Anca MĂRGINEAN

NOTES FOR INTRODUCTION TO ARTIFICIAL INTELLIGENCE

U.T.PRESS
Cluj-Napoca, 2015
ISBN 978-606-737-077-5

Anca MĂRGINEAN

NOTES FOR INTRODUCTION TO ARTIFICIAL INTELLIGENCE

U.T. PRESS
Cluj-Napoca, 2015
ISBN 978-606-737-077-5

Editura U.T.PRESS
Str.Observatorului nr. 34
C.P.42, O.P. 2, 400775 Cluj-Napoca
Tel.:0264-401.999 / Fax: 0264 - 430.408
e-mail: utpress@biblio.utcluj.ro
www.utcluj.ro/editura

Director:	Prof.dr.ing. Daniela Manea
Consilier editorial:	Ing. Călin D. Câmpean

ISBN 978-606-737-077-5
Bun de tipar: 22.06.2015

Preface

This book contains laboratory works related to artificial intelligence domain and it is aimed as an introduction to AI through examples. Its main focus are the students from the third year of Computer Science Department from the Faculty of Automation and Computer Science, Technical University of Cluj-Napoca.

Each laboratory work, except the first one, includes a brief theoretical description of the algorithms, followed by a section of problems, either theoretical or practical. For the theoretical problems, one or more steps of the studied algorithms must be applied. Proper justifications are also required. For the practical ones, two environments are proposed: *Aima3e-Java* and *Pac-Man* projects.

AIMA3e-Java is a Java implementation of the algorithms from Norvig and Russell's Artificial Intelligence - A Modern Approach 3rd Edition. It is available online at `https://github.com/aima-java/aima-java`. Pac-Man projects are educational Python projects aimed for teaching foundational AI concepts, such as search, probabilistic inference, and reinforcement projects. They are available at `http://ai.berkeley.edu/project_overview.html`

Acknowledgements:
This book makes extensive use of educational projects: the sources of Aima3e-java project, and the sources and most of the practical problems for Pac-Man projects are not the property of this book's author. We thank to the authors of these projects.

Contents

List of Figures

Laboratory work 1

Working Environment

These Notes are meant to help you understand and practice the algorithms for *search problems* and *logical agents* studied in *Introduction in Artificial Intelligence* course.

The course support is *Artificial Intelligence - A Modern Approach 2rd Edition, Stuart Russell and Peter Norvig, 2002* [RN02].

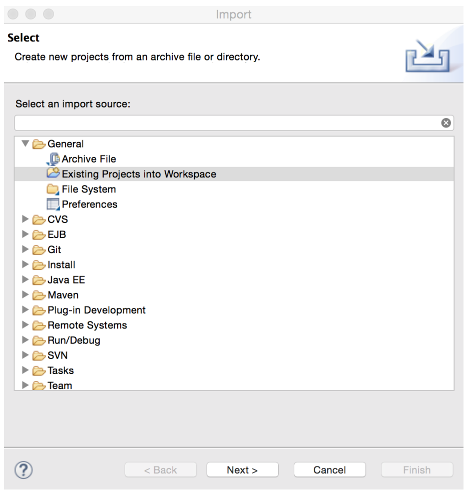
The code for all the used projects is available in $*HOME/LAB_WORKS*.

1.1 AIMA3e-Java project

AIMA3e-Java [aim] is a Java implementation of the algorithms from Norvig and Russell's Artificial Intelligence - A Modern Approach 3rd Edition, Prentice Hall, and it is available at `https://github.com/aima-java/aima-java`.

We recommend working with Eclipse IDE. The main steps for importing AIMA3e-Java projects in your workspace are:

1. Select the menu item *File* → *Import...*
2. In the Import Select dialog, select *General*→*Existing Projects into Workspace*


Import

Select

Create new projects from an archive file or directory.

Select an import source:

3. Press the *Next* button.

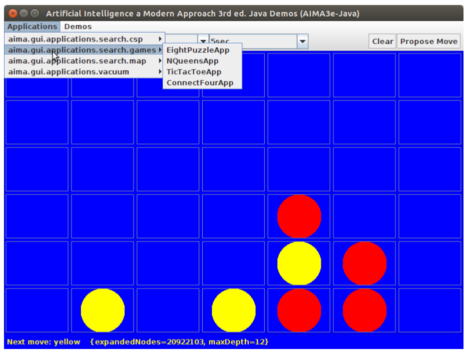
4. In the Import Projects dialog select the *Select archive file*: radio button and *Browse...* to the .zip release file. Select *aima-core, aima-gui, aimax-osm* in case they are not already selected.

5. Press *Finish*.

The *aima-core* project is the baseline project that contains all the current implementations for the algorithms. The *aima-gui* project depends on *aima-core* and contains example GUI and command line demonstrations of the algorithms defined in *aima-core*. The *aimax-osm* project is an extension project that demonstrates how the core algorithms can be used to create Navigation Applications using map data provided by the *Open Street Maps project* [aim].

For more detailed information about setting your workspace, please consult `https://github.com/aima-java/aima-java`

And now let's have fun with some games. The main GUI (figure 1.1) of the Aima3e project should easily be accessible, and you can play with the available applications. Did you observe the number of expanded nodes for the game *Connect four in a row* in figure 1.1?


Artificial Intelligence a Modern Approach 3rd ed. Java Demos (AIMA3e-Java)
Applications
Demos
aima.gui.applications.search.csp
aima.gui.applications.search.games
aima.gui.applications.search.map
aima.gui.applications.vacuum
EightPuzzleApp
NQueensApp
TicTacToeApp
ConnectFourApp
5sec
Clear
Propose Move

```
Next move: yellow    {expandedNodes=20922103, maxDepth=12}
```

Figure 1.1: Aima3e GUI

1.2 Pac-Man projects

Pac-Man projects are education projects from Berkley University, available at `http://ai.berkeley.edu/project_overview.html`. In the game, Pac-Man moves around in a maze and tries to eat as many food pellets as possible, while avoiding the ghosts. In figure 1.2, Pac-Man is the yellow circle, food pellets are the small white dots, and the ghosts are the other two agents with eyes in the above figure. If Pac-Man eats all the food in a maze, it wins. The big white dots at the top-left and bottom-right corner are capsules, which give Pac-Man power to eat ghosts in a limited time window [pac].

For testing search strategies, Pac-Man will be used without ghosts. Pac-Man with ghosts is interesting for adversarial search.

- Play a game of classic Pac-Man (see figure 1.2):

  python pacman.py

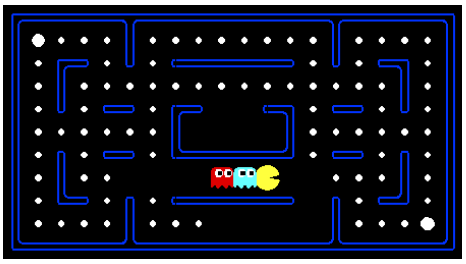


Figure 1.2: Pac-Man game

Add - -frameTime -1 to the command line to run in "demo mode" where the game pauses after every frame.

- Play multiple games in a row with -n. Turn off graphics with -q to run lots of games quickly.

For more details about the rest of the files, please go to `http://ai.berkeley.edu/search.html`.

1.3 Evaluation of your activity

You have to solve all the assigned homeworks. For extra points, choose one item from the following list:

- Propose a search problem and a game that clearly emphasizes the advantages and disadvantages of a search algorithm, adversarial or not.
- Propose a CSP with more than 100 variables and solve it with the studied algorithms. Analyze it with different algorithm and strategies and compare the results.
- Propose 5 mazes on which your final Pac-Man code will be tested. The overall results will be an average of the obtained points on 5 runs on all mazes gathered from all your colleagues.
- Propose "smart" ghosts.

Laboratory work 2

Simple agents

agent = architecture + program

A **rational agent** selects the actions that maximize the expected **utility**.

Simple reflex agent selects actions on the basis of the *current* percept, ignoring the rest of the recent history (figure 2.1).

```
function SIMPLE-REFLEX-AGENT(percept) returns an
   action
 static: rules, a set of condition-action rules

 state <- INTEREPRET-INPUT(percept)
 rule<- RULE-MATCH(state, rules)
 action<-RULE-ACTION[rule]
 return action
```

Simple-reflex Agent

```
function REFLEX-VACUUM-AGENT([location, status])
   returns an action

  if status=Dirty then return Suck
  else if location=A the return Right
  else if location=B then return Left
```

Reflex Vacuum Cleaner

Figure 2.1: Simple-Reflex Agent and Reflex-Vacuum Cleaner

Model-based reflex agent uses a **model** of (a) how the world evolves independently of the agent and (b) how the agent's own action affects the world (2.2).

internal state=keeping track of the part of the world it can't see now

```
function REFLEX-AGENT-WITH-STATE(percept) returns an
   action
 static: state, a description of the current world
    state
         rules, a set of condition-action rules
         action, the most recent action, initialy none

 state <- UPDATE-STATE(state, action, percept)
 rule<- RULE-MATCH(state, rules)
 action<-RULE-ACTION[rule]
 return action
```

Figure 2.2: Model-based reflex agent

Goal-based agent For this agent, goal-based action selection can involve: (a) single action that results in a state that satisfies the goal; (b) long sequence which can be obtained through *planning* or *search*

Utility-based agent uses utility function that maps a state onto a real number which describes the associated degree of happiness [RN02].

2.1 Problems

2.1.1 Theoretical problems

1. Explain why the *model-based reflex* agent is better than *simple reflex* agent for a traffic agent. Please analyze the following two situations: (1) the car in front is braking and (2) the driver intends to change lane.

2. Indicate some situations from traffic world where *utility-based* agent is better suited than *goal-based* agent.

2.1.2 Practical problems

Java

1. Analize the following classes and pay attention to the type of the implemented agent:

Agent	aima.core.agent.Agent.java
	package *aima.core.environment.vacuum*
Table-Driven-Vacuum-Agent	*TableDrivenVacuumAgent*
Table-Driven-Agent	*TableDrivenAgentProgram*
Reflex-Vacuum-Agent	*ReflexVacuumAgent*
Simple-Reflex-Agent	*SimpleReflexAgentProgram*
Model-Based-Reflex-Agent	*ModelBasedReflexAgentProgram*

2. Implement a *model-based reflex* agent for a vacuum cleaner in case there are **two types** of dirt, the robot has a sensor for detecting the type of the dirt from the current square, and there are two *suck* actions.

Python

1. Analize the function *LeftTurnAgent* from *pacmanAgents.py* and identify the type of the implemented agent.

   ```
   python pacman.py - -pacman LeftTurnAgent -k 0
   python pacman.py -l testMaze -p LeftTurnAgent -k 0
   ```

   - *-pacman* (or *-p*) running option allows you to mention the agent that describes the behavior of the player, *-k* 0 means 0 ghosts.

2. Create similar functions for the other types of agents. Example: a simple reflex agent that goes two steps ahead then turns left.

3. Run and explain the provided *ReflexAgent* in *submission.py*:

   ```
   python pacman.py -p ReflexAgent -k 0
   ```

   How well does the agent behave on simple layouts?

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

   Run and observe the agent behavior with two ghosts.

   ```
   python pacman.py -p ReflexAgent -k 2
   ```

   Observation: Default ghosts are random; if you want to use directional ghost, please add -g DirectionalGhost [pac].

Laboratory work 3

Search problems. Uninformed search

Problem-solving agents decide what to do by finding sequences of actions that lead to desirable states [RN02].

3.1 Boat crossing puzzle

A farmer wants to get his cabbage, goat, wolf across a river. He has a boat that only holds two. He cannot leave cabbage and goat alone or the goat and wolf alone. How many river crossings does he need? How can be identified the sequence of needed actions?

$$Farmer \quad Cabbage \quad Goat \quad Wolf$$

There are eight possible actions, which are denoted by a concise set of symbols. For example, the action FG$\triangleright$ means that the farmer will take the goat across to the right bank; F$\triangleleft$ means that the farmer is coming back to the left bank alone.

Search: *An agent with several immediate options of unknown value can decide what to do by first examining different possible sequences of actions that lead to states of known value, and then choosing the best sequence.*

The elements of the search problem for the *boat crossing puzzle* are:

- States, with s_{start} as starting state: the boat, the farmer, the cabbage, the goat and the wolf are on the left side. Which are the rest of the states?
- Actions(s): possible actions $F\triangleright$, $F\triangleleft$, $FC\triangleright$, $FC\triangleleft$, $FG\triangleright$, $FG\triangleleft$, $FW\triangleright$, $FW\triangleleft$
- Cost(s,a): action cost - Each action of type safe river crossing costs 1 unit of time. The actions that result in an eating event cost ∞
- Succ(s,a): successor function - the positions of the farmer, boat, cabbage, wolf and goat after doing a crossing
- IsGoal(s): found solutions? Are all on the right side?

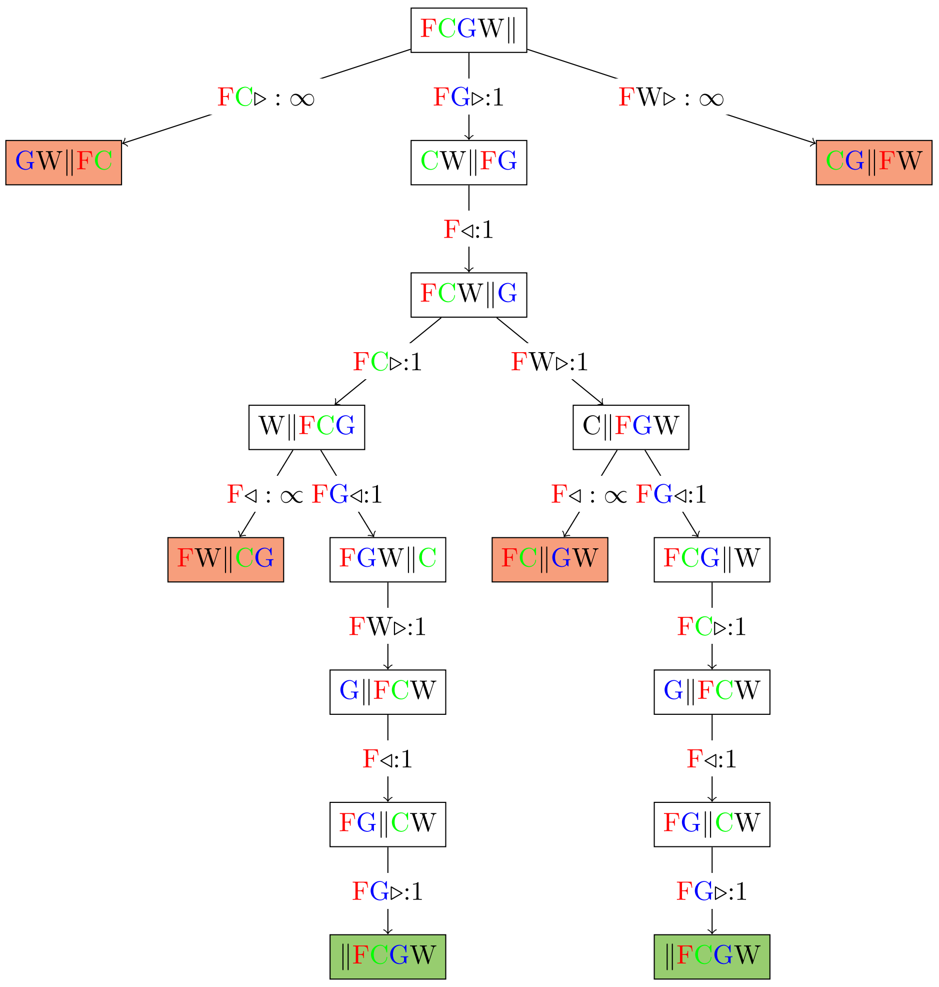


Figure 3.1: Search tree for boat crossing puzzle

3.2 Tree search

The Search tree is generated by the initial state and the successor function that together define the state space. For the boat crossing puzzle, the tree is presented in figure 3.1. The root of the tree is the start state s_{start}, and the leaves are the goal states (IsGoal(s) is true). Each edge leaving a node s corresponds to a possible action $a \in Actions(s)$ that could be performed in state s. The edge is labeled with the action and its cost, written $a \in Cost(s, a)$. The action leads deterministically to the successor state $Succ(s, a)$, represented by the child node.

Each root to leaf path represents a possible action sequence, and the sum of the costs of the edges is the cost of that path. The goal is to find the root to leaf path that has the minimum cost.

Note that in code, we usually **do not** build the search tree as a concrete data structure. The search tree is used merely to visualize the computation of the search algorithms and study the structure of the search problem.

```
function TREE-SEARCH(problem, fringe) returns a
   solution, or failure

 fringe<- INSERT(MAKE-NODE(Initial-State[problem]),
     fringe)
 loop do
   if fringe is empty then return failure
      node<- REMOVE-FRONT(fringe)
      if GOAL-TEST[problem] applied to STATE(node)
          succeeds
            return node
      fringe<- INSERTALL(EXPAND(node,problem), fringe)
 end
```

Tree search algorithm

The common element to all the search strategies is the fringe. The fringe is the collection of nodes that have been generated but not yet expanded. Each element of the fringe is a leaf node.

3.2.1 Uninformed search

Breadth-first search = TREE-Search with an empty fringe that is a FIFO queue. It is a simple strategy in which the root node is expanded first, then all the successors of the root node are expanded next, then their successors, and so on.

BreadthFirstSearch=TREE-Search(problem, FIFO-queue)

Uniform-cost search expands the node n with the lowest path cost (not the shallowest unexpanded node). The algorithm expands nodes in order of increasing path cost

UniformCostSearch=TREE-Search(problem, queue ordered ascending

by path cost)

Depth-first search

DepthFirstSearch=TREE-Search(problem, LIFO-queue)

```
public class DepthFirstSearch implements Search {
  .....
  public List<Action> search(Problem p) {
    return search.search(p, new LIFOQueue<Node>());
        }
  .....
}

public class BreadthFirstSearch implements Search {
 ....
 public List<Action> search(Problem p) {
        return search.search(p, new FIFOQueue<Node>());
        }
 ....
}
public class UniformCostSearch extends PrioritySearch {
 public UniformCostSearch(QueueSearch search) {
        super(search, createPathCostComparator());
}
 private static Comparator<Node> createPathCostComparator() {
  return new Comparator<Node>() {
    public int compare(Node node1, Node node2) {
        return (new Double(node1.getPathCost()).compareTo(new
            Double(node2
          .getPathCost())));
    }    };
  }
}
```

Listing 3.1: Snippet from aima3e-java

Observation You can see from the above snippet that the classes *DepthFirstSearch* and *BreadthFirstSearch* from Aima3e-Java are quite similar. The most important difference is the type of the used queue. In case of *UniformCostSearch*, the search uses a priority queue which involves a comparator on path costs.

Depth-limited search is depth-first search with a predetermined depth limit l. That is, nodes at depth l are treated as if they have no successors.

Iterative deepening depth-first search = Depth-Limited Search with depth increasing from 0 to ∞

Example. Test different search strategies for the problem of Travelling from Arad to Bucharest with the application RouteFindingAgent from Aima3e.

3.3 Problems

3.3.1 Theoretical problems

1. Consider Pac-Man game (figure 3.2). In case the goal is to reach a position (x, y), mention which are the states, actions, successor function and goal test? What about if the goal is to eat all the dots?

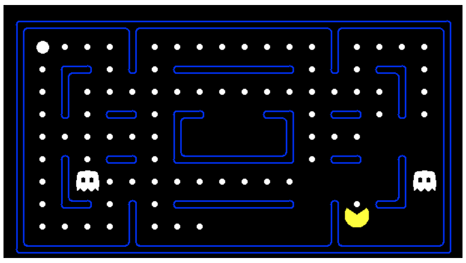
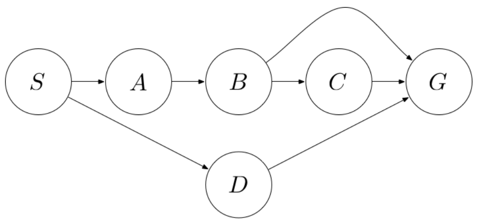


Figure 3.2: The world state includes every detail of the environment

Hint: In the first case, states are (x, y) location. In the second case, the goal test is *all dots are false*, where a dot is considered true if the food dot was not eaten.

2. Which is the solution identified with breadth-first search for state space graph G_1? In case of equality, the nodes are chosen alphabetically: $S \rightarrow X \rightarrow A$ is expanded before $S \rightarrow X \rightarrow B$.



$$\begin{tikzcd} S \arrow[r] \arrow[drr] & A \arrow[r] & B \arrow[r] \arrow[rr, bend left] & C \arrow[r] & G \\ & & D \arrow[urr] & & \end{tikzcd}$$

Figure 3.3: State-space search in graph G_1

- $S \rightarrow A \rightarrow B \rightarrow G$
- $S \rightarrow A \rightarrow B \rightarrow C \rightarrow G$
- $S \rightarrow D \rightarrow G$

3. Which is the solution for depth-first search for the state space graph G_1 in figure 3.3? In case both A and B are good choices for expanding X, $S \rightarrow X \rightarrow A$ is expanded before $S \rightarrow X \rightarrow B$.

- $S \rightarrow A \rightarrow B \rightarrow G$
- $S \rightarrow A \rightarrow B \rightarrow C \rightarrow G$
- $S \rightarrow D \rightarrow G$

4. Which is the solution for uniform cost search for the state space graph in figure 3.4? ($S \rightarrow X \rightarrow A$ is expanded before $S \rightarrow X \rightarrow B$).

- $S \rightarrow A \rightarrow B \rightarrow G$
- $S \rightarrow A \rightarrow B \rightarrow C \rightarrow G$
- $S \rightarrow D \rightarrow G$

5. Which is the next expanded node during the building of the search tree from figure 3.5? Consider all studied search algorithms.

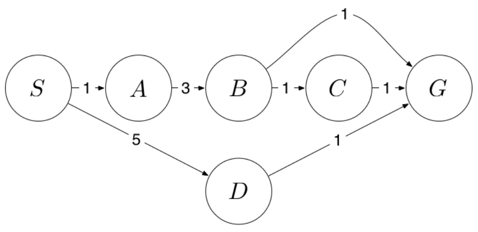


Figure 3.4: State-space search in graph G_2

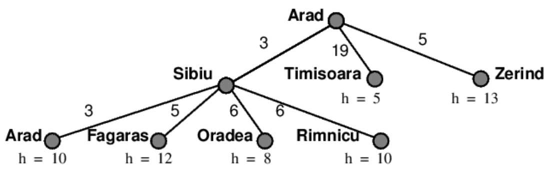


Figure 3.5: Partially expanded search tree

6. Which are the solution of the search with depth and breadth-first search strategies for the state space graph G_3 in figure 3.6?

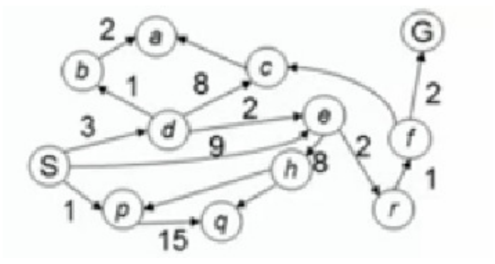


Figure 3.6: State-space search in graph G_3

3.3.2 Practical problems

Java

1. Run *Route Finding Agent* with different strategies. Go to *aima-gui/src/main/java/aima/gui/applications/*.
2. Study the following classes in order to be able to implement your own search problems.

Simple-Problem-Solving-Agent	aima.core.search.framework. SimpleProblemSolvingAgent
Romania	aima.core.environment.map. SimplifiedRoadMapOfPartOfRomania
Tree-Search	aima.core.search.framework.TreeSearch
Graph-Search	aima.core.search.framework.GraphSearch
Node	aima.core.search.framework.Node
Queue	aima.core.search.framework.Queue
Breadth-First-Search	aima.core.search.uninformed.BreadthFirstSearch
Uniform-Cost-Search	aima.core.search.uninformed.UniformCostSearch
Depth-first Search	aima.core.search.uninformed.DepthFirstSearch
Depth-Limited-Search	aima.core.search.uninformed.DepthLimitedSearch
Iterative-Deepening -Search	aima.core.search.uninformed. IterativeDeepeningSearch
Bidirectional search	aima.core.search.uninformed.BidirectionalSearch

3. Change the map of Romania, by adding at least Cluj-Napoca, Turda, Gherla, TgMures, and Alba-Iulia and test different search strategies on the new search problem.

4. Implement the search problem for the graph in figure 3.4. Solve it with different strategies.

5. Implement the search problem for solving the Boat crossing puzzle.

Python

1. In the folder search from pacman, there is a version without ghosts. The Pacman agent has to reach a particular location and to collect food efficiently [pac].

The files that are to be modified:	
search.py searchAgents.py	The search algorithm will be here. The search-based agents are implemented here.
Useful files:	
pacman.py game.py util.py	The Pacman GameState type is defined here. It is the main file for running the game AgentState, Agent, Direction, Grid types are defined here. It describes the logic behind how the Pacman world works. Data structure that can be used for implementing search algorithms.

Test GoWestAgent on two different scenario:

```
python pacman.py - -layout testMaze - -pacman GoWestAgent
```
a trivial reflex agent on a linear maze
```
python pacman.py - -layout tinyMaze - -pacman GoWestAgent
```
the same agent on a maze that requires turning
```
python pacman.py -h
```

2. Find a fixed food dot using depth first search. In *searchAgents.py*, there is a fully implemented SearchAgent. It plans out a path through Pacman's world and then executes that path step-by-step. Before implementing your own tree search and depth first search, please check *tinyMazeSearch*. This is a search algorithm implemented in *search.py*.

python pacman.py -l -p SearchAgent -a fn=tinyMazeSearch

- On the existing code, if you run python pacman.py -l tinyMaze -p SearchAgent you wil get *Method not implemented: depthFirstSearch at line 90 of* $*pacmanHome*$*/berkley_pacman/search/search.py*
- Test your resulting code for three mazes:
  python pacman.py -l tinyMaze -p SearchAgent
  python pacman.py -l mediumMaze -p SearchAgent
  python pacman.py -l bigMaze -z .5 -p SearchAgent
- Pacman board will show an overlay of the explored states, and the order in which they were explored. Brighter red means earlier exploration. Analyze the exploration order and whether Pacman actually goes to all explored nodes.

3. Implement breadth first search for the same problem python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs.

   Test it on *eight puzzle problem*: python eightpuzzle.py

4. Implement Uniform Cost graph search in *search.py*. In *searchAgents.py* there are described three agents with different cost functions: SearchAgent, StayEastSearchAgent and StayWestSearchAgent. Test them and your uniform cost graph search on the medium and mediumScaryMaze.

   python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
   python pacman.py -l mediumDottedMaze -p StayEastSearchAgent

Laboratory work 4

Informed search

Problem definition is extended with problem specific knowledge as desirability degree of a node [RN02].

4.1 Heuristic based search

Heuristic $h(n)$ gives an estimation of the distance from the node n to the goal.

GreedyBestFirstSearch is *Tree-Search/Graph-Search*(Problem, priority queue according to $f(n)$, where $f(n) = h(n)$) - Forward cost

Reminder: Uniform Cost Search uses a priority queue according to *g(n)* - Backward cost.

A* is *Tree-Search/Graph-Search*(Problem, priority queue according to $f(n)$, where $f(n) = h(n) + g(n)$)

Heuristics must be admissible. In order to identify them, the problem can be relaxed, as in can be observed in figure 4.1.

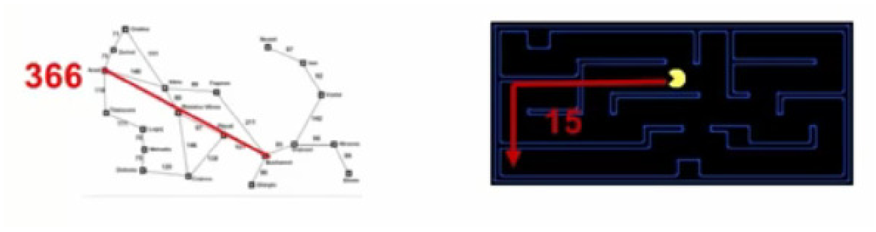


Figure 4.1: Relaxed problems for heuristic identification

4.2 Local search

In case the path is not important, but the final goal state is, Local search is appropriate. It uses heuristics in order to choose the next configuration.

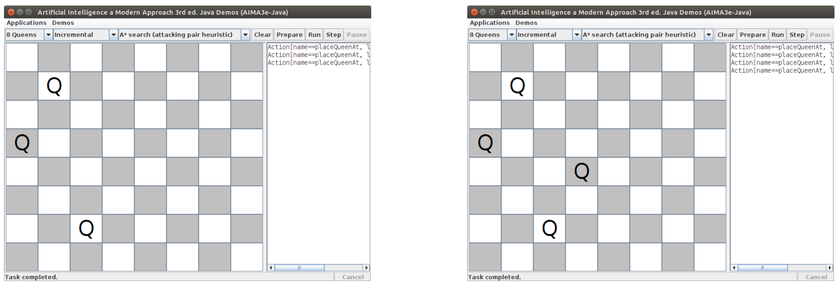

Artificial Intelligence a Modern Approach 3rd ed. Java Demos (AIMA3e-Java)
Applications Demos
8 Queens
Incremental
A* search (attacking pair heuristic)
Clear Prepare Run Step Pause
Action[name==placeQueenAt,
Q
Task completed.
Cancel

Figure 4.2: Two succesive states with A* search

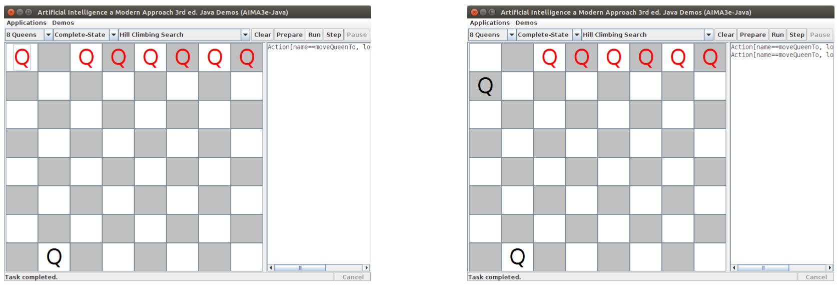

Artificial Intelligence a Modern Approach 3rd ed. Java Demos (AIMA3e-Java)
Applications Demos
8 Queens
Complete-State
Hill Climbing Search
Clear Prepare Run Step Pause
Action[name==moveQueenTo,
Q
Task completed.
Cancel

Figure 4.3: Two succesive states with Hill Climbing (local search)

The state space is a set of complete configurations. In figures 4.2 and 4.3 you can observe the differences between incremental state (as used in previous chapter) and complete state ([RN02]).

Hill-climbing

```
function HILL-CLIMBING(problem) returns {a state that
    is a local maximum}
   inputs: problem
   local variables:current  - a node
                   neighbor - a node
 current<-MAKE-NODE(INITIAL-STATE[problem])
 loop do
    neighbor <- a highest-valued successor of current
    if VALUE[neighbor] ≤ VALUE[current] then return
       STATE[current]
    current<-neighbor
 end
```

Note: For n-queens problem, in local search:

- each state = 8 queens on the table (different to tree-search, where the number of queens increases from 0 to 8) - *complete-state* formulation
- succesor function = all the states which can be obtained by changing the position of only one queen. If we consider to be known that on each column we need for sure a queen, but the row is variable, then there are 8 x 7 = 56 succesors

- heuristic function h= number of pair of queens that are attacking each other.

Simulated-annealing

```
function SIMULATED-ANNEALING(problem, schedule)
   returns a solution state
 inputs: problem - a problem
         schedule - a mapping from time to "
            temperature"
 local variables: current - a node
                  next - a node
                  T - a "temperature" controlling prob
                     . of downward steps
 current←MAKE-NODE(INITIAL-STATE[problem])
 for t←1 to ∞ do
   T←schedule[t]
   if T = 0 then return current
   next←a randomly selected successor of current
   $\Delta E$ ← VALUE[next] - VALUE[current]
   if $\Delta E$>0 then current←next
   else  current← next only with probability $e^{\Delta}E/T$
 end
```

We mentione here some examples of the use of informed search in planning, and specially hill-climbing:

- Planning as heuristic search [BG01] (HSP): the heuristics are automatically extracted from Strips description of the planning domains, where Strips is a formal language which allows description of actions in terms of preconditions and effects. An example of a STRIPS description of the Fly action is:

  $Action(Fly_STRIPS(p, from, to),$

  PRECOND : $At(p, from) \wedge Plane(p) \wedge Airport(from) \wedge Airport(to)$

  EFFECT : $\neg At(p, from) \wedge At(p, tp)$

- Fast Forward [Hof01] (FF): similar to HSP, in FF the planning problems are treated as search problem in state space, with a heuristic function. This heuristic function is also extracted automatically from the domain description, after relaxing the planning problem. The relaxed planning problem is obtained by ignoring parts of the actions' specification.

4.3 Problems

4.3.1 Theoretical problems

1. Pacman: The three figures from 4.4 are the results of three searches applied on the problem of getting to a certain food dot. Identify which search is done in each figure? (Brigter red means early exploration).

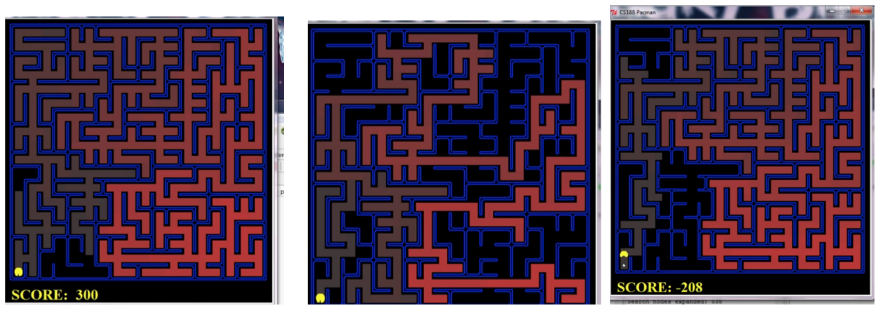


Figure 4.4: Which search is used?

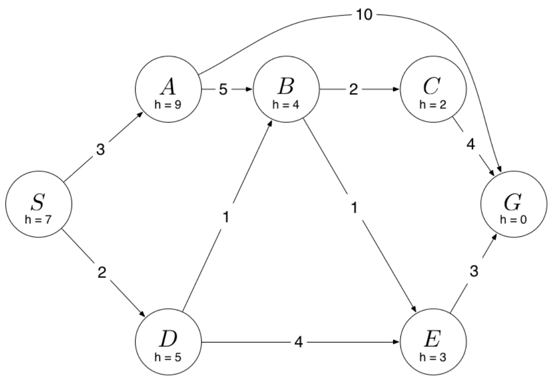
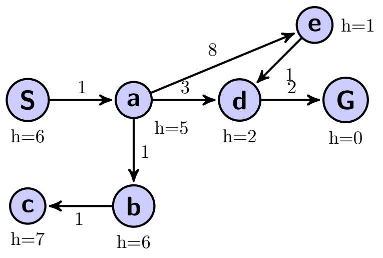


Figure 4.5: The graph G_4 with costs and heuristic values

2.
   - Which is the path found with Greedy Best First search for the graph in figure 4.5?
     - $S \rightarrow A \rightarrow G$
     - $S \rightarrow A \rightarrow B \rightarrow C \rightarrow G$
     - $S \rightarrow D \rightarrow B \rightarrow C \rightarrow G$
     - $S \rightarrow D \rightarrow B \rightarrow E \rightarrow G$
     - $S \rightarrow D \rightarrow E \rightarrow G$
   - Is the heuristic admisible? Which is the path identified with A* search?

3. Which are the paths identified with $GreedyBestFirst$, $UniformCost$ and A* for the following state space graph ?



4. Consider the Pac-Man game, with the goal of eating all the food dots. Consider that each action has a cost of 1. Choose the heuristics that are admisible:
   - total number of remain food dots
   - the distance to the closest dot
   - the distance to the farthest food dot
   - distance between the closest dot plus distance to the farthest dot

4.3.2 Practical problems

Java

1. Analyse the implementations for the studied search strategies:

	package *aima.core.search.informed*
Best-First search	*BestFirstSearch.java*
Greedy best-First search	*GreedyBestFirstSearch.java*
A* Search	*AStarSearch.java*
Hill-Climbing	*HillClimbingSearch.java*
Simulated-Annealing	*SimulatedAnnealingSearch.java*

2. Eight puzzle Game.
   - Run *aima.gui.demo.search.EightPuzzleDemo* and analyze the paths identified with *Greedy Best First Search*, *A** and *Simulated Anneasling Search*.

     The starting and goal configurations are:

1	2	5		1	4	2
3	4			7	5	8
6	7	8		3		6

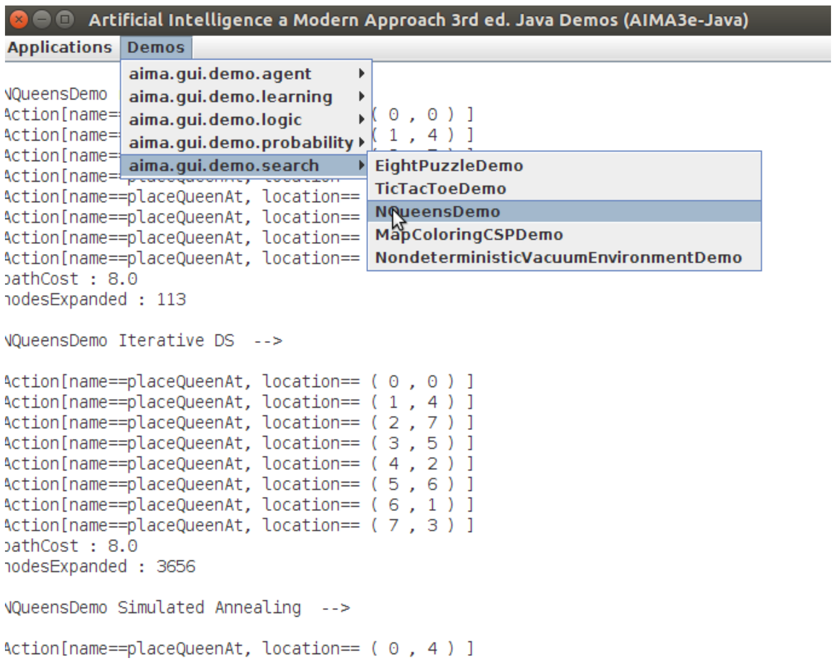
     Notice the difference between the solutions in case *MisplacedTileHeuristic* and *ManhattanHeuristic* are used.
   - Go to the methods: *eightPuzzleGreedyBestFirstDemo(), eightPuzzleGreedyBestFirstManhattanDemo(), eightPuzzleAStarDemo(), eightPuzzleAStarManhattanDemo(), eightPuzzleSimulatedAnnealingDemo()* from EightPuzzleDemo class and change the goal configuration to

	8	7
6	5	4
3	2	1

     Run the search on the new problem and compare the results with the runs from the previous point.

3. Run *aima.gui.demo.search.NQueensDemo* (figure 4.6) and compare the solutions

   Optional: change from n queens to n rooks and n knives.



```
NQueensDemo
Action[name=                                      ( 0 , 0 ) ]
Action[name=                                      ( 1 , 4 ) ]
Action[name=
Action[name=
Action[name==placeQueenAt, location==
Action[name==placeQueenAt, location==
Action[name==placeQueenAt, location==
Action[name==placeQueenAt, location==
pathCost : 8.0
nodesExpanded : 113

NQueensDemo Iterative DS  -->

Action[name==placeQueenAt, location== ( 0 , 0 ) ]
Action[name==placeQueenAt, location== ( 1 , 4 ) ]
Action[name==placeQueenAt, location== ( 2 , 7 ) ]
Action[name==placeQueenAt, location== ( 3 , 5 ) ]
Action[name==placeQueenAt, location== ( 4 , 2 ) ]
Action[name==placeQueenAt, location== ( 5 , 6 ) ]
Action[name==placeQueenAt, location== ( 6 , 1 ) ]
Action[name==placeQueenAt, location== ( 7 , 3 ) ]
pathCost : 8.0
nodesExpanded : 3656

NQueensDemo Simulated Annealing  -->

Action[name==placeQueenAt, location== ( 0 , 4 ) ]
```

Figure 4.6: Aima3e-java: running n queens

4. Analyse on Route Finding Agent application (aima.gui.applications.search.map.RouteFindingAgentApp) different informed and uninformed search. Make a comparative analysis on the number of expanded nodes and the path cost of the solution.

   Observation: from Arad to Bucharest through Sibiu, the travel distance is 450, while through RimnicuValcea is 418 (see figure 4.7).

5. Games: Nqueens, EightPuzzleApp. Run *aima.gui.applications.search.games.NQueensApp* and *EightPuzzleApp* and compare the results of all the studied search strategies.

   Note: On medium configuration of EightPuzzleApp, observe the difference between the number of *expandedNodes* and *maxQueueSize*:

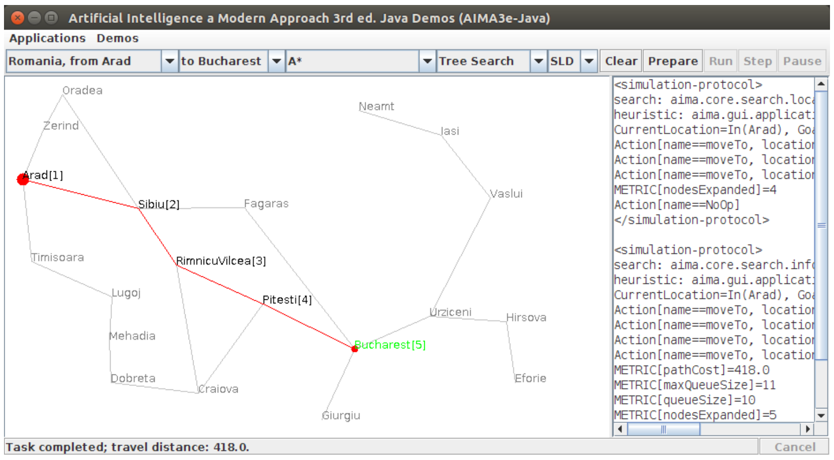
   - in case of A* with MispacedTileHeuristic: 25, respectively 21,
   - in case of Breadth FirstSearch(GraphSearch): 288, respectively 199.

Python

1. Implement A^* search. Test it on the problem of Pacman getting to a certain food dot. Use *Manhattan distance heuristic* which is implemented as *manhattanHeuristic* in searchAgents.py [pac].

   In the end, you should be able to run

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



```
<simulation-protocol>
search: aima.core.search.loca
heuristic: aima.gui.applicati
CurrentLocation=In(Arad), Goa
Action[name==moveTo, locatio
Action[name==moveTo, locatio
Action[name==moveTo, locatio
METRIC[nodesExpanded]=4
Action[name==NoOp]
</simulation-protocol>

<simulation-protocol>
search: aima.core.search.info
heuristic: aima.gui.applicati
CurrentLocation=In(Arad), Goa
Action[name==moveTo, locatio
Action[name==moveTo, locatio
Action[name==moveTo, locatio
Action[name==moveTo, locatio
METRIC[pathCost]=418.0
METRIC[maxQueueSize]=11
METRIC[queueSize]=10
METRIC[nodesExpanded]=5
```

```
Task completed; travel distance: 418.0.
```

Figure 4.7: Aima3e-java: running route finding with A*

- Compare the number of explored states and the solutions between Depth-first Search, Breadth-first Search and A*.
- Identify mazes where one method behaves much better than the others and explain why.

2. Consider the problem of eating all dots (in the absence of ghosts and power food). Implement one or more heuristics and test it on different scenarios.

   Observation: A* with a null heuristic is equivalent to uniform-cost search. By running python pacman.py -l testSearch -p AStarFoodSearchAgent, before implementing an heuristic, the result should be an optimal solution with a total cost of 7.

   AStarFoodSearchAgent is equivalent to using *-p SearchAgent -a fn=astar, prob=FoodSearchProblem,heuristic=foodHeuristic*. *FoodSearchProblem* is implemented in searchAgents.py.

   Therefore, you have to fill in *foodHeuristic* in *searchAgents.py*.

   Please consider for testing the maze in figure 4.8.

   python pacman.py -l trickySearch -p AStarFoodSearchAgent

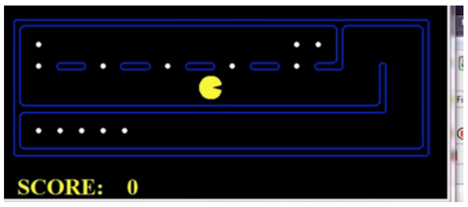


Figure 4.8: An interesting position of food dots

3. In the original Pacman problem, it is possible to have food that gived power to escape from ghosts for a limited time. Consider that the special

food dost are placed in the corners and the goal is to eat all four special food placed in corners (the ghosts are still not present).

You have to formulate a new problem *CornersProblem* in *searchAgent.py*. Choose a state representation which encodes information proper for the new goal. Maybe you also need a new heuristic.

In the end, you should be able to test it with

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

Laboratory work 5

Adversarial search

Adversarial search problems (games) are specific for competitive environments, in which the agents' goals are in conflict.

Zero-sum games of perfect information = deterministic, fully observable environment in which there are two agents whose actions must alternate and in which the utility values at the end of the game are always equal or opposite.

Game tree = tree formed from the initial state and the legal moves for each side.

Utility function = objective function, or payoff function, gives a numeric value for the terminal states.

example: in chess, win is +1, loss is -1, draw is 0.

Solution of an adversarial search problem is an **optimal strategy**. This leads to outcomes at least as good as any other strategy when one is playing an infallible opponent [RN02].

5.1 MiniMax search

MINIMAX-VALUE(n)=

$$= \begin{cases} UTILITY(n) & \text{if n is terminal state} \\ max_{s \in Succesor(n)} MINIMAX-VALUE(s) & \text{if n is nod MAX} \\ min_{s \in Succesor(n)} MINIMAX-VALUE(s) & \text{if n is nod MIN} \end{cases}$$

```
function MINIMAX-DECISION(state) returns an action
inputs: state, current state in game

v← MAX-VALUE(state)
return the action in SUCCESORS(state) with value v
---------------------------------------------------
function MAX-VALUE(state) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
$v \leftarrow -\infty$
for s in SUCCESORS(state) do
  v← MAX(v, MIN-VALUE(v))
```

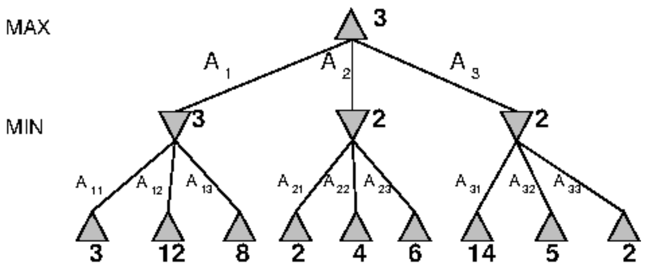


Figure 5.1: An example of game tree

```
return v
--------------------------------------------------
function MIN-VALUE(state) returns a utility value
if TERMINAL-TEST(state) then return UTILITY(state)
$v \leftarrow \infty$
for s in SUCCESORS(state) do
  v← MIN(v, MAX-VALUE(v))
return v
```

MINIMAX-DECISION

Figure 5.1 gives an example of MinimaxSearch game tree.

5.2 Alpha-beta pruning

Optimizes Minimax search tree, by pruning nodes that do not affect the values of the parent nodes.

Figure 5.2 shows how the pruning is done on Minimax search.

```
function ALPHA-BETA-DECISION(state) returns an action
return the a in ACTIONS(state) maximizing MIN-VALUE(
   RESULT(a, state), $-\infty, \infty$)
--------------------------------------------------
function MAX-VALUE(state, $\alpha$, $\beta$) returns an utility
   value
 inputs: state, current state in game
    $\alpha$ the value of the best alternative for MAX along
       the path to state
    $\beta$ the value of the best alternative for MIN along
       the path to state

  if TERMINAL-TEST(state) then return UTILITY(state)
  v← $-\infty$
  for a, s in SUCCESORS(state) do
        v← MAX(v, MIN-VALUE(s, $\alpha$,$\beta$))
        if v $\geq$ $\beta$ then return v
        $\alpha \leftarrow$ MAX($\alpha$, v)
  return v
```

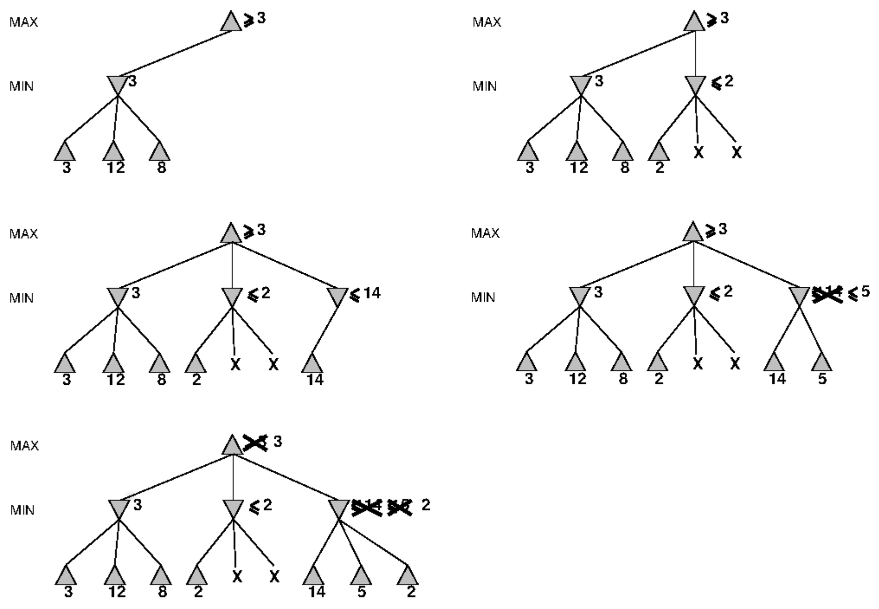


Figure 5.2: Steps of Alpha-Beta pruning

```
-----------------------------------------------------
function MIN-VALUE(state,$\alpha$,$\beta$ returns a utility value
    same as MAX-VALUE but with roles of $\alpha$, $\beta$ reversed
}
```

5.3 Cut-off test

- For limited resources, the search can be done with the use of CUTOFF-TEST instead of TERMINAL-TEST, and
- replace UTILITY function with EVAL. EVAL estimates the desirability of a state and it must:
  - order the terminal states in the same way with utility function,
  - its computation time must be reduced,
  - for non-terminal state, EVAL must be correlated with the chances of winning from this state.

  EVAL could be a weighted linear function of some characteristics: $Eval(s) = w_1 f_1(s) + w_2 f_2(s) + ... + w_n f_n(s)$

Example For chess, for the two non-terminal states from figure 5.3, how good is an EVAL function as a weighted combination of available pieces, where the associated values are: pawn=1, knight and knave=3, rook=5, queen=9?

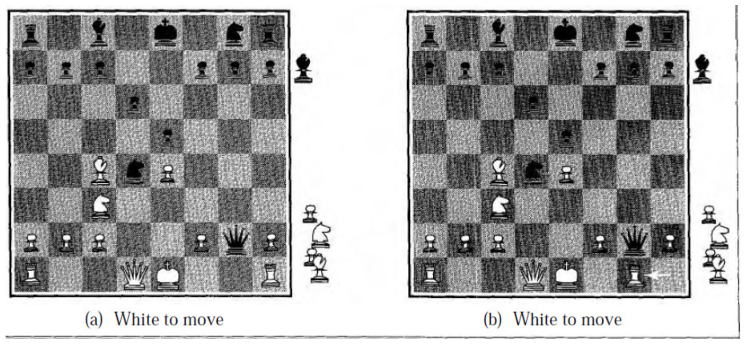
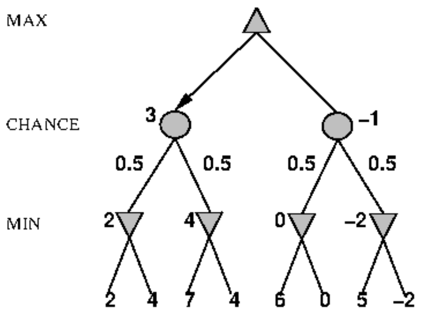


Figure 5.3: Two non-terminal states in chess that are to be evaluated with EVAL

5.4 Games that include an element of chance

Games mirror the unpredictability by including a random element, such as dice. For games that include chance, the game tree includes also chance nodes:



The algorithm changes to:

EXPECTIMINIMAX = MINIMAX + chance nodes

EXPECTIMINIMAX-VALUE= Σ P(S)*EXPECTIMINIMAX-VALUE(SUCCESORS(STATE))

```
if state is a MAX node then
  return the highest EXPECTIMINIMAX-VALUE of
      SUCCESORS(state)
if state is a MIN node then
  return the lowest EXPECTIMINIMAX-VALUE of SUCCESORS
      (state)
if state is a MAX node then
  return the average of EXPECTIMINIMAX-VALUE of
      SUCCESORS(state)
```

Question: Which is the best move in the tree game from 5.5, if all the branches from each chance node have the same probability? Answer: The right move. Explain why.

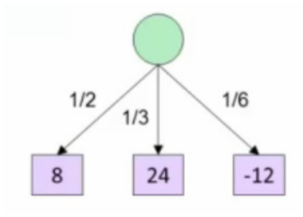


Figure 5.4: A game with the expected value v=1/2 * 8+ 1/3 * 24 + 1/6 * -12

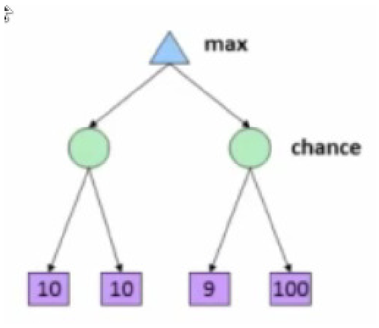
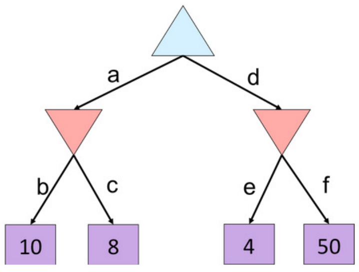


Figure 5.5: Game tree with chance nodes

5.5 Problems

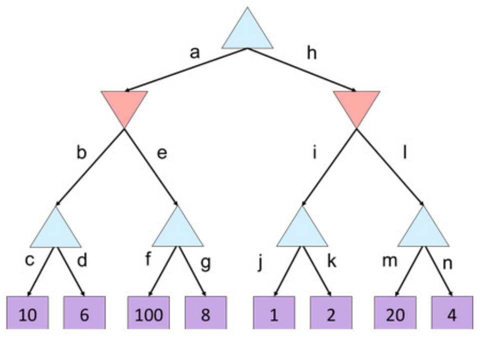
5.5.1 Theoretical problems

1. Which branch is pruned in the following game tree?



   Answer: f

2. Which branch is pruned in the following game tree?



3. Apply ExpectiMinimax on the game tree from figure 5.6. The branches of the chance nodes have the same probability.

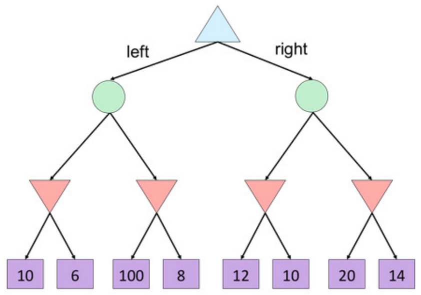


Figure 5.6: Another game tree with chance nodes

- Which is the value of the game?
- Which is the optimal action for MAX?

5.5.2 Practical problems

Java

1. Analyze the implementation of *MiniMax Search* from *aima.core.search. adversarial.MinimaxSearch.java*.

   Run *aima.gui.demo.search.TicTacToeDemo* which uses this search. Build on paper the first four layers of the tree built during the run.

   Do the same thing for *Alpha-beta Search*.

2. Play TicTacToe against an agent that implements MiniMax, AlphaBeta, IncrementalAlphaBeta. For this, run the class *aima.gui.applications.search. games.TicTacToeApp*.

   - Starting from the following configuration (figure 5.7), draw the trees that are built with MiniMax and AlphaBeta. Make a comparison between these two algorithms in terms of expanded nodes.

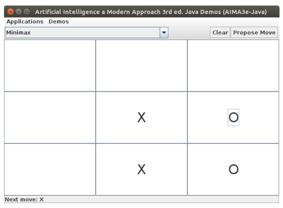

Artificial Intelligence a Modern Approach 3rd ed. Java Demos (AIMA3e-Java)
Applications Demos
Minimax
Clear
Propose Move
X
O
X
O
Next move: X

Figure 5.7: TicTacToe almost final configuration

- Run ConnectFourApp from the package *aima.gui.applications.search.games*. Look carefully at the reported number of expanded nodes. How can you justify its large range?

Python
Different to the previous chapter, now we consider also the ghosts as agents. A brief description of the files in package is:

multiAgents.py	It is where all of the pac-man algorithms will reside, so here you have to do your changes.
pacman.py	he main file that runs Pac-Man games. This file also describes a Pac-Man GameState type.
game.py	The logic behind how the Pac-Man world works. This file describes several supporting types like AgentState, Agent, Direction, and Grid.
util.py	Useful data structures for implementing search algorithms.
Not so important files:	
graphicsDisplay.py	Graphics for Pac-Man
graphicsUtils.py	Support for Pac-Man graphics
textDisplay.py	ASCII graphics for Pac-Man
ghostAgents.py	Agents to control ghosts
keyboardAgents.py	Keyboard interfaces to control Pac-Man
layout.py	Code for reading layout files and storing their contents

1. Reminder **Reflex Agent**. Improve the function *evaluationFunction* of ReflexAgent in *multiAgents.py*. This function evaluates actions. The provided reflex agent code has some helpful examples of methods that query the GameState for information. A capable reflex agent will have to consider both food locations and ghost locations to perform well [pac]. Your agent should easily and reliably clear the testClassic layout:

   pacman.py -p ReflexAgent -l testClassic -n 10 -q.

   Test your agent on *openClassic* layout.

2. Write an adversarial search agent in the provided *MinimaxAgent* class stub in *multiAgents.py*. Your minimax agent should work with any number of ghosts, so you'll have to write an algorithm that is slightly more general than what appears in the textbook. In particular, your minimax tree will have multiple min layers (one for each ghost) for every max layer.

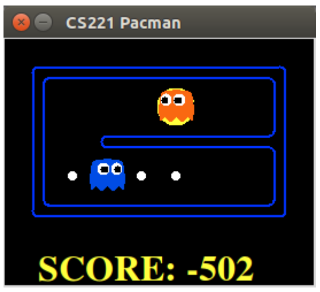
   Your code should also expand the game tree to an arbitrary depth -a depth=4. Score the leaves of your minimax tree with the supplied *self.evaluationFunction*, which defaults to *scoreEvaluationFunction*. MinimaxAgent extends MultiAgentAgent, which gives access to self.depth and self.evaluationFunction. Make sure your minimax code makes reference to these two variables where appropriate as these variables are populated in response to command line options [pac].

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

3. Make a new agent that uses alpha-beta pruning to more efficiently explore the minimax tree, in AlphaBetaAgent. Compare running at depth 3 with minimax at depth 2.

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

4. Consider the ghosts as chance nodes. Assume you will only be running against RandomGhost ghosts, which choose amongst their getLegalActions uniformly at random. The expected behavior is: if Pac-Man perceives that he could be trapped but might escape to grab a few more pieces of food, he'll at least try [pac].



```
python pacman.py -p AlphaBetaAgent -l trappedClassic -a depth=3 -q -n 10
python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10
```

Laboratory work 6

Constraint satisfaction problems

This laboratory work will be studied in two consecutive weeks: in the first week, the backtracking search will be studied, and in the second, the local search with minimum-conflict heuristic.

Different to problems analyzed in the previous chapters, CSPs introduces states and goal test which conform to a standard, structured, and very simple representation.

Variables - a set of variables X_i; each X_i has a nonempty **domain** D_i.

Constraints - each constraint C_i involves some subset of variables and specifies the allowable combinations of values for that subset

A state of the problem is defined by an assignment of values to some or all of the variables.

Consistent or legal assignment = an assignment that does not violate any constraints.
Solution = a complete assignment (in which each variable has a value) and which satisfies all the constraints

Example: map coloring (figure 6.1)

Variables:	WA, NT, Q, NSW, V, SA, T
Domains:	$D_i = \{red, green, blue\}$
Constraints:	adjacent regions must have different colors

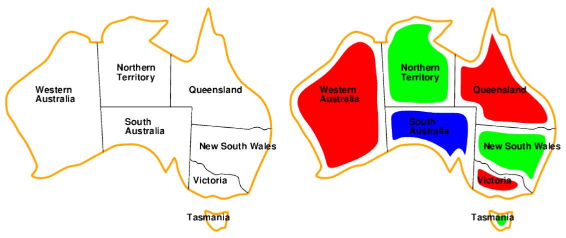
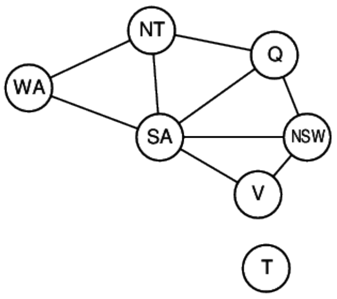


Figure 6.1: Australia: map coloring

CSPs with *Binary constraints* can be represented in *constraints graph*:



6.1 Backtracking search and improvements

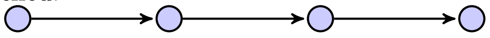
6.1.1 Backtracking search

DFS + variable ordering + fails in violation

```
function BACKTRACKING-SEARCH(csp) returns solution/
   failure
    return RECURSIVE-BACKTRACKING({},csp)

function RECURSIVE-BACKTRACKING(assignment, csp)
   returns soln/failure
 if assignment is complete then return assignment
 var<-SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp],
    assignment, csp)
 for each value in ORDER-DOMAIN-VALUES(var,
    assignment, csp) do
   if value is consistent with assignment given
      CONSTRAINTS[csp] then
      add {var = value } to assignment
      result<- RECURSIVE-BACKTRACKING(assignment, csp)
      if result <>failure then return result
      remove {var = value} from assignment
 return failure
```

Example. Apply backtracking search on the following constraint graph. Consider that the nodes are selected from left to right and each node can take one value from a domain with 3 values. Two adjacent nodes can not have the same value. *How many times will the algorithm go back?* **0** times, since no failure is reached.



Improvements. General purpose improvements of the algorithm are possible, considering the following questions:

- which variable must be assigned first?

- in which order must the values be tried?
- can the inevitably failure be detected sooner?

6.1.2 Minimum remaining values

Selection of the **variable** with the fewest "legal" values is called the minimum remaining values (MRV) heuristic ([BR75]). It also has been called the *most constrained variable* or *fail-first* heuristic, the latter because it picks a variable that is most likely to cause a failure soon, thereby pruning the search tree.

If there is a variable X with zero legal values remaining, the MRV heuristic will select X and failure will be detected immediately. In this way, it is avoided the pointless searches through other variables which always will fail when X is finally selected [RN02].

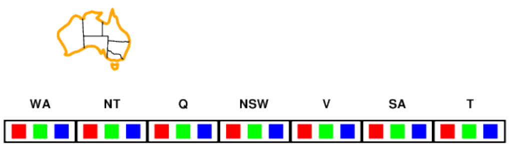
In case of equality between variables according to MRV, Degree heuristic [Bré79] tries to reduce the branching factor on future choices by selecting the variable that is involved in the largest number of constraints on other unassigned variables.

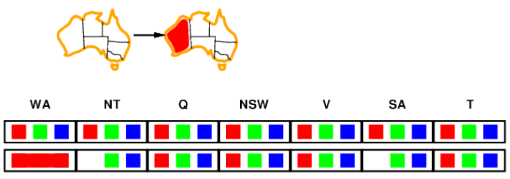
6.1.3 Least constraining value

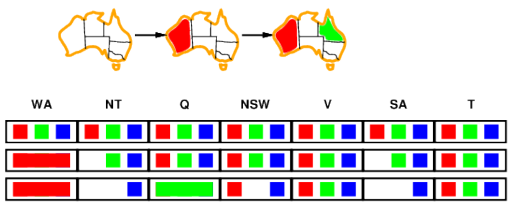
It prefers the **value** that rules out the fewest choices for the neighboring variables in the constraint graph.
Example. Suppose $WA = red$ and $NT = green$, and that the choice is done for the variable Q. *Blue* would be a bad choice, because it eliminates the last legal value left for Q's neighbor, SA. The least-constraining value heuristic therefore prefers *red* to *blue*.

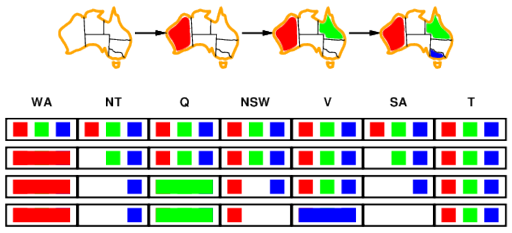
6.1.4 Forward checking

Whenever a variable X is assigned, the *forward checking* process looks at each unassigned variable Y that is connected to X by a constraint and deletes from Y's domain any value that is inconsistent with the value chosen for X.
Example. Forward checking does the following steps after the assignment $WA = red$:

1. 

2. 

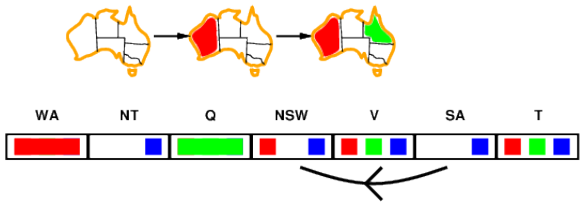
3. 

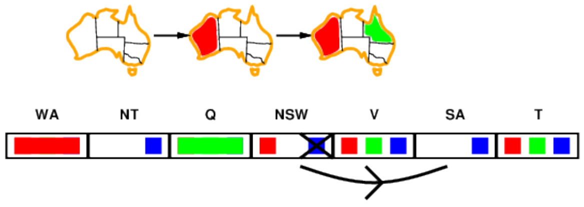
4. 

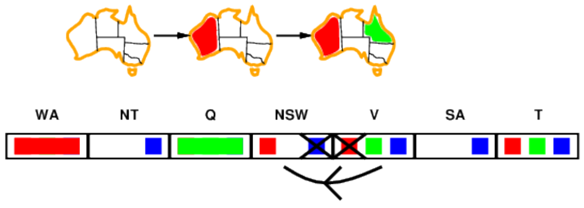
Observation: *NT* and *SA* have the same color from step 3, but the *forward checking* misses this. Solution: *constraint propagation.*

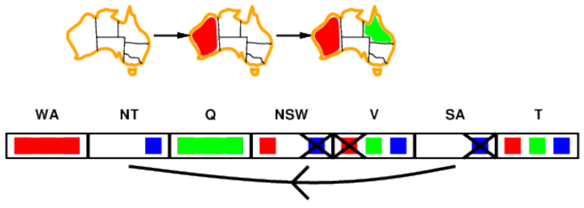
6.1.5 Arc consistency

Is the most simple constraint propagation. Arc refers to a directed arc in the constraint graph: e.g. the arc from SA to NSW. Given the current domains of SA and NSW, the arc is consistent if, for every value x of SA, there is some value y of NSW that is consistent with x.

1. 

2. 

3. 

4. 

Arc consistency checking can be applied either as a *preprocessing step* before the beginning of the search process, or as a *propagation step* (like forward checking) after every assignment during search.

AC-3, uses a queue to keep track of the arcs that need to be checked for inconsistency. Each arc (X_i, X_j) in turn is removed from the queue and checked; if any values need to be deleted from the domain of X_i, then every arc (X_k, X_i) pointing to X_i must be reinserted on the queue for checking [RN02].

```
function AC-3(csp) returns {CSP, posibil cu domenii
   reduse}
    inputs: csp - o problema CSP binara cu variabilele
        X1, X2, ..., Xn
    local variables: queue, o coada de arce, initial
       toate arcele din csp

 while queue is not empty do
     (Xi, Xj)<-REMOVE-FIRST(queue)
     if REMOVE-INCONSISTENT-VALUES(Xi, Xj) then
        for each Xk in NEIGHBORS[Xi] do
             add (Xk, Xi) to queue
------------------------------------------------------
function REMOVE-INCONSISTENT-VALUES(Xi,Xj) returns
   true iff succeeds
   removed<-false
   for each x in DOMAIN[Xi] do
       if no value y in DOMAIN[Xj] allows (x,y) to
          satisfy the constraint Xi -> Xj
          then delete x from DOMAIN[Xi];
             removed<-true
    return removed
```

6.2 Local search: minimum number of conflicts

The initial state may be chosen randomly or by a greedy assignment process that chooses a minimal-conflict value for each variable in turn. In choosing a new value for a variable, the most obvious heuristic is to select the value that results in the minimum number of conflicts with other variables given the rest of the current assignment.

Observation On the n-queens problem, if you don't count the initial placement of queens, the runtime of min-conflicts is roughly independent of problem size. It solves even the million-queens problem in an average of 50 steps (after the initial assignment).

6.3 Use of CSPs

- Planning through CSP is a solution described in [DK01]. The planning graph of the Graphplan algorithm is converted into a CSP and solved using standard CSP solvers.

For the problem of changing the tire, the planning graph, together with the mutex relations, is described in 6.2.

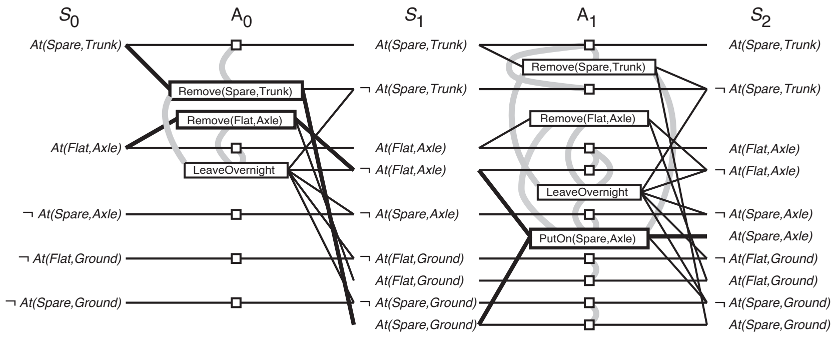


Figure 6.2: Planning graph for the spare tire problem [RN02]

Mutex relations between actions are explained by:

- inconsistent effects: $Remove(Spare, Trunk)$ is mutex with $LeaveOvernight$ since they have $At(Spare, Ground)$, respectively $\neg At(Spare, Ground)$ as effects
- interference $Remove(Flat, Axle)$ is mutex with $LeaveOvernight$ since one of its precondition, $At(Flat, Axle)$, is the negation of the effect of $LeaveOvernight$
- competing needs: $PutOn(Spare, Axle)$ is mutex with $Remove(Flat, Axle)$ because $At(Flat, Axle)$ is the precondition of one action, while its negation of the other one

Mutex relations between literals at the same level is named inconsistent support. There are two cases for inconsistent support:

- one literal is the negation of the other
- each possible pair of actions that could achieve the two literals is mutually exclusive.

$At(Spare, Axle)$ is mutex with $At(Flat, Axle)$ in S_2 since the only way to obtain $At(Spare, Axle)$ is by doing $PutOn(Spare, Axle)$ which at his turn is mutex with the persistence action on $At(Flat, Axle)$

- Crosswords puzzles [Gin11], Timetable creation [ZL05]
- Constraint programming - CLPFD constraint solver library from YAP and SWI-Prolog, Gurobi.

6.4 Problems

6.4.1 Theoretical problems

1. Which graph from figure 6.3 is the correct constraint graph for the following problem: There are 6 presenters for an event: A, B, C, D, E, F.

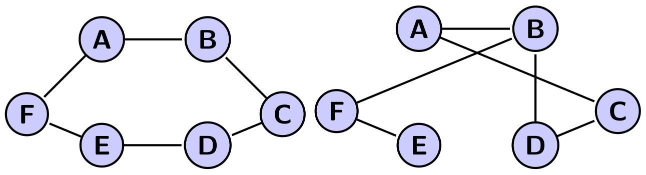


Figure 6.3: Which is the good constraint graph?

The events for A and B are scheduled at moment 1, for C and D at 2 and for E and F at 3. The number of available rooms is 2. The following conditions must also be true:

- the presenters scheduled at the same time, can not be assigned in the same room
- A and C can not be assigned in the same room
- B and F can not be assigned in the same room
- B and D can not be assigned in the same room

2. Suppose the configuration 6.4 was reached with *backtracking search* with *forward checking*. The color for the node in the middle is assigned now. What is the configuration reached after *forward checking*? Explain why?

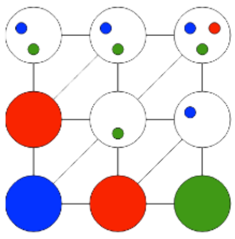
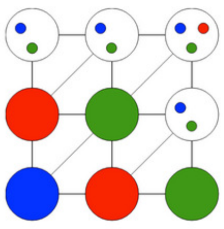
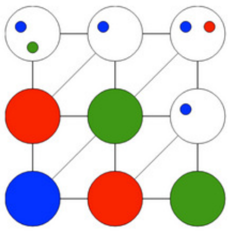
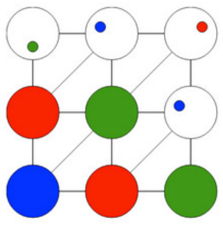


Figure 6.4: Node coloring: the node in the middle is colored green

1.  2.  3. 

Answer: 2.

3. Consider AC-3 run on a CSP with 4 variables: A, B, C, D. Consider that the following arcs are in the queue:
   (a) $A \rightarrow B$
   (b) $A \rightarrow C$
   (c) $B \rightarrow A$

(d) $B \rightarrow D$

(e) $C \rightarrow D$

(f) $D \rightarrow C$

When checking the consistency of the arc $A \rightarrow B$ some values are removed. Which arcs will be added into the queue as a consequence?

(a) $A \rightarrow D$

(b) $B \rightarrow D$

(c) $C \rightarrow A$

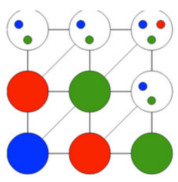
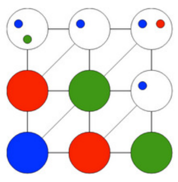
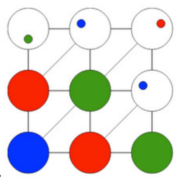
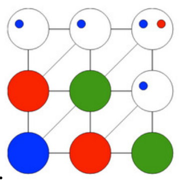
(d) $C \rightarrow D$

(e) $D \rightarrow A$

(f) $D \rightarrow B$

(g) none

4. Consider the partial assignment from figure 6.4. After the assignment of the variable in the middle and application of AC-3, which is the resulted assignment?

1.  2.  3.  4. 

5. Apply *Arc Consistency* on the map coloring problem, with the following initial assignment: WA=red, V=blue. Does the solution exists?

6. You must arrange three statues in an exhibit hall: an ice carving of a swan (i), a gold lion (g), and a marble abstract piece (m). There are three tables, 1, 2, and 3, arranged in a row, with 1 closest to the door and 3 farthest into the exhibit hall. It is a hot day and so the ice carving cannot be nearest the door. Your manager also informs you that it will look bad to have to animal sculptures on adjacent tables. Reality tells you that each table must have a different sculpture. If we formulate this problem as a binary CSP with variables X1, X2 , and X3 , each with domain i, g, m:

(a) What are the unary constraint(s) (list them explicitly)
Answer: $X1 = i$

(b) What are the binary constraint(s) (list them explicitly)
Answer:
(X1 , X2) {(i, m), (m, i), (g, m), (m, g)}
(X2 , X3) ∈ {(i, m), (m, i), (g, m), (m, g)}
(X1 , X3) ∈ {(i, m), (m, i), (g, m), (m, g), (i, g), (g, i)}
Assume we enforce the unary constraint(s) in pre-processing for the remaining parts:

(c) Which variable will be assigned first by the MRV heuristic?
Answer: $X1$ - *explain why*

(d) If we assign X3 = i, show the domains of the remaining variables after *forward checking.*

(e) If no variables are assigned, show the initial domains after running arc consistency.

(f) If its a cool day, and we drop the requirement that the ice swan cannot be nearest the door, what are the initial domains after running arc consistency?

7. Suppose you have a set of n actions, with a partial order $\succ$ between them, i.e. if $a_i \succ a_j$ then a_i must be executed after a_j. Find sequences that respect this partial order.

8. Consider the schedule of movies at TIFF movies festival from Cluj. Identify three types of constraints that a person could have, propose a way to model it, and give a solution for a particular day from the festival.

6.4.2 Practical problems

Java:

1. Study the classes

CSP	*aima.core.search.csp.CSP*
Map CSP	*aima.core.search.csp.MapCSP*
AC-3	*aima.core.search.csp.AC3Strategy*
Backtracking-Search	*aima.core.search.csp.BacktrackingStrategy*
Min-Conflicts	*aima.core.search.csp.MinConflictsStrategy*

- Run the Map coloring application demo on Australia's map with *aima.gui.demo.search.MapColoringCSPDemo.* Change the CSP to another scenario, similar to MapCSP class.
- Run *aima.gui.applications.search.csp.MapColoringApp* and analyse different algorithms on different starting configurations (figure 6.5). Compare the performances of the simple version of the algorithm *backtracking* with the improved versions with *MRV*, *LCV*, *Forward checking*, *AC-3.*

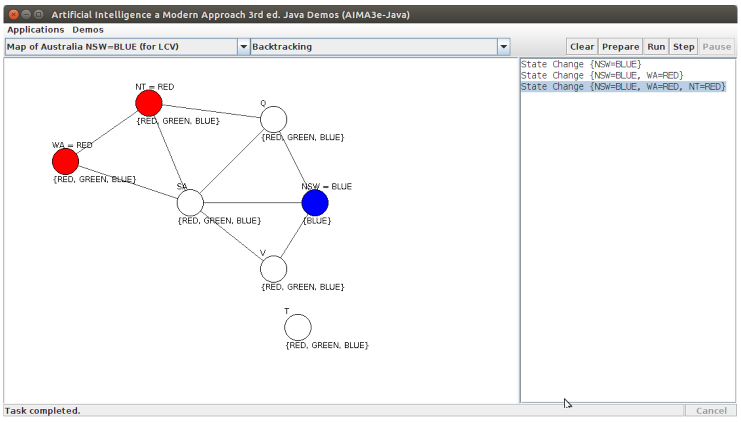
2. Solve the cryptarithmetic problem from figure 6.6. You can use java classes from aima3e-java

3. Solve Einstein puzzle:

(a) There are 5 houses (along the street) in 5 different colors: blue, green, red, white and yellow.

(b) In each house lives a person of a different nationality: Brit, Dane, German, Norwegian and Swede.

(c) These 5 owners drink a certain beverage: beer, coffee, milk, tea and water, smoke a certain brand of cigar: Blue Master, Dunhill, Pall Mall, Prince and blend, and keep a certain pet: cat, bird, dog, fish and horse.



Map of Australia NSW=BLUE (for LCV)
Backtracking

```
State Change {NSW=BLUE}
State Change {NSW=BLUE, WA=RED}
State Change {NSW=BLUE, WA=RED, NT=RED}
```

Task completed.

Figure 6.5: Aima3e-java: application for CSP

```
    T W O
  + T W O
  F O U R
```

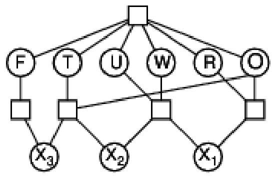


Figure 6.6: Cryptarithmetic problem and its constraint graph

(d) No owners have the same pet, smoke the same brand of cigar, or drink the same beverage.

(e) Hints:

i. The Brit lives in a red house.

ii. The Swede keeps dogs as pets.

iii. The Dane drinks tea.

iv. The green house is on the left of the white house (next to it).

v. The green house owner drinks coffee.

vi. The person who smokes Pall Mall rears birds.

vii. The owner of the yellow house smokes Dunhill.

viii. The man living in the house right in the center drinks milk.

ix. The Norwegian lives in the first house.

x. The man who smokes blend lives next to the one who keeps cats.

xi. The man who keeps horses lives next to the man who smokes Dunhill.

xii. The owner who smokes Blue Master drinks beer.

xiii. The German smokes Prince.

xiv. The Norwegian lives next to the blue house.

xv. The man who smokes blend has a neighbor who drinks water.

Python

Use the package from the folder $\$LAB\_WORKS/ch6$ or `https://labix.org/python-constraint`. Short usage manual:

```
>> from constraint import *
>> problem = Problem()
>> problem.addVariable(”a”, [1,2,3])
>> problem.addVariable(”b”, [4,5,6])
>> problem.getSolutions()
[{’a’: 3, ’b’: 6}, {’a’: 3, ’b’: 5}, {’a’: 3, ’b’: 4},
{’a’: 2, ’b’: 6}, {’a’: 2, ’b’: 5}, {’a’: 2, ’b’: 4},
{’a’: 1, ’b’: 6}, {’a’: 1, ’b’: 5}, {’a’: 1, ’b’: 4}]
>> problem.addConstraint(lambda a, b: a*2 == b, (”a”, ”b”))
>> problem.getSolutions() [{’a’: 3, ’b’: 6}, {’a’: 2, ’b’: 4}]
>> problem = Problem()
>> problem.addVariables([”a”, ”b”], [1, 2, 3])
>> problem.addConstraint(AllDifferentConstraint())
>> problem.getSolutions()
[{’a’: 3, ’b’: 2}, {’a’: 3, ’b’: 1}, {’a’: 2, ’b’: 3},
{’a’: 2, ’b’: 1}, {’a’: 1, ’b’: 2}, {’a’: 1, ’b’: 3} ]
```

1. Use this package in order to solve n-queens problem.

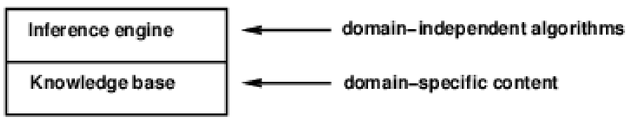
2. Use this package in order to solve magic square problem.

3. Use this package in order to solve sudoku.

4. Use this package in order to solve Battleship Puzzle, with or without Hints obtained during the game.

Laboratory work 7

Propositional Logic



Knowledge base is a set of sentences in a formal language
Inference is the procedure which derive new sentences from old ones
Logical entailment is a relation between sentences that satisfy the following condition: a sentence follows 1ogically from another sentence or set of sentences: $KB \models \alpha$.

Logics are formal languages used to represent knowledge in order to extract new conclusions. A logic must define:

- Syntax of the language - all the sentences that are well formed
- Semantics of the language - the truth of each sentence with respect to each possible world [RN02].

7.1 Propositional Logic

Is the most simple logic. The rules to describe sentences well-formed (**syntax**) in Propositional Logic are:

- Atomic sentences P_1, P_2 - single proposition symbols
- If S is a sentence, $\neg S$ is also a sentence: (negation $\neg$)
- If S_1 and S_2 are sentences, $S_1 \land S_2$ is also a sentence (conjunction $\land$)
- If S_1 and S_2 are sentences, $S_1 \lor S_2$ is also a sentence (disjunction $\lor$)
- If S_1 and S_2 are sentences, $S_1 \implies S_2$ is also a sentence (implication $\Rightarrow$)
- If S_1 and S_2 are sentences, $S_1 \Leftrightarrow S_2$ is also a sentence (biconditional $\Leftrightarrow$)

The **semantics** is:

$\neg S$	is true iff	S	is false		
$S_1 \wedge S_2$	is true iff	S_1	is true *and*	S_2	is true
$S_1 \vee S_2$	is true iff	S_1	is true *or*	S_2	is true
$S_1 \Rightarrow S_2$	is true iff	S_1	is false *or*	S_2	is true
and	is false iff	S_1	is true *and*	S_2	is false
$S_1 \Leftrightarrow S_2$	is true iff	$S_1 \implies S_2$	is true *and*	$S_2 \implies S_1$	is true

and the truth tables for the connectives is:

P	Q	¬ P	P ∧ Q	P ∨ Q	P ⇒ Q	P ⇔ Q
false	false	true	false	false	true	true
false	true	true	false	true	true	false
true	false	false	false	true	false	false
true	true	false	true	true	true	true

7.1.1 Wumpus world

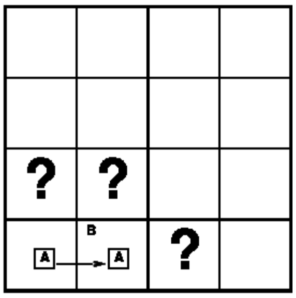
Read the complete description of the *Wumpus world* from the chapter *Propositional Logic* from the book [RN02]. Here there are some examples of sentences in propositional Logic:

$P_{i,j}$ is true if there is a pit in $[i, j]$.
$B_{i,j}$ is true if there is a breeze in $[i, j]$

Knowledge base after the agent visits two squares:

$$\neg P_{1,1}$$
$$\neg B_{1,1}$$
$$B_{2,1}$$
$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$$
$$B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$$

Question: From the current Knowledge base, is it sure that $P_{2,2}$ is safe? What about $P_{1,2}$?



In the next two sections, two inference algorithms for answering to this question are described. Try to apply both of them on the described knowledge base from the Wumpus World.

7.2 Resolution

7.2.1 Resolution inference rule

Resolution inference rule has as input *two clauses* which contain *two complementary literals* and as *output* a new clause. The new clause contains all the literals of the two original clauses except the two complementary literals. It applies only to disjunctions of literals, therefore before its application, the knowledge base must be converted to Conjunctive Normal Form (CNF).

$$\frac{\ell_1 \vee \cdots \vee \ell_k, \qquad m_1 \vee \cdots \vee m_n}{\ell_1 \vee \cdots \vee \ell_{i-1} \vee \ell_{i+1} \vee \cdots \vee \ell_k \vee m_1 \vee \cdots \vee m_{j-1} \vee m_{j+1} \vee \cdots \vee m_n}$$

, where ℓ_i si m_j are complementary literals.

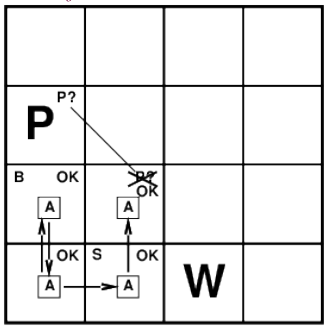


Figure 7.1: Perceptions in wumpus world after the agent does 5 steps

Resolution inference rule in action for figure 7.1:

$$\frac{P_{1,3} \vee P_{2,2}, \qquad \neg P_{2,2}}{P_{1,3}}$$

What is the explanation in plain English?

7.2.2 Conversion to CNF

Please follow the conversion to CNF of the following sentence:

$$B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$$

1. Remove $\Leftrightarrow$, replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \implies \beta) \wedge (\beta \implies \alpha)$.

$$(B_{1,1} \implies (P_{1,2} \vee P_{2,1})) \wedge ((P_{1,2} \vee P_{2,1}) \implies B_{1,1})$$

2. Remove $\Rightarrow$, replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$.

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg(P_{1,2} \vee P_{2,1}) \vee B_{1,1})$$

3. Introduce $\neg$ inside the brackets with the use of *de Morgan* rules and double negation:

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge ((\neg P_{1,2} \wedge \neg P_{2,1}) \vee B_{1,1})$$

4. Apply the rules of distributivity of connectives ($\vee$ and $\wedge$):

$$(\neg B_{1,1} \vee P_{1,2} \vee P_{2,1}) \wedge (\neg P_{1,2} \vee B_{1,1}) \wedge (\neg P_{2,1} \vee B_{1,1})$$

7.2.3 The resolution algorithm

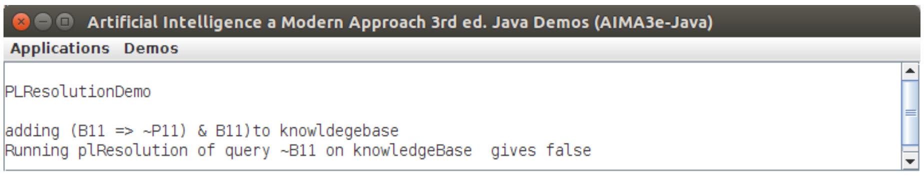
It uses the principle of **proof by contradiction**. It proves $KB \models \alpha$ by providing a contradiction for $KB \wedge \neg\alpha$, or in other terms it proves the unsatisfiability of the knowledge base extended with the negation of the target sentence.

```
function PL–RESOLUTION(KB,α) returns true or false
inputs: KB, the knowldge base, a sentence in propositional logic
        α, the query, a sentence in propositional logic

clauses←the set of clauses in the CNF representation of KB ∧ ∼ α
new← {}
loop do
  for each $C_i, C_j$ in clauses do
    resolvents←PL–RESOLVE($C_i, C_j$)
    if resolvents contains the empty clause then return true
    new ← new ∪ resolvents
  if new ⊆clauses then return flase
  clauses ← clauses ∪ new
```

Question: Identify the resolution step for the knowledge base from *aima.gui.demo.logicPLResolutionDemo* from *aima3e-java*. The output can be observed in figure 7.2.
Observe the main steps of a *Knowledge-based agent* as they are implemented in aime3e-java demo:

```
private static PLResolution plr = new PLResolution()
.....
{
 KnowledgeBase kb = new KnowledgeBase();
 String fact = "(B11 => ~P11) & B11)";
 kb.tell(fact);
 displayResolutionResults(kb, "~B11");
}
 .....
private static void displayResolutionResults(KnowledgeBase kb,
    String query) {
 PLParser parser = new PLParser();
 ... plr.plResolution(kb, parser.parse(query));
        }
```


Artificial Intelligence a Modern Approach 3rd ed. Java Demos (AIMA3e-Java)
Applications Demos

```
PLResolutionDemo

adding (B11 => ~P11) & B11)to knowldgebase
Running plResolution of query ~B11 on knowledgeBase  gives false
```

Figure 7.2: Aima3e-java: demo for *Resolution* on *Propositional Logic*

7.3 Forward chaining algorithm

It applies only on Horn-clauses. A Horn clause is a disjunction of literals of which at most one is positive. The *Horn clause* can be written as implications: the premise is a conjunction of positive literals and the conclusion is a single positive literal.

Definite clauses are Horn clauses with exactly one positive literal. This literal is the head and the negative literals form the body of the clause ([RN02]).

```
function PL–FC–ENTAILS?(KB, q) returns true or false
 inputs: KB – the knowledge base, a set of propositional Horn
     clauses
         q – the query, a proposition symbol
 local variables:
    count –  a table, indexed by clause, initially the number of
        premises
    inferred – a table, indexed by symbol, each entry initially
        false
    agenda – a list of symbols, initially the symbols known in KB
while agenda is not empty do
  p ← POP(agenda)
  unless inferred[p] do
     inferred[p] ← true
     for each Horn clause c in whose premise p appears do
       decrement count[c]
       if count[c]=0 then do
         if HEAD[c]=q then return true
         PUSH(HEAD[c], agenda)
return false
```

7.4 Problems

7.4.1 Theoretical problems

1. Show the steps of the Forward-chaining algorithm for the following knowledge base. The question is $KB \models Q$ $P \implies Q$
   $L \wedge M \implies P$
   $B \wedge L \implies M$
   $A \wedge P \implies L$
   $A \wedge B \implies L$
   A
   B

2. Consider the agent from wumpus world did 7 steps and it had the perceptions described in figure 7.3. Write the corresponding knowledge base and prove that $S_{1,3}$ is false, where $S_{i,j}$ is true if the square (i,j) is safe.

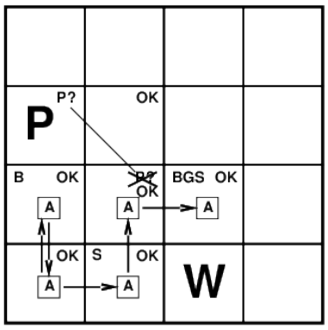


Figure 7.3: Perceptions in wumpus world after the agent does 7 steps

3. Consider the Einstein puzzle. Is it possible to represent it in propositional logic? if the answer is yes, please give a knowledge base in propositional logic.

7.4.2 Practical problems

Java

1. Study the following files from aima3e-java:

KBAgent	*aima.core.logic.propositional.agent.KBAgent*
ConvertToCNF	*aima.core.logic.propositional.visitors.ConvertToCNF*
Resolution	*aima.core.logic.propositional.inference.PLResolution*
ForwardChaining	*aima.core.logic.propositional.inference.PLFCEntails*

2. Implement an agent that finds pits in wumpus world (figure 7.4). Use the demo of *Resolution algorithm* from *aima.gui.demo.logicPLResolutionDemo*. Identify the situations where *Forward Chaining* is not applicable.

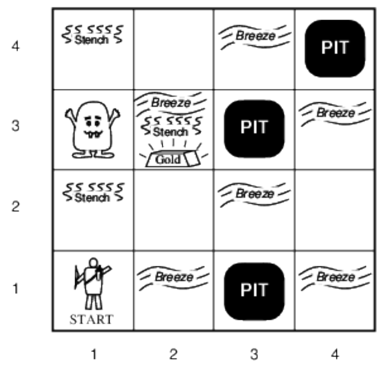


Figure 7.4: Wumpus world

3. Create a knowledge base for the following puzzle (the puzzle was presented by Raymond Smully in [Smu87]) and try to prove that A is a knave with one of the two inference algorithms from this lab.

    The Island of Knights and Knaves has two types of inhabitants: knights, who always tell the truth, and knaves, who always lie. One day, three inhabitants (A, B, and C) of the island met a foreign tourist and gave the following information about themselves:

    1. A said that B and C are both knights.
    2. B said that A is a knave and C is a knight.
    What types are A, B, and C?

4. Another puzzle from the same book [Smu87] is about Ork and Bog. Try to solve it by yourselves and then check if propositional logic is expressive enough to describe it. It case it is, implement it for the same demo from *aima3e-java*.

On Ganymede – a satellite of Jupiter – there is a club known as the Martian-Venusian Club. All members are either from Mars or from Venus, although visitors are sometimes allowed. An earthling is unable to distinguish Martians from Venetians by their appearance. Also, earthlings cannot distinguish either Martian or Venusian males from females, since they dress alike. Logicians, however, have an advantage, since the Venusian women always tell the truth and the Venusian men always lie. The Martians are the opposite; the Martian men tell the truth and the Martian women always lie.
One day a visitor met two Club members, Ork and Bog, who made the following statements:
1. Ork: Bog is from Venus.
2. Bog: Ork is from Mars.
3. Ork: Bog is male.
4. Bog: Ork is female.
Where are Ork and Bog from, and are they male or female?

Laboratory work 8

Test your AI knowledge with AI games

8.1 Warlight AI Challenge

Formerly known as Conquest, Warlight `http://theaigames.com/competitions/warlight-ai-challenge` is a competition of bots playing a Risk-like game.

You are given a starting package and you have to create a bot for the game. The support bots are written in C, C#, C++, Clojure, Go, Haskell, Java, JavaScript, Perl, PHP, Prolog, Python, Ruby, Scala and Tcl.

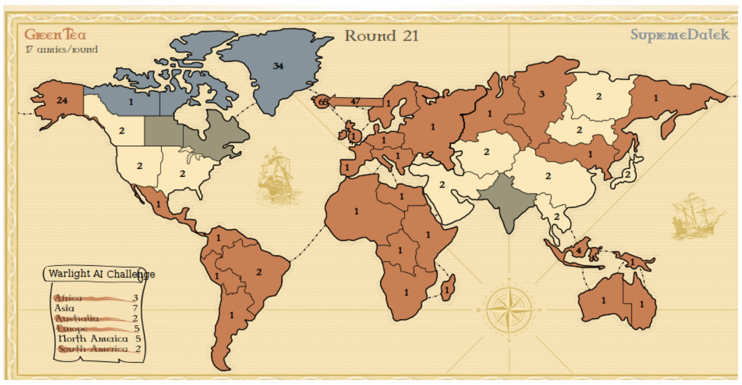


Figure 8.1: Warlight competition

An example of a bot that tries to place 1 army on each region of the map and then try to attack/transfer 1 army from each region to the next neighbouring region is given in the following lines:

```
import java.util.Scanner;

public class MyBot {
    private Scanner scan = new Scanner(System.in);
    public void run()
    {
        while(scan.hasNextLine()) {
            String line = scan.nextLine();
```

```
                if(line.length() == 0) {
                    continue;
                }
                String[] parts = line.split(" ");
                if(parts[0].equals("pick_starting_regions")) {
                    System.out.println( "give me randomly" );
                }
                else if(parts.length == 3 && parts[0].equals("go")) {
                    String output = "";
                    if(parts[1].equals("place_armies")) {
                        for(int i=1; i<=42; i++) {
                            output.concat("myBot place_armies " + i + "
                                 1,");
                        }
                    }
                    else if(parts[1].equals("attack/transfer")) {
                        for(int i=1; i<=41; i++) {
                            output.concat("myBot attack/transfer " + i
                                + " " + i+1 + " 1,");
                        }
                    }
                    System.out.println(output);
                }
            }
        }

        public static void main(String[] args) {
            (new MyBot()).run();
        }
}
```

8.1.1 Warlight AI Challenge 2

Is the successor to Warlight AI Challenge `http://pub.theaigames.com/competitions/warlight-ai-challenge-2`. The most important doiffernce is that the map are randomly generated.

8.2 Texas Hold'em

Is a poker variant, and the challenge is hosted on `theaigames.com/competitions/`. The supported languages are C, C#, C++, Clojure, Go, Haskell, Java, JavaScript, PHP, Perl, Prolog, Python, Ruby, Scala and Tcl, from which, at the ongoing challenge, the following languages are used Java, Python, C#, C++, JavaScript, Go, Scala.

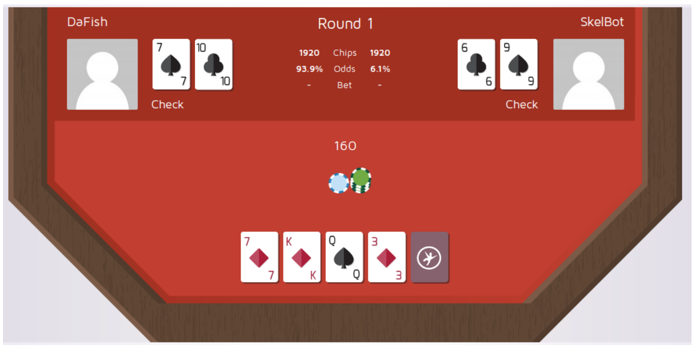
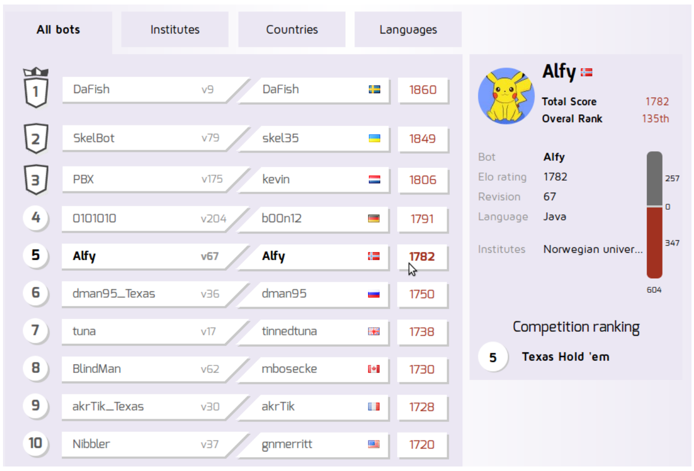


Figure 8.2: TexasHold'em competition



All bots | Institutes | Countries | Languages

1	DaFish	v9	DaFish		1860
2	SkelBot	v79	skel35		1849
3	PBX	v175	kevin		1806
4	0101010	v204	b00n12		1791
5	**Alfy**	v67	**Alfy**		1782
6	dman95_Texas	v36	dman95		1750
7	tuna	v17	tinnedtuna		1738
8	BlindMan	v62	mbosecke		1730
9	akrTik_Texas	v30	akrTik		1728
10	Nibbler	v37	gnmerritt		1720

Alfy

Total Score	1782
Overal Rank	135th

Bot	Alfy
Elo rating	1782
Revision	67
Language	Java
Institutes	Norwegian univer...

257
0
347
604

Competition ranking

5 Texas Hold 'em

8.3 CODECUP 2015

The game for the 2015 edition is Ayu 8.3. The bot can be also tested on local competitions supported by Caia, not only on the official site `www.codecup.nl/`. Programming languages are Pascal, C, C++, Java, Python, Python3, Haskell, Javascript.

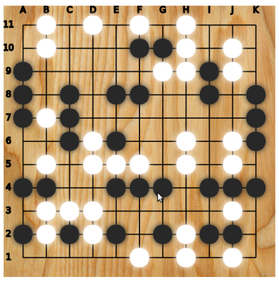


Figure 8.3: Ayu game from Codecup 2015

8.4 Battlecode

It is MIT's competition, `http://www.battlecode.org/`, real-time strategy game. Two teams of robots manage resources and attack each other with different kinds of weapons. Each robot functions autonomously; under the hood it runs a Java virtual machine loaded up with its team's player program. The required knowledge for this game exceeds the studied algorithms, since the robots in the game must communicate and work together to accomplish their goals.

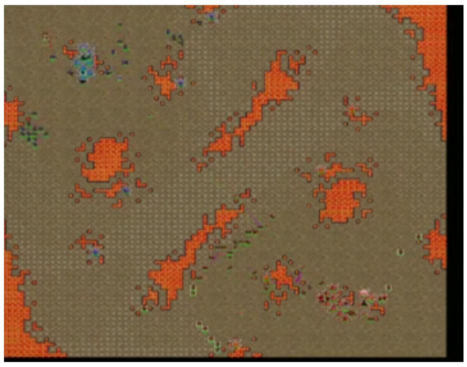


Figure 8.4: Battlecode game

Example of structure of simple Robot player is given bellow:

```java
import battlecode.common.*;
import java.util.*;

public class RobotPlayer {
        static RobotController rc;
        static Team myTeam;
        static Team enemyTeam;
        static int myRange;
        static Random rand;
        static Direction[] directions = {Direction.NORTH, Direction
            .NORTH_EAST, Direction.EAST, Direction.SOUTH_EAST,
            Direction.SOUTH, Direction.SOUTH_WEST, Direction.WEST,
            Direction.NORTH_WEST};

        public static void run(RobotController tomatojuice) {
                rc = tomatojuice;
                rand = new Random(rc.getID());

                myRange = rc.getType().attackRadiusSquared;
                MapLocation enemyLoc = rc.senseEnemyHQLocation();
                Direction lastDirection = null;
                myTeam = rc.getTeam();
                enemyTeam = myTeam.opponent();
                RobotInfo[] myRobots;

                while(true) {
                        try {
                                rc.setIndicatorString(0, "This is
                                    an indicator string.");
                                rc.setIndicatorString(1, "I am a "
                                    + rc.getType());
                        } catch (Exception e) {
                                System.out.println("Unexpected
                                    exception");
                                e.printStackTrace();
                        }

                        if (rc.getType() == RobotType.HQ) {
                                try {
                                        int fate = rand.nextInt
                                            (10000);
                                        myRobots = rc.
                                            senseNearbyRobots
                                            (999999, myTeam);
                                        int numSoldiers = 0;
                                        int numBashers = 0;
                                        int numBeavers = 0;
                                        int numBarracks = 0;
                ....
        }

        // This method will attack an enemy in sight, if there is
            one
        static void attackSomething() throws GameActionException {
                RobotInfo[] enemies = rc.senseNearbyRobots(myRange,
                    enemyTeam);
                if (enemies.length > 0) {
                        rc.attackLocation(enemies[0].location);
                }
        }

        // This method will attempt to move in Direction d (or as
            close to it as possible)
        static void tryMove(Direction d) throws GameActionException
             {
                .....
        }

        // This method will attempt to spawn in the given direction
             (or as close to it as possible)
        static void trySpawn(Direction d, RobotType type) throws
            GameActionException {
                ......

        }

        // This method will attempt to build in the given direction
             (or as close to it as possible)
        static void tryBuild(Direction d, RobotType type) throws
            GameActionException {
                .......
        }

}
```

Bibliography

[aim] Aima3e-java `https://github.com/aima-java/aima-java`.

[BG01] Blai Bonet and Hctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(12):5 – 33, 2001.

[BR75] James R Bitner and Edward M Reingold. Backtrack programming techniques. *Communications of the ACM*, 18(11):651–656, 1975.

[Bré79] Daniel Brélaz. New methods to color the vertices of a graph. *Commun. ACM*, 22(4):251–256, April 1979.

[DK01] Minh Binh Do and Subbarao Kambhampati. Planning as constraint satisfaction: Solving the planning graph by compiling it into {CSP}. *Artificial Intelligence*, 132(2):151 – 182, 2001.

[Gin11] Matthew L Ginsberg. DR. FILL: crosswords and an implemented solver for singly weighted CSPs. *Journal of Artificial Intelligence Research*, pages 851–886, 2011.

[Hof01] Jrg Hoffmann. Ff: The fast-forward planning system. *AI magazine*, 22:57–62, 2001.

[pac] Pac-man projects `http://ai.berkeley.edu/project_overview.html`.

[RN02] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (2nd Edition)*. Prentice Hall, December 2002.

[Smu87] Raymond M. Smullyan. *Forever Undecided: A Puzzle Guide to Gdel*. Oxford University Press, 1987.

[ZL05] Lixi Zhang and SimKim Lau. Constructing university timetable using constraint satisfaction programming approach. In *Proceedings of the International Conference on Computational Intelligence for Modelling, Control and Automation and International Conference on Intelligent Agents, Web Technologies and Internet Commerce Vol-2 (CIMCA-IAWTIC'06) - Volume 02*, CIMCA '05, pages 55–60, Washington, DC, USA, 2005. IEEE Computer Society.