Department of Computer Science
Technical University of Cluj-Napoca

Introduction to Artificial Intelligence

Adrian Groza, Radu Razvan Slavescu and Anca Marginean

Contents

1 Problem-solving agents **5**
- 1.1 Python programming language . . . 5
- 1.2 Pac-Man framework . . . 5
- 1.3 Solutions to exercises . . . 9

2 Uninformed search **10**
- 2.1 Let's find the food-dot. Search problem . . . 10
- 2.2 Random search agent . . . 11
- 2.3 Tree seach and graph search . . . 13
- 2.4 Depth first, breadth first and uniform costs search strategies . . . 13
- 2.5 Solutions to exercises . . . 17

3 Informed search **19**
- 3.1 A* algorithm . . . 19
- 3.2 Admissible and consistent heuristics . . . 19
- 3.3 Finding all corners. Eating all food . . . 21
- 3.4 Solutions to exercises . . . 23

4 Adversarial search **24**
- 4.1 Reflex agent . . . 24
- 4.2 Minimax algorithm . . . 26
- 4.3 Alpha-beta prunning . . . 27
- 4.4 Solutions to exercises . . . 30

5 Propositional logic **31**
- 5.1 Getting started with Prover9 and Mace4 . . . 31
- 5.2 First example: Socrates is mortal . . . 31
- 5.3 A more complex example: FDR goes to war . . . 34
- 5.4 Wumpus world in Propositional Logic . . . 35
- 5.5 Solutions to exercises . . . 37

6 Models in propositional logic **38**
- 6.1 Formalizing puzzles: Princesses and tigers . . . 38
- 6.2 Finding more models: graph coloring . . . 41
- 6.3 Solutions to exercises . . . 43

7 First Order Logic **46**
- 7.1 First Example: Socrates in First Order Logic . . . 46
- 7.2 FDR goes to war in FOL . . . 47
- 7.3 Alligator and Beer: finding models in FOL . . . 50
- 7.4 Solutions to exercises . . . 52

8 Inference in First-Order Logic **53**
8.1 Revisiting resolution and skolemization . . . 53
8.2 Paramodulation and demodulation . . . 54
8.3 Let's find the Wumpus! . . . 57
8.4 Solutions to exercises . . . 58

9 Constraint satisfaction problems **60**
9.1 Solving CSP by consistency checking . . . 60
9.2 Solving CSP with stochastic local search . . . 63
9.3 Solving CSP with satisfiability solvers . . . 65
9.4 Solutions to exercises . . . 69

10 Classical planning **71**
10.1 Planning domains . . . 71
10.2 Planning Domain Definition Language . . . 74
10.3 Fast Downward planner . . . 75
10.4 Solutions to exercises . . . 76

11 Heuristics for planning **79**
11.1 Defining heuristics by relaxation . . . 79
11.2 Search engines and heuristics in the Fast Downward planner . . . 81
11.3 Introducing costs to actions . . . 83
11.4 Modelling planning domains . . . 84
11.5 Solutions to exercises . . . 87

12 Planning and acting in the real world **89**
12.1 Planning domains with uncertainty . . . 89
12.2 Conformant planning . . . 90
12.3 Contingent planning . . . 91
12.4 Solution to exercises . . . 94

13 Knowledge representation with event calculus **95**
13.1 Discrete Event Calculus reasoner . . . 95
13.2 Reasoning modes in event calculus . . . 96
13.3 Carying a newspaper example . . . 97
13.4 Solutions to exercises . . . 99

A Brief synopsis of Linux **100**

B LaTeX– let's roll! **109**

C Fast and furios tutorial on Python **112**

Introduction

This tutorial follows the structure of the AIMA textbook. The tutorial proposes exercises for the first twelve chapters of the textbook and has three parts.

The first part deals with agents that solve problems by searching. Uninformed search strategies (depth first search, breadth first search, uniform cost search) are compared against informed search strategies (A* algorithm). The strength of A* algorithm is evidentiated through various admissible and consistent heuristics. The search strategies are presented through a game-based framework, that is the *Pacman* agent. The Pacman game allows students to also practice adversarial search. This part relies on the search tutorial used at Berkeley `http://ai.berkeley.edu/project_overview.html`. The programming language for this part is Python, as this language has currently rised as the main platform for artificial intelligence.

The second part focuses on knowledge representation based on propositional and first order logic. *Prover9* theorem prover and *Mace4* satisfiability tool are used to illustrate concepts like: resolution, unsatisfiability, models, or conjuctive normal form. The chapter presenting constraint satisfaction problems allows students to trace local search algorithms: random walk, hill climbing, stochastic hill climbing, greedy descendent with random restarts or simmulating annealling.

The third part deals with planning agents. The *Fast Downward* planning system is used to solve classical planning problems. To role of heuristics in planning is stressed through various experiments and running scenarios. We also consider partial observability and nondeterminism. To handle these, we need conformant planning. Moreover, ability to observe aspects of the current state is handled by contingent planning.

The proposed exercises are rather ambitious. Some exercises are straitforward, but others are open toward stimulating research in the eager students. The rationale was to accomodate the heterogenity of the learners. Focus is both on programming and on modelling the reality into a formal representation. Students should be aware that computing interacts with many different domains. Solutions to many artificial intelligence tasks require both computing skills and domain knowledge. This laboratory best serves as a test to see whether you have at the moment the potential to propose solutions to realistic scenarios and validate your solutions with a running prototype. The assignments are designed to provide you with an insight of the research methodology by developing your critical thinking, enhancing your ability to work independently and develop your technical writing. Side effects of completing this tutorial include the pleasure of discovering the strength of Linux for scientific experiments and also the most powerful framework for scientific editing, that is Latex.

Chapters 1-4 and the appendix on Python are written by Anca Marginean. Chapters 5-8 and the appendix on Linux and LaTex are written by Radu Razvan Slavescu. Chapter 9 is a joint work between Radu Razvan Slavescu and Adrian Groza. Chapters 10-13 are written by Adrian Groza.

Chapter 1

Problem-solving agents

Learning objectives for this week are:

1. To get used to or recall Python programming language
2. To run Pacman framework

1.1 Python programming language

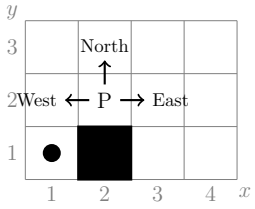
The first four laboratory works use a framework developed in Python. You need basic skills of writing Python. Anexa C is a brief overview of the main elements you will use. Either you are a Python guru, or a Python beginner, read it and do the exercises.

1.2 Pac-Man framework

The Pac-Man projects were developed for UC Berkeley's introductory artificial intelligence course. They were released to other universities for education use `http://ai.berkeley.edu/project_overview.html`. *Search*, *Adversarial search*, *Reinforcement learning*, *Probabilistic inference with hidden Markov model*, *Machine learning with Naive Bayes or Perceptron* are some AI techniques applied in the projects. During this semester you will study the first two.

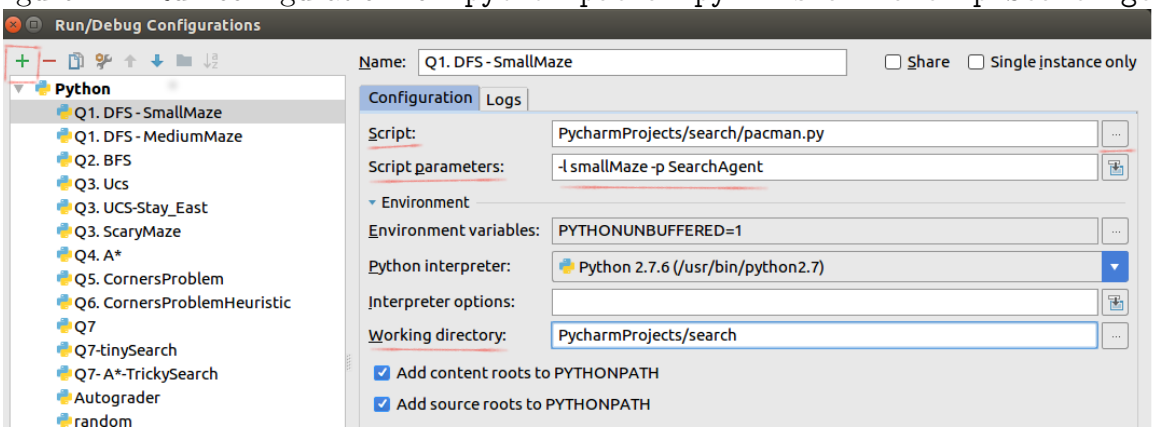
Pac-man is a game with more entities: Pacman, ghosts, food-dots and power-pellets. The player controls Pacman in its quest of eating all the food-dots. Pacman dies if it eaten by a ghost, but if Pacman eats a power-pellet, it wil have temporary ability to eat also the ghosts. Our aim is to build agents that control Pacman and win. Possible actions are North, South, East, West, Stop, depending on the presence of walls (see figure 1.1). With each step, Pacman looses 1 point, at each food-dot eaten it gets 10 points, on finishing the game it get 500 points.

Figure 1.1: Legal actions for Pacman when there is a wall



Legal actions

Figure 1.2: Run configuration for: `python pacman.py -l smallMaze -p SearchAgent`



For the study of search problems and agents a simpler version of Pacman is used where the ghosts are not present. Download the code from `https://s3-us-west-2.amazonaws.com/cs188websitecontent/projects/release/search/v1/001/search.zip` and extract it into your own folder.

You can play a game of Pacman by running the script `pacman.py`:

```
python pacman.py
```

You can see the available options for the script `pacman.py` with:

```
python pacman.py -h
```

The layout must be specified with option $-l$ or $--layout$. The agent is specified with $-p$ or $--pacman$. The layouts are defined in the *Layout* folder. The agents are defined in `searchAgents.py` file.

Using PyCharm

You can use any text editor for editing the files, or if you prefer working in an IDE, you can use Pycharm.

Possible steps in using PyCharm:

1. Start *PyCharm* and create a new project with the location in the folder `search` and with `python 2.7` as project interpreter.

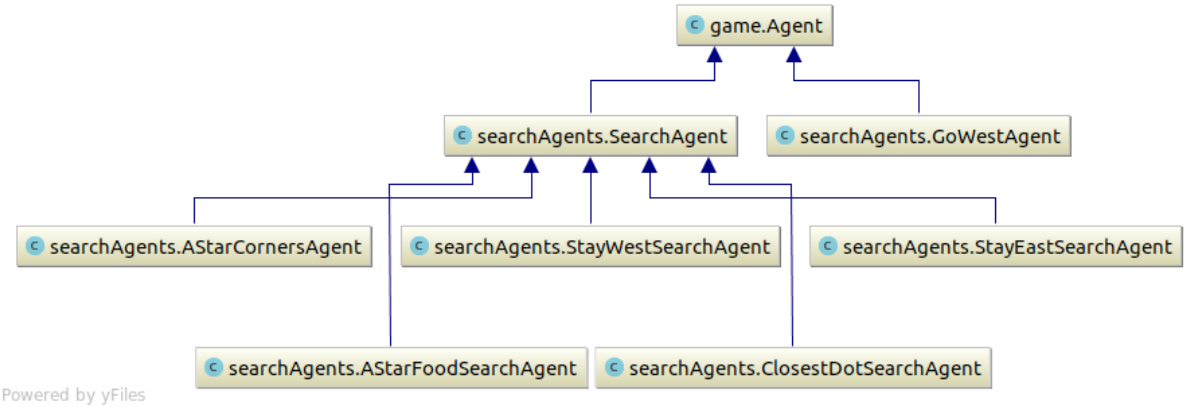
2. Run `pacman.py`: Right click on `pacman.py` and run. This will create a default Run Configuration. You can run the current configuration with SHIFT+F10.

3. You can change or add new Run configurations from the menu *Run >> Edit configurations* or from the *Select Run Configuration* drop-down list in the right upper corner (see figure 1.2).

```
 $ pycharm.sh
File >> New Project>> Pure Python
  <choose search folder from your own folder for Location>
  <select Python 2.7 for Interpreter>

Run >> Edit Configurations >> + >> Choose Python >> Edit the
   parameters: Script, Script parameters, and Working directory
```

Figure 1.3: Types of search agents from `SearchAgens.py`.



There are more alternative running ways for Pacman: from a terminal outside PyCharm, from Pycharm terminal, or with Pycharm Run configurations. In order to open Pycharm terminal use ALT+F12 or go the bottom-left corner of PyCharm window. For all exercises, the script is `pacman.py`, but the parameters will be different, mainly for the *layout* and the *agent*. Since you will run pacman with more options, you should create a Run Configuration for each combination. The file *commands.txt* from *search* folder contains a list of the most common commands.

Content of `search` folder

In the extracted folder *search* there are more files. Each file, class or method includes comments which are very important for you, mainly for classes or methods which are to be changed. You have to change only in the special are marked with `YOUR CODE HERE`.

- Files to be changed
  - `search.py` - description of an abstract class *SearchProblem* and several functions where you are supposed to add your code. The main methods of `SearchProblem` class are:
    - `getStartState(self)` - it returns the initial state
    - `isGoalState(self, state)` - it checks whether a state is a goal state and returns true or false
    - `getSuccessors(self, state)` - it takes a state and returns a list of legal successors together with the requires actions and cost.
    - `getCostOfActions(self, actions)` - it takes a sequence of actions and returns its cost.
  - `searchAgents.py` - includes the search-based agents, already implemented or ToBe implemented search problems, and heuristics.
    - search problem: There are two classes derived from `SearchProblem`:
      - `PositionSearchProblem` - a search problem in which Pacman needs to find a certain position or positions. Class `AnyFoodSearchProblem` is derived from this and describes the problem of finding all the food.
      - `CornersProblem` - a search problem in which Pacman needs to find all the corners.

    * agents: There are two important agents defined, a `SearchAgent` with more derived classes (figure 1.3), and a very simple reflex agent `GoWestAgent`.

- Files which include worth reading parts
  - `pacman.py` - the main file for running Pacman games. Read the description of GameState type which specifies the full game state.
  - `game.py` - the logic behind how the Pacman world works. Important types: *AgentState*, *Agent*, *Direction*, *Grid*.
  - `util.py` - data structures which are recommended to be used when implementing the search algorithms

In order to better understand the structure of the project, you can create class diagrams in PyCharm (Right click on a file >> Diagrams). Do not forget, you will most likely change only the files `search.py` and `searchAgents.py`.

Exercise 1.1 *Analyze the implementation of* `GoWestAgent` *from* `searchAgents.py` *and run the agent with the following commands:*

```
python pacman.py -l testMaze -p GoWestAgent
python pacman.py -l tinyMaze -p GoWestAgent
```

Run them from: i) PyCharm terminal; ii) PyCharm Run configuration; iii) system's terminal. Does Pacman reach the food from (1,1) for both layouts? Why?

Exercise 1.2 *Watch some movies with Pacman helped by search algorithm. At the end of the first four labs you will be able to solve similar problems.*

- *Why do we need search?* `http://cs-gw.utcluj.ro/~anca/iia/why_search.ogv`,
- *Different layouts solved with different search algorithms* `http://cs-gw.utcluj.ro/~anca/iia/examples.ogv`,
- *Comparison of more strategies on the same layout* `http://cs-gw.utcluj.ro/~anca/iia/comparison_search_strategies.ogv`,
- *pacman and more ghosts* `http://cs-gw.utcluj.ro/~anca/iia/multipacman.ogv`.

1.3 Solutions to exercises

Solution for exercise 1.1
GoWestAgent is a reflex agent which goes $West$ when it can, otherwise it *Stops*. This behaviour allows the agent to win for $testMaze$, but to loose when the maze is changed. Our aim is to write agents which can efficiently find their ways in any maze.

Chapter 2

Uninformed search

This lab will introduce search algorithm and search strategies for problem-solving agents. You will implement them in Pac-Man framework and apply them for a Position Search problem: helping Pac-Man find a certain foot-dot. Before dwelving into this, you will do some exercises to get familiar with Pac-Man framework.

Learning objectives for this week are:

1. Understand agents that solve problems by searching
2. Understand tree search and graph search
3. Develop uniformed search strategies: depth-first search, breadth-first search, uniform cost

2.1 Let's find the food-dot. Search problem

In the first problem Pac-Man needs to find a certain food dot. It is known the initial Pac-Man position and the position of the food-dot. The only possible actions for Pac-Man are going North, South, East, or West.

The solution to this problem is a sequence of actions on Pacman board. The cost of the solution is the total sum of actions' cost. Positions $(x,\ y)$ determine the state space. The layouts described in the folder *layout* mention the initial Pac-Man position, the food-dot position and walls.

Open `search.py` and read *tinyMazeSearch* function. Run the command

```
1 python pacman.py -l tinyMaze -p SearchAgent -a fn=tinyMazeSearch --
    frameTime=1
```

or add a new configuration with corresponding arguments. If you want to slow down the movement of Pac-Man, use the option frametime.

Exercise 2.1 *Read the output of your previous command. How many nodes were expanded? Which is the total cost of the found solution?*

Exercise 2.2 *Why the agent finds the dot? Change the maze (with another one from* layouts *folder). Does it work? Does the Pacman find the food?*

The problem is completely formalized in the class `PositionSearchProblem` from `searchAgents.py`. Conceptually it is a *search problem*, therefore its class extends the class `Search-`

`Problem`. Search-based agents are able to solve *search problems*, meaning they are able to compute sequences of actions for a certain layout. The most important advantage of using search problems and search-based agents is that the agents are problem-independent. Therefore, the same agent can solve Pacman food problem, but also eight-puzzle problem, once the problems are modelled as search problems.

A *search problem* can be defined by:

- initial state
- possible actions
- transition model: Result(s,a). A *successor* state is a state reachable from a given state by a single actions.
- goal test: it is true only in goal states.
- path cost

All the search problems from Pac-Man project are described in these terms. For food-dot problem, the initial state is the initial Pacman position. Possible actions are North, South, East, West, where for each action it is described the new position $(x',\ y')$ reached after doing the action from $(x,\ y)$. The goal test returns true when Pacman is in the same position with the food-dot. In the default problem, each action's cost is 1, therefore the path's cost is equal to the number of actions. The solution to a search problem is a sequence of actions which if executed from the initial state, reaches a goal state.

Exercise 2.3 *Read from AIMA what are and how to formalize Search problems in sections 3.1.1 and 3.1.2.*

2.2 Random search agent

The final aim of this laboratory work is to solve Pacman food problem, but before that, you will do some exercises which help you understand the structure of the code which is relevant for you.

Exercise 2.4 *Open* **`searchAgents.py`** *and go to the class* PositionSearchProblem. *Read the comments. Identify the main elements of a search problem: initial state, goal test, successor and cost of actions. Identify the order in which the legal actions are returned in getSuccessors method.*

Exercise 2.5 *Go to depthFirstSearch function from* **`search.py`** *and uncomment the existing lines:*

```
1 print "Start:", problem.getStartState()
2 print "Is␣the␣start␣a␣goal?", problem.isGoalState(problem.
     getStartState())
3 print "Start's␣successors:", problem.getSuccessors(problem.
     getStartState())
```

Run again

```
1  python pacman.py -l smallMaze -p SearchAgent
```

Figure 2.1: Initial position of Pacman in *smallMaze* layout

```
Start: (11, 6)
Is the start a goal? False
Start's successors: [((11, 7), 'North', 1), ((12, 6), 'East', 1),
   ((10, 6), 'West', 1)]
```

The initial position is (11,6) and it is not a goal state (figure 2.1). `problem.getSuccessors( problem.getStartState() )` returns a list of three tuples, one for each legal action. Each tuple contains the next state, the action and the action's cost. Only three actions are legal due to the fact that there is wall on south.

Exercise 2.6 *Go to depthFirstSearch function from* **`search.py`**. *Get the succesors of the initial state and print the state, the action and the cost for each successor. Run again with smallMaze.*

Exercise 2.7 *Go to depthFirstSearch function from* **`search.py`**. *Comment the code from the previous exercises. Comment util.raiseNotDefined() and similar to* TinyMazeSearch*, add to* depthFirstSearch *function:*

```
from game import Directions
w = Directions.WEST
return [w, w]
```

Run again

```
 python pacman.py -l smallMaze -p SearchAgent
```

Important: each search function must return a list of legal actions. Otherwise, you will get an error.

Exercise 2.8 *Go to depthFirstSearch function from* **`search.py`**. *Return a sequence of two legal actions from the initial state.*

Exercise 2.9 *Go to depthFirstSearch function from* **`search.py`**. *Create a new data-structure with two components: name and cost. Create two instances of the new data structure and add them to a* Stack *described in* **`util.py`**. *Pop an element from the stack and print it.*

Exercise 2.10 ***Random search agent***. *As a baseline, we will create an agent which searches a solution randomly: it just picks one legal action at each step of search. Write a new search function in* **`search.py`** *similar to* **`tinyMazeSearch`** *function. The function returns a list of actions from the initial state to goal state, each action being randomly selected from the set of legal actions.*

2.3 Tree seach and graph search

The general algorithms for solving search problems [23] are described in the following listings:

```
function TREE-SEARCH(problem) returns a solution, or failure
initialize the frontier using the initial state of problem

loop do
   if the frontier is empty then return failure
   choose a leaf node and remove it from the frontier
   if the node contains a goal state
       then return the corresponding solution
   expand the chosen node, adding the resulting nodes to the frontier
```

```
function GRAPH-SEARCH(problem) returns a solution, or failure
 initialize the frontier using the initial state of problem
 initialize the explored set to be empty

loop do
  if the frontier is empty then return failure
  choose a leaf node and remove it from the frontier
  if the node contains a goal state
    then return the corresponding solution
  add the node to the explored set
  expand the chosen node, adding the resulting nodes to the frontier
     only if not in the frontier or explored set
```

We recommend using of the following structure for nodes ([23]) in tree/graph search algorithm:

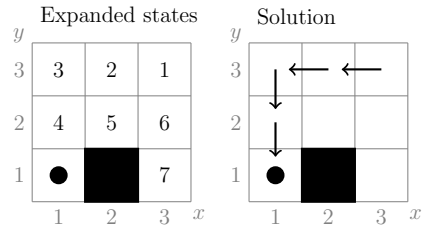
- state: the state in the state space to which the node corresponds;
- parent: the node in the search tree that generated this node;
- action: the action that was applied to the parent to generate the node;
- path-cost: the cost of the path from the initial state to the node

Exercise 2.11 *Read from AIMA section 3.3. about Tree search and Graph search as general methods for searching for solutions.*

The strategies for choosing the node to be extended strongly influences the length of the solution, the time and space required for searching. You must implement three uninformed search strategies and compare their application on Pacman food-dot problem.

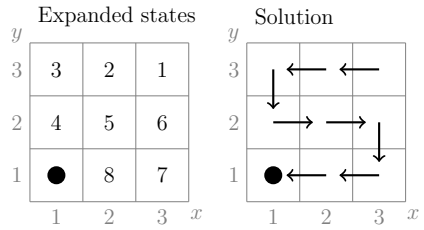
2.4 Depth first, breadth first and uniform costs search strategies

Depth-first search is a tree/graph-search with the frontier as LIFO (stack). Breadth-first search is a tree/graph-search with the frontier as FIFO (queue). Uniform cost search is a search with the frontier as a priority queue, meaning that it expands the node with the lowest path cost $g(n)$.



Nodes added to the frontier	Extracted node from frontier
	(3, 3)
(3, 3)→(3, 2)(South) (3, 3)→(2, 3)(West)	
	(2, 3)
(2, 3)→(2, 2)(South) (2, 3)→(1, 3)(West)	
	(1, 3)
(1, 3)→(1, 2)(South)	
	(1, 2)
(1, 2)→(1, 1)(South) (1, 2)→(2, 2)(East)	
	(2, 2)
(2, 2)→(3, 2)(East)	
	(3, 2)
(3, 2)→(3, 1)(South)	
	(3, 1)
No succesors which were not already expanded	
GOAL	(1, 1)

Solution: West, West, South, South



Nodes added to the frontier	Extracted node from frontier
	(3, 3)
(3, 3)→(3, 2)(South) (3, 3)→(2, 3)(West)	
	(2, 3)
(2, 3)→(2, 2)(South) (2, 3)→(1, 3)(West)	
	(1, 3)
(1, 3)→(1, 2)(South)	
	(1, 2)
(1, 2)→(1, 1)(South) (1, 2)→(2, 2)(East)	
	(2, 2)
(2, 2)→(2, 1)(South) (2, 2)→(3, 2)(East)	
	(3, 2)
(3, 2)→(3, 1)(South)	
	(3, 1)
(3, 1)→(2, 1)(West)	
	(2, 1)
(2, 1)→(1, 1)(West)	
GOAL	(1, 1)

Solution: West, West, South, East, East, South, West, West

Figure 2.2: Expanded states and solution for DFS on similar layouts: with/without wall

Exercise 2.12 *Read from AIMA about uninformed search strategies: Depth first search (section 3.4.3), breadth-first search (section 3.4.1) and uniform cost search (section 3.4.2).*

Exercise 2.13 *Compare the behavior of DFS on a simple layout of 3x3 with/without a wall in position (2,1) from figure 2.2. The number on each cell indicate the order in which the positions are explored, while the arrow indicate the actions from the solution. Think about ways to reduce the number of expanded states and to improve the quality of the solution.*

In case of breadth-first search, the goal test from graph search can be applied to each node before inserting the element in the frontier rather then when it is extracted from the frontier. Note that breadth-first search always has the shallowest path to every node on the frontier.

In `search.py` you can find `depthFirstSearch`, `breadthFirstSearch`, and `uniformCostSearch` functions. In order to obtain maximum score for the activity of this lab, you need to implement these three strategies. The first three questions from the autograder check the identified solution and the order in which the nodes are explored. You need to obtain 9 points for

these. Implement them as graph-searches with different types of data-structures for the frontier. it is recommended to use `Stack`, `Queue` and `PriorityQueue` classes implemented in file `util.py`. Write your code as general as possible and use methods from the *SearchProblem* class `getStartState`, `isGoalState`, `getSuccessors`.

Observation: If you implement DFS as a graph search, there will be minor differences between the three strategies DFS, BFS and UCS.

Exercise 2.14 Question 1 *In* `search.py`*, implement* ***Depth-First search*** *(DFS) algorithm in function* depthFirstSearch. *Don't forget that DFS graph search is graph-search with the frontier as a LIFO queue (Stack).*

- *Test your solution on more layouts:*

```
python pacman.py -l tinyMaze -p SearchAgent
python pacman.py -l mediumMaze -p SearchAgent
python pacman.py -l bigMaze -z .5 -p SearchAgent
```

- *Are the solutions found by your DFS optimal? Explain your answer*
- *Run autograder* `python autograder.py` *and check the points for* Question 1.

For more details, go to project page `http://ai.berkeley.edu/search.html`.

Exercise 2.15 Question 2 *In* `search.py`*, implement* ***Breadth-First search*** *algorithm in function* $breadthFirstSearch$.

- *Similar to DFS, test your code on mediumMaze and bigMaze by using the option* $-a\ fn = bfs$

```
1 python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
```

- *Is the found solution optimal? Explain your answer.*
- *Run autograder* `python autograder.py` *and check the points for Question 2.*

Exercise 2.16 Question 3 *In* `search.py`*, implement uniform-cost graph search algorithm in* $uniformCostSearch$ *function.*

- *Test it with* $mediumMaze$ *and* $bigMaze$

```
1 python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
```

- *Compare the results to the ones obtained with DFS. Are the solutions different? Is the number of extended(explored) states smaller? Explain your answer.*
- *Consider that some positions are more desirable than others. This can be modeled by a cost function which sets different values for the actions of stepping into positions. Identify in* `searchAgents.py` *the description of agents* $StayEastSeachAgent$ *and* $StayWestSearchAgent$ *and analyze the cost function. Why the cost* $.5 * *x$ *for stepping into* (x, y) *is associated to* $StayWestAgent$*?*
- *Run the agents* $StayEastSeachAgent$ *and* $StayWestSearchAgent$ *on* $mediumDottedMaze$ *and* $mediumScaryMaze$ *with uniform cost search.*

```
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
```

For more details, go to project page ***http://ai.berkeley.edu/search.html*** *.*

- *Run autograder* ***python autograder.py*** *and check the points for Question 3.*

Don't forget, a set of commands for running and testing Pacman are included in `commands.txt`. Once you implement BFS, you can test it on the eight puzzle problem with `python eightpuzzle.py`.

2.5 Solutions to exercises

Solution for exercise 2.1.
The number of expanded nodes is 0, since the sequence of actions is not computed, it is hard-coded. The cost of the path is 8, equal with the number of required actions from the initial position to (1,1).

Solution for exercise 2.2.
Since the sequence of actions is hard-coded, on a new layout most probable you will get an exception "Illegal action". For example for *smallMaze*, Pacman has wall in south of its initial position, and the first actions from the sequence is South, therefore you get an exception.

Solution for exercise 2.5.

```
1 print "Initial␣state␣is", problem.getStartState()
2 for succ in problem.getSuccessors(problem.getStartState()):
3     (state, action, cost) = succ
4      print "Next␣state␣could␣be", state, "␣with␣action␣", action,
           "and␣cost␣", cost
```

Solution for exercise 2.8.

```
1 (next_state, action, _) = problem.getSuccessors(problem.
     getStartState())[0]
2 (next_next, next_action, _) = problem.getSuccessors(next_state)
     [0]
3 print "A␣possible␣solution␣could␣start␣with␣actions␣", action,
     next_action
4 return [action, next_action]
5 #util.raiseNotDefined()
```

The last line is commented because we want to see what happens when these two actions are executed.

Solution for exercise 2.9.

```
node1 = CustomNode("first", 3) # creates a new object
node2 = CustomNode("second", 10)
print "Create a stack"
my_stack = util.Stack() # creates a new object of the class Stack
    defined in file util.py
print "Push the new node into the stack"
my_stack.push(node1)
my_stack.push(node2)
print "Pop an element from the stack"
extracted = my_stack.pop() # call a method of the object
print "Extracted node is ",extracted.getName()," ", extracted.
   getCost()
util.raiseNotDefined()
```

The class is defined also in `search.py`

```
class CustomNode:

    def __init__(self, name, cost):
        self.name = name # attribute name
        self.cost = cost # attribute cost

    def getName(self):
        return self.name

    def getCost(self):
        return self.cost
```

Solution for exercise 2.10.
At each step in the search, the agent asks for the successors of the current state and chooses one randomly. Remember that each successor is a tuple of state, action, cost. The state of the chosen successor is the new current state in search, while the action is added to the solution.

```
def randomSearch(problem):
    current = problem.getStartState()
    solution =[]
    while (not (problem.isGoalState(current))):
        succ = problem.getSuccessors(current)
        no_of_successors = len(succ)
        random_succ_index = int(random.random()*no_of_successors)
        next = succ[random_succ_index]
        current = next[0]
        solution.append(next[1])
    print "The solution is ", solution
    return solution
```

Add `rs = randomSearch` to the end of `search.py` and run with option $-a\ fn = rs$

```
python pacman.py -l tinyMaze -p SearchAgent -a fn=rs
```

Chapter 3

Informed search

What can an agent do when no single action will achieve its goal? SEARCH. Although BFS, DFS and UCS are able to find solutions, they do not do it efficiently. They are called *uninformed* algorithms since they use only problem definition and no other information. The search can be reduced in many situations with some guidance on where to look the solutions. *Informed algorithms* use additional information about the problem in order to reduce the search.

Learning objectives for this week are:

1. To implement A^* algorithm
2. To compare search strategies: DFS, BFS, UCS, A*
3. To formulate your own search problem
4. To define admisible and consistent heuristics for A^*

3.1 A* algorithm

Informed search strategy uses problem-specific knowledge beyond the definition of the problem itself. Best-first search is Graph-search in which a node is selected for expansion based on an *evaluation function*. Evaluation function f is an *estimation* of the real cost.

A^* algorithm is the most widely known best-first search algorithm. The evaluation of a node combines the cost to earch the node from the initial state with the estimated cost to get from the node to the goal.

$$f(n) = g(n) + h(n)$$

$h(n)$ is the heuristic. An admisible and consistent heuristic guarantees the optimality of A^*.

3.2 Admissible and consistent heuristics

An admissible heuristic never overestimates the cost to reach the goal. A consistent heuristic meets the following condition for each nodes n and n':

$$h(n) \leq c(n, a, n') + h(n')$$

Exercise 3.1 *Read from AIMA section 3.5.2 - Minimizing the total estimated solution cost.*

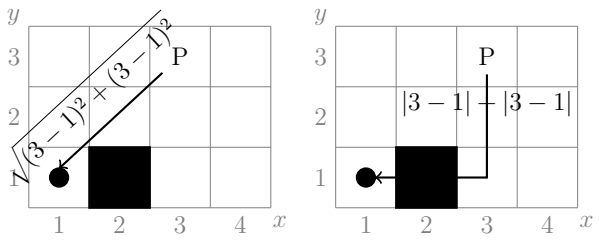


Figure 3.1: Euclidian and Manhattan Distance

For Pacman food-dot problem, good heuristic are Euclidian and Mahattan distances between two positions (see figure 3.1). Let's analyze the behaviour of A^* with Manhattan Distance as heuristic on the layout 4x3 from figure 3.1, with Pacman in $(3, 3)$ and food-dot in $(1, 1)$.

n^{th}	Nodes added to frontier	Expanded	g	h	f
1		(3, 3)			
	(3, 3)→(3, 2)(South)		1	3	4
	(3, 3)→(4, 3)(East)		1	5	6
	(3, 3)→(2, 3)(West)		1	3	4
2		(3, 2)			
	(3, 2)→(3, 1)(South)		2	2	4
	(3, 2)→(4, 2)(East)		2	4	6
	(3, 2)→(2, 2)(West)		2	2	4
3		(2, 3)			
	(2, 3)→(2, 2)(South)		2	2	4
	(2, 3)→(1, 3)(West)		2	2	4
4		(3, 1)			
	(3, 1)→(4, 1)(East)		3	3	6
5		(2, 2)			
	(2, 2)→(1, 2)(West)		3	1	4
6		(1, 3)			
	(1, 3)→(1, 2)(South)		3	1	4
7		(1, 2)			
	(1, 2)→(1, 1)(South)		4	0	4
8		(1, 1)			

Solution: West, South, West, South

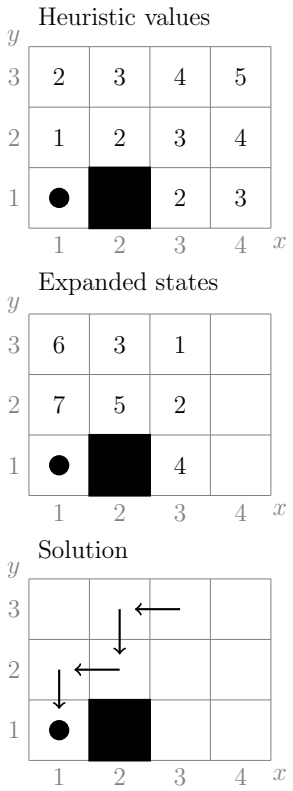


Figure 3.2: A^* algorithm on simple layout. Heuristic Values, Expansion order, Solution.

In figure 3.2 you can observe the order in which the nodes are added into and extracted from the frontier. You can observe also that the optimal solution has cost 4. During the search, the cost of the paths increases, while the heuristic value decreases as we get closer to the goal. The value of every heuristic in goal must be zero. None of the states on the right of the initial position of Pacman are expanded, since their value for $f = 6$ is greater than the cost of the solution 4. Many states with f value equal to the cost of the solution are expanded. In case there are more states on the frontier with the same f value, any state can be extracted.

Exercise 3.2 *Question 4. Go to* aStarSearch *in* **`search.py`** *and implement A^* search algorithm. A^* is graphs search with the frontier as a priorityQueue, where the priority is given by the function $g = f + h$*

- Test your implementation by searching for a solution for finding a food dot with the use of Manhattan Distance heuristic by using the option *heuristic*

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,
    heuristic=manhattanHeuristic
```

- Does A^* and UCS find the same solution or they are different?
- Does A^* finds the solution with fewer expanded nodes than UCS?
- Test with autograder `python autograder.py` that you obtain 3 points for Question 4.

For more details, go to the project's page `http://ai.berkeley.edu/search.html#Q4`

3.3 Finding all corners. Eating all food

A^* algorithm is a very powerful algorithm. In order to prove this, you will work with another two *search problems*: firstly, Pacman needs to find all four corners, secondly, Pacman needs to eat all the food-dot available. Don't forget the walls. When it comes to finding all corners, you will practice your ability to describe search problems. Then, you will deal with the efficiency of heuristics. Better the heuristics, lesser the number of expanded nodes. You are asked to propose heuristics for both problems: all corners and all food-dot.

Exercise 3.3 *Question 5. Pacman needs to find the shortest path to visit all the corners, regardless there is food dot there or not. Go to* CornersProblem *in* ***searchAgents.py*** *and propose a representation of the state of this search problem. It might help to look at the existing implementation for PositionSearchProblem. The representation should include only the information necessary to reach the goal. Read carefully the comments inside the class* CornersProblem.

- *Test your implementation with BFS - remember that BFS finds the optimal solution in number of steps (not necessarilly in cost). The cost for each action is the same for this problem.*

```
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=
    CornersProblem
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=
    CornersProblem
```

For hint and more details, go to the project's page ***http://ai.berkeley.edu/search.html#Q5****.*

For *mediumCorners*, BFS expands a big number - around 2000 search nodes. It's time to see that A^* with an admissible heuristic is able to reduce this number.

Exercise 3.4 *Question 6. Implement a consistent heuristic for CornersProblem. Go to the function* cornersHeuristic *in* ***searchAgent.py****.*

- *Test it with*

```
$python pacman.py -l mediumCorners -p SearchAgent -a fn=
    aStarSearch,prob=CornersProblem,heuristic=cornersHeuristic
or
$python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
```

The heuristic is tested for being consistent, not only admissible.

Grading depends on the number of nodes expanded with your solution.

Number of nodes expanded	*Grade*
more than 2000	*0/3*
at most 2000	*1/3*
at most 1600	*2/3*
at most 1200	*3/3*

Exercise 3.5 ***Question 7*** *Propose a heuristic for the problem of eating all the food-dots. The problem of eating all food-dots is already implemented in FoodSearchProblem in* `searchAgents.py`*.*

- *Test your implementation of A* on FoodSearchProblem by running*

```
1 python pacman.py -l testSearch -p AStarFoodSearchAgent
2 identical to
3 python pacman.py -l testSearch -p SearchAgent -a fn=astar,prob=
    FoodSearchProblem,heuristic=foodHeuristic.
```

  The existing heuristic foodHeuristic returns 0 for each node (trivial heuristic), therefore the previos running is the same with UCS. For the layout testSearch the optimal solution is of length 7, while for tinySearch layout is of length 27. The number of expanded nodes for tinySearch is very large: around 5000 nodes.

- *Go to foodHeuristic function in* `searchAgents.py` *and propose an admissible and consistent heuristic. Test it on trickySearch layout. On this layout, UCS explores more than 16000 nodes.*

- *Test with autograder* `python autograder.py`*. Your score depends on the number of expanded states by A* with your heuristic.*

Number of expanded nodes	*Grade*
more than 15000	*1/4*
at most 15000	*2/4*
at most 12000	*3/4*
at most 9000	*4/4*
at most 7000	*5/4*

For maximum score, you need to get in autograder 3 points for Question 4, 3 points for Question 5, 3 points for Question 6, and 4 points for Question 7.

Exercise 3.6 *For FoodSearch problem, compare the results of the application of your solutions on different layouts and different algorithms. You can also propose new layouts.*

Layout	*Algorithm*	*No of Expanded nodes*	*Path cost*	*Time*
tinySearch	*DFS*	*59*	*41*	*0.0s*
	UCS	***5057***	*27*	*3.1s*
	*A**	***974***	*27*	*0.6s*
trickySearch				
....				

3.4 Solutions to exercises

Solution for exercise 3.2.
Pay attention to

- the number of arguments of *aStarSearch* function
- the number of arguments of the heuristics: two heuristics are defined for *PositionSearchProblem* in *searchAgents.py* - Manhattan Heuristic and Euclidian Heuristic
- the real cost of a node from the initial state does not depend on the heuristic; only the path from the initial state to the goal state through that node depends on the heuristic
- if you already implemented DFS or UCS as graph search, for A^* the changes should relate mainly to the frontier

The answer should be yes for the question whether A^* finds the solution with fewer expanded nodes than UCS. For *bigMaze*, you could get 549 with A^* vs. 620 with UCS search nodes expanded, but these values can differ according to the order you put into the frontier the successor nodes.

Chapter 4

Adversarial search

Learning objectives for this week are:

1. To understand how optimal decision can be taken in multi-agent environment
2. To implement evaluation functions of a state in Pacman game with ghosts
3. To implement MINIMAX algorithm and Alpha-beta prunning

We will extend the Pacman scenario with multi-agents. That is, we will consider also the ghosts. Download the multiagent framework for Pacman from `https://s3-us-west-2.amazonaws.com/cs188websitecontent/projects/release/multiagent/v1/002/multiagent.zip`. The description of the project is at `http://ai.berkeley.edu/multiagent.html`.

Pacman can do five actions: *North*, *South*, *East*, *West*, and *Stop*. Pacman can eat power pellet and scare the ghosts: for a certain number of moves, the ghosts will be scared and Pacman can eat them. Rules for the score are: eating one food: +10 points, win/loose +500/-500 points, each time step: -1 point. All the rules are defined in class *PacmanRules* from `pacman.py`.

Exercise 4.1 *Read the comments in class GameState from* ***`pacman.py`****. The class specifies the full game state.*

Exercise 4.2 *Short python exercise: use list comprehension for computing the following sets:*

$$S = \{x^2 | x \in \{0...9\}\}$$

$$T = \{1, 13, 16\}$$

$$M = \{x | x \in \ S \ and \ x \in \ T \ and \ x \ even\}$$

4.1 Reflex agent

We create here a reflex agent which chooses at its turn a random action from the legal ones. Note that this is different from the random search agent, since a reflex agent does not build a sequence of actions, but chooses one action and executes it. The random reflex agent appears in listing 4.1.

Listing 4.1: Random reflex agent

```
1 class RandomAgent(Agent):
2
```

```
    def getAction(self, gameState):
        legalMoves = gameState.getLegalActions()
          # Pick randomly among the legal
        chosenIndex = random.choice(range(0, len(legalMoves)))
        return legalMoves[chosenIndex]
```

Exercise 4.3 *Add the class* `RandomAgent` *to* `multiagent.py`*. Run it with:*

```
python pacman.py -p RandomAgent -l testClassic
```

In some situations, Pacman wins, but most of the time it looses. A better reflex agent is already described in `multiagent.py`.

Exercise 4.4 *Run and analyze ReflexAgent from* `multiagent.py`*.*

```
python pacman.py -p ReflexAgent
python pacman.py -p ReflexAgent -l testClassic
```

Exercise 4.5 *Read the content of functions getAction, evaluationFunction, scoreEvaluation−Function.*

This reflex agent chooses its current action based only on its current perception. The *ReflexAgent* gets all its legal actions, computes the scores of the states reachable with these actions and selects the states that results into the state with the maximum score. In case more states have the maximum score, it will choose randomly one. The agent still looses many times.

Exercise 4.6 *Think about how you could improve the agent by using the variables newFood, newGhostState, newScaredTimes available for all the successor states. Print these variables in function evaluationFunction of the* `ReflexAgent`*.*

Exercise 4.7 *For each legal action, print the position of Pacman, the position of ghosts, and the distance between Pacman and the ghosts.*

Exercise 4.8 *Question 1. Improve the ReflexAgent such that it selects a better action. Include in the score food locations and ghost locations. The layout* `testClassic` *should be solved more often.*

- *Test your solution on testClassic layout.*

```
python pacman.py -p ReflexAgent -l testClassic
```

- *You can speed up the animation with the option frameTime. The number of ghost is given with option k.*

```
python pacman.py --frameTime 0 -p ReflexAgent -k 1 -l mediumClassic
python pacman.py --frameTime 0 -p ReflexAgent -k 2 -l mediumClassic
```

For an average evaluation function, the agent will loose for two ghosts.

- *Test your solution with* `autograder` *for* `Question 1`*.*

```
python autograder.py -q q1
```

or

```
python autograder.py -q q1 --no-graphics
```

Grading is computed as follows

Out of 10 runs on `openClassic` *layout:*	
0	*the agent times out or looses all the time*
1	*the agent wins at leat 5 times*
2	*the agent wins all 10 games*
+1	*the average score is* > 500
+2	*the average score is* > 1000

More details can be found at `http://ai.berkeley.edu/multiagent.html`.

4.2 Minimax algorithm

In case the world where the agent plans ahead includes other agents which plan against it, *adversarial search* can be used. One agent is called MAX and the other one MIN. $Utility(s, p)$ (called also payoff function or *objective function*) gives the final numeric value for a game that ends in terminal state s for player p. For example, in chess the values can be +1, 0, $\frac{1}{2}$. The *game tree* is a tree where the nodes are game states and the edges are moves. MAX's actions are added first. Then, for each resulting state, the action of MIN's are added, and so on. A game tree can be seen in figure 4.1.

Optimal decisions in games must give the best move for MAX in the initial state, then MAX's moves in all the states resulting from each possible response by MIN, and so on. Minimax value ensures optimal strategy for MAX. The algorithm for computing this value is given in listing 4.2. For the agent in figure 4.1, the best move is a_1 and the minimax value of the game is 3 (see chapter 5 from AIMA).

$MINIMAX(s) =$
=UTILITY(s) if TERMINAL-TEST(s)
$=max_{a \in Actions(s)}$ MINIMAX(RESULT(s,a)) if PLAYER(s)= MAX
$=min_{a \in Actions(s)}$ MINIMAX(RESULT(s,a)) if PLAYER(s)=MIN

Listing 4.2: Minimax algorithm

```
function MINIMAX–DECISION (state) returns an action
   return arg max_{a∈ACTIONS(state)} MIN–VALUE(RESULT(state, a))

function MAX–VALUE(state) returns a utility value
 if TERMINAL–TEST(state) then return UTILITY(state)
 v ← −∞
 for each a in ACTIONS(state) do
   v ← MAX(v, MIN–VALUE(RESULT(state, a)))
return v

function MIN–VALUE(state) returns a utility value
 if TERMINAL–TEST(state) then return UTILITY(state)
 v ← ∞
 for each a in ACTIONS(state) do
   v ← MIN(v,MAX–VALUE(RESULT(state, a)))
return v
```

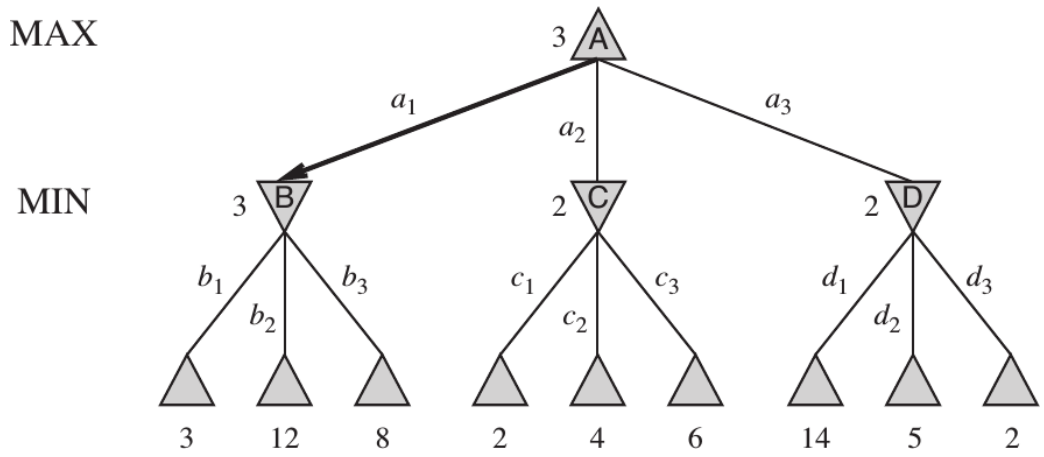


Figure 4.1: Minimax algorithm for two agents

Result(*state*, *a*) is the state which results from the application of action *a* in *state*. Minimax algorithm generates the entire game search space. Imperfect real-time decisions involve the use of cutoff test based on limiting the depth for the search. When the CUTOFF test is met, the tree leaves are evaluated using an heuristic evaluation function instead of the utility function.

H-MINIMAX(s,d) =
=EVAL(s) if CUTOFF-TEST(s, d)
$=max_{a \in Actions(s)}$ H-MINIMAX(RESULT(s,a), d+1) if PLAYER(s)= MAX
$=min_{a \in Actions(s)}$ H-MINIMAX(RESULT(s,a), d+1) if PLAYER(s)=MIN

Exercise 4.9 *Question 2. Implement H-Minimax algorithm in MinimaxAgent class from* `multiAgents.py`*. Since it can be more than one ghost, for each max layer there are one or more min layers.*

- *Test your implementation at certain depth and layouts:*

```
python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4
```

- *Test your implementation with* `autograder` *for* `Question 2`

```
python autograder.py -q q2
```

For more hints go to `http://ai.berkeley.edu/multiagent.html`.

Exercise 4.10 *Test Pacman on* `trappedClassic` *layout and try to explain its behaviour.*

```
python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3
```

Why Pacman rushes to the ghost? For random ghosts minimax behaviour could be improved.

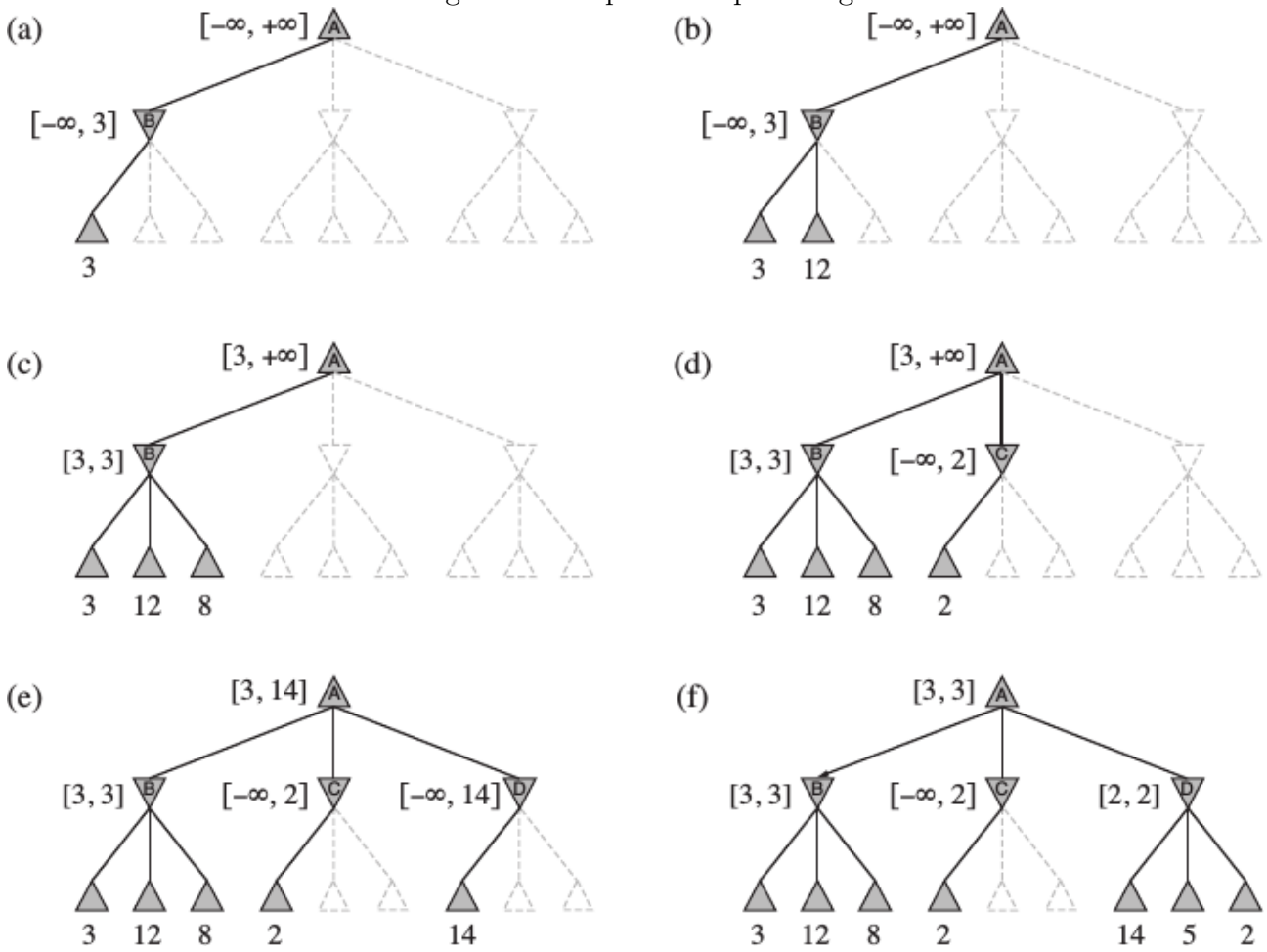
4.3 Alpha-beta prunning

In order to limit the number of game states from the game tree, alpha-beta $\alpha - \beta$ prunning can be applied, where

α = the value of the best (highest value) choice there is so far at any choice point along the path for MAX

β = the value of the best (lowest-value) choice there is so far at any choice point along the path for MIN

Figure 4.2: Alpha-beta prunning



Listing 4.3: Alpha-beta prunning.

```
function ALPHA–BETA–SEARCH (state) returns an action
  v ← MAX–VALUE(state , −∞,∞)
return the action in ACTIONS(state ) with value v

function MAX–VALUE(state , α,β) returns a utility value
if TERMINAL–TEST(state) then return UTILITY(state)
v ← −∞
for each a in ACTIONS(state) do
 v ← MAX(v, MIN–VALUE(RESULT(s,a), α,β))
 if v ≥ β then return v
 α ← MAX(α, v )
return v

function MIN–VALUE(state , α,β) returns a utility value
if TERMINAL–TEST(state) then return UTILITY(state)
v ← +∞
for each a in ACTIONS(state) do
 v ← MIN(v, MAX–VALUE(RESULT(s,a),α,β))
 if v ≤ α then return v
 β ← MIN(β, v )
return v
```

Exercise 4.11 *Question 3. Use alpha-beta prunning in AlphaBetaAgent from* `multiagents.py` *for a more efficient exploration of minimax tree.*

- Test your implementation of `smallClassic` layout. Similar to exercise 4.9, there is one MAX agent and possible more MIN agents. $alpha - beta$ prunning with depth 3 will run comparable to *minimax* at depth 2. On `smallClassic` the time should be at most a few seconds per move.

```
python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic
```

- Test your implementation with `autograder` for `Question 3`.

```
python autograder.py -q q3
```

or

```
python autograder.py -q q3 --no-graphics
```

Don't forget that the minimax value obtained with alpha beta prunning is the same with the value obtained for minimax algorithm (both at the same depth). One constraint given by the `autograder` is that you must not prune on equality. In theory, you can also allow for prunning on equality and invoke alpha-beta once on each child of the root node.

In order to obtain the maximum score you must obtain 14 for the first three questions.

4.4 Solutions to exercises

Solution for exercise 4.2.

```
1 s=[x**2 for x in range(0,9)]
2 t=[1,13,16]
3 m=[x for x in s if x % 2 == 0 and x in t]
4 s
5 #[0, 1, 4, 9, 16, 25, 36, 49, 64]
6 m
7 #[16]
```

Solution for exercise 4.7.
Add the following lines into *evaluationFunction* of *ReflexAgent*:

```
1  distanceToGhosts = [manhattanDistance(newPos, gp) for gp in
      successorGameState.getGhostPositions()]
2  print "New position", newPos
3  print "Ghost positions", successorGameState.getGhostPositions()
4  print "Distance to ghosts", distanceToGhosts
```

The *Grid* class (used for food) has the method *asList*() which returns a list of positions.

Observations for exercise 4.9
One Pacman move together with all ghost responses make one ply in the game tree. So a search of depth two involves Pacman and ghosts moving two times.
The function *getAction* from *MinimaxAgent* class returns the minimax action from the current game state with depth having the value $self.depth$. You must be able to limit the game tree to an arbitrary depth mentioned with option $-a\ depth = 4$. The depth is stored in $self.depth$ attribute. In order to give the score for the leaves of the minimax tree use $self.evaluationFunction$.
It is normal Pacman to lose in some cases. For layout *minimaxClassic* the minimax values of the initial state are: 9, 8, 7, −492 for depths 1, 2, 3, and 4.

Chapter 5

Propositional logic

This lab will begin presenting how proofs can be build automatically, starting from a set of axioms assumed to hold. Programs which achieve this are called automated theorem provers, or just provers for short. The one we will use to illustrate the idea is called Prover9 [18].

Learning objectives for this week are:

1. To see the structure of a Prover9 input file corresponding to a problem
2. To understand the output returned by the prover
3. To be able to explain, step by step, the proof obtained

5.1 Getting started with Prover9 and Mace4

Prover9 searches for proofs; Mace4, for counterexamples. The sentences they operate on could be written in propositional, first-order or equational logic. This first lab is concerned with sentences written solely in propositional logic. Here, we have true or false propositions, like P or Q, but no sentences of type $\forall x P(x)$.

Installing Prover9 and Mace4

Download the current command-line version of the tool (LADR-2009-11A), which is available at `https://www.cs.unm.edu/~mccune/mace4/download/LADR-2009-11A.tar.gz`. Unpack it, change directory to LADR-2009-11A, then type `make all` and follow the instructions on the screen.

Do not forget to add the `bin` folder to you `PATH`, such that you should be able to start Prover9 by simply typing `prover9` in a terminal, regardless your current directory.

5.2 First example: Socrates is mortal

Let us assume the following hold:

1. If someone's name is Socrates, then he must be a human.
2. Humans are mortal.
3. The guy over there is called Socrates.

Now try to prove that Socrates is a mortal.

The assumptions in this tiny, well known knowledge base could be represented formally using different types of logics. Right now, we employ the simplest one, called Propositional Logic, which comprises variable names for sentences plus a set of logical operators like $\wedge, \vee, \neg, \rightarrow, \leftrightarrow$.

In our example, we'll use the following set of sentences/propositions:

S: The guy over there is called Socrates

H: Socrates is a human

M: Socrates is mortal

The Prover9 input file containing the implementation is presented in Listing 5.1.

Listing 5.1: Knowledge on Socrates' mortal nature

```
1  assign(max_seconds, 30).
2  set(binary_resolution).
3  set(print_gen).
4
5  % 1. If someone's name is Socrates, then he must be a human.
6  % 2. Humans are mortal.
7  % 3. The guy over there is called Socrates.
8  % 4. Prove that Socrates is a mortal.
9
10 % S: the guy is called Socrates
11 % H: Socrates is a human
12 % M: Socrates is mortal
13
14
15 formulas(assumptions).
16  S.
17  S -> H.
18  H -> M.
19 end_of_list.
20
21 formulas(goals).
22 M.
23 end_of_list.
```

Exercise 5.1 *Type* `prover9 -f socrates.in` *in order to run Prover9 with* `socrates.in` *as an input file. Redirect the output to* `socrates.out` *and examine it. Have you obtained a proof of your goal? (the text* `Exiting with 1 proof` *or alike in the file/on the screen should indicate this).*

Input file explained

The input files for Prover9 comprise some distinct parts. In this Section, we will explain the meaning of each part in the input file for the example in Listing 5.1.

The first part contains some flags. For example, `assign(max_seconds, 30)` limits the processing time at 30 seconds, while `set(binary_resolution)` allows the use of the binary resolution inference rule (`clear(binary_resolution)` would do the opposite). Writing `set(print_gen)` instructs Prover9 to print all clauses generated while searching for the proof. For now, we don't focus on flags, just assume they are given in each exercise. Further, when we might get to fine tuning them, we'll use at the comprehensive description provided in the All Prover9 Options of the online manual (https://www.cs.unm.edu/ mccune/mace4/manual/2009-11A/).

Comment lines start with the % symbol.

The part between `formulas(assumptions)` and the corresponding `end_of_list` contains the actual knowledge base, i.e., the sentences which are assumed to be true. Sentence S means, as already mentioned, that the guy is called Socrates, while $S \rightarrow H$ says that if you are Socrates,

then you are a human. You can use − for negation, & for logical conjunction and | for logical disjunction. Each sentence (called formula) ends with a dot.

The part between `formulas(goals)` and the corresponding `end_of_list` state the goal the prover must demonstrate, namely M in our case.

Note: By default, Prover9 uses names starting with u, v, w, x, y, z to represent variables in clauses; thus, for the time being, please avoid using sentence names starting with them.

Output file explained

The output file produced (i.e., `socrates.out`) starts with some information on the running process, a copy of the input and a list of the formulas that are not in clausal form. Then, these clauses are processed according to the algorithm for transforming them into the Clausal Normal Form and we get:

```
4 S.  [assumption].
5 -S | H.  [clausify(1)].
6 -H | M.  [clausify(2)].
7 -M.  [deny(3)].
```

Please remember the equivalence between $P \to Q$ and $\neg P \vee Q$, which is used for instance in line 5. This is what `clausify` does. One should also notice in line 7 the goal has been added in the negated form $\neg M$. This is called *reductio ad absurdum*: if one both accepts the axioms and denies the conclusion, then a contradiction will be inferred. Prover9 actually searches for such a contradiction by repeatedly applying inference rules over the existing clauses till the empty clause is obtained.

The SEARCH section of the output lists all clauses inferred during search for the proof.

The PROOF section shows just those which actually helped in building the demonstration. For the example above:

```
1 S -> H # label(non_clause).  [assumption].
2 H -> M # label(non_clause).  [assumption].
3 M # label(non_clause) # label(goal).  [goal].
4 S.  [assumption].
5 -S | H.  [clausify(1)].
6 -H | M.  [clausify(2)].
7 -M.  [deny(3)].
8 H.  [resolve(5,a,4,a)].
9 -H.  [resolve(7,a,6,b)].
10 $F.  [resolve(9,a,8,a)].
```

Lines 1-4 are the original ones, 5 and 6 are the clausal forms of 1 and 2, while 7 is the goal denial. Line 8 shows how the resolution is applied over clauses 5 and 4 (Prover9 calls this "binary resolution"). Propositional resolution inference rules says that if we have $P \vee Q$ and $\neg Q \vee R$, we can infer $P \vee R$. The first literal in clause 5 ($-S$, hence the index a) and the first literal in clause 4 are "resolved" and the inferred clause is H. If P is absent, this inference rule is called "unit deletion", or, if R is absent, "back unit deletion". Line 10 shows the derived contradiction.

Exercise 5.2 *Add a clause specifying that Socrates is a philosopher (use sentence symbols P for "philosopher"). Run Prover9 again and take a look at the generated clauses and at the produced proof. Is the set of generated clauses different in this case? How about the proof?*

5.3 A more complex example: FDR goes to war

Given:

1. If your name is FDR, then you are a politician [1].
2. The name of the guy addressing the Congress is FDR.
3. A politician can be isolationist or interventionist [2]
4. If you are an interventionist, then you will declare war.
5. If you are an isolationist and your country is under attack, then you will also declare war
6. The country is under attack.

we intend to prove that war will be declared.

Exercise 5.3 *Implement the knowledge above, save it in the file* ***fdr-war.in*** *and use Prover9 to show that war will be declared. Then, try to prove that war will not be declared. Have you succeeded both times?*

Exercise 5.4 *Let us add now the following piece to our knowledge base: the country is not under attack (*`-attack`*). The knowledge base contains a contradiction now. Let's try to show that war will be declared. Then, try to prove that war will not be declared. Have you succeeded both times?*

As mentioned earlier, Prover9 seeks for a contradiction in the body of clauses which incorporates the initial knowledge base and the negated conclusion. If the initial knowledge base is consistent (with no contradiction), then the only source of contradiction would be the negate conclusion which has just been added. But, if the initial knowledge base already contains a contradiction, then Prover9 will eventually derive it, no matter what the added conclusion might be. So, in this case, everything would be provable. Mace4 can help to detect if there exists at least one model of a knowledge base, which would mean it is not contradictory. For example, if no `-attack` is present in the knowledge base, we can comment out the goal line and do `mace4 -c -f fdr-war.in` and, if we get a model of the knowledge base, then we can be sure it contains no contradiction. Listing 5.2 shows such a model.

Listing 5.2: A model for FDR knowledge base example

```
interpretation( 2, [number=1, seconds=0], [

        relation(attack, [ 1 ]),

        relation(declare_war, [ 1 ]),

        relation(fdr, [ 1 ]),

        relation(interventionist, [ 0 ]),

        relation(isolationist, [ 1 ]),

        relation(politician, [ 1 ])
]).
```

[1] During Franklin Delano Roosevelt (FDR)' Presidency, the American public was divided between interventionists and isolationists: people who supported, respectively rejected the US involvment in WWII.

[2] Usually you can't be both, but some people can.

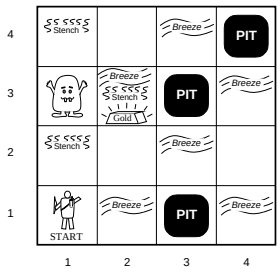


Figure 5.1: Wumpus world.

In the output produced by Mace4, `relation(attack [1])` means that in the model built, `attack` has value TRUE (`[0]` would mean FALSE). The current model number is specified by `number`.

Given a knowledge base KB and a goal G, the ideal situation is to have at least one model for KB and no model for $KB \wedge \neg G$. This means that G could be proved based on KB alone. If KB has no models, Pover9 can find a proof for both G and $-G$. On the other hand, if a model for $KB \wedge \neg G$ is found, then no proof for G can be build based on KB alone. In this latter case, we will say we found counterexample. Mace4 can do this job, maybe in the same time with Prover9 searching for a proof.

5.4 Wumpus world in Propositional Logic

A 4x4 grid contains one monster called wumpus and a number of pits [23]. Pits must be avoided and so should be the wumpus room as long as this is alive. We say a room is safe if you can find neither a pit, nor a live wumpus inside. Squares adjacent to wumpus are smelly, ans so is the square with the wumpus. Squares adjacent to a pit are breezy. Glitter iff gold is in the same square. Shooting kills wumpus if you are facing it. Shooting uses up the only arrow. Grabbing picks up gold if in same square. Releasing drops the gold in same square.

Exercise 5.5 *We will use the chessboard-like notation for this problem. Sentence $-Sij$ will mean there is no smell in the square on column i, line j. Let us assume the agent knows $-S11$, $-S21$ and $S12$. He also knows $-B11$, $-B12$, $B21$. Write down sentences to express "if there is a wumpus in (1,2), then there is a smell in (1,1)". Write down sentences to express "if I can smell something in (1,1), then there is a wumpus in (1,2) or in (2,1)". Can you prove the wumpus is in (1,3)?*

Exercise 5.6 *The agent is in the initial state (corner (1,1)) and perceives no smell and no breeze. No other perceptions are available, so you should drop all of them from your knowledge base. Can you prove now the following squares are safe for the agent to go there: (1,2); (2,1); (2,2)? Let's assume the agent goes in one of the squares he has proven to be safe and gets some perceptions from there. List the whole set of rooms which could be proven to be safe, given the new information available. Repeat the process until no safe room can be added.*

Exercise 5.7 *Write down a rule which says: a room is safe if there is no pit in it and there is no alive wumpus in it. Can you prove room (1,3) is safe, using only percepts obtained from safe rooms? Can you write enough rules for the agent to safely navigate among the whole castle, unti he gets the gold?*

5.5 Solutions to exercises

Solution to exercise 5.2 Yes - P is also inferred, even if not relevant. No - the proof remains the same, as only some of the generated sentences are part of it and P is not among them.

Solution to exercise 5.3 You should be able to automatically prove the sentence "War will be declared", but not the sentence "War will not be declared". The message "SEARCH FAILED" and "Exiting with failure" should be seen on the screen. That means Prover9 has stopped searching and failed to find a proof.

Solution to exercise 5.5

Listing 5.3: First wumpus problem

```
1  assign(max_seconds, 30).
2
3  formulas(assumptions).
4  -S11.
5  -S21. %colon 2, line 1
6  S12.
7  -B11.
8  B21.
9  -B12.
10 W11 -> S11.
11 W11 -> S12.
12 W11 -> S21.
13 W12 -> S12.
14 W12 -> S11.
15 W12 -> S22.
16 W12 -> S13.
17 W21 -> S21.
18 W21 -> S11.
19 W21 -> S22.
20 W21 -> S13.
21 W22 -> S22.
22 W22 -> S21.
23 W22 -> S12.
24 W22 -> S23.
25 W22 -> S32.
26 W13 -> S13.
27 W13 -> S14.
28 W13 -> S23.
29 W13 -> S12.
30 W31 -> S31.
31 W31 -> S21.
32 W31 -> S32.
33 W31 -> S41.
34 S12 -> W11 | W22 | W13 | W12.
35 end_of_list.
36
37 formulas(goals).
38 W13.
39 end_of_list.
```

Chapter 6

Models in propositional logic

The problem of efficiently finding models for formulas is present in many fields belonging to Computer Science and particularly in Artificial Intelligence.

Learning objectives for this week are:

1. To understand how to encode knowledge in a form suitable for Prover9 and Mace4
2. To see how obtaining models for a given set of clauses is connected with building proofs

Henceforth, we will use for our experiments a knowledge base named KB, comprising one or more clauses connected by a logical AND. We will focus on testing whether KB has at least one model (a set of assignments of truth values to its propositional variables which make it true) and whether a given goal sentence g could be proven based on the KB content.

We say a proposition is $satisfiable$ if it has at least one model; otherwise, it is $unsatisfiable$. A proposition is *valid* if it is true no matter the truth values assigned to its components.

6.1 Formalizing puzzles: Princesses and tigers

We will start with formalizing a logical puzzle described in [22], who borrowed it from [24].

A prisoner is given by the King holding him the opportunity to improve his situation if he solves a puzzle. He is told there are three rooms in the castle: one room contains a lady and the other two contain a tiger each. If the prisoner opens the door to the room containing the lady, he will marry her and get a pardon. If he opens a door to a tiger room though, he will be eaten alive. Of course the prisoner wants to get married and be set free than being eaten alive.

The door of each room has a sign bearing a statement that may be either true or false. The sign on the door of the room containing the lady is true and at least one of the signs on the doors of the rooms containing tigers is false. The signs say respectively:

- (The sign on the door of room #1): There is a tiger in room #2.
- (The sign on the door of room #2): There is a tiger in this room.
- (The sign on the door of room #3): There is a tiger in room #1.

Exercise 6.1 *At a job interview, you are asked to solve this puzzle. Can you tell in which room the lady is?*

Let's try to formalize what we know as an input for Prover9/Mace4, except won't specify the goal we intend to prove (a.k.a. the conjecture). Once done this, we will use Mace4 to see how many models it can find. If we obtain just one model (which is usually the case when dealing with this type of puzzles), we can see the solution immediately; for example, we will know the lady is in room #1. Then, we can add this finding as a goal to the input file and use Prover9 to build a proof for it. If we end up with more than one model, there is still hope: either in all of them the lady is in the same room, or we can ask for more clues.

Exercise 6.2 *How can we express the fact that there exists a lady in the castle? Write this information alone in the* `oneLadyTwoTigers-step1.in` *file, then call Mace4 to generate as many models as it can for this tiny KB. The command you need is:* `mace4 -c -n 2 -m -1 -f oneLadyTwoTigers-step1.in | interpformat`*. The* `-f` *option specifies the input file for Mace4;* `-m -1` *tells to generate as many models as possible;* `-n 2` *specifies a model complexity measure, while* `-c` *asks Mace4 (for compatibility purposes) to ignore Prover9 options it does not understand. The* `interpformat` *command is used here for pretty printing the models. How many models do you get? How come?*

Now, we will tell the system the lady is unique (i.e., there is only one lady in the castle). The knowledge base looks like in listing 6.1:

Listing 6.1: There is precisely one lady in the castle.

```
formulas(assumptions).
 %there is a lady in room 1, 2 or 3
 l1 | l2 | l3.
 %no lady in more than 1 room; the princess is unique
 l1 -> -l2.
 l1 -> -l3.
 l2 -> -l1.
 l2 -> -l3.
 l3 -> -l1.
 l3 -> -l2.
end_of_list.

formulas(goals).
end_of_list.
```

Exercise 6.3 *How many models for the new KB does Mace4 produce now?*

We go on by specifying that there are 2 tigers in the castle, in separate rooms, and there is no tiger in the room where the lady stays (see listing 6.2).

Listing 6.2: The lady and the tigers stay in separate rooms.

```
formulas(assumptions).
 %there is a lady in room 1, 2 or 3
 l1 | l2 | l3.
 %no lady in more than 1 room; the princess is unique
 l1 -> -l2.
 l1 -> -l3.
 l2 -> -l1.
 l2 -> -l3.
 l3 -> -l1.
 l3 -> -l2.

 %there are 2 tigers in two of the rooms 1, 2 or 3
 t1 & t2 | t2 & t3 | t1 & t3.
```

```
14
15  %no tiger in the room where the lady stays
16  l1 -> -t1.
17  l2 -> -t2.
18  l3 -> -t3.
19  %do we also have to write t1 -> -l1 and alike, or do we already have that?
20
21 end_of_list.
22
23 formulas(goals).
24 end_of_list.
```

Exercise 6.4 *We wrote* `l1 -> -t1` *in order to say "if the lady is in room #1, then there is no tiger in the first room". Do we also have to write* `t1 -> -l1` *in order to say "tiger in room #1 means no lady in room #1"? Why (not)?*

So far, we have formalized the common sense knowledge about the field. Although obvious, this knowledge is vital for the systems to be able to build models and to exclude some of those inconsistent with the problem statement. Further, we will encode the clues on the doors. This part requires a certain effort from the programmer as in Propositional Logic we are not allowed to use sentences *about some other sentences*: we can't simply say "if John is from country C, then, for every sentence P John says, $\neg P$ is true".

Thus, we will need to interpret the meaning of the sentence on each door and tell Prover9 the result of this interpretation. As an example, if we assume the princess is in room #3, then the clue on the door is true, so we jump to the conclusion a tiger is present in room #1. What we will state is precisely this: `l3 -> t1`. The full input file is given in listing 6.3:

Listing 6.3: One lady and two the tigers: full KB.

```
1  formulas(assumptions).
2  %there is a lady in room 1, 2 or 3
3  l1 | l2 | l3.
4  %no lady in more than 1 room; the princess is unique
5  l1 -> -l2.
6  l1 -> -l3.
7  l2 -> -l1.
8  l2 -> -l3.
9  l3 -> -l1.
10 l3 -> -l2.
11
12 %there are 2 tigers in two of the rooms 1, 2 or 3
13 t1 & t2 | t2 & t3 | t1 & t3.
14
15 %no tiger in the room where the lady stays
16 l1 -> -t1.
17 l2 -> -t2.
18 l3 -> -t3.
19
20
21 %clue on door #1: there is a tiger in room #2
22 l1 -> t2.
23
24 %clue on door #2: there is a tiger here
25 l2 -> t2.
26
27 %clue on door #3: there is a tiger in room #1
28 l3 -> t1.
29
```

```
30 %at least one of the clues on tiger rooms lies.
31 (t1 & t2) -> (-t2 | -t2).
32 (t2 & t3) -> (-t2 | -t1).
33 (t1 & t3) -> (-t2 | -t1).
34
35 end_of_list.
36
37 formulas(goals).
38 end_of_list.
```

Exercise 6.5 *Ask Mace4 to generate all models. How many do you get? Is the lady in the same room in all of them? If so, add the position of the lady to the input file as a conjecture and ask Prover9 to prove it. Take a look at the proof.*

Exercise 6.6 *Let's drop the assumption that at least one of the clues on the tiger room lies. Ask Mace4 to generate all models in this situation. How many do you get? Is the lady in the same room in all of them?*

Let us assume we have one room containing the lady, one containing a tiger and one empty room. If the prisoner opens the door to the room containing the lady, he will marry her and get a pardon. If he opens a door to a tiger room though, he will be eaten alive. If he opens the door of the empty room, he will go on with staying in prison.

The sign on the door of the room containing the lady is true. The sign on the door of the room containing tigers is false. We don't know anything about the falsehood of the sign on the empty room The signs say respectively:

- (The sign on the door of room #1): Room #3 is empty.
- (The sign on the door of room #2): The tiger is in room #1.
- (The sign on the door of room #3): This room is empty.

Exercise 6.7 *Do you know in which room the lady is now?*

Exercise 6.8 *Implement the puzzle and repeat all tasks stated in exercise 6.5.*

6.2 Finding more models: graph coloring

Typically, puzzles of this sort have a unique solution: we have just one model consistent with all conditions. We'll explore now situations when many more models exist. Consider for example the task of coloring each node of a graph with one color from a given set, such that neighbor nodes have different colors. We will model this as a set of sentences in Propositional Logic (a knowledge base) and employ Mace4 once again for finding models of it.

Exercise 6.9 *Let us assume we are given a set of 4 nodes a, b, c, d connected by a total of 5 edges, with the edge (b, c) being the only one missing from the 4-nodes clique. We are also given a set of 3 colors $\{Red, Green, Blue\}$. Write down sentences in Propositional Logic to formalize the coloring restrictions described above, then ask Mace4 to find all models for the knowledge base. You should carefully express the following:*

- *each node has assigned at least a color*
- *no node can have more than one color*

- *neighbors cannot have the same color*

How many models do you get?

Exercise 6.10 *Do the same tasks as in exercise 6.9 for a graph with 10 nodes, using 6 colors. The edge set is up to you. Test the number of models produced and the running time.*

6.3 Solutions to exercises

Solution to exercise 6.1: The lady is in the first room.

Solution to exercise 6.2:

Listing 6.4: Step 1: there are some ladies in the castle

```
1 formulas(assumptions).
2  %there is a lady in room 1, 2 or 3
3  l1 | l2 | l3.
4 end_of_list.
5
6 formulas(goals).
7  %l2.
8 end_of_list.
```

You should get 7 models. So far, nobody tells the system there is only one lady. All we know is there exists at least one.

Solution to exercise 6.3: 3 models: lady in room #1, or in room #2, or #3.

Solution to exercise 6.4: No. We have already expresses that. Check that $p \to q$ and $\neg q \to \neg p$ are equivalent, then use the equivalence between $p \to q$ and $\neg p \vee q$.

Solution to exercise 6.5: 1 model: lady in room #1.

Solution to exercise 6.6: 2 models: `l1,t2,t3` and `t1,t2,l3` respectively.

Solution to exercise 6.7: The lady is once again in the first room.

Solution to exercise 6.8:

Listing 6.5: One lady and one tiger plus an empty room

```
assign(max_seconds, 30).

formulas(assumptions).

 %there is a lady in room 1, 2 or 3
 l1 | l2 | l3.
 %no lady in more than 1 room; the princess is unique
 l1 -> -l2.   l1 -> -l3.   l2 -> -l1.
 l2 -> -l3.   l3 -> -l1.   l3 -> -l2.

 %there is 1 tiger in one of the rooms 1, 2 or 3
 t1 | t2 | t3.

 %no tiger in more than 1 room; the tiger is unique as well
 t1 -> -t2.   t1 -> -t3.   t2 -> -t1.
 t2 -> -t3.   t3 -> -t1.   t3 -> -t2.

 %we have one empty room
 e1 | e2 | e3.
 %but no more than one
 e1 -> -e2.   e1 -> -e3.   e2 -> -e1.
 e2 -> -e3.   e3 -> -e1.   e3 -> -e2.

 %no tiger in the room where the lady stays
 l1 -> -t1.   l2 -> -t2.   l3 -> -t3.

 %the room where the lady stays is not empty
 l1 -> -e1.   l2 -> -e2.   l3 -> -e3.

 %the room where a tiger stays is not empty
 t1 -> -e1.   t2 -> -e2.   t3 -> -e3.

 %the clue on the lady's room is true; on the tiger's room is false
 %the clue on the empty room is either false or true

 %clue on door #1: room #3 is empty
 l1 -> e3.    t1 -> -e3.

 %clue on door #2: the tiger is in room #1
 l2 -> t1.    t2 -> -t1.

 %clue on door #3: this room is empty
 l3 -> e3.    t3 -> -e3.
end_of_list.

formulas(goals).
 %l1.
end_of_list.
```

Solution to exercise 6.9:

Listing 6.6: Graph coloring: 4 nodes and 5 edges

```
formulas(assumptions).
 %each node has assigned at least a color
 a_Red | a_Green | a_Blue.
 b_Red | b_Green | b_Blue.
 c_Red | c_Green | c_Blue.
 d_Red | d_Green | d_Blue.

 %no node can have more than one color
 a_Red -> -a_Blue.
 a_Red -> -a_Green.
 a_Green -> -a_Blue.

 b_Red -> -b_Blue.
 b_Red -> -b_Green.
 b_Green -> -b_Blue.

 c_Red -> -c_Blue.
 c_Red -> -c_Green.
 c_Green -> -c_Blue.

 d_Red -> -d_Blue.
 d_Red -> -d_Green.
 d_Green -> -d_Blue.

 %neighbors cannot have the same color
 a_Red -> -b_Red.  a_Red -> -c_Red.
 a_Red -> -d_Red.  a_Green -> -b_Green.
 a_Green -> -c_Green.  a_Green -> -d_Green.
 a_Blue -> -b_Blue.  a_Blue -> -c_Blue.
 a_Blue -> -d_Blue.

 b_Red -> -d_Red.  b_Blue -> -d_Blue.
 b_Green -> -d_Green.

 c_Red -> -d_Red.  c_Blue -> -d_Blue.
 c_Green -> -d_Green.
end_of_list.

formulas(goals).
end_of_list.
```

Chapter 7

First Order Logic

It might be tedious to model reasonably large problems in Propositional Logic (PL from now on), as you need to write a huge amount of propositions to encode the available knowledge. First Order Logic (FOL in short) offers you some more operators, which allow a more compact way of writting the same knowledge.

Learning objectives for this week are:

1. To get familiar with encoding knowledge in FOL for Prover9.
2. To see how you can deal with ordered sets in FOL, without using the set of natural numbers.

One drawback of Propositional Logic it that it does not allow you to write general enough sentences. For example, you cannot simply write something like "If there is a pit in square (i, j), then you can feel breeze in square (i, j)". If both i and j are in the range 1..4, Propositional Logic will force you to write 16 sentences of the type $P23 \rightarrow B23$. This makes the knowledge base to grow rapidly and to become unmanageable.

First Order Logic addresses this by allowing the $\forall$ (`forall`) quantifier. All the PL sentences in the example above could now be collapsed into just one FOL proposition:

$$\forall\ i, j : pit(i, j) \rightarrow breeze(i, j)$$

Here, *pit* and *breeze* are two predicates, each of arity 2, which are true iff the room number (i, j) contains a pit or a breeze respectively. The arguments i and j are called *bound* variables. They are assumed to belong to a known domain (for example, here the domain is the set $\{1, 2, 3, 4\}$ for each of them).

7.1 First Example: Socrates in First Order Logic

We will formalize once again the knowledge about Socrates in Lab #5, but this time, we'll do it in FOL. Please recall that:

1. Socrates is a human.
2. All humans are mortal.

Our target is to prove that Socrates is a mortal.

First, we need to write down the constants involved, which reffer particular objects in the domain of discourse. We have here just one person, namely Socrates, so we use all in all one constant: `Socrates`.

Then, we need to identify the predicates we might need. One of them would be predicate `human(x)`, which is true iff the object x is a human being. Another predicate which we will need is `mortal(x)`, which is true iff its argument x is mortal.

In both cases, we assume the domain of value for x is known. For our example, this could be the set of all creatures who are alive. Hence, `human(Socrates)` is true, but `human(Viserion)` [1] or `human(Incitatus)` [2] are false. Similarly, `mortal(Socrates)` is true (as we will prove soon), while `mortal(Zeus)` is arguably false.

Given these, the Prover9 implementation is given in listing 7.1.

Listing 7.1: Socrates KB in FOL.

```
set(binary_resolution).
set(print_gen).

% 1. Socrates is a human.
% 2. All humans are mortal.
% 3. Prove that Socrates is a mortal.

% constant: Socrates
% predicate human(x): x is a human
% predicate mortal(x): x is mortal


formulas(assumptions).
 human(Sokrates).
 human(x) -> mortal(x).
end_of_list.

formulas(goals).
 mortal(Sokrates).
end_of_list.
```

Exercise 7.1 *Run the example in listing 7.1, which is provided in file* `socratesFOL.in`*. Look at the output and try to understand the transformations performed by Prover9 over the input before the search for a proof begins. Carefully identify the original form and the transformed form for implications.*

Now focus on the PROOF section and follow, step by step, the produced proof. Try to understand the meaning of the following marks: `assumption, clausify, goal, resolve, deny`*.*

There are two conventions for naming the constants and variables involved in the predicates. The default convention states that the names of the variables in clauses start with (lower case) 'u' through 'z'. But if you set the flag called `prolog_style_variables`, they will start with (upper case) 'A' through 'Z', as in Prolog. Setting the flag is achieved by adding the following line to your input file: `set(prolog_style_variables).`

We will use the Prolog convention once, in the next example. Then, for the rest of this lab, we will stick with the former convention.

7.2 FDR goes to war in FOL

We try now to represent a slightly larger knowledge base in FOL: the statements about FDR in lab #5. We know that:

[1] One of the dragons of Daenerys Targaryen from The Game of Thrones

[2] The horse of the Roman emperor Caligula, whom he appointed a priest

1. FDR is a politician.
2. A politician could be an isolationist or an interventionist (maybe both!).
3. An interventionist would declare war.
4. If US is under attack, even an isolationist would declare war.
5. US is under attack.

We employ the following set of constants and predicates:

- constant: `fdr` (meaning FDR)
- constant: `country_us` (meaning US)
- predicate: `under_attack(x)` (meaning country x is under attack)
- predicate: `politician(x)` (meaning x is a politician)
- predicate: `interventionist(x)` (meaning x is an interventionist)
- predicate: `isolationist(x)` (meaning x is an isolationist)
- predicate: `declare_war(x)` (meaning x will declare war)

The Prover9 implementation is given in listing 7.2.

Listing 7.2: FDR declares war - FOL version.

```
% FDR is a politician.
% A politician could be an isolationist or an interventionist (maybe both!).
% An interventionist would declare war.
% If US is under attack, even an isolationist would declare war.
% US is under atack.

% constant: fdr (meaning FDR)
% constant: country_us (meaning US)
% predicate: under_attack(x) (meaning country x is under attack)
% predicate: politician(x)  (meaning x is a politician)
% predicate: interventionist(x) (meaning x is an interventionist)
% predicate: isolationist(x) (meaning x is an isolationist)
% predicate: declare_war(x) (meaning x will declare war)

formulas(assumptions).
 under_attack(country_us).
 politician(fdr).
 politician(x) -> isolationist(x) | interventionist(x).
 interventionist(x) -> declare_war(x).
 isolationist(x) & under_attack(country_us) -> declare_war(x).
end_of_list.

formulas(goals).
 declare_war(fdr).
end_of_list.
```

Exercise 7.2 *Run the example in the file* ***fdr-warFOL.in****, which contains the example in listing 7.2 and repeat the tasks specified in exercise 7.1.*

In listing 7.3 you can see the same problem modeled using the backward implication operator (which is similar to the `:-` symbol in Prolog). Please note the name of variables and constants, which follow the Prolog convention.

Listing 7.3: FDR declares war - FOL version with backward implications.

```
% FDR is a politician.
% A politician could be an isolationist or an interventionist (maybe both!).
% An interventionist would declare war.
% If US is under attack, even an isolationist would declare war.
% US is under atack.

% constant: fdr (meaning FDR)
% constant: us (meaning US)
% predicate: under_attack(x) (meaning country x is under attack)
% predicate: politician(x)  (meaning x is a politician)
% predicate: interventionist(x) (meaning x is an interventionist)
% predicate: isolationist(x) (meaning x is an isolationist)
% predicate: declare_war(x) (meaning x will declare war)

formulas(assumptions).
 under_attack(us).
 politician(fdr).
 (isolationist(P) | interventionist(P)) <- politician(P).
 declare_war(X) <- interventionist(X).
 declare_war(X) <- isolationist(X) & under_attack(us) .
end_of_list.

formulas(goals).
 declare_war(fdr).
end_of_list.
```

Now, listing 7.4 shows you how some predefined operators of Prover9 could be redefined. We did this with 3 operators, in order to make them look similar with their Prolog counterparts.

For instance, the line

```
redeclare(backward_implication, "COLON_MINUS").
```

means the usual operator for the backward implication, i.e., $\leftarrow$, is replaced by the string `"COLON_MINUS"`, which resembles the `:-` construction in Prolog (simply redefining $\leftarrow$ as `:-` failed as the colon character is used in Prover9 for the cons operator). Now, instead of writting `a(x) <- b(x)`, we'll have to write `a(x) COLON_MINUS b(x)`.

In line with this, the disjunction operator (`|`) was replaced "SEMICOLON" and the conjunction operator (`&`) became "COMMA".

Listing 7.4: FDR declares war - Prolog-style syntax.

```
% FDR is a politician.
% A politician could be an isolationist or an interventionist (maybe both!).
% An interventionist would declare war.
% If US is under attack, even an isolationist would declare war.
% US is under atack.

% constant: fdr (meaning FDR)
% constant: us (meaning US)
% predicate: under_attack(x) (meaning country x is under attack)
% predicate: politician(x)  (meaning x is a politician)
% predicate: interventionist(x) (meaning x is an interventionist)
% predicate: isolationist(x) (meaning x is an isolationist)
% predicate: declare_war(x) (meaning x will declare war)
```

```
14
15 set(prolog_style_variables).
16 redeclare(backward_implication, "COLON_MINUS").
17 redeclare(disjunction, "SEMICOLON").
18 redeclare(conjunction, "COMMA").
19
20 formulas(assumptions).
21  under_attack(us).
22  politician(fdr).
23  (isolationist(P) SEMICOLON interventionist(P)) COLON_MINUS politician(P).
24  declare_war(X) COLON_MINUS interventionist(X).
25  declare_war(X) COLON_MINUS isolationist(X) COMMA under_attack(us) .
26 end_of_list.
27
28 formulas(goals).
29  declare_war(fdr).
30 end_of_list.
```

Exercise 7.3 *Run the example in the file* `fdr-warFOL-Prolog.in`*, which contains the example in listing 7.4 and make sure you got one proof. Extract the clauses in the* `assumptions` *part, replace the three string-type operators with their Prolog versions (e.g.,* `COLON_MINUS` *with* `:-` *and alike) and save the them in a file called* `fdr.pl`*. Could you load the* `fdr.pl` *file in Prolog and ask if FDR will declare war? Will you get the same answer as in Prover9? Why (not)?*

7.3 Alligator and Beer: finding models in FOL

The problem we will model is a variation of the puzzle widely known as Einstein's riddle. Surely Uncle Google does the best he can to help you find the solution. It's time to ask Mace4 to do the same. For this instance of the problem, the latter is probably faster than the former (both implementation and running time included).

There are five houses in a row: a, b, c, d and e. Each has a different color (amber, beige, cyan, denim, emerald) and an owner of different nationality. The owners have differrent cars, pets and preffer different drinks. We know that:

1. The Austrian lives in the amber house.
2. Cider is drunk in the middle house.
3. The Belgian owns the bulldog.
4. The Czech lives in the first house on the left.
5. The man who owns a Dacia lives next to the man with the eagle.
6. The Czech lives next to the denim house.
7. The Bugatti owner has a cat.
8. The person in the beige house drives a Cadillac.
9. Advocaat is drunk in the cyan house.
10. The Dane drinks eiswein.
11. The Estonian drives an Edonis.

12. The Cadillac is always parked in front of the house next to the one with a donkey.
13. The cyan house is immediately to the right of the emerald house.
14. The Aston Martin owner drinks daiquiri.

We want to know:

- Where is the alligator?
- Who drinks beer?

Exercise 7.4 *Can you answer the two questions?*

We want to model this problem in FOL, then to ask Mace4 to find a model. To this end, we will use 5 constants, namely {`a, b, c, d, e`} to denote each of the five houses. For the other information, we are going to use several predicates of arity one. Hence, `amber(x)` will mean that house `x` has color amber (where `x` is either `a, b, c, d` or `e`), while `austrian(x)` will mean the person in house `x` is of Austrian nationality.

We must model the common sense knowledge about this problem (e.g., to teach Prover9 what "neighbor" means), then to tell that each house has at least one color, but no more. In the first attempt, we will do this without using numbers, just FOL constructs.

Then, we will encode all clues using the predicates mentioned above.

Eventually, we will save everyting in the file `alligatorBeer1.in` and call Mace4 with command `mace4 -c -f alligatorBeer1.in | interpformat`. The model produced should give us the solution.

Exercise 7.5 *Using a predicate `differentFrom(x,y)`, write down enough sentences to express the idea that `a,b,c,d,e` are distinct from one another. You should come up with a couple of clauses like `differentFrom(a,b)`. Then, write a clause to say the "differentFrom" relation is symmetrical.*

Exercise 7.6 *Using a predicate `rightneighbor(x,y)`, write down enough sentences to express the fact that "b is immediately to the right of a" for every pair in `a,b,c,d,e`. You should come up with a couple of clauses like `rightneighbor(a,b)`. Then, write clauses to say that "it is not the case that `rightneighbor(a,b)`". Complete the rest of the exercises and try to figure out whether the clause `rightneighbor(a,b)` is necessary.*

Exercise 7.7 *Define a predicate `neighbor(x,y)` which says that you are the neighbor of someone either if you live just to his right or he lives just to your right (i.e., you live just to his left).*

Exercise 7.8 *Write down clauses to express that each house has at least one nationality, pet, drink, color, car. E.g., `austrian(x) | belgian(x) | czech(x) | dane(x) | estonian(x).`*

Exercise 7.9 *Specify that each property applies to at most one house. How can you do that?*

Exercise 7.10 *Now try to model the rest of the clues. You may want to write down clauses like `austrian(x)` $\leftrightarrow$ `amber(x).` Call Mace4 to generate the models (hopefully there is only one of them) and explore them in order to extract the required answers.*

7.4 Solutions to exercises

Hint for exercise 7.3: think about Horn clauses.

Solution for exercise 7.4: The Estonian. The Czech.

Hint for exercise 7.9: try to model this information in the following manner: "if we have two houses x and y and both are inhabited by an Austrian, then x and y must be the same house".

Solution for exercise 7.5 - 7.10.
See file `alligatorBeer1.in`.

Chapter 8

Inference in First-Order Logic

Adding numbers and an object equality operator would help Prover9 express more easily some knowledge. Some more inference rules are needed in order to deal with these new elements.

Learning objectives for this week are:

1. To get familiar with resolution and the way clauses are preprocessed before actually performing it
2. To see how basic arithmetic is managed in Prover9
3. To get familiar with paramodulation inference rule

8.1 Revisiting resolution and skolemization

Resolution

Recall the task of proving colonel West is a criminal as stated in [23], page 336.

The law says it is a crime for an American to sell weapons to hostile nations. The country Nono, an enemy of America, has some missiles. All of its missiles were sold to it be Colonel West, who is American.

We possess the following background knowledge: missiles are weapons; enemies of America are hostile.

Exercise 8.1 *Implement in Prover9 the problem about colonel West and prove he is a criminal. For every predicate, specify the meaning of each of its arguments.*

In the proof produced, we have:

```
 8 -Missile(x) | -Owns(Nono,x) | Sells(West,x,Nono).  [clausify(2)].
9 Owns(Nono,M1).  [assumption].
...
12 -Missile(M1) | Sells(West,M1,Nono).  [resolve(8,b,9,a)].
```

Clause 12 is obtained by performing a resolution over clauses 8 and 9. Here, `8,b` means the second part of clause number 8, namely `-Owns(Nono,x)` (clause parts are separated by `|`). The second sentence needed by the resolution is the first (and only) part of 9, hence the `9,a` notation. The result is clause 12.

Exercise 8.2 *Try to follow and explain at least one more resolution step in the proof.*

Understanding Skolemization

Now let's focus on Curiosity problem ([23], page 354). We know that:

1. Everyone who loves all animals is loved by someone.
2. Anyone who kills an animal is loved by no one.
3. Jack loves all animals.
4. Either Jack or Curiosity killed the cat, who is named Tuna.
5. Cats are animals. (background knowledge)

Question: Did Curiosity kill the cat?

Prover9 uses `all` and `exists` for the logical $\forall$ and $\exists$ respectively. By default, variables in a clause are universally quantified, so `all` could be omitted. Please be careful to the quantifier scoping: parentheses could be of great help. The best practice is probably to write `exists x ( )` and just after that to fill in the spaces inside the pparentheses

Exercise 8.3 *Implement in Prover9 the problem about who killed the cat. Try to prove the goal "Curiosity killed Tuna". Now change the goal to "There exists someone who killed Tuna" and try to see who this is. Do you get the correct answer?*

Exercise 8.4 *In the proof produced, we have:* `Animal(f1(x)) | Loves(f2(x),x)`. *What do* `f1(x)` *and* `f2(x)` *mean?*

8.2 Paramodulation and demodulation

We consider as an example a part of a royal family. Elizabeth II is the mother of two sons: Charles and Andrew. Charles is the father of William and Harry. Elizabeth II is the Queen of UK. Prince William has become Duke of Cambridge on his wedding day.

The goal is to prove the Queen is the grandparent of the Duke of Cambridge. The implementation is given in listing 8.1.

Listing 8.1: The Queen and the Duke of Cambridge

```
set(paramodulation).

% constants: ElizabethII,
% predicate Mother(x,y): x is motehr of y

formulas(assumptions).
  Mother(ElizabethII, Charles).
  Mother(ElizabethII, Andrew).
  Father(Charles, William).
  Father(Charles, Harry).
  Father(x,y) | Mother(x,y) -> Parent(x,y).
  Parent(x,y) & Parent(y,z) -> Grandparent(x,z).
  William = DukeOfCambridge.
  Queen = ElizabethII.
end_of_list.

formulas(goals).
  Grandparent(Queen,DukeOfCambridge).
end_of_list.
```

Listing 8.2: The Queen and the Duke of Cambridge - the proof

```
1 Father(x,y) | Mother(x,y) -> Parent(x,y) # label(non_clause).  [assumption].
2 Parent(x,y) & Parent(y,z) -> Grandparent(x,z) # label(non_clause).  [assumption].
3 Grandparent(Queen,DukeOfCambridge) # label(non_clause) # label(goal).  [goal].
```

```
4  4 -Mother(x,y) | Parent(x,y).  [clausify(1)].
5  5 Mother(ElizabethII,Charles).  [assumption].
6  7 -Father(x,y) | Parent(x,y).  [clausify(1)].
7  8 Father(Charles,William).  [assumption].
8  10 -Parent(x,y) | -Parent(y,z) | Grandparent(x,z).  [clausify(2)].
9  11 William = DukeOfCambridge.  [assumption].
10 12 DukeOfCambridge = William.  [copy(11),flip(a)].
11 13 Queen = ElizabethII.  [assumption].
12 14 -Grandparent(Queen,DukeOfCambridge).  [deny(3)].
13 15 -Grandparent(ElizabethII,William).  [copy(14),rewrite([13(1),12(2)])].
14 16 Parent(ElizabethII,Charles).  [resolve(4,a,5,a)].
15 18 Parent(Charles,William).  [resolve(7,a,8,a)].
16 20 -Parent(Charles,William).  [ur(10,a,16,a,c,15,a)].
17 21 $F.  [resolve(20,a,18,a)].
```

Run the example and take a look at the proof, which is copied in Listing 8.2. Clause 15 (`15 -Grandparent(ElisabethII,William). [copy(14),rewrite([13(1),12(2)])].`) is obtained from clause 14 by applying demodulation (aka rewriting), using clause 13 and 12. As stated in the Prover9 documentation [18], demodulation is the process of using a set of oriented equations to rewrite (simplify, canonicalize) terms. The first part, `[13(1)`, tells that the left side of clause 13 (i.e., `Queen`) gets replaced by its right side (`ElizabethII`) inside clause 14. The part `12(2)` says that `DukeOfCambridge` is replaced by `William`. The numbers 1 and 2 indicate the position of rewriting (although this is not fully documented in Prover9).

The ability to deal with equality adds some power to Mace4. Another improvement would be the possibility to do simple arithmetic. By doing `set(arithmetic)`, you give Mace4 access to operators like `+,*,/` or `<.>,<=,>=`. A full description of operators is given in [18].

We consider Einstein's riddle again, but this time using arithmetic operators and properties. The implementation in `alligatorBeer2.in` follows the approach in [18]. First, we do:

```
set(arithmetic).
assign(domain_size, 5).
```

which will allow us define the "right neighbor" relation and to tell the five houses are numbered as `{0,1,2,3,4}` respectively. Then, we specify that objects in the same category are mutually distinct:

```
list(distinct).
 [Alligator,Bulldog,Cat,Donkey,Eagle]. % pets are distinct
 [Aston_Martin,Bugatti,Cadillac,Dacia,Edonis].  % cars are distinct
 ...
end_of_list.
```

To define neighbors, we write:

```
right_neighbor(x,y) <-> x+1 = y.
neighbors(x,y) <-> right_neighbor(x,y) | right_neighbor(y,x). % y left/right
```

The clues are simply written like this:

```
Austrian = Amber.
```

Exercise 8.5 *Solve the puzzle written in this new manner, by typing*
`mace4 -c -f alligatorBeer1.in | interpformat`
Can you tell who drinks beer?

The output contains an interpretation consisting of a set of functions and relations, which actually describe a model produced by Mace4. For example, `function(Cat,[2])` tells us "We have a cat in house 2" (which is actually the house in the middle).

If you call `interpformat` with the `tex` option (`interpformat tex`), you can generate the TEX output, which could be used further in a LATEX file. An excerpt of the output produced by the call

```
mace4 -c -f alligatorBeer1.in | interpformat tex
```

is given in listing 8.3.

Listing 8.3: TEX output for Einstein's Riddle

```
1 \begin{table}[H]  \centering % size 5
2 Amber: 2 \hspace{.5cm}
3 Cat: 2 \hspace{.5cm}
4 \dots
5 \end{table}
```

The pdf obtained from the generated tex, which obviously needs editing, can be seen at the end of this file.

The following two examples are taken from the distribution kit of [18]. The first example asks you to show that a group where $x * x = e$ for all x is commutative. You can find the implementation in `x2.in`. Take a look at that, then generate the proof. File `x2.out` contains such a proof. Inside, you can find an example of applying an inference rule called paramodulation. This is basically the resolution modified to deal with equality.

If we look at lines 2, 3, 4 and 7:

```
2 e * x = x.  [assumption].
3 x' * x = e.  [assumption].
4 (x * y) * z = x * (y * z).  [assumption].
7 x' * (x * y) = y.  [para(3(a,1),4(a,1,1)),rewrite([2(2)]),flip(a)].
```

we can see caluse 7 is obtained by applying paramodulation over clauses 3 and 4. Inside `para(3(a,1),4(a,1,1))`, 1 refers to the left side and 2 to right side of the equality, while index `a` refers to the part of the clause. Given this, we can see that the part `x' * x` in 3 will replace the `(x * y)` part in the left side of 4. This makes `x'` to unify with `x` and `x` with `y`, getting us `(x' * x) * z = x' * (x * z)`. Further, `(x' * x)` is replaced by `e`. The next step is to apply clause 2 and simplify `e * z` to `z`; the last step is to flip the sides of eequalityand obtain `x' * (x * z) = z`, which is basically the clause 7 modulo one variable renaming. The whole bunch of steps is actually one application of paramodulation.

Exercise 8.6 *Drop the condition $x * x = e$ for all x and try to prove or disprove the group remains commutative.*

Exercise 8.7 *Can you prove $\sqrt{2}$ is not a rational number?*

Prover9, if fed the file `sqrt2.in`, can generate the proof.

Exercise 8.8 *Take a look at the input file and see how the domain is modeled. Generate the proof. Try to follow and explain at least one more paramodulation step in the proof. Delete the clause `2 != 1` from the input and see if you get the same result.*

Note: You can find many problems implemented at Thousands of Problems for Theorem Provers [25] (or TPTP for short, `http://www.cs.miami.edu/~tptp`). The program `tptp_to_ladr` might help you translate between formats.

8.3 Let's find the Wumpus!

Exercise 8.9 *Consider the Wumpus world scenario again. We assume* `(1,1)` *is safe, there is neither breeze, nor stench, nor gold there and we have a total of 3 pits, 1 wumpus and 1 gold treasure. Questions:*

1. *calculate how many different worlds are there which obey the stipulated conditions*
2. *estimate how many rules you need to write in your knowledge base to find the Wumpus if you stick with the Propositional Logic*
3. *write down a knowledge base for the same problem using now the First Order Logic and arithmetic operators and try to find the Wumpus by successively visiting safe rooms and adding the perceptions obtained there to your knowledge base.*

8.4 Solutions to exercises

Solution for exercise 8.1 is given in listing 8.4. It closely follows the solution in [23].

Listing 8.4: Colonel West is a criminal

```
% Russell, R. and Norvig, P. (2010). Artificial intelligence: A modern
% approach, 3rd Edition. pp. 336. Prentice Hall: Upper Saddle River, NJ.
%
% 1. It is a crime for an American to sell weapons to hostile nations.
% 2. The country Nono, an enemy of America, has some missiles.
% 3. All of its missiles were sold to it be Colonel West, who is American.
% 4. Missiles are weapons (background information)
% 5. Enemies of America are hostile (background information)
% Prove that West is a criminal
%
% constants: West (col. West), Nono (country), M1 (missile)
% predicate American(x): x is American
% predicate Weapon(x): x is a weapon
% predicate Sells(x,y,z): x sells object y to z
% predicate Hostile(x): x is hostile
% predicate Owns(x,y): entity x owns object y
% predicate Missile(x): x is a missile
% predicate Weapon(x): x is a weapon
% predicate Enemy(x,America): x is an enemy of America
% predicate Criminal(x): x is a criminal

set(print_gen).

formulas(assumptions).
 American(x) & Weapon(y) & Sells(x,y,z) & Hostile(z) -> Criminal(x). %1
 Owns(Nono,M1). %2
 Missile(M1). %2
 Enemy(Nono,America). %2
 Missile(x) & Owns(Nono,x) -> Sells(West,x,Nono). %3
 American(West). %3
 Missile(x) -> Weapon(x). %4
 Enemy(x,America) -> Hostile(x). %5
end_of_list.

formulas(goals).
 Criminal(West).
end_of_list.
```

Solution for exercise 8.3 is given in listing 8.5. It closely follows the solution in [23].

Listing 8.5: Curiosity killed the cat

```
% Russell, R. and Norvig, P. (2010). Artificial intelligence: A modern
% approach, 3rd Edition. pp. 354. Prentice Hall: Upper Saddle River, NJ.
%
% 1. Everyone who loves all animals is loved by someone.
% 2. Anyone who kills an animal is loved by no one.
% 3. Jack loves all animals.
% 4. Either Jack or Curiosity killed the cat, who is named Tuna.
% 5. Cats are animals. (background knowledge)
% Did Curiosity kill the cat?
%
% constants: Jack, Curiosity, Tuna
% predicate Animal(x): x is an animal
% predicate Loves(x,y): x loves y
% predicate Kills(x,y): x kills y
% predicate Cat(x): x is a cat

set(binary_resolution).

formulas(assumptions).
 all x (all y (Animal(y) -> Loves(x,y)) -> exists y (Loves(y,x))). %1
 all x (exists z (Animal(z) & Kills(x,z)) -> all y (-Loves(y,x))). %2
 all x (Animal(x) -> Loves(Jack,x)). %3
 Kills(Jack,Tuna) | Kills(Curiosity,Tuna). %4
 Cat(Tuna). %4
 all x (Cat(x) -> Animal(x)). %5
end_of_list.

formulas(goals).
 Kills(Curiosity,Tuna).
end_of_list.
```

Hint for exercise 8.4: remember Skolem functions.

Hint for exercise 8.9: $C_{13}^3 * 13 * 15$ (positions for pits, wumpus, gold respectively).

Advocaat: 3 Amber: 2 Aston_Martin: 4 Austrian: 2 Beige: 0 Belgian: 4
Bugatti: 2 Bulldog: 4 Cadillac: 0 Cat: 2 Cider: 2 Cyan: 3 Czech: 0
Dacia: 1 Daiquiri: 4 Dane: 1 Denim: 1 Donkey: 1 Eagle: 0 Edonis: 3
Eiswein: 1 Emerald: 4 Estonian: 3 Alligator: 3 Beer: 0

neighbors:	0	1	2	3	4
0	0	1	0	0	0
1	1	0	1	0	0
2	0	1	0	1	0
3	0	0	1	0	1
4	0	0	0	1	0

right_neighbor:	0	1	2	3	4
0	0	1	0	0	0
1	0	0	1	0	0
2	0	0	0	1	0
3	0	0	0	0	1
4	0	0	0	0	0

Table 8.1: The model for Einstein's riddle produced by `interpformat tex`

Chapter 9

Constraint satisfaction problems

Learning objectives for this week are:

1. To build constraint network for a set of constraints.
2. To trace the arc consistency algorithm.
3. To see how local search is used to solve CSP
4. To learn that CSP can be solved by satisfiability.

9.1 Solving CSP by consistency checking

This section helps you to understand the arc consistency algorithm. We will use the Consistency Based CSP Solver from http://www.aispace.org/downloads.shtml. Start the solver with:

```
1 java -jar constraint.jar
```

Let $\mathcal{D}_X$ domain of variable X. An arc $\langle X, r(X,Y)\rangle$ is consistent if for each value $x \in \mathcal{D}_X$ there is some value $y \in \mathcal{D}_Y$ such that $r(x,y)$ is satisfied. Consistency of an arc $\langle X, r(X,Y)\rangle$ is obtained by removing all values $x \in \mathcal{D}_X$ for which there is no corresponding value $y \in \mathcal{D}_Y$ that satisfies the constraint.

Arc consistency can be used to reduce to domain of the variables. Let $A < B$ be a constraint between the variables A and B with $\mathcal{D}_A = \{4,5,6,7,8\}$ and variable $\mathcal{D}_B = \{2,3,4,5,6\}$. Note that for some values in the domain of A there does not exist a consistent value in B's domain satisfying the constraint $A < B$. The corresponding constraints network in Figure 9.1 has two arcs: $\langle A, A < B\rangle$ and $\langle B, A < B\rangle$

Exercise 9.1 *What is domain of A and B such that the constraint network to be consistent. Build the constraint network in the CSP solver. Using the* `Step` *mechanism, check if the solver gives the same solution as you did.*

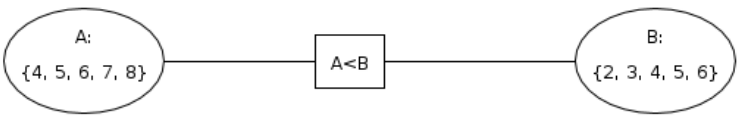


Figure 9.1: A constraint network with two variables and one binary constraint. Arc consistency algorithm is able to reduce the domains of A and B.

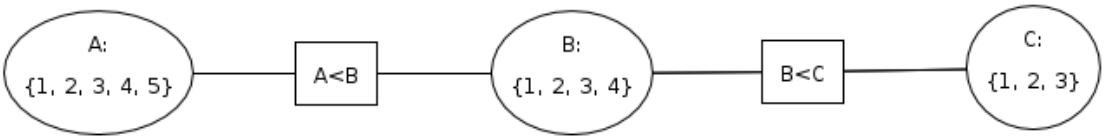


Figure 9.2: A constraint network with three variables and two binary constraints. Arc consistency algortihm is able to find the unique solution.

Arc consistency can be complemented with *domain splitting* to identify a solution. That is, a consistent assignment of unique values for each variable. Hence, to solve a CSP, two steps are taken:

1. Simplify the CSP using arc consistency; and,

2. If the problem is not solved, select a variable whose domain has more than one element, split it, and recursively solve each case (that is domain splitting).

For some CSP, domain splitting is not required, as arc consistency finds itself a solution. Let the variables A, B, C with $\mathcal{D}_A = \{1, 2, 3, 4, 5\}$, $\mathcal{D}_B = \{1, 2, 3, 4\}$ and $\mathcal{D}_C = \{1, 2, 3\}$. Let the binary constraints $A < B$ and $B < C$. The constraints network in Figure 9.2 has four arcs: $\langle A, A < B\rangle$, $\langle B, A < B\rangle$, $\langle B, B < C\rangle$, $\langle C, B < C\rangle$. One possible computation flow is:

i) 1 is removed from the domain of B because of arc $\langle B, A < B\rangle$.
ii) 4 and 5 are removed from the domain of A because of arc $\langle A, A < B\rangle$.
iii) 1 and 2 are removed from the domain of C because of arc $\langle C, B < C\rangle$.
iv) 3 and 4 are removed from the domain of B because of arc $\langle B, B < C\rangle$.
v) 2 and 3 are removed from the domain of A because of arc $\langle A, A < B\rangle$

The arc consistency algorithm is able to find the unique solution $A = 1$, $B = 2$, $C = 3$.

Exercise 9.2 *Build the constraint network in the CSP solver. Use the* `Step` *mechanism to trace each step performed by the arc consistency algorithm. Change the default order of making an arc consistent. You can do this by selecting yourself the arc to check. Did you manage to find an order for checking arc consistency for which the solution is found in less than 5 steps?*

When the arc consistency algorithm terminates and some domains have multiple values, there is no guarantee that a solution exists. This is one limitation of the arc consistency, as illustrated by exercise 9.3.

Exercise 9.3 *Let the variables A, B, C with $\mathcal{D}_A = \mathcal{D}_B = \mathcal{D}_C = \{1, 2, 3\}$ and the constraints: $A \neq B$, $A \neq C$ and $B \neq C$.*

1. *Build the corresponding constraint network and trace the arc consistency algorithm. When algorithm ends, figure out if a solution is still possible. How many solutions are there? You can use* `autosolve` *button to find all solutions.*

2. *Restrict the domain of A to $\{1, 2\}$. Trace the arc consistency algorithm. How many solutions exist?*

3. *Additionally, restrict the domain of B to $\{1, 2\}$. Trace the arc consistency algorithm. How many solutions exist?*

4. *Additionally, restrict the domain of C to $\{1, 2\}$. Trace the arc consistency algorithm. How many solutions exist?*

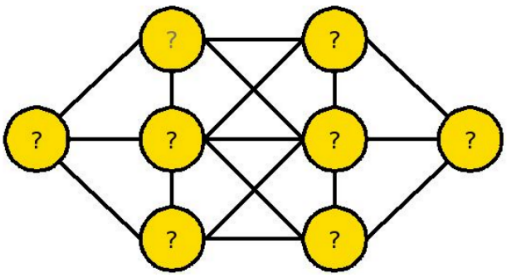


Figure 9.3: Using heuristics to solve CSP.

The order in which arcs are selected does impact the computation. Computation amount is also influenced by the order in which values are picked for the current variable. Some heuristics are usefull here. *Minimum-remaining-values* (MRV) heuristic picks the variable with the smallest domain. MRV choses the variable that is most likely to cause a failure soon, thereby pruning the search tree. *Degree heuristic* picks the variable involved in the largest number of constraints on other unassigned variables. *Least-constraining-value* heuristic prefers the value that rules out the fewest choices for the neighboring variables.

Consider the CSP in Figure 9.3. You have to place numbers 1 through 8 on nodes such that: 1) each number appears exactly once and 2) no connected nodes have consecutive numbers. One general strategy stressed by Bartak in [2] is: *"Deal with hard cases first: they can only get more difficult if you put them off."*

Exercise 9.4 *For the CSP in figure 9.3 identify the variables, their domains, and the constraints involved. Assume that the variables are selected based on the degree heuristic. Which node will be selected first? What value would you pick from the domain of that node?*

The following exercise 9.5 is for students taking gingko biloba daily. It is better to solve this exercise at the end of the class, if time available. Note that you have to define a global constraint on all variables (to model that nodes have distinct numbers). To increase speed, you can save the partial model in xml and work directly on this file.

Exercise 9.5 *Build the constraint network from Figure 9.3. Can you solve the problem without using the* `Split domain` *option in the solver?*

It is easy to translate satisfiability problems in CSP. Let the formula in conjunctive normal form:

$$(\neg A \vee B) \wedge (\neg C \vee A) \wedge (A \vee B \vee D) \wedge C$$

The corresponding constraint hypergraph appears in the left part of figure 9.4. Note that there is one unary constraint ($C = true$), two binary constraints ($\neg A \vee B$), $\neg C \vee A$) and one ternary constraint $A \vee B \vee D$. Applying the arc consistency algorithm, gives the solution depicted in the right part of figure 9.4.

Exercise 9.6 *Build yourself the contraint network for* $(\neg A \vee B) \wedge (\neg C \vee A) \wedge (A \vee B \vee D) \wedge C$. *Trace the arc consistency algorithm. Split the domain of variable* D *to obtain solutions. How many solution are there. Check if Mace4 returns the same number of models.* `Auto arc consistency`

The following exercise 9.7 is taken from http://www.aispace.org/.

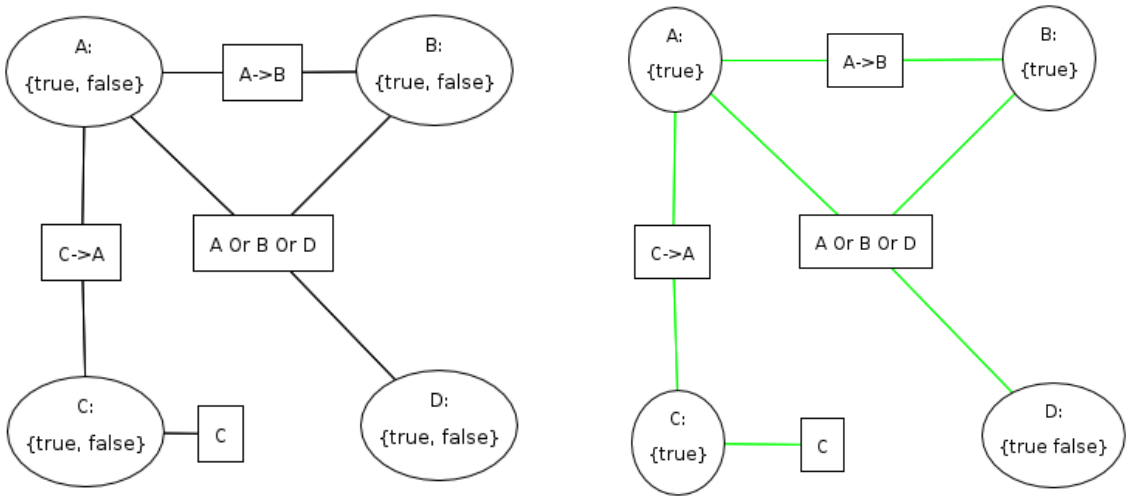


Figure 9.4: Boolean formula as CSP.

Exercise 9.7 *A family of four needs to figure out how each family member will commute to work or school given several constraints. The family consists of a mother, father, son and daughter. Each family member can bicycle or ride in the car. Additionally the son has a pogo stick he can use for commuting to school. The assignment of transportation modes to family members is subject to the following constraints:*

- *There are only two bicycles.*
- *The car can only hold three people.*
- *The son and daughter must take the same mode of transportation.*
- *The son and daughter can only go by car if at least one of the parents is going by car.*

Identify the variables and their domains. Solve the CSP yourself. Model the problem in the CSP solver. Is it able to find the same solution as you did? How many solutions are there?

9.2 Solving CSP with stochastic local search

This section helps you to understand how CSP can be solved with various local search algorithms: Random Walk, Greedy Descent, and Simmulating Annealing. We will use the Stochastic Local Search Based CSP Solver from http://www.aispace.org/downloads.shtml. Start the solver with:

```
1 java -jar hill.jar
```

The running scenario is 5-Queens problem. That is, placing five queens on chess-like board of size five, such that no queen attacks another one. Load the sample problem `5queens.xml` developed Knoll et. al [15] and available from the `Sample CSP` menu. Here, queen A stays on column A, or queen B on column B. The constraint `Queen1` defines conflicts between queens on consecutive columns: (A, B), (B, C), (C, D), and (D, E). The constraint `Queen2` defines conflicts between queens at a distance of two columns between them: (A, C), (B, D), (C, E). The constraint `Queen3` defines conflicts between queens at a distance of three columns: (A, D), (B, E). There is only one `Queen4` constraint between the queen on column A and the queen on E.

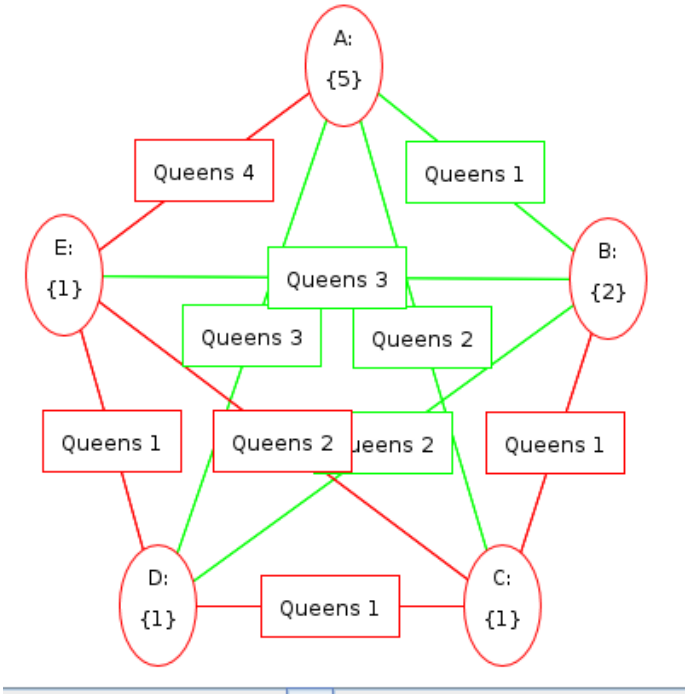


Figure 9.5: Initialising the 5-queens problem.

First, the CSP must be initialized. Here, a value is assigned to each variable in the CSP from its domain. Assume the random initialisation in Figure 9.2. The red arcs show which queens atack each others. These arcs are inconsistent according to the custom `Queeni` constraints. The total number of conflicts in the network is five.

For local search, a neighbour is one change to variable assignment. In the 5-Queens problem, a neighbour state is generated by changing the row of one queen only. Here, each state has four possible neighbours. To generate a neighbour, the degree heuristic picks the most conflicting variable. The queens introducing the most conflicts are C and E, each with three conflicts. Assume that queen C is selected, the corresponding neibours are for $C = 2$, $C = 3$, $C = 4$ and $C = 5$.

Exercise 9.8 *Solve the 5-Queens problem by hand using the support of the CSP solver. You can mannually select both the queen to be moved and the value for it. After initialisation, simply select a variable to manually change its value. If you get stuck feel free to re-initialize the problem. What do you think - to find a solution, would be better to strugle to solve a CSP which is very close to the solution (i.e only one conflict remaining) or would be better to simply restart the board?*

`Batch Run` mode can be used to calculate the runtime distribution and plot the percentage of successes against the number of steps.

Exercise 9.9 *Select the Greedy Descent algorithm from the* ***`Hill options -> Algorithm options`*** *menu. Does it find a solution within the default number of steps? What about in 1000 steps?*

Recall that Greedy Descent is an incomplete algorithm. That is, it is not guaranteed to find a solution even given infinite number of steps. That is why the runtime distribution flattens out.

You already know that random restarts improve Greedy Descent by removing its incompleteness.

Exercise 9.10 *Select now the Greedy Descent with Random Restarts. Does it perform better?*

Exercise 9.11 *Select now the Greedy Descent with all options. Does it perform better? Investigate which is the optimal percent for randomness in case of the 8-queens problem.*

Exercise 9.12 *Select now the Simulating Annealing. Does it perform better?*

Exercise 9.13 *You might find interesting the Traffic Flow exercise at http://www.aispace.org/exercises/exercise4-c-2.shtml*

9.3 Solving CSP with satisfiability solvers

We will rely on your favorite satisfiability solver, that is Mace4.

N-Queens problem

The two implementations we present are borrowed from [18] and use the arithmetic facilities of Mace4. The solution in listing 9.1 uses the `Q(x,y)` predicate to express that there is a queen in row `x`, column `y`. The solution in listing 9.2 employs function `Q(x)=y` to say the same.

Listing 9.1: N-queens problem using predicates

```
set(arithmetic).

formulas(assumptions).

% Relation Q(x,y) means there is a queen at row x, column y.

 all x exists y Q(x,y).           % Each row has at *least* one queen.

 Q(x,y1) & Q(x,y2) -> y1 = y2.    % Each row has at most one queen.

 Q(x1,y) & Q(x2,y) -> x1 = x2.    % Each column has at most one queen.

 Q(x1,y1) & Q(x2,y2) & (x2 + -x1 = y2 + -y1) ->
            x1 = x2 & y1 = y2.     % Each \ diagonal has at most one queen.

 Q(x1,y1) & Q(x2,y2) & (x1 + -x2 = y2 + -y1) ->
            x1 = x2 & y1 = y2.     % Each / diagonal has at most one queen.

end_of_list.
```

Listing 9.2: N-queens problem using functions

```
set(arithmetic).

formulas(assumptions).

% In this representation, Q(i)=n means that Row i Column n has a queen.
% The constraint that no queens can be in the same row is always satisfied
% in this representation, because Q is a function; that is,
% Q(x) != Q(z) -> x != z  is always satisfied.

% Note that there is no binary "minus" operation, so we have to write x + -y.

x != z -> Q(x) != Q(z).   % No 2 queens in the same column.

x != z  ->  z + -x != Q(z) + -Q(x).  % No 2 queens in \ diagonal.
```

```
15
16 x != z  ->  z + -x != Q(x) + -Q(z).  % No 2 queens in / diagonal.
17
18 end_of_list.
```

Exercise 9.14 *Test both solutions using the* `-n8` *option in the command line. Make sure you understand the obtained* `function` *and* `relation` *(if any). Check them. Then, add the* `-m -1` *option to generate all models. How many of them are there?*

Exercise 9.15 *Repeat the tasks in 9.14 with different values of* `n`*.*

Cryptarithmetic

As a student at TUCN, you got used to send secret messages to your parents. In such a message, you asked for some more money.

```
  S E N D +
  M O R E
---------
M O N E Y
```

Your parents will use Mace4 to decript your request. The solution is given in listing 9.3 and uses `C0,...` for the values of the carry. The comments are pretty self-explanatory.

Listing 9.3: SEND+MORE

```
1  set(arithmetic).
2
3  assign(domain_size, 10).  % domain is {0,1,2,3,4,5,6,7,8,9}.
4
5  list(distinct). %distinct letters for distinct digits
6
7    [D,E,M,N,O,R,S,Y]. % set of all letters
8
9  end_of_list.
10
11 formulas(assumptions).
12
13   D + E + C0 = Y + 10 * C1.
14   N + R + C1 = E + 10 * C2.
15   E + O + C2 = N + 10 * C3.
16   S + M + C3 = O + 10 * M.
17
18   % No initial carry
19   C0 = 0.
20
21   % Leading zeros not allowed
22   S != 0.
23   M != 0.
24
25 end_of_list.
```

Exercise 9.16 *Test the example and see how many solutions you obtain. How much money your parents should send to you?*

Exercise 9.17 *Model the same problem, without introducing suplimentary variables such as* `C0`*,* `C1`*,* `C2`*,* `C3`*. Test your solution with Mace4.*

Exercise 9.18 *Implement the problem SEND+MOST=MONEY and see how many solutions you obtain. Which is the maximum value you are hoping to receive from your loving parents? Which is the minimum value your parents most probable send to you?*

Exercise 9.19 *Implement in Mace4 the CSP in section 6.1 from [23]. That is TWO + TWO = FOUR. Would be possible to model the same CSP in the contraint.jar solver?*

Exercise 9.20 *Following the approach in listing 9.3, implement the problem*

ZWEI+VIER+VIER+VIERZIG+VIERZIG=NEUNZIG (In German: 2+4+4+40+40=90)

and see how many solutions you obtain and in how much time. Then implement CSP without variables for carry and see the new running time.

Sudoku

The example in listing 9.4 adapts the approach in [23] to the 4×4 case.

Listing 9.4: Sudoku 4×4

```
1
2  formulas(assumptions).
3
4   %a number must be assigned to each square
5   a0 = 0 | a0 = 1 | a0 = 2 | a0 = 3.  %first line, first column
6   a1 = 0 | a1 = 1 | a1 = 2 | a1 = 3.  %second line, first column
7   a2 = 0 | a2 = 1 | a2 = 2 | a2 = 3.
8   a3 = 0 | a3 = 1 | a3 = 2 | a3 = 3.
9
10  b0 = 0 | b0 = 1 | b0 = 2 | b0 = 3.  %second line, first column
11  b1 = 0 | b1 = 1 | b1 = 2 | b1 = 3.
12  b2 = 0 | b2 = 1 | b2 = 2 | b2 = 3.
13  b3 = 0 | b3 = 1 | b3 = 2 | b3 = 3.
14
15  c0 = 0 | c0 = 1 | c0 = 2 | c0 = 3.
16  c1 = 0 | c1 = 1 | c1 = 2 | c1 = 3.
17  c2 = 0 | c2 = 1 | c2 = 2 | c2 = 3.
18  c3 = 0 | c3 = 1 | c3 = 2 | c3 = 3.
19
20  d0 = 0 | d0 = 1 | d0 = 2 | d0 = 3.
21  d1 = 0 | d1 = 1 | d1 = 2 | d1 = 3.
22  d2 = 0 | d2 = 1 | d2 = 2 | d2 = 3.
23  d3 = 0 | d3 = 1 | d3 = 2 | d3 = 3.
24
25  %no more than one number is assigned to a square
26  (u = x & u = y) -> x = y.
27
28
29  %defining the "mutually distinct" relation for 4 variables
30  alldiff(x,y,z,u) -> (x != y) & (x != z) & (x != u) &
31                      (y != z) & (y != u) & (z != u).
32
33  %Sudoku rule: numbers in a line must be different from each other
34  alldiff(a0,a1,a2,a3). %first line
35  alldiff(b0,b1,b2,b3). %second line
36  alldiff(c0,c1,c2,c3).
37  alldiff(d0,d1,d2,d3).
38
39  %Sudoku rule: numbers in a column must be different from each other
40  alldiff(a0,b0,c0,d0). %first column
```

```
41  alldiff(a1,b1,c1,d1).
42  alldiff(a2,b2,c2,d2).
43  alldiff(a3,b3,c3,d3).
44
45  %Sudoku rule: numbers in squares must be different from each other
46  alldiff(a0,a1,b0,b1). %top left square
47  alldiff(a2,a3,b2,b3). %top right square
48  alldiff(c0,c1,d0,d1). %bottom left square
49  alldiff(c2,c3,d2,d3). %bottom right square
50
51  %initial hints
52 %b1 = 2.
53 %b3 = 1.
54 %c0 = 3.
55 %c2 = 2.
56
57  %initial hints - alternative problem
58 %b1 = 0.
59 %b3 = 2.
60 %c0 = 3.
61 %c2 = 0.
62
63 end_of_list.
```

Exercise 9.21 *Look in Listing 9.4 and identify how the Sudoku constraints are modeled. Test the example on two different set of clues, by commenting the clue lines appropriately.*

Exercise 9.22 *Extend the example to the 9×9 case and test it on the initial clues in [23] page 213.*

Exercise 9.23 *Model the exercise in Mace4. Does Mace4 find the same number of models as the CSP solver?*

9.4 Solutions to exercises

Solution to exercise 9.2
The constraint network appears in file `simple3.xml`. There is no solution in less than five steps.

Solution to exercise 9.3

1. The constraint network appears in file `different.xml`. Arc consistency algorithm does not reduce the domains of A, B or C. There are 3!= 6 solutions now.
2. There are 4 solutions now.
3. There are 2 solutions now.
4. With $\mathcal{D}_A = \mathcal{D}_B = \mathcal{D}_C = \{1, 2\}$, there is no solution. However, arc consistency is not able to figure this out.

Solution to exercise 9.4
MRV will select the nodes in the middle, as they are the most restricted ones. Each of these two nodes has six constraints. Least-constraining-value heuristic will put either 1 or 8, as these values rules out the fewest choices for the neighboring variables. Both of them have only one consecutive number in the given domain. With the above heuristics, a solution is found without backtracking.

Solution to exercise 9.6
There are two solutions: 1) $A = B = C = D = true$ and 2) $A + B + C + true, D = false$.

Follow the steps at http://www.aispace.org/exercises/exercise4-b-1.shtml.

Solution for exercise 9.14: 92

Solution for exercise 9.17:

Listing 9.5: SEND+MORE

```
1  set(arithmetic).
2  assign(domain_size, 10).  % domain is {0,1,2,3,4,5,6,7,8,9}.
3
4  list(distinct). %distinct letters for distinct digits
5
6   [D,E,M,N,O,R,S,Y]. % set of all letters
7
8  end_of_list.
9
10 formulas(assumptions).
11
12  1000 * S + 100 * E + 10 * N + D +
13  1000 * M + 100 * O + 10 * R + E =
14  10000 * M + 1000 * O + 100 * N + 10 * E + Y.
15
16 % Leading zeros not allowed
17   S != 0.
18   M != 0.
19
20 end_of_list.
```

Solution for exercise 9.18: see file `SEND_MOST_MONEY.in`

Solution for exercise 9.20: see file `ZWEI_VIER_VIER_VIERZIG_VIERZIG_NEUNZIG.in`

Chapter 10

Classical planning

We focus here on classical planning: offline (static), discrete, deterministic, fully observable, single-agent, sequential (plans are sequences of actions), domain-independent. Planners use a model of an application domain (such as blocks world) and a description of a specific problem (initial state and goals) to compute a plan.

Learning objectives for this week are:

1. To write STRIPS domains in Planning Domain Definition Language.
2. To get used with the Fast Downward planning system.

10.1 Planning domains

Benchmark domains in planning have been continuously used in International Planning Competitions [5]. Classical examples (such as Assembly, Blocks, Grid, Gripper, Hanoi, Logistics, Travel Salesman, Tire-world) are described at https://fai.cs.uni-saarland.de/hoffmann/ff-domains.html, while more modern ones (such as Parking, CityCar, Barman, Genome) at https://helios.hud.ac.uk/scommv/IPC-14/domains.html. We restrict here to three planning domains: sliding tiles, blocks world and the gripper robot.

Sliding-tiles puzzle

The domain contains tiles that should be slided from a start position to a goal one, as depicted in Figure 10.1. The `sliding-tile` domain in listing 10.1 contains four predicates and one action. The position of a tile is formalised by predicate `(tile-at ?tile ?row ?col)`. Blank position is encoded by `(is-blank ?row ?col)`. The grid is encoded with `(next-row ?r1 ?r2)` and `(next-column ?c1 ?c2)`. The `move-tile-down` action has four parameters: the `?tile` which is moved from the current row `?old-row` to next row `?new-row`, and the column `?col`. The preconditions check if the new position is blank. In that case, two facts are removed from

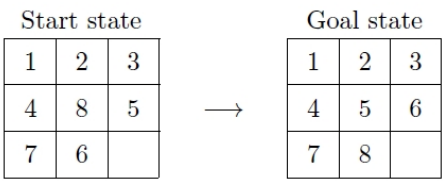


Figure 10.1: Sliding-tiles puzzle.

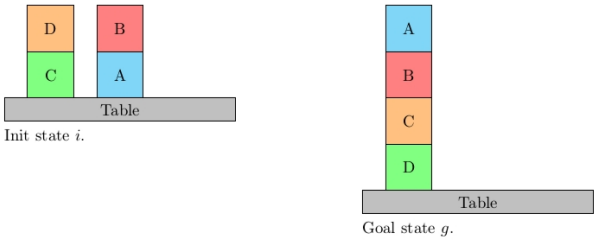


Figure 10.2: Blocks world domain.

the current state (current tile position and current blank) and two new facts are added (the new tile position and new blank).

Listing 10.1: Example of Sliding-Tiles puzzle.

```
1  (define (domain sliding-tile)
2    (:predicates
3        (tile-at ?t ?r ?c)
4        (is-blank ?r ?c)
5        (next-row ?r1 ?r2)
6        (next-column ?c1 ?c2))
7    (:action move-tile-down
8        :parameters    (?tile ?old-row ?new-row ?col)
9        :precondition (and (next-row ?old-row ?new-row)
10                          (tile-at ?tile ?old-row ?col)
11                          (is-blank ?new-row ?col))
12       :effect       (and (not (tile-at ?tile ?old-row ?col))
13                          (not (is-blank ?new-row ?col))
14                          (tile-at ?tile ?new-row ?col)
15                          (is-blank ?old-row ?col)))
```

Exercise 10.1 *Following the model in listing 10.1, define three actions:* `move-tile-up`*,* `move-tile-right`*, and* `move-tile-left`*.*

Blocks world

The domain contains blocks that need to be re-assembled on a table with unlimited space. Basic model uses three actions: 1) moving a block from the table to another block, 2) moving a block from another block to the table, 3) moving a block from another block to another block.

Listing 10.2: Example of the blocks world.

```
1  (define (domain blocksworld)
2  (:predicates (clear ?x)
3               (on-table ?x)
4               (on ?x ?y))
5  (:action move-b-to-b
6    :parameters    (?bm ?bf ?bt)
7    :precondition (and (clear ?bm) (clear ?bt) (on ?bm ?bf))
8    :effect       (and (not (clear ?bt)) (not (on ?bm ?bf))
9                       (on ?bm ?bt) (clear ?bf)))
```

The formalisation in listing 10.2 uses three predicates: `(clear ?x)` checks if block `?x` is clear, `(on-table ?x)` verifies if block `?x` is on table, while `(on ?x ?y))` if block `?x` is on block `?y`. The action in lines 5-9 moves a block `?bm` from the top of another block `?bf` on a new block `?bt`. As preconditions, the block to move should not have anything above, that is `(clear`

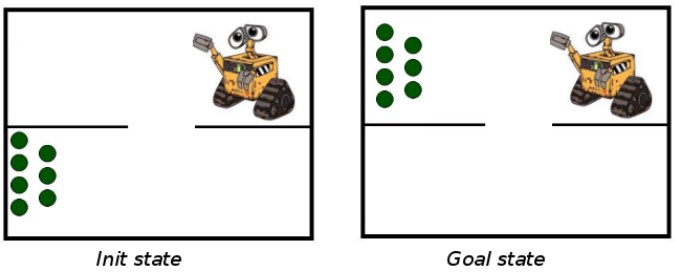


Figure 10.3: Gripper domain.

`?bm`). Similarly for the block `?bt`: `(clear ?bt)`. The third precondition `(on ?bm ?bf)` forces `?bm` to be on top of `?bf`. If preconditions are meet, there are two negative effects and two positive effects. Negative effects remove two facts from the successor state: the new block is no longer clear `(not (clear ?bt))`, while `?bm` is no longer on top of `?bf`. Positive effects add two facts to the successor state: `?bm` is on top of `?bt`, and `?bf` become clear.

Exercise 10.2 *Following the model in listing 10.2, define the actions* `move-b-to-t` *that moves a block from another block to table, respectively* `move-t-to-b` *that moves a block from the table on another block.*

Gripper

A robot moves a set of balls between two rooms (Fig. 10.3). Two gripers are used to grip two balls at the time. There are only three actions: move, pick, and drop.

Listing 10.3: Example of the gripper domain.

```
(define (domain gripper-strips)
   (:predicates (room ?r)
                (ball ?b)
                (gripper ?g)
                (at-robby ?r)
                (at ?b ?r)
                (free ?g)
                (carry ?o ?g))

   (:action move
       :parameters  (?from ?to)
       :precondition (and  (room ?from) (room ?to) (at-robby ?from))
       :effect (and  (at-robby ?to)
                     (not (at-robby ?from))))



   (:action pick
       :parameters (?obj ?room ?gripper)
       :precondition  (and  (ball ?obj) (room ?room) (gripper ?gripper)
```

Action `move` moves the robot from the room `?from` to the room `?to` under the condition that the robot is located in room `?from`, given by `(at-robby ?from)`. Action `pick` checks if a ball `?b` lays in the same room `?room` with the robot and the robot has a free gripper. The positive effect asserts that the gripper carries the object `(carry ?obj ?gripper)`. The negative effects state that the object is no longer in the location `(not (at ?obj ?room))`, and the gripper is no longer free `(not (free ?gripper))`.

Exercise 10.3 *Following the model in listing 10.3, define the action* `drop` *with three parameters* `?obj`, `?room`, `?gripper`.

As you have already noted, we used a specific language to formalise planning domains. This is the Planning Domain Definition Language.

10.2 Planning Domain Definition Language

Planning Domain Definition Language (PDDL) is the input language of most planning tools. The PDDL language supports many different levels of expressivity starting with STRIPS (STanford Research Institute Problem Solver) or ADL (Action Description Language). For instance, STRIPS allows only specifications of preconditions (what must be established before the action is performed) and postconditions (what is established after the action is performed). ADL extends STRIPS with types, negative preconditions, disjunctive preconditions, equality, quantified preconditions or conditional effects. Recent formalisations include durative actions, fluents, preferences, functions. As each planner implements only a subset of PDDL, attention is needed when modelling a domain for a particular planning tool. For the BNF description of PDDL3.0 consult [6].

Planning tasks formalised in PDDL are separated into two files: domain and problem to solve. The domain file specifies the predicates and available actions (see listing 10.4).

Listing 10.4: Syntax for a PDDL domain.

```
(define (domain DOMAIN_NAME)
  (:requirements [:strips] [:equality] [:typing] [:adl] ...)
  [(:types T1 T2 T3 T4 ...)]
  (:predicates (PREDICATE_1_NAME [?A1 ?A2 ... ?AN])
               (PREDICATE_2_NAME [?A1 ?A2 ... ?AN])
               ...)

  (:action ACTION_1_NAME
    [:parameters (?P1 ?P2 ... ?PN)]
    [:precondition PRECOND_FORMULA]
    [:effect EFFECT_FORMULA]))
```

The problem definition specifies which domain it belongs to, the existing objects in the problem instance, the initial state of the world, and the goal (see listing 10.5).

Listing 10.5: Syntax for a PDDL problem.

```
(define (problem PROBLEM_NAME)
  (:domain DOMAIN_NAME)
  (:objects OBJ1 OBJ2 ... OBJ_N)
  (:init ATOM1 ATOM2 ... ATOM_N)
  (:goal CONDITION_FORMULA))
```

As an example, the `eight-puzzle` problem in listing 10.6 starts by specifying the domain needed to solve the problem. The objects are represented by eight tiles, three rows and three columns. The initial state specifies the grid (i.e., `next-row row1 row2)`) and position of each tile (i.e., `(tile-at tile8 row1 col1)`). The goal's state uses the operator `and` to specify a conjunction of facts that should be true.

Listing 10.6: Example of a problem for the Sliding-Tiles domain.

```
(define (problem eight-puzzle)
  (:domain sliding-tile)
  (:objects
     tile1 tile2 tile3 tile4 tile5 tile6 tile7 tile8
```

```
5       row1 row2 row3
6       col1 col2 col3)
7     (:init
8       (next-row row1 row2)          (next-column col1 col2)
9       (next-row row2 row3)          (next-column col2 col3)
10      (tile-at tile8 row1 col1)     (tile-at tile4 row2 col2)
11      (tile-at tile7 row1 col3)     (tile-at tile3 row2 col1)
12      (tile-at tile6 row1 col2)     (tile-at tile2 row3 col1)
13      (tile-at tile5 row2 col3)     (tile-at tile1 row3 col2)
14      (is-blank row3 col3))
15    (:goal
16     (and
17      (tile-at tile1 row1 col1)    (tile-at tile2 row1 col2)
18      (tile-at tile3 row1 col3)    (tile-at tile4 row2 col1)
19      (tile-at tile5 row2 col2)    (tile-at tile6 row2 col3)
20      (tile-at tile7 row3 col1)    (tile-at tile8 row3 col2))))
```

Exercise 10.4 *For the domain in listing 10.1, define the problem illustrated in Fig. 10.1.*

Exercise 10.5 *For the domain in listing 10.2, define the problem illustrated in Fig. 10.2.*

Exercise 10.6 *For the domain in listing 10.3, define the problem illustrated in Fig. 10.3.*

10.3 Fast Downward planner

Fast Downward (FD) is a platform for implementing and evaluating planning algorithms. Among many running options provided by FD [9], we limit here to the two most relevant: the search algorithm and the heuristic that guides the search. With these two parameters, the usual call is:

```
1 ./fd Domain.pddl Problem.pddl \
2   --heuristic "h=ff()" \
3   --search "astar(h)"
```

FD gets the domain and the planning problem to be solved in PDDL syntax. In the above example, A^* algorithm with the `ff()` heuristic are used to compute a plan. The plan is saved in files named `sas.plan.1`.

Exercise 10.7 *Generate by hand the plan for problem in Fig. 10.1. Does FD planner compute the same plan?*

Exercise 10.8 *Run the FD planner for the 8-sliding tiles domain with various init and goal states. Extend the problem to 15-tiles, that is 4 rows and 4 columns.*

Exercise 10.9 *Run the FD planner for the blocks world domain with various init and goal states.*

Exercise 10.10 *Run the FD planner for the gripper domain with various init and goal states.*

Exercise 10.11 *Model the Wumpus world in PDDL. Used FD planner to generate plans for the three problems in figure 10.4. Assume that the environment is fully observable.*

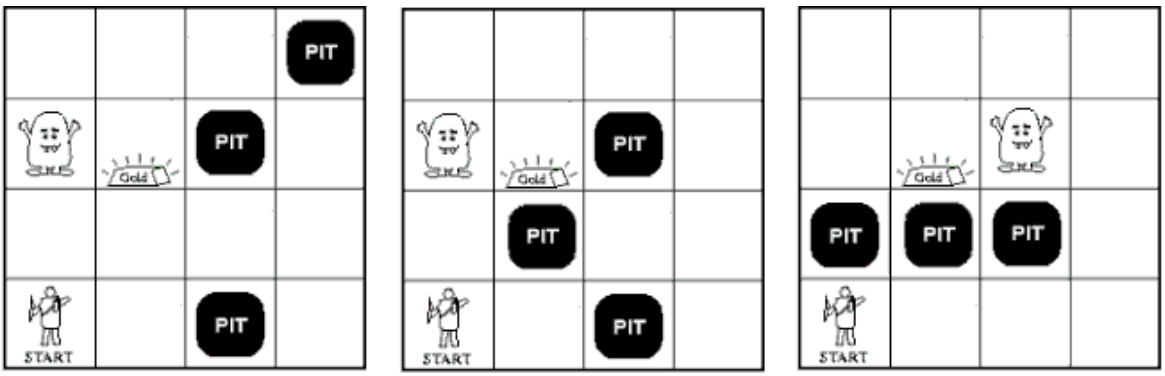


Figure 10.4: Modelling Wumpus world in PDDL.

10.4 Solutions to exercises

Solution to exercise 10.1

```
1   (:action move-tile-up
2     :parameters (?tile ?old-row ?new-row ?col)
3     :precondition (and (next-row ?new-row ?old-row)
4                        (tile-at ?tile ?old-row ?col)
5                        (is-blank ?new-row ?col))
6     :effect (and (not (tile-at ?tile ?old-row ?col))
7                  (not (is-blank ?new-row ?col))
8                  (tile-at ?tile ?new-row ?col)
9                  (is-blank ?old-row ?col)))
10
11  (:action move-tile-right
12    :parameters (?tile ?row ?old-col ?new-col)
13    :precondition (and (next-column ?old-col ?new-col)
14                       (tile-at ?tile ?row ?old-col)
15                       (is-blank ?row ?new-col))
16    :effect (and (not (tile-at ?tile ?row ?old-col))
17                 (not (is-blank ?row ?new-col))
18                 (tile-at ?tile ?row ?new-col)
19                 (is-blank ?row ?old-col)))
20
21  (:action move-tile-left
22    :parameters (?tile ?row ?old-col ?new-col)
23    :precondition (and (next-column ?new-col ?old-col)
24                       (tile-at ?tile ?row ?old-col)
25                       (is-blank ?row ?new-col))
26    :effect (and (not (tile-at ?tile ?row ?old-col))
27                 (not (is-blank ?row ?new-col))
28                 (tile-at ?tile ?row ?new-col)
29                 (is-blank ?row ?old-col)))
```

Solution to exercise 10.2

```
(:action move-b-to-t
  :parameters (?bm ?bf)
  :precondition (and (clear ?bm) (on ?bm ?bf))
  :effect (and (not (on ?bm ?bf))
               (on-table ?bm) (clear ?bf)))

(:action move-t-to-b
  :parameters (?bm ?bt)
  :precondition (and (clear ?bm) (clear ?bt) (on-table ?bm))
  :effect (and (not (clear ?bt)) (not (on-table ?bm))
               (on ?bm ?bt))))
```

Solution to exercise 10.3

```
   (:action drop
       :parameters  (?obj  ?room ?gripper)
       :precondition  (and  (ball ?obj) (room ?room) (gripper ?gripper)
                            (carry ?obj ?gripper) (at-robby ?room))
       :effect (and (at ?obj ?room)
                    (free ?gripper)
                    (not (carry ?obj ?gripper)))))
```

Solution to exercise 10.4

```
(define (problem eight-puzzle-from-figure)
  (:domain sliding-tile)
  (:objects
     tile1 tile2 tile3 tile4 tile5 tile6 tile7 tile8
     row1 row2 row3
     col1 col2 col3)
  (:init
     (next-row row1 row2)          (next-column col1 col2)
     (next-row row2 row3)          (next-column col2 col3)
     (tile-at tile1 row1 col1)     (tile-at tile8 row2 col2)
     (tile-at tile2 row1 col2)     (tile-at tile5 row2 col3)
     (tile-at tile3 row1 col3)     (tile-at tile7 row3 col1)
     (tile-at tile4 row2 col1)     (tile-at tile6 row3 col2)
     (is-blank row3 col3))
  (:goal
   (and
    (tile-at tile1 row1 col1)    (tile-at tile2 row1 col2)
    (tile-at tile3 row1 col3)    (tile-at tile4 row2 col1)
    (tile-at tile5 row2 col2)    (tile-at tile6 row2 col3)
    (tile-at tile7 row3 col1)    (tile-at tile8 row3 col2))))
```

Solution to exercise 10.5

```
(define (problem blocks-4)
    (:domain blocks)
    (:objects a b c d)
    (:init (clear d) (clear b)
           (on-table c) (on-table a)
           (on d c) (on b a))
    (:goal (and (on a b) (on b c) (on c d))))
```

Solution to exercise 10.6

```
(define (problem gripper-7)
   (:domain gripper-strips)
   (:objects rooma roomb b1 b2 b3 b4 b5 b6 b7 left right)
   (:init (room rooma) (room roomb)
          (gripper left) (gripper right)
          (ball b1) (ball b2) (ball b3)
          (ball b4) (ball b5) (ball b6) (ball b7)
          (at-robby rooma)
          (free left)(free right)
          (at b1 rooma) (at b2 rooma)
          (at b3 rooma) (at b4 rooma)
          (at b5 rooma) (at b6 rooma) (at b7 rooma))
   (:goal (and (at b1 roomb) (at b2 roomb)
               (at b3 rooma) (at b4 roomb)
               (at b5 roomb) (at b6 roomb) (at b7 roomb))))
```

Solution to exercise 10.7.
With
`./fd` *LAB/puzzle − simple.pddl*`LAB/tiles_p01.pddl`
`--heuristic "h=ff()"`
`--search "astar(h)"`
the plan has 4 steps:

```
move-tile-right tile6 row3 col2 col3 (1)
move-tile-down tile8 row2 row3 col2 (1)
move-tile-left tile5 row2 col3 col2 (1)
move-tile-up tile6 row3 row2 col3 (1)
Plan length:  4 step(s).
Plan cost:  4
Initial state h value:  6.
```

Chapter 11

Heuristics for planning

Heuristics are used to guide the search in large state spaces.

Learning objectives for this week are:

1. To investigate the role of heuristics in searching for a plan.
2. To assign costs to each action.
3. To decide on your own running scenario related to planning.

11.1 Defining heuristics by relaxation

Consider that the eight-sliding-tiles domain and the problem depicted in Fig. 10.1 are located in folder `LAB`.

```
./fd $LAB/puzzle-simple.pddl $LAB/tiles_p02.pddl \
    --heuristic "h=ff()" \
    --search "astar(h)"
```

Here, `tiles_p02.pddl` is the problem in listing 10.6. The resulted plan partially appears in listing 11.1. Note that the initial estimation of $f = 26$ steps is exact - the computed plan has indeed 26 steps. The cost of the plan is the same with its length, because each action has a default cost value of 1.

Listing 11.1: Plan generated by the FastDownward planner with A^* and ff.

```
move-tile-down tile5 row2 row3 col3 (1)
move-tile-down tile7 row1 row2 col3 (1)
.....................................
move-tile-left tile8 row3 col3 col2 (1)
Plan length: 26 step(s).
Plan cost: 26
Initial state h value: 26.
```

A more costly solution is obtained with a modified version of A^*, called Weighted A^* (WA^*):

```
./fd $LAB/puzzle-simple.pddl $LAB/tiles_p02.pddl \
--heuristic "h=ff()" \
--search "astar(weight(h, 3))"
```

The obtained plan has 40 steps. The initial estimated cost was $f = 78$, given by $f = g + w \cdot h = 0 + 3 \cdot 26$.

Exercise 11.1 *Which is the length of the plan generated with* `ff` *heuristic and* WA^* *with* $w = 2$*?*

Now consider that some tiles are glued in place and cannot be moved. To handle this, moving actions should additionally check in the preconditions that the tile is not glued, given by `(not (glued ?tile))` in listing 11.2.

Listing 11.2: In the glued-sliding-tiles actions additionally check if the the tiles are not glued.

```
(:action move-tile-down
   :parameters (?tile ?old-row ?new-row ?col)
   :precondition (and (is-tile ?tile)
                      (not (glued ?tile))
                      (is-row ?old-row)
                      (is-row ?new-row)
                      (is-column ?col)
                      (next-row ?old-row ?new-row)
                      (tile-at ?tile ?old-row ?col)
                      (is-blank ?new-row ?col))
   :effect (and (not (tile-at ?tile ?old-row ?col))
                (not (is-blank ?new-row ?col))
                (tile-at ?tile ?new-row ?col)
                (is-blank ?old-row ?col)))
```

Consider the glued-fifteen-puzzle task [17].

```
$ ./fd tile/glued.pddl tile/glued01.pddl\
--heuristic "h=cg()" \
--search "eager_greedy(h,preferred=h)"
```

Note that a `cg()` heuristic is used instead of `ff()`, and the eager-greedy algorithm instead of A^*. A plan of 1987 steps is computed.

Now consider that you can remove tiles from the frame and reinsert them at any blank location. The `remove-tile-from-frame` action has three effects: i) the tile is removed from its position, ii) that position becomes blank, and iii) the tile is not outside the frame. Note in listing 11.3 the new predicate `(is-outside-frame ?tile)`.

Listing 11.3: In the cheating-tile domain tiles can be removed from the frame.

```
(:action remove-tile-from-frame
   :parameters (?tile ?row ?col)
   :precondition (and (is-tile ?tile)
                      (is-row ?row)
                      (is-column ?col)
                      (tile-at ?tile ?row ?col))
   :effect (and (not (tile-at ?tile ?row ?col))
                (is-blank ?row ?col)
                (is-outside-frame ?tile)))
```

Exercise 11.2 *Following the model in listing 11.3, extend the sliding tiles domain with the action* `insert-tile-into-frame` *with three parameters:* `?tile`*,* `?row`*,* `?column`*.*

Exercise 11.3 *Find yourself a plan for the problem in file* `cheating_p01.pddl`*. Does FD planner compute the same plan?*

By running FD on the `cheat.pddl` domain [17] with the problem in Fig 10.1

```
./fd $LAB/cheat.pddl $LAB/cheathing_p01.pddl \
--heuristic "h=cg()" \
--search "astar(h)"
```

the following plan is obtained:

Listing 11.4: Plan computed for the cheating domain.

```
remove-tile-from-frame tile8 row2 col2 (1)
move-tile-left tile5 row2 col3 col2 (1)
remove-tile-from-frame tile6 row3 col2 (1)
insert-tile-into-frame tile6 row2 col3 (1)
insert-tile-into-frame tile8 row3 col2 (1)
Plan length: 5 step(s).
Plan cost: 5
Initial state h value: 6.
```

Exercise 11.4 *Can you obtain a better plan with FD?*

11.2 Search engines and heuristics in the Fast Downward planner

Search engines

Three search algorithms used by FD are detailed next: Astar, Weighted A*, and enforced hill climbing.

Astar. A^* firstly expands nodes with the lowest cost function $f = g + h$. If several nodes have the same f-cost, A* must implement some tiebreaking policy. One such policy is to favor of nodes with least h-estimation. This is also indicated by the following call:

```
--heuristic h=evaluator
--search eager(tiebreaking([sum([g(), h]), h], unsafe_pruning=false),
               reopen_closed=true, f_eval=sum([g(), h]))
```

In the call above, the tiebreaking mechanism uses a list of two elements. If a node has the same sum for `g()` and `h`, the node with the lowest h is expanded. The above call is equivalent to `fd --search astar(evaluator)`.

Exercise 11.5 *FD's tiebreaking strategy for A* is* `[h, fifo]`*. Here, the 2nd-level tiebreaking is fifo. However, [1] has reported that* `[h, lifo]` *outperforms* `[h, fifo]`*. Run some experiments on tiebreaking strategies for A*. Do you tend to confirm the results reported in [1]?*

Generic call for A^* is:

```
astar(eval, mpd=false, pruning=null(), cost\_type=NORMAL,
      bound=infinity, max\_time=infinity)
```

Operations on evaluators (such as *max*, *sum*, or *weight*) can be applied: `max(evals)`, `sum(evals)`, `weight(eval, w(int))`. Here `evals` is a list of evaluators.

Weighted A*. (WA^*) interpolates between A^* and greedy by applying BFS with the cost function $f(n) = (1-\alpha) \cdot g(n) + \alpha \cdot h(n)$, where $\alpha \in [0, 1]$. FD uses the alternative representation obtained by replacing α with $\frac{w}{w+1}$, where $w \geq 1$:

$$f(n) = g(n) + w \cdot h(n)$$

A possible call is `lazy_wastar(evals, w=2)`. Note that for $w = 0$ we have uniform-cost search, while for $w = 1$ we have A^*.

Enforced hill climbing. Recall that hill climbing randomly selects the best successor to each state and restarts are needed when a path became too long. Instead, enforced form of hill climbing (EHC)[11] exploits both local and global search. EHC evaluates all direct successors of the current state s. If none of these successors has a better heuristic value, EHC looks at the successor's successors. This local search process ends when a state s' with $h_{FF}(s') < h_{FF}(s)$ is found. The path to s' is added to the current plan, and search continues with s' as the new starting state. Thus, each search iteration performs complete breadth-first search for a state with strictly better evaluation.

Heuristics in the FD planner

The following heuristics use a relaxed version of the problem in which the negative effects (delete list) for each applicable action is removed. The effect is that once an atom is achieved, it stays achieved.

Max (h_{max})

h_{max} is the maximum of the accumulated costs of the paths to the goal in the relaxed problem. h_{max} is admissible,

Additive (h_{add})

h_{add} is the sum of the costs needed to achieve each subgoal in the goal G. h_{add} computes the cost $c(s, g)$ to reach each proposition $g \in G$ from the state s and then sums the costs. The cost of each proposition g is determined by the index of the first layer in which it appears. h_{add} is not admissible.

Goal count (h_{gc})

h_{gc} counts the number of unachieved goals. h_{gc} is neither admissible nor consistent.

Fast forward (h_{ff})

At AIPS-2000[1], Fast Forward (FF) planner outperformed all the other fully automatic systems. This distinguished performance was due to its h_{ff} heuristic. FF uses forward search guided by the "ignoring delete list" relaxation. The value $h_{ff}(s)$ is the number of actions needed to achieve the goal from state s when assuming the delete lists are all empty. Differently for h_{add} and h_{max}, h_{ff} actually solves the relaxed problem.

Finding a solution for the relaxed problem depends on the implementation of the h_{ff} algorithm. However, computing optimal relaxed solution length is NP hard. The trick is to use Graphplan algorithm [4] that finds a reasonable plan in polynomial time, but with no guarantee that the solution is optimal. After running a relaxed version of Graphplan, the length of the generated plan is used for h_{ff} evaluation.

With a heuristic function computed in polynomial time, the difficulty is now on the number of states for each h_{ff} needs to be computed. Hoping to reach the goal by evaluating few states as possible, an option is hill climbing. h_{ff} uses an enforced form of hill climbing [11] with a prunning strategy called "helpful actions". h_{ff} considers helpful only those applicable actions that add at least one goal at the lowest layer of the relaxed solution. The successors of any state s in breadth-first search are actually restricted to $Helpful(s)$. This "helpful action" pruning

[1] http://www.cs.toronto.edu/aips2000/

does not preserve completeness. That's why, when a solution is not found, one can simply switch to a WA* algorithm, that is complete.

For other heuristics such as the Causal Graph heuristic (h_{cg}), the eager student is referred to [8] where it was firstly introduced. The heuristic has been later generalised to Context-Enhanced Additive heuristic (h_{cea}) [10].

Combination of heuristics

FD allows combination of heuristics through operators such as `max(h1,h2)` or `sum(h1,h2)`. Note that maximum of admissible heuristics is admissible. Sum yields a stronger heuristic, but not necessarily admissible.

Some predefined combination of heuristics are saved as aliases in the `src/driver/aliases.py module`. For instance, the `seq-sat-lama-2011` configuration can be run with:

```
./fd --alias seq-sat-lama-2011 misc/tests/benchmarks/gripper/prob01.pddl
```

To see all available aliases run:

```
./fast-downward.py --show-aliases
```

Exercise 11.6 *Run FD planner on the 8-sliding-tiles problem depicted in figure 3.4, page 71, AIMA 3rd edition.*

1. *Which is the length of the optimal plan?*
2. *Play with the FD options to obtain the optimal plan or a plan close to it. Write the length of the obtained plan and the running parameters.*
3. *Identify the best plan obtained by one of your colleagues. Write the running parameters for this case.*

11.3 Introducing costs to actions

FD planner supports `:action-costs` requirement from PDDL 3.1. Consider the domain in listing 11.5. Note that `:requirements` include the action costs. Two functions are defined: `weight` with one parameter (that is a tile) and `total cost`. The `increase` keyword is used to update the cost of an action.

Listing 11.5: Sliding tiles domain with costs.

```
(define (domain weighted-sliding-tile)

 (:requirements :strips :action-costs)

 (:predicates
  (next-row ?r1 ?r2)
  (next-column ?c1 ?c2)
  (tile-at ?t ?r ?c)
  (is-blank ?r ?c))

 (:functions
  (WEIGHT ?t) - number
  (total-cost) - number)

 (:action move-tile-down
   :parameters (?tile ?old-row ?new-row ?col)
```

```
    :precondition (and (next-row ?old-row ?new-row)
                       (tile-at ?tile ?old-row ?col)
                       (is-blank ?new-row ?col))
    :effect (and (not (tile-at ?tile ?old-row ?col))
                 (not (is-blank ?new-row ?col))
                 (tile-at ?tile ?new-row ?col)
                 (is-blank ?old-row ?col)
                 (increase (total-cost) (WEIGHT ?tile))))
 )
```

Assume that the odd tiles are somehow more difficult to be moved. In listing 11.6, odd tiles have a cost of 2 while even tiles a cost of 1.

Listing 11.6: Assigning costs to each tile: odd tiles are difficult to move.

```
(define (problem 8-puzzle-with-costs)
  (:domain weighted-sliding-tile)
  (:objects
   tile1 tile2 tile3 tile4 tile5 tile6 tile7 tile8
   row1 row2 row3 col1 col2 col3)

  (:init
   (next-row row1 row2)         (next-row row2 row3)
   (next-column col1 col2)      (next-column col2 col3)
   (tile-at tile8 row1 col1)    (tile-at tile6 row1 col2)
   (tile-at tile4 row1 col3)    (tile-at tile5 row2 col1)
   (tile-at tile7 row2 col2)    (tile-at tile2 row2 col3)
   (tile-at tile3 row3 col1)    (tile-at tile1 row3 col2)
   (is-blank row3 col3)

   (= (total-cost) 0)

  ;; Odd tiles are difficult to slide.
   (= (WEIGHT tile1) 2) (= (WEIGHT tile2) 1)
   (= (WEIGHT tile3) 2) (= (WEIGHT tile4) 1)
   (= (WEIGHT tile5) 2) (= (WEIGHT tile6) 1)
   (= (WEIGHT tile7) 2) (= (WEIGHT tile8) 1)
  )

  (:goal (and (is-blank row1 col1)))

  (:metric minimize (total-cost))
)
```

Exercise 11.7 *Solve the problem in listing 11.6 by hand. Which is the cost of your plan? Ask FD to compute a plan with minimum cost.*

Exercise 11.8 *Consider that `tile4` is almost glued. You need 7 tries to slide it. That is, its cost is 7. Does FD is able to find the optimal plan in this case?*

Exercise 11.9 *Consider that the `shoot(arrow)` action in the Wumpus world has a cost of 10. Which is the minimum cost plan computed by Fd for the third problem in Figure 10.4?*

11.4 Modelling planning domains

Your task is to model a planning domain in PDDL. You can either extend a given domain, or write one from scratch. You should also create several problems for your domain, and verify

that at least one planning engine can solve them. Feel free to use ADL features (conditional effects, quantified preconditions, etc.) as long as these features are supported by your planner. You might want to validate the obtained plans with the *validator* tool. Then you will conduct empirical investigation of the performance of different planning engines and heuristics, using the domain you created in the first part. To do this, you will create a set of problems of increasing size and complexity, and you will measure runtime. You will identify some parameters that you can use to increase problem's complexity (like number of objects in the init state or number of predicates in the goal. Next, you will create problems for your domain by scaling up your parameters. Feel free to use your favorite programming language to automatically generate this experimental problems. During experiments, it is reasonable to set a time limit between 5 and 10 minutes. Additionally, feel free to introduce some incomplete knowledge in the init state or nondeterministic effects. You will see in the following chapter how to handle incompleteness and non-determinism with conformant and contingent planning.

Browse the available repositories with the aim to find yourself an idea for an original domain.

Exercise 11.10 *Investigate classical planning domains (such as Assembly, Blocks, Grid, Gripper, Hanoi, Logistics, Travel Salesman, Tire-world) described at https://fai.cs.uni-saarland.de/hoffmann/ff-domains.html.*

Exercise 11.11 *Investigate modern planning domains (such as Parking, CityCar, Barman, Genome) at https://helios.hud.ac.uk/scommv/IPC-14/domains.html.*

Exercise 11.12 *Investigate the domains from the* `ipc` *directory.*

Action Description Language (ADL) extends STRIPS with:

- negative preconditions: robot can move a ball if it is NOT carrying anything.
- disjunctive preconditions: (you can travel by bus if you have a ticket or a season ticket or you sent an SMS
- disjunctive goals: you want to have `(on a b)` or `(on b a)`
- conditional effects

You can enact full ADL expressivity using `(:requirements :adl)`. An example of using ADL appears in `airport-adl`.

A comprehensive set of planning domains can be obtained by querying the following API: http://api.planning.domains/. You might find helpful the online PDDL editor provided by at http://lcas.lincoln.ac.uk/fast-downward. Note that this editor does not allow you to change the planner's parameters.

Use the plan validator VAL (https://github.com/KCL-Planning/VAL) [14] to independently verify that the generated plans are correct. VAL can automatically produce a LATEX report of the plan validation. The report includes a step by step account of plan validation, plan repair advice if necessary and Gantt charts.

```
1 $validate -l domain.pddl problem.pddl plan.pddl
```

The Gantt chart shows the times over which actions are active, their duration, concurrent activity and dependency.

```
1 ./validate -l tile/puzzle.pddl tile/puzzle01.pddl sas\_plan
```

Exercise 11.13 *Identify a domain that you will formalise in PDDL. Start writing basic predicates and actions. Use the bottom up approach: after writing a simple action, immediately test it with appropriate init and goal states.*

You might find useful the template in listing 11.7:

Listing 11.7: Template for modelling your scenario.

```
(define (domain Template)
(:requirements :strips :typing :action-costs :adl)

(:types type1 - object)

(:predicates (pred0 ?x - type1))

(:action act1
  :parameters (?x - type1)
  :precondition (and (pred0 ?x))
  :effect (and (not (pred0 ?x))))
```

11.5 Solutions to exercises

Solution to exercise 11.1.
The plan has 32 steps, computed with:

```
./fd $LAB/puzzle-simple.pddl $LAB/tiles_p02.pddl
--heuristic "h=ff()"
--search "astar(weight(h, 2))"
```

Solution to exercise 11.2.

```
1  (:action insert-tile-into-frame
2    :parameters (?tile ?row ?col)
3    :precondition (and (IS-TILE ?tile)
4                       (IS-ROW ?row)
5                       (IS-COLUMN ?col)
6                       (is-outside-frame ?tile)
7                       (is-blank ?row ?col))
8    :effect (and (not (is-outside-frame ?tile))
9                 (not (is-blank ?row ?col))
10                (tile-at ?tile ?row ?col)))
```

Solution to exercise 11.4
A plan of length four is obtained with A^* and ff.

```
./fd $LAB/cheat.pddl $LAB/cheating_p01.pddl
    --heuristic "h=ff()"
    --search "astar(weight(h,1))"
```

The computed plan is:

```
    remove-tile-from-frame tile6 row3 col2 (1)
    move-tile-down tile8 row2 row3 col2 (1)
    move-tile-left tile5 row2 col3 col2 (1)
    insert-tile-into-frame tile6 row2 col3 (1)
    Plan length:  4 step(s).
    Plan cost:  4
    Initial state h value:  7.
```

Solution to exercise 11.6
AIMA states that the solution is 26 steps long.

Solution to exercise 11.7

A plan of cost four is obtained with:
../fd $LAB/weight_simple.pddl$LAB/weight_p01.pddl

```
    --heuristic "h=ff()"
    --search "eager_greedy(h,preferred=h)"
```

The computed plan is:

```
    move-tile-down tile2 row2 row3 col3 (1)
    move-tile-down tile4 row1 row2 col3 (1)
    move-tile-right tile6 row1 col2 col3 (1)
    move-tile-right tile8 row1 col1 col2 (1)
```

Solution to exercise 11.8

A plan of cost five is obtained with:

```
../fd LAB/weight_simple.pddl LAB/weight_p02.pddl
     --heuristic "h=ff()"
     --search "astar(h)"
```

The computed plan is:

```
      move-tile-down tile2 row2 row3 col3 (1)
     move-tile-right tile7 row2 col2 col3 (2)
     move-tile-down tile6 row1 row2 col2 (1)
     move-tile-right tile8 row1 col1 col2 (1)
```

Chapter 12

Planning and acting in the real world

When model real world domains, one should consider partial observability and nondeterminism. For instance, the initial situation may be only partially specified or actions may have nondeterministic effects. To handle these, we need *conformant planning*. Moreover, ability to observe aspects of the current state is handled by *contingent planning*.

Learning objectives for this week are:

1. To model uncertain information into planning domains.
2. To formalise nondeterministic actions.
3. To include observations during plan execution.

12.1 Planning domains with uncertainty

Blocks world with partially observed init state. In blocks world, assume the uncertainty is that the top n blocks on each initial stack are arranged in an unknown order (see listing 12.1).

Listing 12.1: Blocks world with uncertainty in the init state.

```
(define (problem BW-rand-5)
  (:domain blocksworld)
  (:objects b1 b2 b3 b4 b5)
  (:init
     (on-table b1) (on-table b2) (on-table b4)
     (clear b2) (clear b4)
     (unknown (on b5 b1)) (unknown (clear b5)) (unknown (on b5 b3))
     (unknown (on b3 b1)) (unknown (clear b3)) (unknown (on b3 b5))
     (or (not (on b5 b3)) (not (on b3 b5)))
     (or (not (on b3 b5)) (not (on b5 b3)))
     (oneof (clear b5) (clear b3))
     (oneof (on b5 b1) (on b3 b1))
     (oneof (on b5 b1) (on b5 b3))
     (oneof (on b3 b1) (on b3 b5))
     (oneof (clear b5) (on b3 b5))
     (oneof (clear b3) (on b5 b3)))
  (:goal (and (on b2 b5) (on b4 b2) (on b5 b1))))
```

`Unknown` construct expresses the set of all possible assignments to an object.
`(oneof e1 . . . en )` indicates that exactly one of the `ei` effects will take place.

Packets with bomb. This example is described by Kushmerick, Hanks, and Weld (1995). In "packets with bomb" domain [16], a robot is given several packages `pi`, and told that some of them may contain bombs `bi`. Robot's goal is to defuse the bomb, and the only way to do so is to dunk the package containing the bomb in the toilet. Placing a package in the toilet might clog the toilet. The robot can sense if a bomb is within a packet by using the observation action `senseIN` (see listing 12.2). Action `dunk` can be executed if the toilet is not clog or stuck. Note the conditional effect `(defused ?b)` takes place only when the bomb is in the package `(in ?p ?b)`. Important here is the nondeterministic effect `(stuck ?t)`, signaled by the keyword `nondet`.

Without preconditions, action `flash` can be executed at each time instance with the certain effect of unclog the toilet `(not (clog ?t))`. Action `unstick` can also be executed anytime, but its effect is conditioned by the state variable `(stuck ?toilet)`.

Listing 12.2: Packets and bomb domain with sensing actions.

```
(define (domain btnd)
  (:types package bomb toilet)
  (:predicates
     (in ?p - package ?b - bomb)
     (defused ?b - bomb)
     (clog ?toilet - toilet)
     (stuck ?toilet - toilet))

  (:action senseIN
     :parameters (?p - package ?b - bomb)
     :observe (in ?p ?b))

  (:action dunk
     :parameters (?p - package ?b - bomb ?t - toilet)
     :precondition (and (not (clog ?t)) (not (stuck ?t)))
     :effect (and (when (in ?p ?b) (defused ?b))
                  (clog ?t)
                  (nondet (stuck ?t))))

   (:action flush
      :parameters (?t - toilet)
      :effect (not (clog ?t)))

   (:action unstick
      :parameters  (?toilet - toilet)
      :effect (and (when (stuck ?toilet) (not (stuck ?toilet))))))
```

To solve problems such as those formalised in listings 12.1 and 12.2, we need tools able to do conformant planning and contingent planning.

12.2 Conformant planning

Differently for classical planning, conformant planning deals with uncertainty in the initial state. Moreover, actions can be nondeterministic. A solution is a plan that is guaranteed to take us from any initial state to some goal state, no matter what the effect of actions is.

We will use the FF-conformant planner[1]. [13] describes the mechanism of the FF-conformant planner as follows: First, enforced hill-climbing (EHC) is tried. The current search state s is set to the initial state. While $s \neq goal$:

[1] Available at https://fai.cs.uni-saarland.de/hoffmann/cff.html

1. Perform breadth-first search starting from s such that $h(s') < h(s)$. During search: avoid repeated states; cut out states s' with $h(s') = \infty$; and expand only the successors of s' generated by the actions in $H(s)$. $H(s)$ is the set of *helpful actions* to state s, that are those actions that could be selected to start the relaxed plan.

2. If $\neg\exists s'$ with $h(s') < h(s)$ then fail, else $s := s'$.

Second, if EHC fails, apply complete best-first search by expanding all states in order of increasing h value. Here, $h(s)$ is the number of actions in a relaxed plan to the goal state. The PDDL representation is translated into CNF, hence a satisfiability solver (such as Chaff [19] or DPLL) is used.

The planner gets the operator file (that is the domain) and the fact file, (that is the planning task. An example of calling the planner is:

```
./Conf-FF -o cff-tests/blocks/domain.pddl -f cff-tests/blocks/p1.pddl
```

Exercise 12.1 *Run the examples provided in the* `cff-tests` *directory.*

12.3 Contingent planning

Contingent planning generates conditional plans under uncertainty about the initial state and action effects, but with the ability to observe some aspects of current state. Hence,

$$contingent\ planning = conformant\ planning + observations \tag{12.1}$$

In the non-deterministic blocks world, we have uncertainty about the initial arrangement of the top n blocks on each stack, and one must observe the block positions before proceeding. In the non-deterministic bomb domain, flushing a package non-deterministically deletes a new fact (`unstuck(toilet)`) that then must be re-achieved.

We will use Contingent-FF planner[2]. Contingent-FF adds observations and uncertainty to sequential STRIPS with conditional effects [12].

Observation actions. Beside the normal actions, there are now special observation actions (see listing 12.3). Each observation action splits the belief state and introduces two branches into the plan: one for the positive effect and one for the negative effect.

Listing 12.3: Sensing actions in the blocks domain.

```
(:action senseON
   :parameters (?b1 ?b2)
   :observe (on ?b1 ?b2))

(:action senseCLEAR
   :parameters (?b1)
   :observe (clear ?b1))
```

Exercise 12.2 *Following the model in listing 12.3, define the sensing action* `senseOnTable` *used to observe if a block is on table.*

In listing 12.4 what is known is that block `b1` is on table and nothing is on `b2`. We don't know if `b2` is on table, if `b1` is clear and if `b2` is on `b1`. What we do know is either `b1` is clear or `b2` is on `b1`.

[2]Available at https://fai.cs.uni-saarland.de/hoffmann/cff.html

Listing 12.4: Conformant task in the blocks domain.

```
(define (problem BW-2)
(:domain blocksworld)
(:objects b1 b2)
(:init
    (on-table b1)
    (clear b2)
    (unknown (on-table b2))
    (unknown (clear b1))
    (unknown (on b2 b1))
    (oneof (clear b1) (on b2 b1)))
(:goal (and (on b1 b2) )))
```

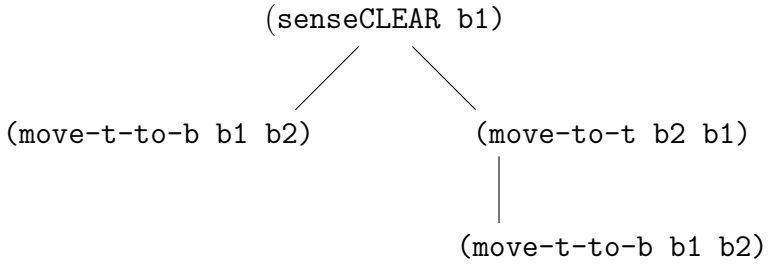
The planner gets the operator file (that is the domain) and the fact file (that is the planning task. An example of calling the planner is:

```
./Contingent-FF -o contff-tests/blocks/domain.pddl -f contff-tests/blocks/p1.pddl
```

The resulted plan has three layers (noted below with 0, 1 and 2) and four actions:

```
-------------------------------------------------------
  0||0 --- SENSECLEAR B1 --- TRUESON: 1||0 --- FALSESON: 1||1
-------------------------------------------------------
  1||0 --- MOVE-T-TO-B B1 B2 --- SON: 2||-1
  1||1 --- MOVE-TO-T B2 B1 --- SON: 2||0
-------------------------------------------------------
  2||0 --- MOVE-T-TO-B B1 B2 --- SON: 3||-1
-------------------------------------------------------
```

The plan is more clear presented in the following diagram:



Here, the left branch is the `true` son, while the right branch is the `false` son.

Exercise 12.3 *Using Contingent-FF, can you find another plan?*

Nondeterministic effects. Normal actions can have non-deterministic effects. FF-contingent is limited to a single unconditional non-deterministic effect per action.

A problem for the "bomb and toilet" nondeterministic domain (btnd) appears in listing 12.5. There are two bombs, four packages and one toilet. We know that bomb `b0` is in package `p0`. We also know that `b1` is in one of the remaining three packages `p1`, `p2` or `p3`. Goal is to defuse both bombs.

Listing 12.5: Nondeterministic bomb domain.

```
(define (problem btnd-4)
   (:domain btnd)
   (:objects
      b0 b1 - bomb
      p0 p1 p2 p3 - package
      t0 - toilet)
```

```
7     (:init
8        (in p0 b0)
9        (unknown (in p1 b1))
10       (unknown (in p2 b1))
11       (unknown (in p3 b1))
12       (oneof (in p1 b1) (in p2 b1) (in p3 b1)))
13 (:goal (and (defused b0) (defused b1))))
```

By running the Contingent-FF planner with the default settings:

```
1 ./Contingent-FF -o contff-tests/blocks/domain.pddl -f contff-tests/blocks/p1.pddl
```

the plan in listing 12.6 is obtained.

Listing 12.6: A linear plan for the nondeterministic bomb domain.

```
1  ------------------------------------------------
2    0||0 --- DUNK P3 B1 T0 --- SON: 1||0
3  ------------------------------------------------
4    1||0 --- FLUSH T0 --- SON: 2||0
5  ------------------------------------------------
6    2||0 --- UNSTICK T0 --- SON: 3||0
7  ------------------------------------------------
8    3||0 --- DUNK P0 B0 T0 --- SON: 4||0
9  ------------------------------------------------
10   4||0 --- UNSTICK T0 --- SON: 5||0
11 ------------------------------------------------
12   5||0 --- FLUSH T0 --- SON: 6||0
13 ------------------------------------------------
14   6||0 --- DUNK P2 B1 T0 --- SON: 7||0
15 ------------------------------------------------
16   7||0 --- UNSTICK T0 --- SON: 8||0
17 ------------------------------------------------
18   8||0 --- FLUSH T0 --- SON: 9||0
19 ------------------------------------------------
20   9||0 --- DUNK P1 B1 T0 --- SON: 10||-1
21 ------------------------------------------------
```

Cost of the plan is given by the number of layers (10) and number of actions (10). That is, the plan is linear with no sensing actions.

Exercise 12.4 *Using Contingent-FF, can you find better plan? Does the plan include observation actions? Is it the optimal plan?*

Exercise 12.5 *Run the examples provided in the* `contff-tests` *directory.*

Exercise 12.6 *Solve the planning task described in section 11.2 AIMA, third edition.*

12.4 Solution to exercises

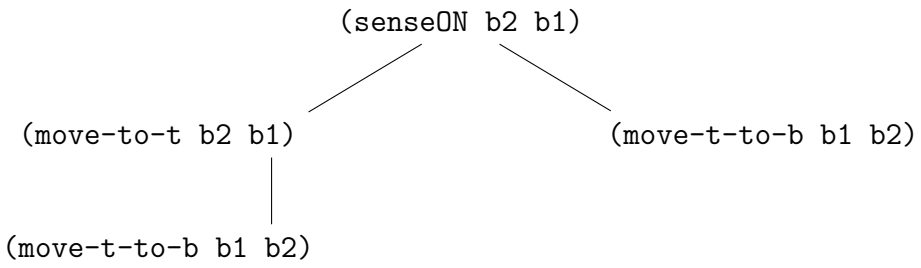
Solution to exercise 12.2.

```
(:action senseONTABLE
      :parameters (?b1)
      :observe (on-table ?b1))
```

Solution to exercise 12.3
Searching with AO* instead of Greedy AO* with

```
./Contingent-FF -o contff-tests/blocks/domain.pddl
                -f contff-tests/blocks/p1.pddl
                -a 0
```

the following plan was found:



Here, the left branch is the `true` son, while the left branch of the `false` son.

Solution to exercise 12.4.
With:

```
./Contingent-FF -o contff-tests/btnd/domain.pddl
                -f contff-tests/btnd/p4.pddl
                -a 0 -w 3
```

you obtain:

```
-----------------------------------
0——0 — DUNK P0 B0 T0 — SON: 1——0
-----------------------------------
1——0 — UNSTICK T0 — SON: 2——0
-----------------------------------
2——0 — FLUSH T0 — SON: 3——0
-----------------------------------
3——0 — SENSEIN P1 B1 — TRUESON: 4——0 — FALSESON: 4——1
-----------------------------------
4——0 — DUNK P1 B1 T0 — SON: 5——-1
4——1 — DUNK P3 B1 T0 — SON: 5——0
-----------------------------------
5——0 — UNSTICK T0 — SON: 6——0
-----------------------------------
6——0 — FLUSH T0 — SON: 7——0
-----------------------------------
7——0 — DUNK P2 B1 T0 — SON: 8——-1
-----------------------------------
```

As the number of layers is 8 and number of actions 9, this plan is better. Note the `SensoOn` observation action. Given the size of the problem, try to find by hand the optimal plan.

Chapter 13

Knowledge representation with event calculus

The commonsense knowledge about the effects of events on fluents is modelled with event calculus.

Learning objectives for this week are:

1. To see the structure of a DECReasoner input corresponding to a domain description.
2. To perform reasoning tasks in event calculus: prediction, abduction, postdiction.
3. To understand and describe common sense reasoning in event calculus.

13.1 Discrete Event Calculus reasoner

We will use the Discrete Event Calculus Reasoner (DECReasoner)[1]. The tool supports various types of reasoning: deduction, temporal projection, abduction, planning, postdiction, and model finding [20]. DECReasoner translates the domain into a satisfiability (SAT) problem. The SAT solver that we will use is Relsat[2]. Relsat [3] is already compiled for you in the `solvers` directory.

Basic notions of event calculus are `events` and `fluents`. An event `e` is an action that may occur in the world, such as a robot picking an object. An event may occur or happen at a time instance. A fluent `f` is a property of the world that is varies in time, such as the location of the robot. The commonsense knowledge about the effects of events on fluents is modelled with the following predicates: `HoldsAt(f, t)`, `Happens(e, t)`, `Initiates(e, f, t)`, `Terminates(e, f, t)`. For the Semantics of these predicates consult [23] or [21].

Test the DECReasoner with:

```
1  cd decreasoner
2  python
3 >>> import decreasoner
4 >>> decreasoner.test()
```

The model shows the fluents that are true at timestep zero. For the following timesteps, fluents that become true are signaled with ("+"), and fluents that become false with ("-"). To show all fluents that are true or false at each timestep, you need to add the following line in

[1] Available at http://decreasoner.sourceforge.net

[2] The diligent student should recall that Mace4 is also a SAT solver.

the input file: `option timediff off`. Variables start with lowercase letters, constants with uppercase letters.

13.2 Reasoning modes in event calculus

There are three reasoning modes for event calculus: prediction, abduction and postdiction.

Prediction

For prediction (or temporal projection) we start with an initial state and some events and then reason about the state that results from the events.

```
1  agent Nathan
2  fluent Awake(agent)
3  event WakeUp(agent)
4  event FallAsleep(agent)
5
6  Initiates(WakeUp(agent),Awake(agent),time).
7  Terminates(FallAsleep(agent),Awake(agent),time)
8
9  !HoldsAt(Awake(Nathan),0).
10
11 Happens(WakeUp(Nathan),1).
12
13 completion Happens
14
15 range time 0 3
16 range offset 1 1
```

In listing 13.2, `Nathan` is not awake at timestep 0. The event `WakeUp` happens at timestep 1. Based on this narrative, the reasoner can infer that `Nathan` is awake at timestep 2, given by `HoldsAt(Awake(Nathan),2)`. Based on the common law of inertia, `Nathan` will be also awake at time step 3. Note that the reasoning is limited to the time interval 0 to 3, specified with`range time 0 3`.

```
1  1 model
2  ---
3  model 1:
4  0
5  1
6  Happens(WakeUp(Nathan), 1).
7  2
8  +Awake(Nathan).
9  3
```

Planning in event calculus

For abduction (or adductive planning), we start with an initial state and a final state and then reason about the events that lead from the initial state to the final state

```
1  >>> decreasoner.run('examples/Mueller2006/Chapter2/Sleep2.e')
```

```
1  agent Nathan
2  fluent Awake(agent)
3  event WakeUp(agent)
4  event FallAsleep(agent)
```

```
5
6  Initiates(WakeUp(agent),Awake(agent),time).
7  Terminates(FallAsleep(agent),Awake(agent),time)
8
9  !HoldsAt(Awake(Nathan),0).
10 HoldsAt(Awake(Nathan),1).
```

The reasoner infers that the action `Happens(WakeUp(Nathan),0)` should be executed. This is the only action that leads the agent from the init state `!HoldsAt(Awake(Nathan),0)` to the goal state `HoldsAt(Awake(Nathan),1)`.

You can test this by loading the file `Sleep2.e`:

```
1 >>> decreasoner.run('examples/Mueller2006/Chapter2/Sleep2.e')
```

Note that a STRIPS operator can be translated into the language of event calculus. The action `op` below

```
1 (:action op
2    (:precondition: (and c1 c2)
3    (:effect: (and p1
4                   p2
5                  (not n1)
6                  (not n2))))
```

is similar to the following set of event calculus formulas:

```
c1 & c2 ⇒ Terminates(op, n1, t)
c1 & c2 ⇒ Terminates(op, ni, t)
c1 & c2 ⇒ Initiates(op, p1, t)
c1 & c2 ⇒ Initiates(op, pi, t)
```

Postdiction

We start with some events that lead to a state and then reason about the state prior to the events.

```
1 >>> decreasoner.run('examples/Mueller2006/Chapter2/Sleep3.e')
```

Exercise 13.1 *Run and analyse the prediction reasoning task from the* ***monkey and banana*** *scenario in the* ***example/GiunchigliaEtAl2004****. The scenario is described in [7].*

13.3 Carying a newspaper example

This example is introduced in [21].

Example 1 *In the living room, Lisa picked up a newspaper and walked into the kitchen. Where did the newspaper end up? (It ended up in the kitchen.)*

```
1 object Newspaper
2 agent Lisa
3 room LivingRoom, Kitchen
4
5 event LetGoOf(agent,object)
6 event PickUp(agent,object)
7 event Walk(agent,room,room)
8
```

```
fluent InRoom(object,room)
fluent Holding(agent,object)

Initiates(Walk(agent,room1,room2),InRoom(agent,room2),time)

room1 \neq room2 -> Terminates(Walk(agent,room1,room2),InRoom(agent,room1),time)

HoldsAt(InRoom(agent,room),time)
HoldsAt(InRoom(object,room),time) ->
Initiates(PickUp(agent,object),Holding(agent,object),time)

HoldsAt(Holding(agent,object),time) ->
Terminates(LetGoOf(agent,object), Holding(agent,object),time)

HoldsAt(Holding(agent,object),time) ->
Initiates(Walk(agent,room1,room2),InRoom(object,room2),time)

HoldsAt(Holding(agent,object),time) & room1 \neq room2 ->
Terminates(Walk(agent,room1,room2),InRoom(object,room1),time)


HoldsAt(InRoom(object,room1),time) &
HoldsAt(InRoom(object,room2),time) -> room1=room2

Happens(PickUp(Lisa,Newspaper),0)

Happens(Walk(Lisa,LivingRoom,Kitchen),1)

HoldsAt(InRoom(Lisa,LivingRoom),0)
HoldsAt(InRoom(Newspaper,LivingRoom),0)
\neg HoldsAt(Holding(Lisa,Newspaper),0)
\neg HoldsAt(Holding(agent,agent),time)
```

```
 Model 1:
0
InRoom(Newspaper, LivingRoom).
InRoom(Lisa, LivingRoom).
Happens(PickUp(Lisa, Newspaper), 0).
1
+Holding(Lisa, Newspaper).
Happens(Walk(Lisa, LivingRoom, Kitchen), 1).
2
-InRoom(Newspaper, LivingRoom).
-InRoom(Lisa, LivingRoom).
+InRoom(Newspaper, Kitchen).
+InRoom(Lisa, Kitchen).
```

13.4 Solutions to exercises

Solution for exercise 13.1.

```
decreasoner.run('examples/GiunchigliaEtAl2004/MonkeyPrediction.e')
decreasoner.run('examples/GiunchigliaEtAl2004/MonkeyPlanning.e')
```

The reasoner computes the following plan:

```
Happens(Walk(L3), 0).
Happens(PushBox(L2), 1).
Happens(ClimbOn(), 2).
Happens(GraspBananas(), 3).

decreasoner.run('examples/GiunchigliaEtAl2004/MonkeyPostdiction.e')
```

Appendix A

Brief synopsis of Linux

Here you have some basic, useful Linux commands. This synopsis is not meant to be exhaustive or to always offer the best way to achive one goal. It is rather intended to help a beginner use a Linux system and get a quick start in running the lab applications. The synopsis is mainly concerned with the Fedora distribution (the one which is used in the lab). Unless otherwise specified, the commands are to be typed at a terminal prompt. The argument(s) required should be obvious once you look at the example provided and at its description.

How to begin and end a work session

In the login screen, you should enter: you username (aka login name or account name or simply user) and your password, as provided by the administrator. Henceforthh, the username will be `iia`.

```
login:  iia
password:            < -- Please continue typing even if no echo is shown!
```

Typically, your current directory will be `/home/iia` (we'll call this your *home directory*). If you open a terminal, you will "arrive" in that directory. The home directory is denoted by `~` (the tilde character). Most of the time it is here that you find a file which has been copied from a remote computer (e.g., a file copied to you machine by your teacher), unless otherwise specified by the person who did this (and most of the time your teacher will not bother to specify a different destination).

To quit your session, search for `System -> Logout` in the upper right corner of the graphical interface.

Shut down a system: `poweroff`

Alternatively to typing `poweroff` at the prompt, you may use `System -> Shut down`, also available from the upper right corner of the graphical interface.

Pay attention! Before shutting down a computer, always do:

```
who
```

in a terminal, in order to see whether some other users are connected to your computer (don't forget Linux is actually a multiuser operating system – more users can work on the same machine in the same time and share its resources. See more details in Section A).

Help for a command: `man`

Example:

```
man ls
```
shows a help for the command `ls` (list all files in a folder/directory).

Get the IP address of your computer: `ifconfig`

Your LAN interface is typically called `eth0`. The IP address consists of 4 dot-separated numbers, e.g., `192.168.1.3`

Connecting to another computer via secure shell: `ssh`

Example: If you are working on computer `c1` and want to connect to computer `c2`, in the `iia` account, you should do on `c1`:

```
ssh -X iia@c2
```

then type the password of `iia` on computer `c2` (even if you see no echo when typing). Here, `c2` is either the name or the IP address of the remote computer. You will get acces to a terminal on `c2` and be able to run programs there, including GUI featuring programs (due to the `X` option).

To close (i.e. end) the connection, just type in the remote terminal:

```
exit
```

Copy files on some other computer via secure copy: `scp`

Example: if you want to copy file `f1` from computer `c1` to destination computer `c2`, into account `iia`, open a terminal on `c1`, change directory (see Section A) to the one containing `f1` and type:

```
scp f1 iia@c2:                    <-- Please note the colon at the end!
```

then type the password of `iia` on computer `c2`. `f1` is copied on computer `c2`, in `/home/iia`.

Directories and files

The directory system has a unique root called `/` (similar to `C:` in Windows). File names are **Case Sensitive**. Files have no specific extensions; a dot "`.`" might be a part of the file name (a file could be named `f`, `f.c` or even `f.1.2.3.c`).

Change current directory: `cd`

Example: change directory to `/home/iia/john`:

```
cd /home/iia/john
```

Example: change directory to the parent of the current directory:

```
cd ..
```

Example: change directory to your home directory:

```
cd ~
```

or simply

```
cd
```

Print working directory (the name of the current directory): `pwd`

The command `pwd` prints the complete name of the current directory. Alternatively, you can do `echo $PWD` (see also Section A).

Paths to files

Let us assume that the current directory is `/home/iia` and we create here a folder called `d1`, then, inside `d1`, a file called `f1.txt`. We can reffer to the file `f1.txt` either as `d1/f1.txt` (we will say we use the "relative path") or as `/home/iia/d1/f1.txt` (in this case, we use the "absolute path"). If we type `./d1/f1.txt`, we also use a relative path as a dot means "the current directory", which is, for now, `/home/iia` .

If we now change directory to `d1`, we can reffer to `f1.txt` as simply `f1.txt` or as `./f1.txt` (as you have probably noticed, the current directory is now `/home/iia/d1`), or even as `/home/iia/d1/f1.tx` Please note the absolute path remains the same, while the relative path (not surprisingly) changes.

Here, by "path" we mean the name of a file preceded by the names of the directories containing it.

Symbolic Links: `ln -s`

This command creates a symbolic link (like a Windows shortcut: an alternative name for a file). It's usually used in order to give a "constant" name to a file whose "true" name changes over time (e.g., a library, whose name includes the version number). Example:

```
ln -s /usr/local/lib/myjar.0.1.23.jar myjar.jar
```

Archives: `tar` (or via GUI)

To create a compressed archive called `archive_name.tar.gz`, you should do:

```
tar zcvf archive_name.tar.gz f1 f2 d1
```

where `f1, f2, d1` are either files or directories to be added to the archive. When adding a directory to an archive its whole structure is preserved.

You can do the same job via a graphical interface, by right clicking on the file.

To decompress and extract files from an archive, you may use:

```
tar zxvf archive_name.tar.gz
```

Midnight Commander (a Total Commander-like tool): `mc`

The `mc` command starts a tool which allows you to create, copy, delete files or directories by using the Functional Keys. One can see their functionalities at the bottom of the panels (e.g., `F7` allows creating a new directory).

Environment variables

They contain data which is important for the whole system. Examples:

- `PATH` contains the name of directories, separated by colons, in which a file launched into execution is to be searched
- `CLASSPATH` tells Java applications and JDK tools where to search for classes
- `PWD` gives the present working directory

The `env` command shows all environment variables and their values.

Setting: use `export`

You need to type:

```
VARIABLE=value
export VARIABLE
```

or:

```
export VARIABLE=value
```

Example:

```
export PATH=$PATH:/home/iia/d1
```

says the new value of `PATH` will be its old value (`$PATH`), to which we append (via `:`) the value `/home/iia/d1`

Example:

```
export PATH=$PATH:.
```

appends the value `.` (i.e., the "current directory") to `PATH` ; as you remember, one could always reffer to the current directory by using the character `.` (a dot).

Let us assume now the current directory is `/home/iia` and we create here a folder called `d2` and a file called `f1.sh` in `d2`, then we set the execution rights to `f1.sh` as explained in Section A. In order to run `f1.sh`, we have the following options:

1. using the absolute path: `/home/iia/d2/f1.sh`
2. using the relative path: `./d2/f1.sh` or simply `d2/f1.sh`
3. if the directory containing `f1.sh` (i.e., `/home/iia/d2`) has been added to `PATH`, you can just type `f1.sh` and this will work no matter which your current directory is
4. change directory to `d2` and type `./f1.sh`
5. provided you have changed directory to `d2`, you can do a `f1.sh` but this only works if the current directory, denoted in Linux by a dot, has been added to the `PATH`. If this is not the case and the directory containing `f1.sh` has not been added to the `PATH` either, then simply typing `f1.sh` WILL NOT WORK and will return a `command not found` error.

Let us assume the dot is in the `PATH`, but no other directories have been added there, and you have a file called `f1.sh` in directory `d1` and a file also called `f1.sh` in directory `d2`. If you type just `f1.sh`, you will be able to run `f1.sh`, but the version to be run depends on your current directory. So, if you do `cd /home/iia/d1` then `f1.sh`, you will actually run `/home/iia/d1/f1.sh`. But if you do `cd /home/iia/d2` then `f1.sh`, you will actually run `/home/iia/d2/f1.sh`.

The command `export PATH=$PATH:.` in the example above basicly tells that, if the user tries to run file `x` by typing `x` at the prompt, the shell must search `x` in the directory where the command `x` has been issued, besides the other directories in the `PATH`.

To find out the absolute path to a program `progr` which is run, type:

```
which progr
```

Displaying: `echo $VARIABLE_NAME`

If we want to see which is the current value of a specific environment variable, we may use:

```
echo $VARIABLE_NAME
```

Example:

```
echo $PATH
```

Automatic Setting

If you want directory `/home/iia/d1` to be added to your `PATH` automatically each time you log in, please make sure the following lines are added to the file `/home/iia/.bash_profile`:

```
PATH=$PATH:/home/iia/d1
export PATH
```

(alternatively, you can use the file `.bashrc` in your home directory).

USB memory stick

Usually, when the stick is introduced into the computer, it gets automatically mounted on the file system. This means it temporarily becomes integrated into and visible as a part of the file system. From a terminal, you may access it in `/run/media/iia/THE_NAME_OF_YOUR_STICK/`. If a writting operation has been conducted over a file on the stick (like writting, modifying or deleting a file), then, before removing the stick from the computer, you must unmount (eject) it. A right click on the stick's icon will guide you.

Rights over files

Rights and types of users

In Linux, users of a file are divided into three categories: file owner, denoted by `u`; file owner group `g`; others `o`. The rights a user may have over a file are `r`, `w` and `x` meaning the right to read, write (i.e., modify) and execute (i.e., run) the file. A file that could be executed is either a binary file (for example, the result of compiling and linking a C source), or a text file, aka script, containing operating system commands (e.g., the PATH setting command above).

As an example: if you want to run a binary file or a script, you need to make sure its execution right is set. Otherwise you will get an error message (e.g., `Permission denied`), signaling that you have no permission to run that file.

Displaying rights: `ls -l`

Example:

```
ls -l f1
```

returns:

```
-rwxr-xr-- 1 iia students 32212 Oct 22 13:46 f1
```

The first column of the result contains 9 letters showing the rights over file `f1` of its owner (`rwx`) – so `iia` can read, write and execute the file, group members (`r-x`) – the members of the `students` group can read and execute the file, but can't modify it; the other users (`r--`) can only read it.

Changing rights: `chmod`

Example:

```
chmod ug+r file1
```

grants the owner (user `u`) and her group (`g`) the right ot read (`r`) the file `file1` (if the user who launches this command has the right to grant them).

Revoking the right of writting the file for everyone (user, group members and the others)is done by either of the following two commands:

```
chmod a-w file1
```

```
chmod -R a-w dir1
```

The latter recursively changes the rights over the files and directories in `dir1`, at every nesting level.

Changing file owner and group: `chown` and `chgrp`

Example: to make `iia` the new owner of `file1`, do:

```
chown iia file1
```

Example: to change group for `file1` into `students`, do:

```
chgrp students file1
```

Seek for a file: `locate` or `find`

Example: `locate passwd`

returns the full path to all files in the system whose name includes the string "passwd". It actually looks in a system database which stores data about files, so it might not produce up-to-date results, depending on the moment the database update has been done.

Example: `find /home -name "passwd" -print`

searches recursively in the file system, starting from `/home`, the files called `passwd`.

Search for a string inside a file: `grep`

Example:

```
grep abc file1
```

displays every line in `file1` containing the string `abc`. The string could actually be a regular expression (`grep` actually stands for "get regular expression pattern"), e.g., `f*.txt` which means "all strings starting with `f` and ending in `.txt`".

Editing files: `mcedit`, `gedit`, `kile`

```
mcedit file_name
```

(or `F4` on the file in Midnight Commander). `mcedit` heavily relies on Function keys. One can see their functionalities at the bottom of the panels (e.g., `F7` does a search of a word in the file).

Some other options are:

```
gedit file_name
xemacs file_name
```

For productively writing texts using LaTeX, a very good Integrated Environment is `kile`.

Redirecting commands: >, >> and <

The result of a command can be sent into a file instead of being displyed on the screen. This is done by `> dest`, if you want the original `dest` file to be erased, or `>> dest` if you need it to be appended.

Example:

```
ls -l > all.txt
```

creates a detailed list with all files in the current directory which will be written in the file `all.txt`.

Similarly, one can instruct the system to read from a file `f.in` instead of the keyboard (to redirect the input). This is done by using `< f.in` (e.g., `go < f.in` makes `go` read from `f.in`).

Pipes: |

More programs could be pipelined such that the output of one is used as the input for the next; the operator used is `|`.

Example:

```
ls -l | grep *.c
```

produces a list of all files in a directory (`ls -l`); this list will be the input for `grep *.c`, which searches for all files whose name end in `.c`.

Compiling and running programs

C programs: gcc and make

For `.c` programs, just do:

```
gcc
```

Example: let us assume we have in the file `hw.c` the following C code:

```
#include<stdio.h>

void main() {
    printf("Hello, world!\n");
}
```

you may compile it from command line like this:

```
gcc hello.c -o hello.exe
```

This will produce the output `hello.exe`, which can be run by typing `./hello.exe`

For bigger projects, the following command usually helps:

```
make
```

It assumes there is a file called `Makefile` or `makefile` in the current directory; this file contains the commands to be actually launched for compiling the sources.

Quite often, projects could be compiled and installed using the following procedure:

```
./configure
make
make install
```

where the last line should be run with root priviledges.

Java programs: javac, java and ant

A Java program could be compiled using:

```
javac
```

Example: if we have the following code in the file `HelloWorld.java`:

```
public class HelloWorld {

 public static void main(String[] args) {
        System.out.println("Hello, World!");
```

```
 }

}
```

we can compile it with:

```
javac HelloWorld.java
```

which will produce the file `HelloWorld.class`.

Running the program needs launching the Java Virtual Machine, which is done by:

```
java
```

In order to run the example above, we can do:

```
java HelloWorld
```

If all we have is a `jar` file, e.g., `hw.jar`, we can type:

```
java -jar hw.jar
```

A program which does for a Java project pretty much the same job as `make` for C projects is `ant`.

Processes on the system

Listing processes and their status: `ps`

Example:

```
ps -ef
```

could return something like:

```
...
iia       5712  2692  0 22:54 ?        00:00:00 kdeinit4: kio_file [kdein e
root      5713     2  0 22:55 ?        00:00:00 [kworker/2:2]
iia       5752  2481  0 22:57 pts/0    00:00:00 ps -ef
...
```

showing each program running on your system, together with its "process identifier", or pid, which is a unique number attached to each process, as seen in column 2.

Forcibly stopping a process: `kill`

```
kill -9 pid
```

where `pid` is the unique number of the process you want to stop, as returned by `ps`.

Display a .pdf file from a terminal: `okular` or `evince`

Portable Document Format (.pdf):

```
okular file_name.pdf &
```

or

```
evince file_name.pdf&
```

Browsers: `firefox` or `google-chrome`

Mozilla Firefox, Google Chrome etc. could be launched via the GUI or from a command line, e.g., by typing `google-chrome&` in a terminal.

MS Office-like packages: `LibreOffice` or `Apache OpenOffice`

LibreOffice and Apache OpenOffice are two alternatives for MS Office. Corresponding to Word, Excel and PowerPoint, you have Writer, Calc and Impress.

Appendix B

LaTeX– let's roll!

We will present how to write a `.tex` file and how to generate a `.pdf` from it. This includes writting text, equations, tables, algorithms and handling references. You need little more than one hour to understand the basics of LaTeX and one life to master it :)

Introduction

You should look in the `.tex` file (which is just a text file containing your text plus some formatting macros) as well as in the `.pdf` file generated from it. You will see plain content and some macros. A macro always starts with a backslash. For example, the macro `\section{...}` marks the beginning of a document section (of course, you should replace the dots with the section's actual name).

The comments are introduced by the % sign and will help you understand the meaning of the macros; the comment continues up to the end of line.

The commands needed for generating the `.pdf` are written in the file called `Makefile`; make sure the variables in its first lines are properly set. Then, all you need to do is to type `make` at the shell prompt. Alternatively, you can use LaTeX editors (see below) which provide the compiling commands.

Some basics

First, Romanian letters: ă î ș ț Ă Î Ș Ț â Â ü (even if the latter one does not seem to be in Romanian). In order to write them properly (use a comma instead of a cedilla for ș), you need to add `\usepackage{combelow}` to your document's preamble. Or you can add `\usepackage[utf8x]{inputenc}` to your document's preamble, do a `setxkbmap ro` in a terminal to change the keyboard layout to Romanian, then you can enter Romanian diacritics using `AltGr` (e.g., `AltGr+t`).

We can use bullets:

- number one
- number two

Or we can use numbered items:

1. un (aka 1)
2. dos (aka 2)

To start a new paragraph, you just enter a blank line in the `.tex` file. The first paragraph will have no indentation. This is normal; do not attempt to alter this.

Some characters, like \$, &, %, #, _, {, }, have special meanings in LaTeX; if you want to escape them, place a backslash in front of them. In order to write a backslash in your `.pdf`, put `\backslash` in your `.tex` file. To write < or >, you should enclose them between \$ signs (`$<$` and `$>$` respectively).

When we cite a paper (e.g., paper [26]), we take its bibtex reference from the web, store it in the `.bib` file and label it there (e.g., `gigel`; this label must be unique in the `.bib` file). Then use `\cite{gigel}` in the `.tex` file and let LaTeX handle ordering and cross-referencing. Each time you use an idea, text etc. borrowed from some other author, you must properly cite the source. It is a good time now to take a look in the `.bib` file and see how a bibtex entry of an article looks like. Then, you may want to search the web for the bibtex of an article (e.g., search bibtex "The Anatomy of a Large-Scale Hypertextual Web Search Engine"), grab it and add it to the `.bib` file.

The rest of this paper is structured as follows. Section B describes the implementation details, etc.

Implementation details

If we have algorithms used/modified/implemented/introduced, we can describe them. For instance, algorithm 1. Describing algorithms requires studying the documentation of the `algorithmic` and `algorithm` packages.

Algorithm 1: An algorithm looks like this

if $Committed(G_1, GR, \alpha)$ **then**
  $BRT_\alpha = PredictBRT(G_1, GR, \alpha, C_{GR})$
  $C_\beta = ContextUpdate(C_\beta, o)$
end if
if $utility \geq CommunicationCost(G_2)$ **then**
  $Int.To(G_1, Communicate(G_1, G_2, o))$
end if

We can write equations:

$$\frac{S : a_m \wedge r_b \wedge d}{B : d_c \wedge ((d_e \wedge \neg d_{t_2} \wedge d_{t_{15}}) \vee (\neg d_e \wedge d_{t_2})} \tag{B.1}$$

And big curly brackets:

$$\begin{cases} b = \frac{r}{r+s+2} \\ d = \frac{s}{r+s+2} \\ u = \frac{2}{r+s+2} \end{cases}$$

The notations for "n choose k" and sum look like C_n^k and $\sum_{k=0}^{n} C_n^k$ respectively.

Results

Tables

We summarize the results in a table, e.g., Table B.1. Typically, the table is placed on top of the page where it is reffered for the first time, or on one of the consecutive pages. It is the

Table B.1: Multinomial opinion multiplication

	belief			atomicity		
	poor	*avg*	*good*	*poor*	*avg*	*good*
success	0.1	0.2	0.3	0.4	0.5	0.6
failure	0.1	0.2	0.3	0.4	0.5	0.6

Figure B.1: Saying bye bye

LaTeX compiler decision.

Figures

We can also have figures, like Figure B.1. Some sources claim they tell 1000 words/figure.

Conclusions

Now we know how to write text, tables, references etc.

Acknowledgments

Special thanks to Knuth and Lamport for section B.

How to run the example

The `v1.tex` file contains the source; edit it with

```
kile v1.tex
```

To generate the `.pdf` file, type

```
make
```

And no, we don't attempt to launch the `Makefile`. Just type `make` at the shell prompt and enjoy.

Alternatively, you may press the QuickBuild button in the Kile editor.

Appendix C

Fast and furios tutorial on Python

Python supports multiple programming paradigms, including object-oriented, procedural and functional styles.

- Proper indentation is a must. Blocks are represented with indentation, there is no need to use curly braces.
- You don't need to define the type of variables.
- You don't need to add semicolon at the end of the statements.
- Python is case sensitive
- A minimum set of good practices are mentioned in C.1

There are two ways to run a python code:

- Interactively via an interpreter: `$ python`
- Execute a script - call from the command line: `$ python your_file.py`

Main operators and built-in data types which you are going to use during the lab are mentioned in tables C.2, C.3, C.4, AND C.5. For an exhaustive list, go to `https://www.tutorialspoint.com/python/index.htm` or `https://docs.python.org/2/tutorial/datastructures.html`

Running Python

1. Try basic operation in Python
   (a) Invoke the interpreter in a terminal

   ```
   $ python
   ```

   (b) Try basic operators

Table C.1: Good practice

Variable, functions, methods, packages and modules	$lower_case_with_underscores$
Classes and exceptions	$CapWords$
Protected methods and internal functions	$_single_leading_underscore(self, ...)$
Constants	$ALL_CAPS_WITH_UNDERSCORE$

Table C.2: Operators

Arithmetic	$+, -, *, /, \%. **$	$?2 **3$ 8
Comparison	$==, != , <, >, >=, <=$	
Assignment	$=, + =, - =, * = ...$	$?a = 2$ $?a, b = 2,$ "$Hello$" (multiple assignments)
Logical	*and or not*	

Table C.3: Data types

Numbers	int, flat and complex classes
Strings	marked with single quotes or double quotes
Lists []	on ordered sequence of items which do not need to be of the same type) $?a = [3,' an', [3, 5]]$ $?a[0]$ 3 $?a[-1]$ $[3, 5]$
Tuples ()	an ordered sequence of items same as list; tuples once created cannot be modified $?a = (3,' an',' 10')$
Set {}	on ordered collection of unique items $?a = \{1, 2, 3, 0\}$
Dictionary	an unordered collection of ke-value pairs $?d = \{1 :' one', 2 :' two'\}$ $d[1] =' one'$
Observation: slicing operator [ ] works for string, lists and tuples.	

Table C.4: Main operators on lists

len() Length	$?len([1, 2,' 3'])$ 3
Membership *in* , *not in*	$?a = [1, 2]$ $?1$ *in* a True
Concatenation: +	$?a, b = [1, 2], [3, 4, 5]$ $?b + a$ $[3, 4, 5, 1, 2]$
Negative indexing from back of the list:	$?a = [1, 2, 3]$ $?a[-1]$ 3
Slice operator for adjacent elements $list[start,\ stop]$	$?a = [1, 2, 3, 4]$ $?a[1 : 3]$ [2, 3]

Table C.5: Main methods on lists

Name	Description	Example
pop()	Remove an elements from list	$?a = [1,'a','three']$ $?a.pop()$ $'three'$ $?a$ $[1,'a']$
reverse()	Reverses items from list in place	$?a = ['start','a','three']$ $?a.reverse()$ $?a$ $['three','a','start']$
sort()	Sorts items from list in place	$?a = ['start','a','three']$ $?a.sort()$ $?a$ $['a','start','three']$
count()	Count of how many times obj occurs in list	$?[1,2,3,2].count(2)$ 2
index()	The lowest index in list that obj appears	$?[1,2,3,2].index(2)$ 1
extend()	Appends the contents of seq to list	$?a = [1,2,3]$ $a.extend([4])$ $?a$ $[1,2,3,4]$
append()	Appends object to list	$?a = [1,2,3]$ $?a.extend([4])$ $?a$ $[1,2,3,[4]]$

```
>>>1 + 3
4
>>> "IIA" == "iia"
False
>>> 'iia' + ' 3rd year'
'iia 3rd year'
>>> len('iia')
3
>>> s = 'iia'
>>> print s
>>> print "variable s has the value %s " % s
variable s has the value iia
>>> a,b = 2,5
>>> a+b
7
>>> (a,b) = (2, "iia")
>>> a
2
>>> b
'iia'
```

(c) Built-in data structures: test operators and methods listed in tableC.4

```
>>> roles = ['arya', 'jon', 'daenerys', 'bran']
>>> roles[1]
'jon'
>>> roles[-2]
'daenerys'
>>> roles[2:]
['daenerys', 'bran']
>>> new_roles = ['cersei', 'sansa', 'tyrion']
>>> roles + new_roles
['arya', 'jon', 'daenerys', 'bran', 'cersei', 'sansa', 'tyrion']
>>> roles
['arya', 'jon', 'daenerys', 'bran']
>>> roles.append(new_roles)
>>> roles
['arya', 'jon', 'daenerys', 'bran', ['cersei', 'sansa', 'tyrion']]
>>> roles.remove(new_roles)
>>> roles
['arya', 'jon', 'daenerys', 'bran']
>>> roles.extend(new_roles)
>>> roles
['arya', 'jon', 'daenerys', 'bran', 'cersei', 'sansa', 'tyrion']
>>> roles.pop()
'tyrion'
>>> roles
['arya', 'jon', 'daenerys', 'bran', 'cersei', 'sansa']
>>>

Dictionaries
```

```
>>> actors = {'tyron':"peter", "daenerys":"emilia", "jaime":"nikolaj"}
>>> actors["tyron"]
'peter'
>>> actors.keys()
['daenerys', 'tyron', 'jaime']
>>> actors.values()
['emilia', 'peter', 'nikolaj']
>>> actors.items()
[('daenerys', 'emilia'), ('tyron', 'peter'), ('jaime', 'nikolaj')]
>>> actors.items()[1]
('tyron', 'peter')

>>> dict = {"two:":2, "zero":0,"three":3}
>>> dict["four"] = 4
>>> dict
{'four': 4, 'zero': 0, 'two:': 2, 'three': 3}
>>> min(dict, key = dict.get)
'zero'
```

(d) Control structures

```
>>> a = [("a", 1), ("b", 2), ("c", 3)]
>>> for pair in a:
      print "The pair is ", pair
      (what, value) = pair
      print "Elements of the pair are: ", what, " ", value

The pair is  ('a', 1)
Elements of the pair are:  a   1
The pair is  ('b', 2)
Elements of the pair are:  b   2
The pair is  ('c', 3)
Elements of the pair are:  c   3

>>>if (1==1):
       print "True"
   else:
       print "False"
```

Practice Python and autograder Your activity from the first 4 laboratories can be auto-tested by using an autograder. It is possible to have more question for one exercise and for each question there are more test cases. The original projects are taken from `http://ai.berkeley.edu/project_overview.html`

Copy the files for tutorial autograder from `https://s3-us-west-2.amazonaws.com/cs188websitecontent/projects/release/tutorial/v1/001/tutorial.zip` into your folder. The folder tutorial contains more files from which you need to modifiy only the files `addition.py`, `buyLotsOfFruit.py` and `shopSmart.py`.

(a) Run the autograder and observe the results before any change to the files

```
1 $python autograder.py
```

(b) **Question 1** Change the function from `addition.py` such that it returns the addition of the two parameters. Run again the autograder and check that at Question 1 you get 1 point from 1.

(c) **Question 2** Change the function *buyLotsOfFruit* in `byLotsOfFruit.py` such that it returns the total cost of he order.

- Run

```
1 $python buyLotsOfFruit.py
```

- Run again the autograder and check that at Question 2 you get 1 out of 1 points.

(d) **Question 3** Class `shop.py` describes the main methods for accessing the cost per pound for a fruit and the price of the entire order (similar to teh previous question, but in an object oriented manner). There are more Fruit Shops and all items of an order must be bought from the same shop.

Change the function *shopSmart* from `shopSmart.py` such that it returns the FruitShop where the order costs the least amount. Don't change `shop.py`.

Run

```
1 $python shopSmart.py
```

Run again the autograder and check that at Question 3 you get 1 out of 1 points.

Bibliography

[1] Masataro Asai and Alex S Fukunaga. Tiebreaking strategies for A* search: How to explore the final frontier. In *AAAI*, pages 673–679, 2016.

[2] Roman Bartak. Constraint propagation and backtracking-based search. *Charles Universität, Prague*, 2005.

[3] Roberto J Bayardo Jr and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *AAAI/IAAI*, pages 203–208, 1997.

[4] Avrim L Blum and Merrick L Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1):281–300, 1997.

[5] Amanda Coles, Andrew Coles, Angel García Olaya, Sergio Jiménez, Carlos Linares López, Scott Sanner, and Sungwook Yoon. A survey of the seventh international planning competition. *AI Magazine*, 33(1):83–88, 2012.

[6] Alfonso Gerevini and Derek Long. Plan constraints and preferences in PDDL3. *The Language of the Fifth International Planning Competition. Tech. Rep. Technical Report, Department of Electronics for Automation, University of Brescia, Italy*, 75, 2005.

[7] Enrico Giunchiglia, Joohyung Lee, Vladimir Lifschitz, Norman C. McCain, and Hudson Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 153:49–104, 2004.

[8] Malte Helmert. A planning heuristic based on causal graph analysis. In *ICAPS*, volume 4, pages 161–170, 2004.

[9] Malte Helmert. The fast downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[10] Malte Helmert and Héctor Geffner. Unifying the causal graph and additive heuristics. In *ICAPS*, pages 140–147, 2008.

[11] Jörg Hoffmann. FF: The fast-forward planning system. *AI magazine*, 22(3):57, 2001.

[12] Jörg Hoffmann and Ronen Brafman. Contingent planning via heuristic forward search with implicit belief states. In *Proc. ICAPS*, volume 2005, 2005.

[13] Jörg Hoffmann and Ronen I Brafman. Conformant planning via heuristic forward search: A new approach. *Artificial Intelligence*, 170(6-7):507–541, 2006.

[14] Richard Howey, Derek Long, and Maria Fox. VAL: Automatic plan validation, continuous effects and mixed initiative planning using PDDL. In *Tools with Artificial Intelligence, 2004. ICTAI 2004. 16th IEEE International Conference on*, pages 294–301. IEEE, 2004.

[15] Byron Knoll, Jacek Kisynski, Giuseppe Carenini, Cristina Conati, Alan Mackworth, and David Poole. Aispace: Interactive tools for learning artificial intelligence. In *Proc. AAAI 2008 AI Education Workshop*, page 3, 2008.

[16] Nicholas Kushmerick, Steve Hanks, and Daniel S Weld. An algorithm for probabilistic planning. *Artificial Intelligence*, 76(1):239–286, 1995.

[17] Gabriele Röger Malte Helmert. *A Beginner's Introduction to Heuristic Search Planning*, 2015.

[18] W. McCune. Prover9 and Mace4. `http://www.cs.unm.edu/~mccune/prover9/`, 2005–2010.

[19] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[20] Erik T Mueller. Discrete event calculus reasoner documentation. *Software documentation, IBM Thomas J. Watson Research Center, PO Box*, 704, 2008.

[21] Erik T Mueller. *Commonsense reasoning: an event calculus based approach.* Morgan Kaufmann, 2014.

[22] Gheorghe Păun. *Matematica? Un spectacol!* Editura Ştinţifică şi Enciclopedică, 1988.

[23] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.

[24] Raymond M. Smullyan. *The lady or the tiger? and other logic puzzles, including a mathematical novel that features Coders great discovery.* Alfred Knopf, Inc., 1982.

[25] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure. From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[26] Chen Zhou, Liang-Tien Chia, and Bu-Sung Lee. DAML-QoS ontology for web services. In *ICWS '04: Proceeding's of the IEEE International Conference on Web Services*, page 472, Washington, DC, USA, 2004. IEEE Computer Society.