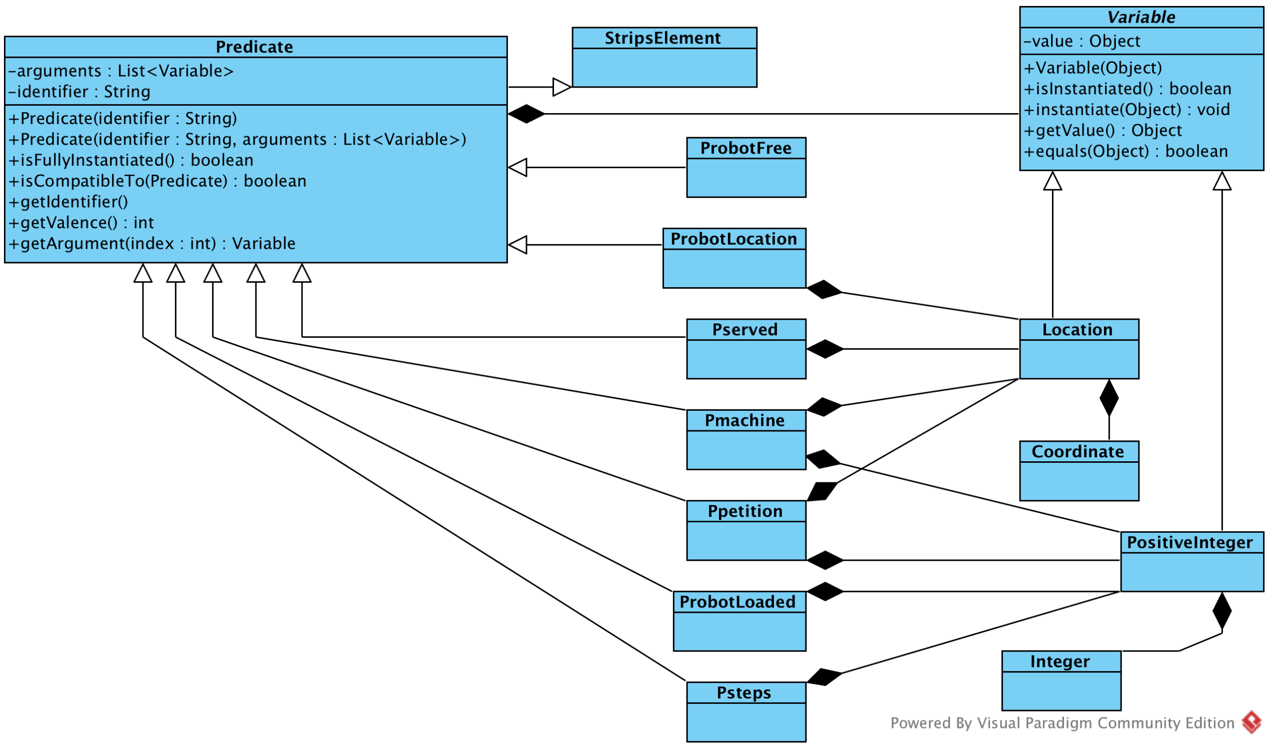

Predicate
-arguments : List<Variable>
-identifier : String
+Predicate(identifier : String)
+Predicate(identifier : String, arguments : List<Variable>)
+isFullyInstantiated() : boolean
+isCompatibleTo(Predicate) : boolean
+getIdentifier()
+getValence() : int
+getArgument(index : int) : Variable
StripsElement
Variable
-value : Object
+Variable(Object)
+isInstantiated() : boolean
+instantiate(Object) : void
+getValue() : Object
+equals(Object) : boolean
ProbotFree
ProbotLocation
Pserved
Location
Pmachine
Coordinate
Ppetition
PositiveInteger
ProbotLoaded
Integer
Psteps
Powered By Visual Paradigm Community Edition

Figure 4: UML Class Diagram of STRIPS classes

4.2 Problem Implementation

It should be noted that none of the classes introduced in the previous subsection 4.1 are implementations of a specific domain dependent problem - they are a general representation of the STRIPS algorithm that works for a large set of problems.

The class `CoffeeHeuristic` is a realization of its abstract super-class `Heuristic`. It implements the heuristics described in the previous section.

The package `problem.coffee.pred` contains implementations of the problem's different predicates which are all subclasses of `Predicate`. The data types of the arguments are implemented in `problem.coffee.pred.args` and are subclasses of `Variable`.

The three operators are implemented in the package `problem.coffee.operators` and are subclasses of `Operator`.

The class `problem.Solve` creates a `Strips` object and uses it to compute a solution for a specific coffee problem, read from a file.

4.3 Heuristic Details

In this section we present some chosen details of our implementation that we consider more relevant. We focus on demonstrating the implementation of the heuristics explained in subsection 3.2. To get a full understanding of the code one might want to look at the complete source code attached to this report (see Appendix A).

The most important heuristic in terms of step minimization is choosing the best coffee machine out of several available ones when instantiating a Machine(o, n) predicate. Listing 2 shows how the STRIPS algorithm finds all possible instantiations of a predicate *singlePred* in the current state. All possible instantiation candidates are then passed to the heuristic, that decides which Machine(o, n) predicate to choose.

Listing 2: Strips method *instantiate*

```java
private void instantiate(Predicate singlePred, Heuristic heuristic) {
  List<Predicate> compatiblePredicates = new ArrayList<>();
  for (Predicate currPred : this.currentState.getPredicates().toList()) {
    // find all compatible predicates in current state:
    if (currPred.isCompatibleTo(singlePred)) {
      compatiblePredicates.add(currPred);
    }
  }
  ...
  // Heuristic chooses one candidate from compatiblePredicates
  Predicate chosenPred = heuristic.choosePredicateForInstantiation(
                                   compatiblePredicates, currentState);

  // instantiate with singlePred with constants of chosenPred
  for (int i = 0; i < chosenPred.getValence(); i++) {
    if (!singlePred.getArgument(i).isInstantiated()) {
      // Update the Java object of the variable.
      // All references to this variable in other predicates
      // will now reference to the instantiated variable.
      singlePred.getArgument(i).instantiate(
                   chosenPred.getArgument(i).getValue());
    }
  }
}
```

In listing 3 we can see, how the heuristic chooses a good machine based on the current location of the robot and the next petition he is trying to serve. Each time a new petition is instantiated (based on a Served(o) predicate), the heuristic object saves the location of this petition in a local variable *lastPetition*. When the heuristic encounters a list of several Machine(o, n) predicates as candidates to instantiate a single predicate, it will choose the machine with the smallest sum of the distance between the current position and the machine plus the distance between the machine and the next position. This makes sure that a machine is picked that leads to the least possible increase in steps.

Listing 3: CoffeeHeuristic method *choosePredicateForInstantiation*

```java
public Predicate choosePredicateForInstantiation(
  List<Predicate> compatiblePredicates, State currentState) {
```

```
  // Save the last instantiated petition to be included in the
  //distance calculation:
  List<Ppetition> petitionsPred = new ArrayList<>();
  for (Predicate pred : compatiblePredicates) {
    if (pred.getIdentifier().equals(Ppetition.ID)) {
      petitionsPred.add((Ppetition) pred);
    }
  }
  if (!petitionsPred.isEmpty()) {
    lastPetition = petitionsPred.get(0);
  }

  // filter out Machine predicates
  List<Pmachine> machinePreds = new ArrayList<>();
  for (Predicate pred : compatiblePredicates) {
    if (pred.getIdentifier().equals(Pmachine.ID)) {
      machinePreds.add((Pmachine) pred);
    }
  }

  if (machinePreds.size() > 0) {
    // Find current position in current state:
    Location currentPos = null;
    for (Predicate pred : currentState.getPredicates().toList()) {
      if (pred.getIdentifier().equals(ProbotLocation.ID)) {
        currentPos = (Location) pred.getArgument(0);
        break;
      }
    }

    // Find best machine:
    Pmachine closestMachine = null;
    int bestDistanceSoFar = Integer.MAX_VALUE;
    for (Pmachine pred : machinePreds) {
      // distance = distance from current position to machine +
      // distance from machine to last petition
      int distance = Location.distance(currentPos, pred.getLocation(),
                     lastPetition.getLocation());
          if (distance < bestDistanceSoFar) {
            closestMachine = pred;
            bestDistanceSoFar = distance;
          }
    }
    return closestMachine;
  }
  ...
}
```

5 Tests

We designed both manually designed three different worlds with increasing complexity and 1000 random worlds with random complexity.

5.1 Manually Created Worlds

The first example `data/examples/test1` is a simple world with only one Machine and one Petition:

```
InitialState=Robot-location(o1);Machine(o16,1);Petition(o26,1);Robot-free; Steps(0);
GoalState=Robot-location(o7);Served(o26);
```

The second example `data/examples/test2` is a more complex world with now two Machines and one Petition:

```
InitialState=Robot-location(o1);Machine(o16,1);Machine(o14,1);Petition(o26,1);
Robot-free; Steps(0); GoalState=Robot-location(o7);Served(o26);
```

The third example `data/examples/test2` is even more complex with two Machines and now four Petitions:

```
InitialState=Robot-location(o1);Machine(o29,1);Machine(o8,1);Petition(o3,1);
Petition(o32,1);Petition(o18,1);Petition(o19,1);Robot-free; Steps(0); GoalState=
Robot-location(o7);Served(o3);Served(o32);Served(o18);Served(o19);
```

5.2 Automatically created worlds

We created 1000 random worlds. Each of these worlds has between 3 and 9 petitions (at least one for each possible n) and between 3 and 9 machines (at least one for each possible n). The generated examples are attached to this report, see appendix A.

6 Results

6.1 Manually Created Worlds

6.2 Automatically created worlds

Figure 8 shows the average result (blue dot) and standard deviation (blue bar) of the step error for different heuristics. The step error is defined as the difference between the steps of the optimal solution and the steps that the planner came up with.

NN-TS means, that the NN-heuristic was used to determine the order of petitions. Random means, that a random order of petitions was used. Closest-M means, that the Machine predicate heuristic was used, while First M corresponds to simply picking the first Machine

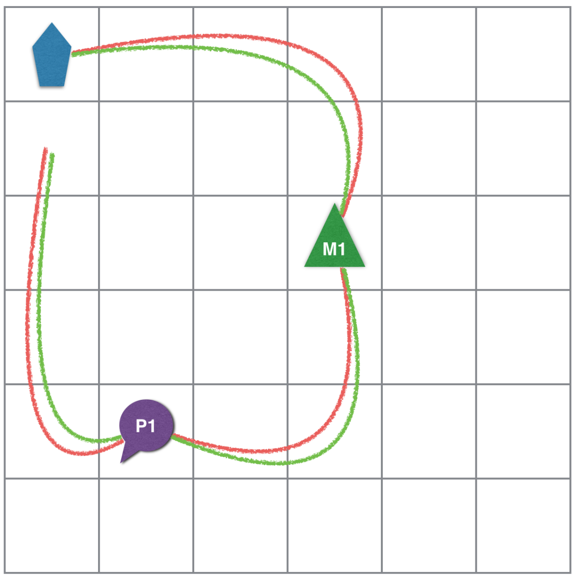


Figure 5: Results for first manually created example

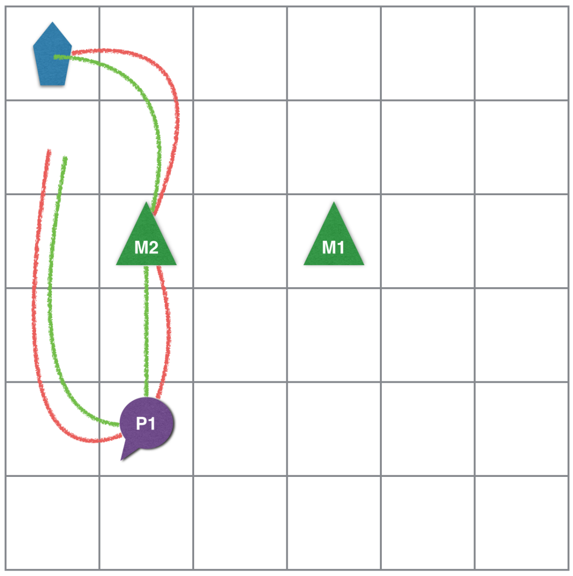


Figure 6: Results for second manually created example

predicate in the list.

As we can see in figure 8, using both heuristics yields the best results on average. Not using the NN-heuristics only worsens the results slightly. Deactivating the Machine-heuristic makes the step error increase dramatically - hence it has the highest impact on success.

Figure 9 shows how often the different heuristic configuration yielded the smallest step error. Again we can see, that using both heuristics yields the best results with a winner ratio of more than 0.6. The values don't add up to 1.0 because sometimes several strategies had the same result.

Figure 10 shows the relationship between the complexity of the world in terms of numbers of petitions and the average step error. As we can see, for increasing the complexity, using both heuristics yields increasingly better results compared to the other strategies.

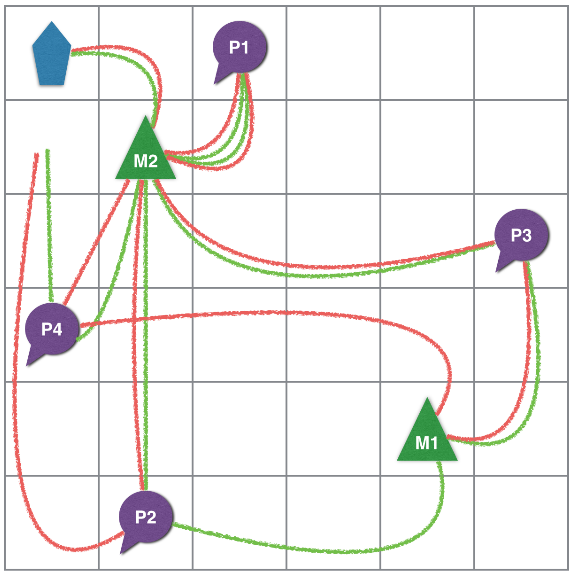


Figure 7: Results for third manually created example

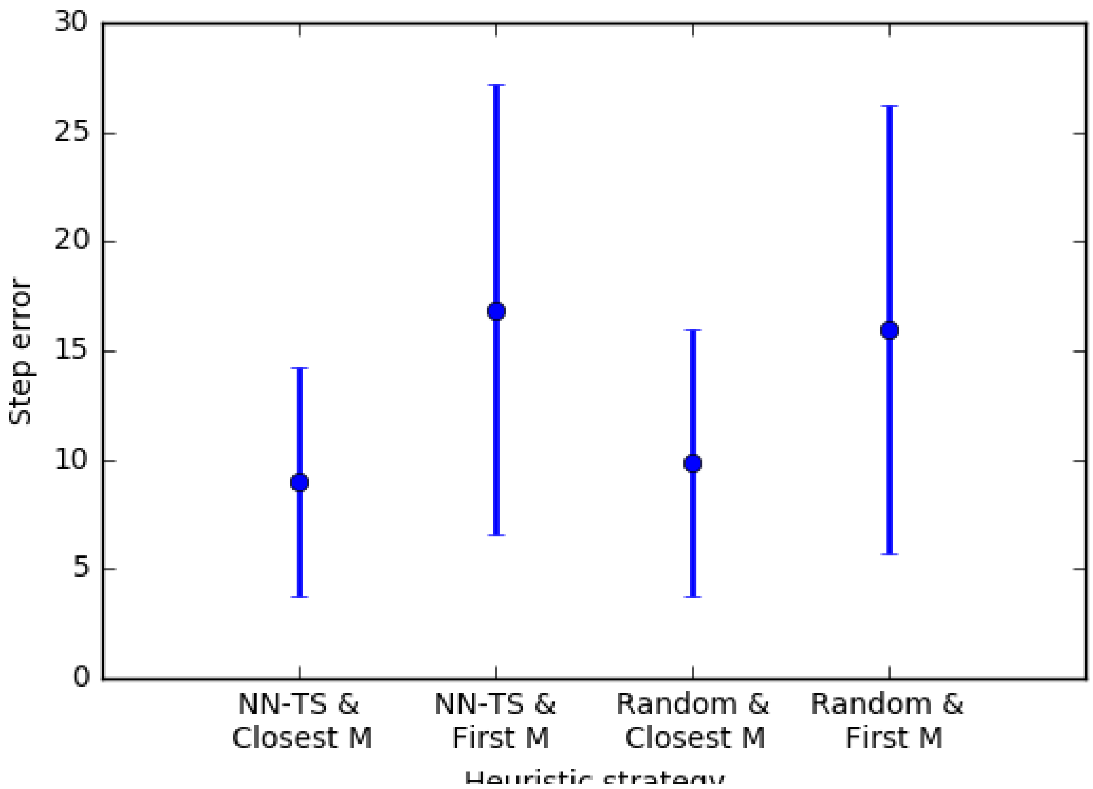


Figure 8: Step Errors for Different Heuristic

Figure 11 shows the relationship between the complexity of the world in terms of numbers of machines and the average step error. There is less correlation between machine complexity and the step error, than we observed with the petition complexity.

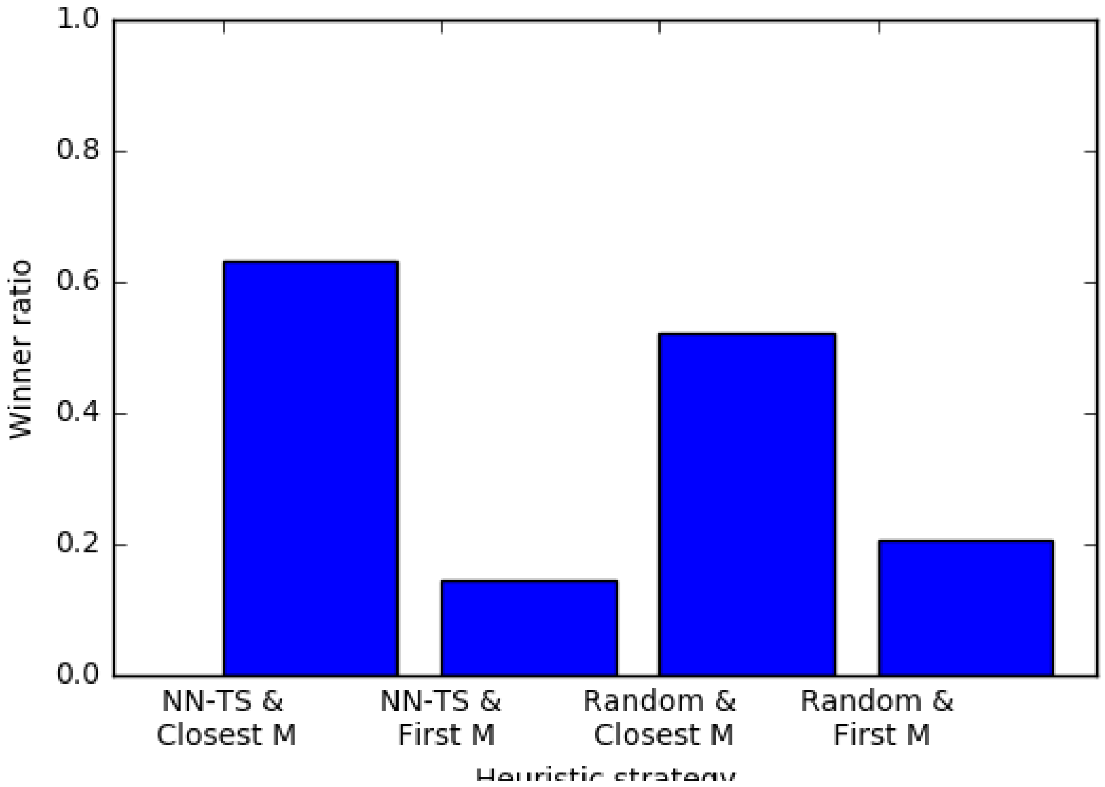


Figure 9: Ratio of Best Solution for Different Heuristic

7 Outlook

The results of our experiments were promising and showed, that even relatively simple heuristics can substantially improve the number of required steps to solve the coffee problem. One possible improvement would be, to use a more sophisticated heuristic to calculate the TSP-like route of the different petitions.

It would also potentially yield interesting results to test the algorithm on larger worlds. Some first experiments on worlds with a size of 100,000 fields with several ten-thousands of petitions have shown that the two used heuristics managed to reduce the number of steps from about 4 million to about 20,000 while only requiring approximately 15% more computation time. The optimal solution could not be computed with brute force in this case since the sheer number of possible permutations exceeds available computation power by far.

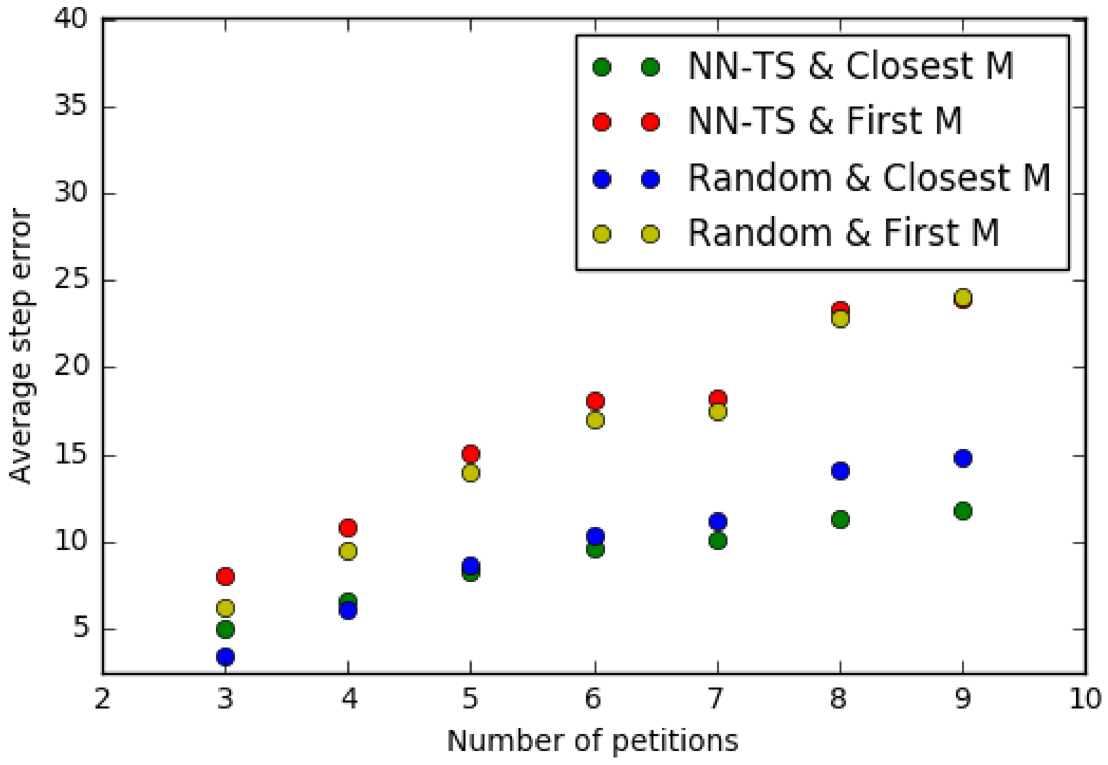


Figure 10: Average Result and Petition Complexity

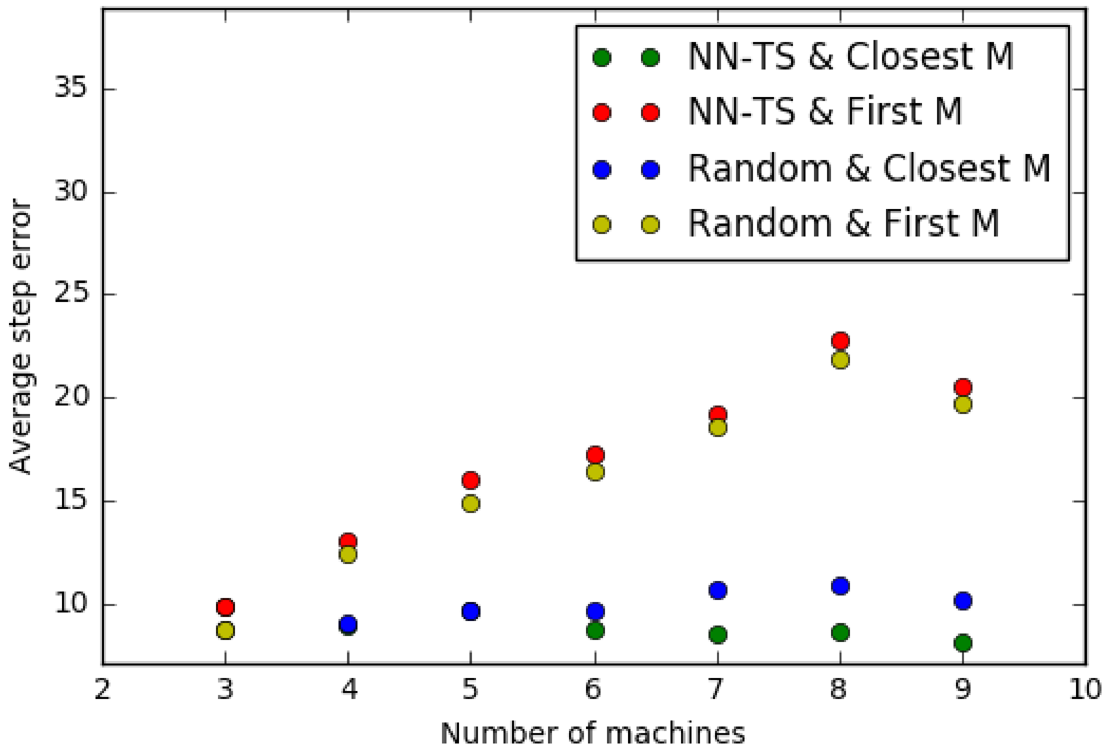


Figure 11: Average Result and Machine Complexity

Appendices

A Included Files

all the code!

image of the UML diagram!

B Eclipse Instructions

1. Import the project via `File > Import > Existing Projects into Workspace`
2. Navigate to class `problem.Solve`
3. Choose the input file for the main method by either:
   - Changing the constant `INPUT_PATH` to the desired path, or
   - Going to `Run > Run Configuration > Arguments > Program arguments` and copying the relative or absolute file path in the first line.
4. Run the main method of `problem.Solve`.
5. The results can be found in the directory `data/out`.

To run the experiment that our results are based on:

1. Navigate to class `problem.coffee.validator.CoffeeExperiments`.
2. Run the main method.
3. Results can be found in the directory `data/generated`:
   - `steps...` contains the number of steps for the different methods. The first four columns correspond to the order in the figures of our result section. The last column are the steps for the optimal solution (calculated with brute force).
   - `test...` contains 1000 different worlds, one per line.
   - `test_matrix...` contains graphical representations of all 1000 worlds.