

S T U D I E N A R B E I T



# Solving Wumpus Worlds With Description Logics

S.M. REZA RASOULI

26.11.2009

*A work by:*

*S.M. Reza Rasouli*



*Informatik Ingenieurwesen*

*[reza.rasouli@tu-harburg.de](mailto:reza.rasouli@tu-harburg.de)*

*Institut für Softwaresysteme (STS), E-16*

*Schwarzenbergstraße 95, Gebäude E, 4. Etage*

*D-21073 Hamburg, Germany*

*Technical University of Hamburg-Harburg (TUHH)*

Chapter 1	
Introduction	5
1.1 Wumpus Problem	6
Chapter 2	
Wumpus TBox	9
Chapter 3	
Search Algorithm	15
Chapter 4	
Implementation Basics	18
Chapter 5	
Implementation	22
5.1 General Class	23
5.2 MyButton Class	23
5.3 Stacking Class	24
5.4 Showing Class	25
5.5 Field Class	26
5.6 Wumpus Class	29
5.7 Agent Class	30
Chapter 6	
Examples	33
Example 6.1	34
Example 6.2	38
Example 6.3	39
Example 6.4	41

Example 6.5	42
<b>Chapter 7</b>	
Evaluation	43
7.1 Size Evaluation	44
7.2 General Evaluation	45
<b>Chapter 8</b>	
Conclusion	46
8.1 Summary	47
8.2 Outlook	47
<b>Chapter 9</b>	
Definitions	48
Boards Signs	49
<b>Chapter 10</b>	
References	50

# Chapter 1

---

## Introduction

---

In this work I am going to solve the wumpus problem with an algorithm based on description logics. Before getting through the work, first the wumpus problem is going to be described and then the TBox related to the solution of this problem is going to be shown. Then the results of the test of the TBox with an ABox related to a game board will be illustrated and in the next step the implementation of the above discussed algorithm that uses the TBox and ABox elements to find a solution for the problem is going to be described. At last the result of the implementation with the help of RACER reasoner will be shown.

## 1.1 Wumpus Problem

The wumpus problem is a game played on a board with  $N \times N$  fields where each field on the board can have different states. The field number 1 is always the starting point. The other fields could have a combination of the following states:

- 1- Wumpus
- 2- Gold
- 3- Pit
- 4- Breeze
- 5- Stench
- 6- Glitter

An example for a  $5 \times 5$  game board could be seen in the figure 1. The start point could only have the Breeze state or Stench state or both or neither of them. If the start point contains a Pit or Wumpus or the Gold, then the game is ended. Therefore to avoid that, the above assumption has been made. There is only one field which contains the Gold and only one field that contains the Wumpus. For an  $N \times N$  board, there are maximum  $N-2$  fields that contain a Pit. Pits and Wumpus could be in the same field, whereas the Gold cannot be in the same field where a Pit or Wumpus is present. All of the neighboring fields of a Pit field contain a Breeze and all of the neighboring fields of a Wumpus field contain a Stench. The possible outcomes for each fields could be seen below:

**Start Point (Field #1):** {S,B} , {S} , {B} , {0}

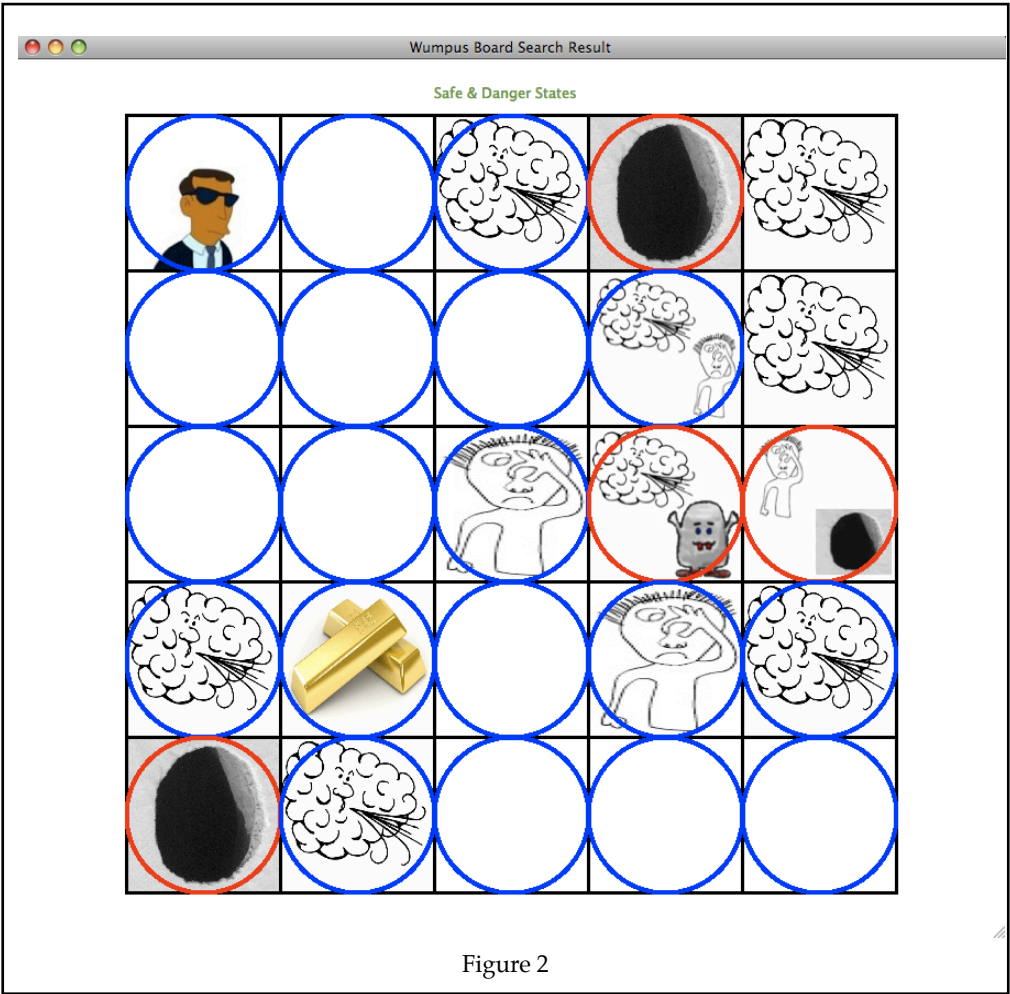
**The rest of the fields :** {S,B} , {S} , {B} , {0}, {S,B,P} , {S,P} , {B,W} , {W,P}, {S,P,B} ,{W,P,B}, {P,B} , {B,G} , {S,G}, {S,B, G}.

The agent A starts the game from the start point and goes through the fields on the

board until it reaches the field that contains the Gold and then returns back to the start point or after searching and realizing that he can not find the Gold, it will return back to the start point. If the agent enters the field that contains the wumpus or a pit, the game is over and the agent has loosed the game. In this work the fields are going to be divided in 2 categories. The safe section and the danger section. Any field that contains the Wumpus or a Pit or both of them, will be categorized under the danger section. The other states will be categorized under the safe section. The start point is always in the safe section. The fields that the agent has no idea about their state will be categorized under none of the 2 sections mentioned above. The agent always tries not to go into a field in the danger section. For the example in figure 1, the safe and danger sections are shown in figure 2. Each of the blue circles show a safe field and each of the red circles show a danger field.

From figure 2 it can be realized that there are some fields that cannot be evaluated by the agent. This is because the agent was not able to be near enough to those fields or doesn't have enough information from the neighboring fields to evaluate the state of this field. As mentioned before, the agent will ignore these fields and as long as it is not able to make a judgment about these fields, it won't go into this fields. The last point that has to be mentioned, regarding the safe and danger states, is that the agent may be able to gather more information about the fields, if the agent

continues the search after finding the gold in a specific field from that field. Although it's obvious that the agent is not able to find a new solution by this method, due to the fact, that loops are not allowed in the solving algorithm, the agent is able to go more through the fields and categorized more fields under safe and danger categories.





# Chapter 2

---

## Wumpus TBox

---

In the following the logics of the wumpus problem will be discussed. The logics of the wumpus problem has been shown on the box below (Wumpus TBox).

The **equation (1)** says that each *field* of the game can be next to maximum 4 neighboring *fields*.

*nextTo* is the relation that connects to fields with each other.

**Equation (2),(3) and (4)** say that *corner*, *edge* and *middle* are all type of *field* and under this category.

**Equation (5)** says that each field that is of the type *corner*, can only be next to 2 *fields*.

**Equation (6)** says that each field that is of the type *edge*, can only be next to 3 *fields*.

**Equation (7)** says that each field that is the type *middle*, can only be next to 4 *fields*.

**Equation (8)** says that there exists a *breeze* next to each field that contains a *pit* and **Equation (9)** says that there exists a *stench* next to each field that contains a *wumpus*.

**Equation (10)** says that the *gold* is in the field where the *glitter* is.

**Equation (11) and (12)** say that *pit* and *wumpus* fields are dangerous fields.

**Equation (13)** say that all of the fields neighboring a field that contains a *pit*, contain a *breeze* and

**Equation (14)** say that all of the fields neighboring a field that contains a *wumpus*, contain a *stench*.

$$field \sqsubseteq \exists \leq 4 nextTo. \top \quad (1)$$

$$corner \sqsubseteq field \quad (2)$$

$$edge \sqsubseteq field \quad (3)$$

$$middle \sqsubseteq field \quad (4)$$

$$corner \sqsubseteq \exists \leq 2 nextTo. field \quad (5)$$

$$edge \sqsubseteq \exists \leq 3 nextTo. field \quad (6)$$

$$middle \sqsubseteq \exists \leq 4 nextTo. field \quad (7)$$

$$breeze \sqsubseteq \exists nextTo.pit \quad (8)$$

$$stench \sqsubseteq \exists nextTo.wumpus \quad (9)$$

$$glitter \sqsubseteq gold \quad (10)$$

$$pit \sqsubseteq danger \quad (11)$$

$$wumpus \sqsubseteq danger \quad (12)$$

$$pit \sqsubseteq \forall nextTo.breeze \quad (13)$$

$$wumpus \sqsubseteq \forall nextTo.stench \quad (14)$$

$$(\neg wumpus) \wedge (\neg pit) \sqsubseteq safe \quad (15)$$

$$safe \not\equiv danger \quad (16)$$

**Equation (15)** says that if a field doesn't contain a *pit* and *wumpus*, is *safe* and it's not dangerous.

**Equation (16)** says that a field is either *safe* or *danger*.

In order to test the TBox<sup>1</sup>, in the first step this TBox will be tested in the *RacerPorter* program by passing an arbitrary game board data to the program and checking the state of each field and its states by Racer reasoner.

Then TBox will be tested again by passing the logic mentioned above to the *RacerPorter* program and adding the info of each field step by step as the program algorithm should evolve in action.

For step one, the game board shown in figure (3) will be passed to the ***RacerPorter***. After loading the board into the *RacerPorter* program, the instances of *pit*, *wumpus*, *gold*, *danger* and *safe* concept will be retrieved as followings:

---

<sup>1</sup> The TBox discussed here could be also described in other ways with more terms and different concepts, but not as efficient as this one.

```

[1] ? (retrieve-concept-instances pit wumpus-family)
[1] > (feld7 feld4)
[2] ? (retrieve-concept-instances wumpus wumpus-family)
[2] > (feld15)
[3] ? (retrieve-concept-instances gold wumpus-family)
[3] > (feld11)
[4] ? (retrieve-concept-instances danger wumpus-family)
[4] > (feld15 feld7 feld4)
[5] ? (retrieve-concept-instances safe wumpus-family)
[6] > (feld16 feld14 feld13 feld12 feld11 feld10 feld9 feld8 feld6 feld5 feld3
      feld2 feld1)
[7] ? (tbox-coherent? wumpus)           [7] > t
[8] ? (abox-consistent? wumpus-family) [8] > t

```

As it is shown in the box above, the RacerPorter recognizes the instances of each concept

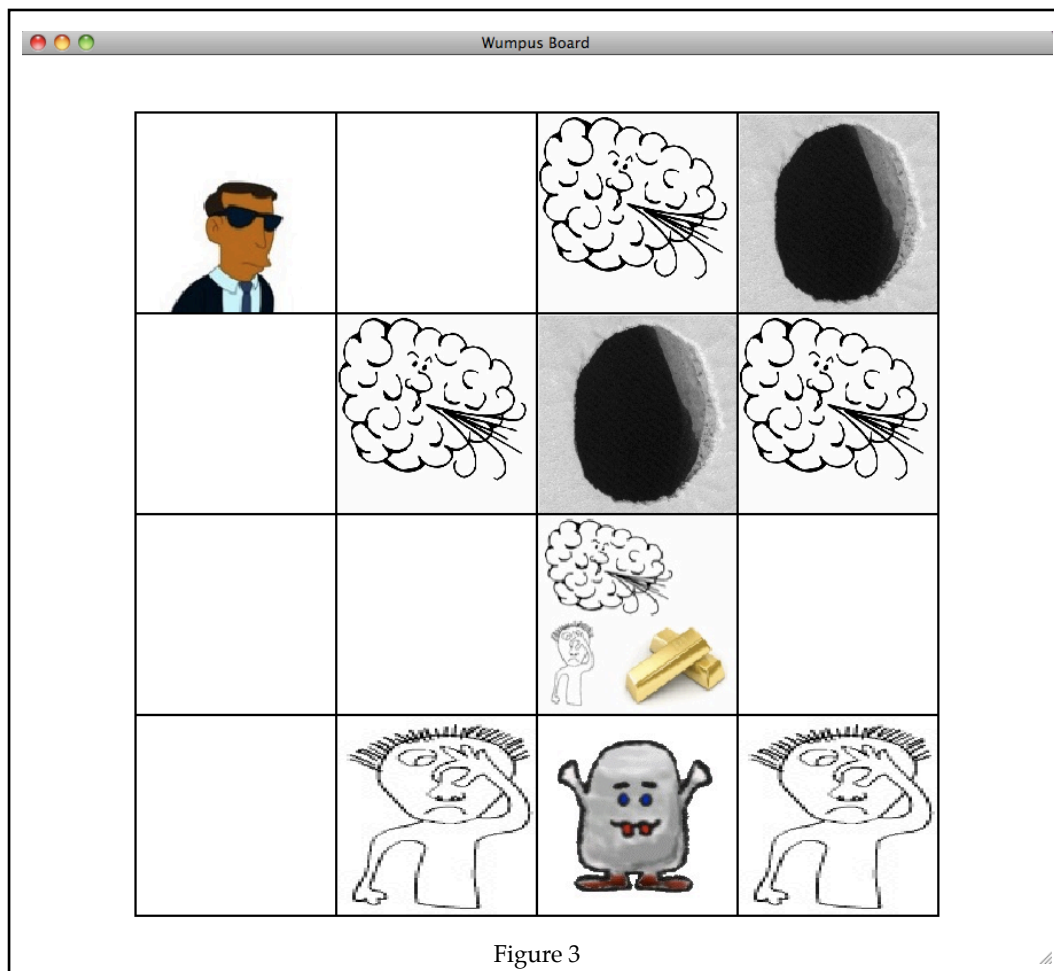


Figure 3

regarding to the example in figure (3) correctly. As it is shown, the TBox is coherent and the ABox regarding to the TBox is consistent.

In the next step, an empty board is going to be loaded to the *RacerPorter* and then the information about each field is going to be given to the program by hand, to observe the facts that the program recognizes, as it goes further, based on the logics of the program.

The board that has been chosen for this step, is a 4x4 board and the information about each field will be inserted to the program, regarding to figure 4 starting from field number 1.

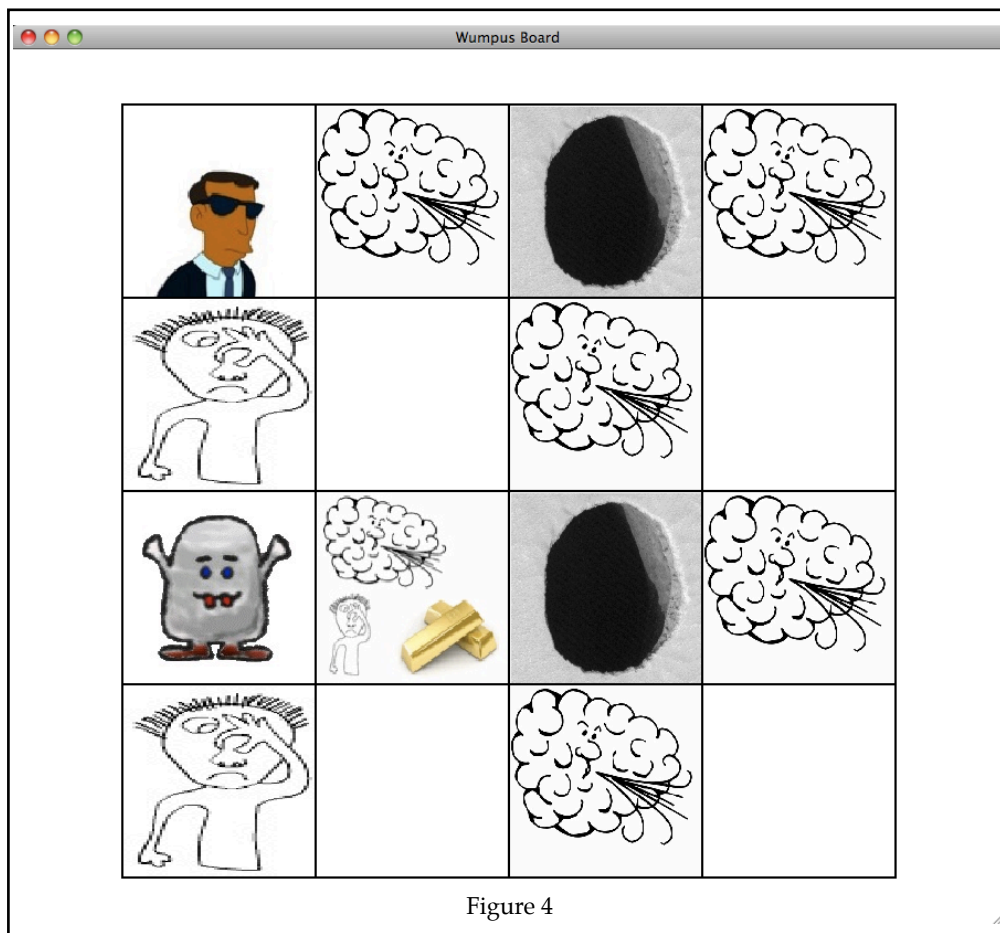


Figure 4

First the information about field number one which is the starting point will be entered. These field is neither an instance of *breeze* nor *stench* and also neither *pit* nor *wumpus* nor *glitter* by default. Since the information about field 1 indicates that the neighboring fields are neither *wumpus* nor *pit*, the program comes to the conclusion that fields number 1,2 and 5 are *safe*, so the agent is able to go to these fields. The next step is entering the information about the neighboring fields. The test starts first from field number 2. This field is an instance of *breeze* but not an instance of any other concepts. These information will be passed to the *RacerPorter*. In the next step the information about field number 5 will be passed. This field is an instance of *stench*, but no any other concepts. It will be observed as the data about field number 5 has been passed to the *RacerPorter*, that the program can quickly realize many information about the other neighboring fields which their data hasn't been passed to the *RacerPorter* yet, as seen in the box below:

```

[14] ? (retrieve-concept-instances pit wumpus-family)
[14] > (feld3)

[15] ? (retrieve-concept-instances wumpus wumpus-family)
[15] > (feld9)

[16] ? (retrieve-concept-instances safe wumpus-family)
[16] > (feld6 feld5 feld2 feld1)

[17] ? (retrieve-concept-instances danger wumpus-family)
[17] > (feld9 feld3)

[18] ? (abox-consistent? wumpus-family)
[18] > t

[19] ? (tbox-coherent? wumpus)
[19] > t

```

Since the only safe state identified so far ( not the ones their data are already in the RacerPorter program), is the field number 6, in the next step the information about this field is going to be passed to the RacerPorter. This field is neither an instance of *stench* nor *breeze*. Therefore new information about some new fields are going to be recognized as follows:

```

[24] ? (retrieve-concept-instances safe wumpus-family)
[24] > (feld10 feld7 feld6 feld5 feld2 feld1)

[25] ? (retrieve-concept-instances danger wumpus-family)
[25] > (feld9 feld3)

```

As it is seen above, with only some few information about a few number of fields in a few steps, most of the information about the game board is going to be gathered which is useful for the search algorithm to follow the right route to find a solution for the wumpus problem. The other advantage is that since the *safe* states remain stable during the search process, the route how the agent has to take to get back to the starting point will be identified easily through the *safe* fields that have been found so far. In the following chapter, the search algorithm will be discussed.

# Chapter 3

---

## Search Algorithm

---

Now that the TBox and the ABox based on the TBox have functioned correctly, an algorithm for finding a route from the starting point to the field that contains *gold* (if such route exists), should be found in order to solve the wumpus problem for an arbitrary game board.

Since the fields have been already categorized in 2 different categories (*safe* and *danger*), I thought of an algorithm where the agent starts the search from the starting point and moves and gathers information about the game board by identifying and entering the *safe* fields only, until it finds the *gold* in one of *safe* fields. (*gold* could not be in a *danger* field).

In the search algorithm designed in this work, after the agent enters a *safe* field, it checks for the wumpus state instances that are present in that field and adds the information to it's ABox and updates it's knowledge by identifying the states of it's neighboring fields. If the agent finds a neighboring field which is *safe*, the search will continue from the next *safe* field until the *gold* has been found. By default, the agent chooses to go to the upper(north) neighboring *safe* field first and continue the search from there, then from the left(west) neighboring *safe* field, then the right(east) neighboring *safe* field and at last the lower(south) neighboring *safe* field. It has to be mentioned that the agent doesn't necessarily visits all of these 4 neighboring fields, if existing, either because one of them is not *safe* or the agent has entered the current *safe* field from that *safe* field.

As it has been observed, the search algorithm is a recursive algorithm, where the agent starts the same algorithm by entering each new field until it finds the *gold* or comes to the conclusion that there is no *safe* path from the starting point to the field which contains *gold*, for example because of the *pits* and *wumpus*, blocking the way to the *gold* or simply because the agent was not able to gather enough information about the board in order to identify the *safe* fields.

Since there could be many different routes to reach the *gold*, the algorithm finds many possible routes from each field. In this work all of the possible routes will be identified and outputted as the agent progresses and each route will be shown separately on the game solution window.

In the following, the algorithm will be described regarded to the example in figure 4.

Since the starting point is always a *safe* field, the algorithm considers the neighboring fields which are number 2 and 5 as *safe*. Since the field in east has a higher priority than a field in south, the agent enters the field number 2, which is a *safe* field. After gathering the information about what the agent percepts in field number 2, the agent considers now the neighboring fields of the field number 2. Since this field has no north field, the next option is the west field, but because the agent has entered the field number 2 from its west field, this field will also be discarded. The next options are the east and south fields (field number 3 and 6), but since the agent is not able to make any judgments, if these 2 fields are *safe* (either one of them or both are *pits*, regarding the *breeze* perceived in field number 2), they will also be discarded.<sup>1</sup> Since there are no any other options left, the agent goes back to the pervious field which here will be field number 1 again.

---

<sup>1</sup> When it is not clear for the agent if a field is *safe* or not, the field will be regarded as passive *danger*, unless the agent has enough information to consider this field as a *safe* field. Therefore if the agent is not sure, whether or not a field is *safe*, it doesn't necessarily mean that the field is *dangerous*.



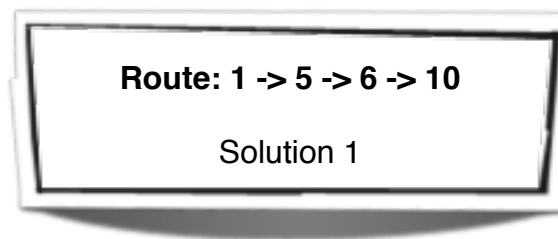
The next possible option from field number 1 is the field in the south which in this case is field number 5. When the agent gathers the information in field number 5, the position of the *wumpus* and *pit* which the *breeze* in field number 2 has been perceived, will become clear based on the TBox and therefore new *safe* fields( here field number 6,7 and 10) will be recognized.

Now the agent can continue its search from the next possible field which here will be field number 6.<sup>1</sup> Here the first option for the agent is going to field number 2, but in order to avoid loops in the search algorithm, which can lead the agent to a loop with no end, this option will be ignored. In the next step, the agent goes to field number 7.

In field number 7 a *breeze* can be perceived and as it was expected, since field number 3 is a *pit*, but it could be possible that the other neighboring fields (here field number 8 and 11) are also *pits* and since the agent does not know yet if these 2 fields are *safe* or not, it avoids going into them and therefore it returns to field number 6.

From here the agent chooses to go to the next *safe* field which is field number 10. After entering this field, the agent perceives a *glitter* which it means the *gold* is in this field.

Now that agent knows which route from starting point he has taken to get to field number 10, a solution has been found for this game board which its route is as follows:



The algorithm then proceeds by continuing the search from the pervious fields until all of the fields have been visited or identified as *safe* or *danger* by the algorithm.

In following I am going to discuss how do I implement the TBox and the algorithm discussed above using Java and Racer.

---

<sup>1</sup> It has to be mentioned that in each step the agent checks if a *glitter* could be perceived in the field that the agent is in there now, which if yes, it means that the *gold* has been found.

# Chapter 4

---

## Implementation Basics

---

In order to implement the search algorithm, I use the Java programming language. Therefore I use the Eclipse software which is a very comfortable and easy software to use, when implementing interfaces with Java.

Before starting with the implementation, the [Racer related classes\(JRacer\)](#) provided on [Racer homepage](#) have been added to the package that is going to include the classes and files of the program. These classes are listed on table 1.

Class	Description
<i>RacerClient</i>	A socket client that opens a socket to a Racer server,
<i>RacerSymbol</i>	Returns a Racer symbol value
<i>RacerKeyword</i>	Returns a Racer keyword value
<i>RacerList</i>	Implements the List for Racer
<i>RacerLiteral</i>	Abstracts class RacerLiteral that extends RacerResult
<i>RacerNull</i>	Returns RacerNull value
<i>RacerNumber</i>	Converts a Racer result to number
<i>RacerResult</i>	Converts a Racer result to string
<i>RacerString</i>	Returns a Racer string value
<i>RacerStubs</i>	Contains Racer commands
<i>RacerClientException</i>	Contains Racer Client Exception terms

Table 1

My implementation adds 7 more classes to the above package, in order to implement the search algorithm discussed before, solving the wumpus problem. These classes are listed in table 2.

The UML diagram of the classes implemented in this work is shown on page 20. The *General* class contains the *main function* where the instances of *Wumpus* and *Agent* are defined and combined with each other, in order to solve different Wumpus problems.

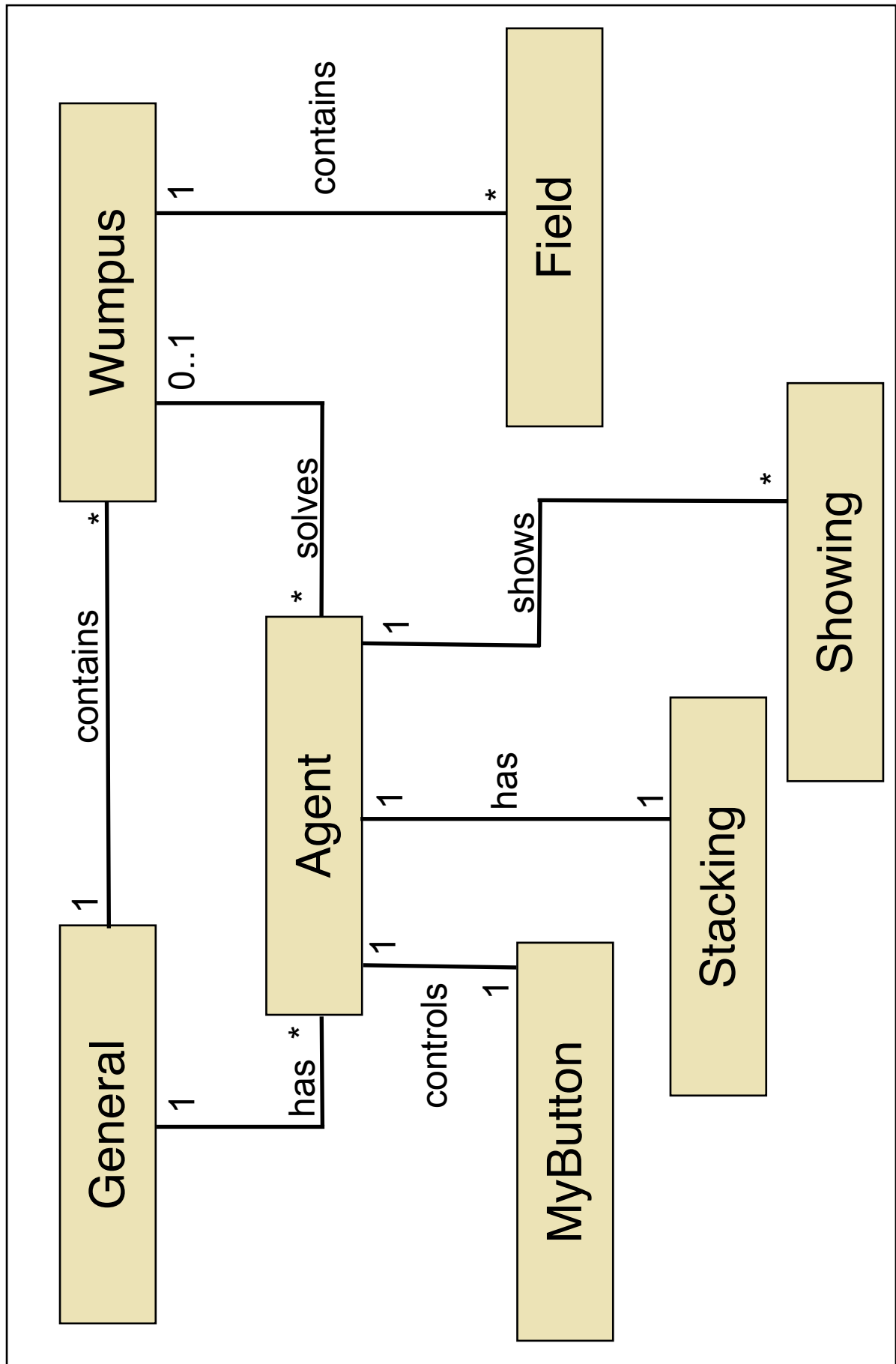
The Wumpus UML diagram indicates that each instance of the class *Agent* solves only one or no wumpus problems and an instance of the class *Wumpus* (a game board) can be solved by many different agents.

Class	Description
<i>General</i>	contains main function
<i>Agent</i>	racer and solution operations
<i>Wumpus</i>	contains field operations
<i>Field</i>	contains field specifications
<i>Showing</i>	controls graphical interface
<i>MyButton</i>	contains button operations
<i>Stacking</i>	contains stack operations

**Table 2**

From the diagram it could be seen that an instance of the class *Wumpus*, could have many *Field* instances, although it has to be mentioned that the number of the instances of the *Field* class related to the *Wumpus* instance should be a quadratic number, due to the quadratic size of the board.

Each instance of the class *Agent* has then one and only one button(an instance of the class *MyButton*) to control the agent's graphical interface, one instance of *Stacking* class, in order to save the route that the agent has taken so far for solving the wumpus problem and different instances of the class *Showing*, which are the graphical windows, that the agent shows it's observations and solutions on them. From here it will be also recognized then that each button (instance of the class *MyButton*), each instance of the *Stacking* class and also each instance of the *Showing* class can be related only to one instance of the class *Agent*.



# Chapter 5

---

## Implementation

---

## 5.1 General Class

This class contains the *main function* and is used to control different agents and wumpus boards in the program. In the main function many wumpus boards and agents could be defined and connected to each other (each agent to only a wumpus board). For example in order to create the board on figure 1, we use the following commands :

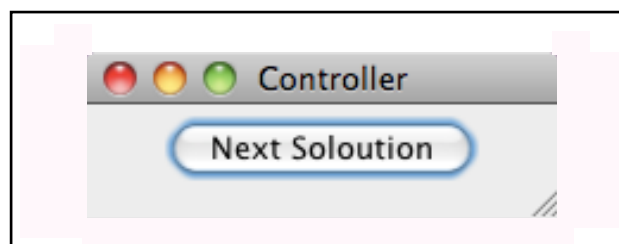
```
Wumpus w_MyBoard = new Wumpus(5,17,14,[4,15,21]);  
.....  
.....  
Agent a_MyAgent = new Agent(w_MyBoard);
```

Here an instance of *Wumpus* is defined by positioning randomly or by purpose the *gold*, *wumpus* and *pits* on the game board related to the wumpus problem. Then an instance of the class *Agent* should be defined and an instance of *Wumpus* should be connected to this agent, by passing the wumpus instance as the input parameter of the agent constructor.

Then the agent could be initialized and its *solve()* function could be called to solve the problem from the field, its number is given as the input parameter of the *solve()* function. At the end, the result of the search will be printed on the Eclipse console.

## 5.2 MyButton Class

This class is an extension of the *JPanel* class which implements the button as an action listener attached to each agent in order to control the results of the search process. By the last solution or by not founding any solutions, the button will disappear automatically. The button is shown below:



The main variable of this class is the following variable:

```
protected int i_Checker = 0;
```

This variable has the role of a semaphore. By default its value is set to zero. A zero value means that the button hasn't been clicked yet and therefore the program waits until this value is zero. By clicking the button, this value will be set to 1, which it signals to other classes (here the class *agent*) that the button has been clicked in order to go to the next solution.

## 5.3 Stacking Class

This class helps the program to have special stack operations in order use in the program, while searching for a solution. As the agent goes forward, it always inserts a new field on his route into the stack. The top of the stack is always the latest field on the route and the bottom of the stack is the field number 1, since the agent starts the search always from this field. When the search from a specific field fails, this field will be removed from the stack with the help of the function `pop()` and the search continues from the latest field on the route that the agent has taken so far.

In this work, an instance of this class is used in the class *agent* and defined for each instance of this class and will be only changed by the agent *class*. The *Showing* class can use this instance of the *Stacking* class, but can't change it.

The constructor of this class has a single integer input which is the size N of the board, so that a stack of the size NxN will be generated in order to be used by the agent. The constructor then initializes the 2 private integer variables of this class which one is *i\_Size*, which is the size of the board and the other one is *i\_Counter*, which is the counter of the stack and it always points to the position of the top of the stack. The private variable *al\_MyList* is then from the type *ArrayList*, which is the stack list itself.

Function	Description
<code>Stacking(int i_Num)</code>	Class constructor which its input is the size of the board
<code>int pop()</code>	Stack pop function which it returns the top of the stack
<code>int stackSize()</code>	Returns the size of the stack (position of the top of the stack)
<code>int stackMem(int i_StackMember)</code>	Returns the elements in the stack positioned at the input parameter
<code>void print()</code>	Prints the elements available in the stack
<code>void push(int i_El)</code>	Stack push function that inserts its input into the stack
<code>boolean hasT(int i_CheckElement)</code>	Checks if the input element is in the stack
<code>boolean isEmpty()</code>	Checks if the stack is empty

The functions implemented in this class could be seen in the table below:

The *print* function prints the stack elements into the Eclipse console when a solution has been founded in the order they have been inserted into the stack which it is actually the route that the agent has taken for the founded solution. The *pop()* function checks if the stack is empty and if not then returns the top of the stack and the *push()* function checks if the stack is not full before inserting a new element into it.

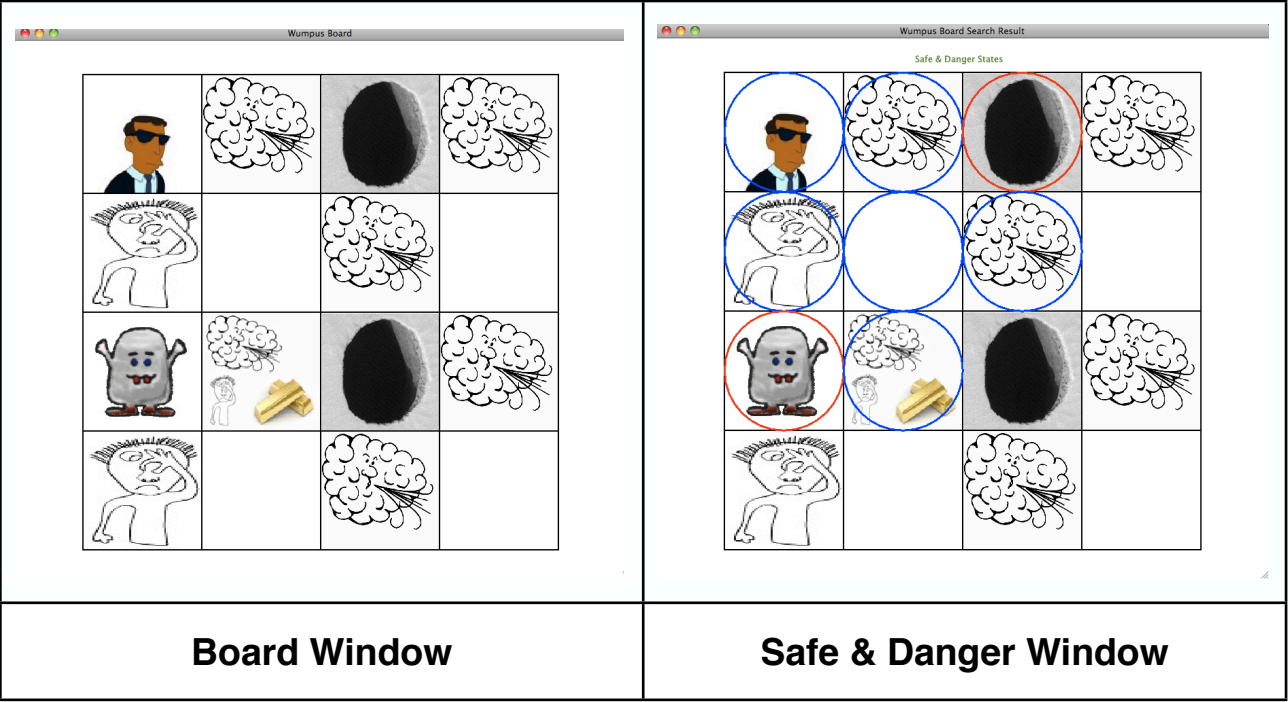


# 5.4 Showing Class

This class controls the graphical interface of the program. The programs has different types of graphical windows:

- 1. **Board Window**
- 2. **Safe & Danger Window**
- 3. **Solution Window**

The **Board Window** illustrates the wumpus game board related to the agent that tries to solve this board then. The **Safe & Danger Window** shows the safe and danger fields founded by the agent at the end of the search. As mentioned before the safe fields will be marked by blue circles and the danger fields will be marked by red circles. The fields that haven't been marked are the fields that either the agent hasn't been close enough to them or the agent hasn't enough information about these fields to categorized them under safe and danger categories. The **Solution Window** illustrates the solutions founded so far by the agent. The window is numbered by the number of the solution and the route from the start point to the field that contains *gold* is illustrated with a green line on the board and will be also printed on the window in green color. To go to the next solution, the button introduced in pervious section, should be clicked. If no solution was found, no **Solution Window** will be opened and the program goes directly to the end of the program and shows the **Safe and Danger Window**. An example for each windows is shown below and in the next page.





Location Variables	State Variables	
<code>int i_X</code>	<code>int i_Mode</code>	<code>boolean b_Pit</code>
<code>int i_Y</code>		<code>boolean b_Gold</code>
<code>int i_East</code>		<code>boolean b_Stench</code>
<code>int i_West</code>		<code>boolean b_Breeze</code>
<code>int i_North</code>		<code>boolean b_Glitter</code>
<code>int i_South</code>		<code>boolean b_Wumpus</code>
<code>int i_Number</code>		<code>boolean b_Current</code>
		<code>boolean b_Safe</code>
		<code>boolean b_Danger</code>

If a *Field* has any of the above states, its boolean value related to that state will be set.

The *i\_Mode* indicates which combination of the states are presented in a *Field*. To distinguish between different states, each state has been valued by a specific binary number as follows:

State	Binary
Breeze	1
Stench	2
Glitter	4
Pit	8
Wumpus	16

The *i\_Mode* variable will then be calculated by the following line code:

```
i_Mode= (1 *getInt(b_Breeze) + 2*getInt(b_Stench) + 4*getInt(b_Glitter)
        + 8*getInt(b_Pit) + 16*getInt(b_Wumpus))
```

For example if a field is a breeze and a pit, its *i\_Mode* variable will be set to 9 or if a field is a stench and breeze and glitter, its *i\_Mode* will be set to 7.

The functions in this class could be categorized again under the same 2 categories as its variables, whereas each category will be then divided into 2 sub-categories.

In the following these functions are illustrated:

Location Functions		State Functions	
Setting Functions	<code>void setNumber(int i_Number)</code>	Setting Functions	<code>void setBreeze(boolean b_X)</code>
	<code>void setEast(int i_Number)</code>		<code>void setStench(boolean b_X)</code>
	<code>void setNorth(int i_Number)</code>		<code>void setGlitter(boolean b_X)</code>
	<code>void setWest(int i_Number)</code>		<code>void setPit(boolean b_X)</code>
	<code>void setSouth(int i_Number)</code>		<code>void setWumpus(boolean b_X)</code>
	<code>void setNumber(int i_Number)</code>		<code>void setDanger(boolean b_X)</code>
	<code>int setX(int i_Num, int i_Size)</code>		<code>void setSafe(boolean b_X)</code>
	<code>int setY(int i_Num, int i_Size)</code>		<code>void setCurrent(boolean b_X)</code>
Check Functions	<code>boolean hasNorth()</code>	Check Functions	<code>void Gold(boolean b_X)</code>
	<code>boolean hasWest()</code>		<code>boolean isWumpus()</code>
	<code>boolean hasEast()</code>		<code>boolean isPit()</code>
	<code>boolean hasSouth()</code>		<code>boolean isGold()</code>
Query Functions	<code>int getNumber()</code>		<code>boolean isStench()</code>
	<code>int getX()</code>		<code>boolean isGlitter()</code>
	<code>int getY()</code>		<code>boolean isBreeze()</code>
	<code>int getNorth()</code>		<code>boolean isDanger()</code>
	<code>int getWest()</code>		<code>boolean isSafe()</code>
	<code>int getEast()</code>		<code>boolean isCurrent()</code>
	<code>int getSouth()</code>		

The inputs of the constructor function *Field*(*int* Num, *int* i\_SizeOfBoard) of this class are the size of the board, which this field is located in it and its position on the board. based on these 2 inputs, the constructor function then calculates the X-Y position of the field on the board and specifies its neighboring numbers, if existed.

The *void calculateMode()* function calculates the mode of the *Field* which it specifies the combination of the states in that field.

The *int getInt(boolean b\_Num)* converts a boolean value into integer by returning 1 for a true input and 0 for a false input.

## 5.6 Wumpus Class

In this class the structure of a wumpus game board will be implemented. An instance of the class *Wumpus* contains a collection(list) of instances from the class *Field*. The *Wumpus* class has 4 different type of constructor functions as listed below:

Nr.	Wumpus Class Constructor Functions
1	<i>Wumpus(int i_InputSize)</i>
2	<i>Wumpus(int i_InputSize, int gold)</i>
3	<i>Wumpus(int i_InputSize, int gold, int wumpus)</i>
4	<i>Wumpus(int i_InputSize, int gold, int wumpus, int[] Pits)</i>

Any instance of the *Wumpus* class needs to have an input which indicates the size of the game board. If nothing but the size of the board is given as the input, the *gold*, *wumpus* and *pits* will be positioned randomly. Therefore function 1 comes here into action. Additionally to the size, the position of the *gold* could be given as the second parameter to *Wumpus* class constructor and the rest will be generated automatically by the constructor function. Here function 2 comes into action. The same is valid for function 3, whereas here the position of the *wumpus* will be given as the third parameter. Finally by the forth constructor function the position of the *pits* will also be added to the board, where these positions are all given by a *ArrayList* of integer. Based on the size of the constructor (N), an *ArrayList* of the NxN instances of class *Field* will be generated, each instance related to a single field of the game board

If the position of the *gold*, *wumpus* and/or *pits* should be generated randomly, the constructor functions prevents positioning of them on field number 1, since this doesn't make any sense, if any of them is positioned on the starting point.

The next important function of this class is the `void createBoard()` function. After the positions of the *gold*, *wumpus* and *pits* have been set, based on these positions, this function then identifies the positions of **breezes**, **stenches** and **glitters** and initializes the states of the fields related to these 3 states. At the end, the `calculateMode()` function will be called for each instance of the class *Field* (which belongs to the *Wumpus*) in order specify its states code combination (`i_Mode` variable)

The `int getBoardSize()` returns the size of the board.

The *Field* `getField(int i_IDX)` returns the field indexed with `i_IDX`.

The `ArrayList<Field> getList()` returns the list of the wumpus fields.

## 5.7 Agent Class

The last class is the *Agent* class. This class is responsible for communication with Racer and for the searching algorithm and finally controlling the graphical windows and printing of the results.

The private variables of this class are listed as follows:

Variables	Description
<code>int i_Port = 8088</code>	Port number for communication with Racer
<code>int i_Size</code>	Size of the game board
<code>int i_SoloutinCounter</code>	Number of solutions founded
<code>String s_IP = "localhost"</code>	IP address of the Racer
<code>RacerClient rc_Racer</code>	Racer client
<code>Wumpus w_MyWumpus</code>	Wumpus board related to the agent
<code>Stacking st_MyStack</code>	Routing stack
<code>JFrame j_Frame</code>	Solution frame
<code>JApplet j_Applet</code>	Solution applet
<code>MyButton cp_newContentPane</code>	Controller button
<code>JFrame j_myFrame</code>	Button frame
<code>JFrame j_Board</code>	Board frame
<code>JApplet j_BoardApplet</code>	Board applet

The *Agent* constructor's input variable is the wumpus board which the agent has to solve. Based on the basic information of the wumpus board(size, ...) the agent initializes its own variables and creates a button to control the solution window. The agent constructor also establishes a connection to the **RacerPro** and logs in into it. At last it shows the pure game board on the board window.

The `void printResult()` function is responsible for printing the results, after the agent finishes the search by sending queries to the **RacerPro** and printing the results on the Eclipse console. An example is shown below:

```
System.out.println("PIT    : +rc_Racer.retrieveConceptInstances
                    (\"pit\", \"wumpus-family\")");
```

The same command could be used for the other wumpus states. This function also illustrates the result of the entire search algorithm (Safe & Danger Fields) when the agent is finished with its search on a new graphical frame window called Safe & Danger window.

The `void mInitialize()` is responsible for the basic settings of the wumpus TBox and ABox in **RacerPro**. At the beginning, the concepts defined in the TBox will be defined in the **RacerPro** and then the TBox commands will be passed to the **RacerPro** as follows:

```
rc_Racer.impliesM("field", "(at-most 4 nextTo *top*)");
rc_Racer.impliesM("corner", "field");
rc_Racer.impliesM("edge", "field");
rc_Racer.impliesM("middle", "field");
rc_Racer.impliesM("corner", "(at-most 2 nextTo)");
rc_Racer.impliesM("edge", "(at-most 3 nextTo)");
rc_Racer.impliesM("middle", "(at-most 4 nextTo)");
rc_Racer.impliesM("breeze", "(some nextTo pit)");
rc_Racer.impliesM("stench", "(some nextTo wumpus)");
rc_Racer.impliesM("glitter", "gold");
rc_Racer.impliesM("pit", "danger");
rc_Racer.impliesM("wumpus", "danger");
rc_Racer.impliesM("pit", "(all nextTo breeze)");
rc_Racer.impliesM("wumpus", "(all nextTo stench)");
rc_Racer.impliesM("(and (not wumpus) (not pit))", "safe");
rc_Racer.conceptDisjointP("safe", "danger", "wumpus");
```

After that based on the position of each field, each field will be marked either as *corner*, *edge* or *middle* and will be passed as an instance of these concepts to the **RacerPro**. At the end based on the size of the board, the *nextTo* relations between the fields will be determined and passed to the **RacerPro**.

The most important function in this class is the `int solve(int i_IDX)` which implements the search algorithm discussed in chapter 3 and therefore is the most important function by the implementation of this work. Th input parameter of this function is the position of the field, where

the functions starts the search from this field. As the function precedes and enters a new field (*safe* field), the solve function will be called recursively to continue the search from this field.

By entering a new *field*, the field number will be added to the route stack and the function checks then, if this field is an instance of *breeze* or not and also if it is an instance of *stench* or not and then adds this data to the **RacerPro**. Then it checks, if this *field* is an instance of *glitter* or not and adds this information to the **RacerPro** too and then based on the TBox and ABox, makes a query to **RacerPro**, if the field contains the *glitter*.

If the *glitter* query is true, it means that the *gold* is in this field and a new solution has been found for the wumpus problem. Then it will be printed on the Eclipse console in which field the *gold* has been found and then the number of solutions will be incremented and the number of the new solution and its related route will be printed on the Eclipse console. Then the route and the result will be shown on the graphical result window. The search function stops here, till the control button is not pressed, which it means the button has the role of a semaphore and the agent waits for its signal before continuing the search from other fields again.

If the *glitter* query is false and the *gold* is not in this field, the search function checks, if this field has a field on its north, west, east and south side. If any field founded on these positions, the function checks then, if the founded field is already on the route stack. If yes, it ignores it and if it is not on the stack, it checks if this field is *safe*. If the field is not *safe*, the function ignores it and goes to the next neighboring field and if it is *safe*, the solve function will be called recursively for this field and the search continues from this field. The order and priority that the agent takes is as follows:

1. North
2. West
3. East
4. South

At the end of the search from a specific field, the field number will be removed from the stack and the functions returns to its call position.



# Chapter 6

---

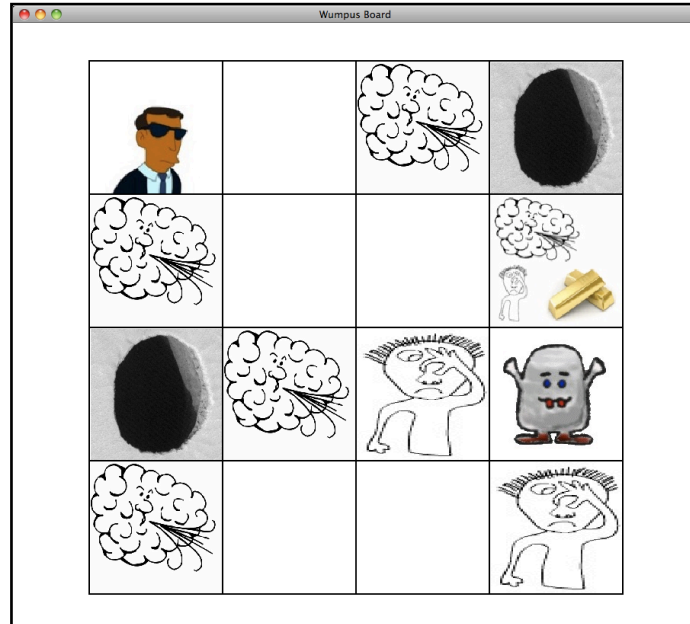
## Examples

---

In this chapter some examples will be shown:

## Example 6.1

In this example a 4x4 game board is shown. The *gold* is positioned at field number 8, with 2 *pits* positioned at fields number 4 and 9 and a *wumpus* at field number 12. The results of the program are as follows:



```
Wumpus GENERATED : 12
Gold GENERATED : 8
PIT GENERATED : 4
PIT GENERATED : 9
```

Solving:

```
Found Gold in Feld8
Soloution 1 -- Rout: 1 2 6 7 8
```

```
Found Gold in Feld8
Soloution 2 -- Rout: 1 2 6 10 11 7 8
```

```
Found Gold in Feld8
Soloution 3 -- Rout: 1 5 6 2 3 7 8
```

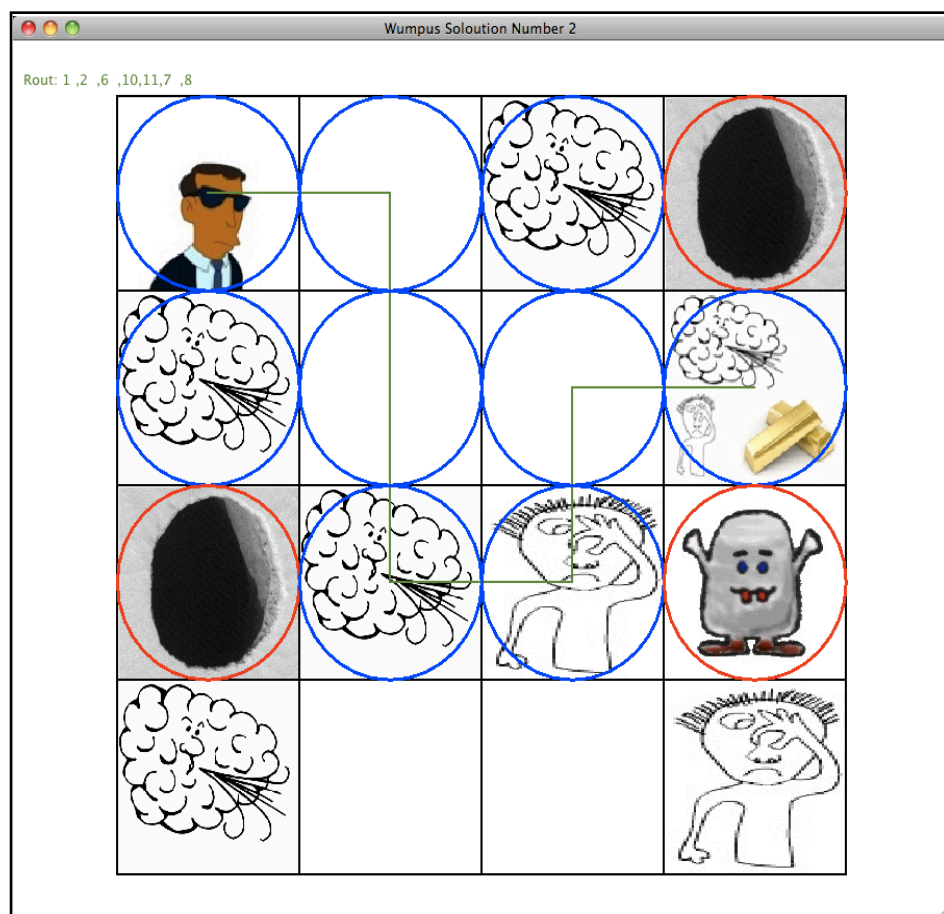
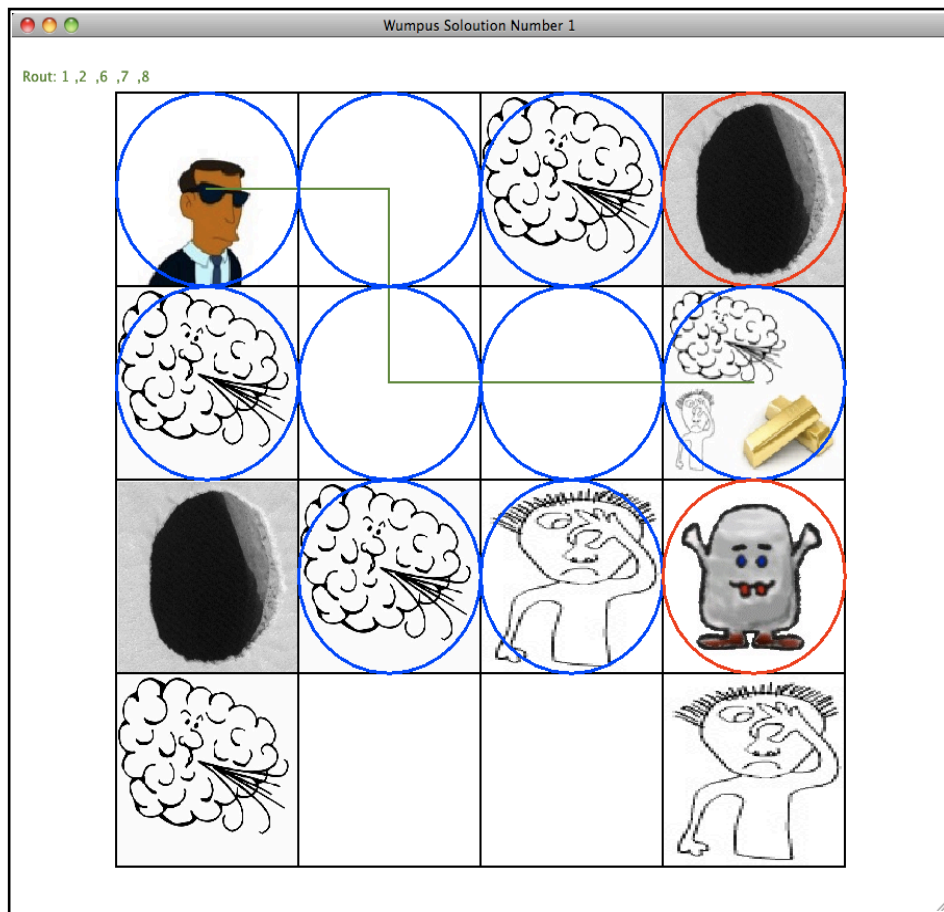
```
Found Gold in Feld8
Soloution 4 -- Rout: 1 5 6 7 8
```

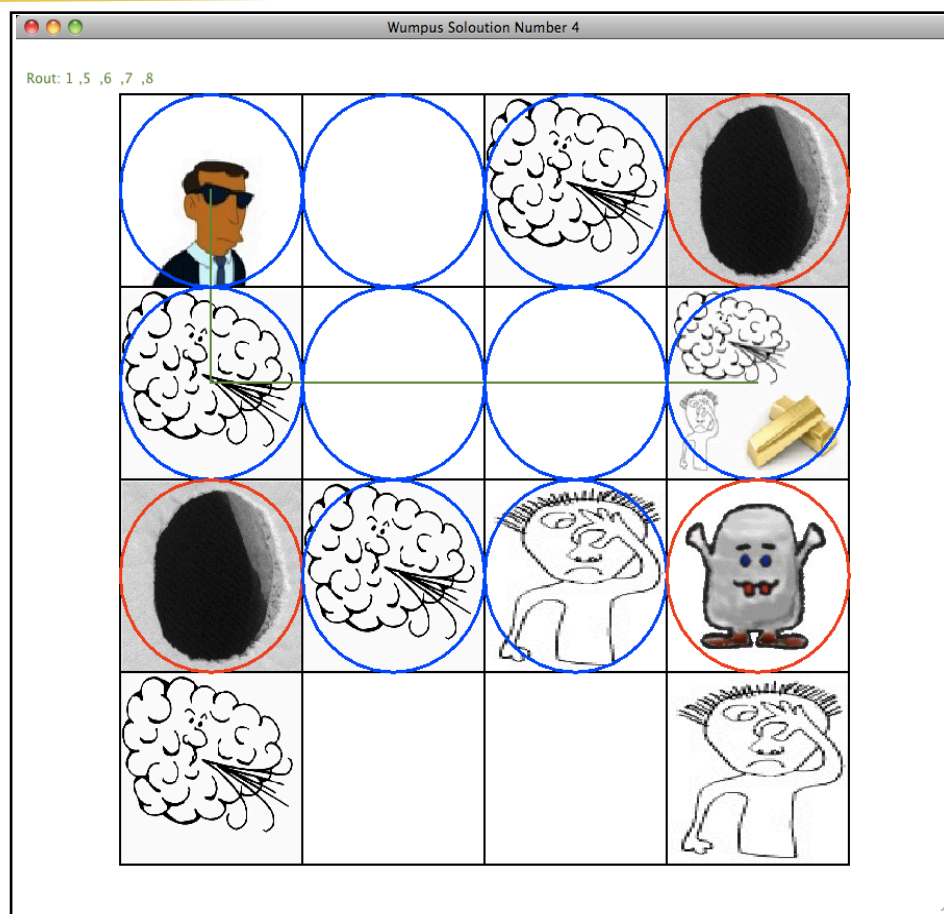
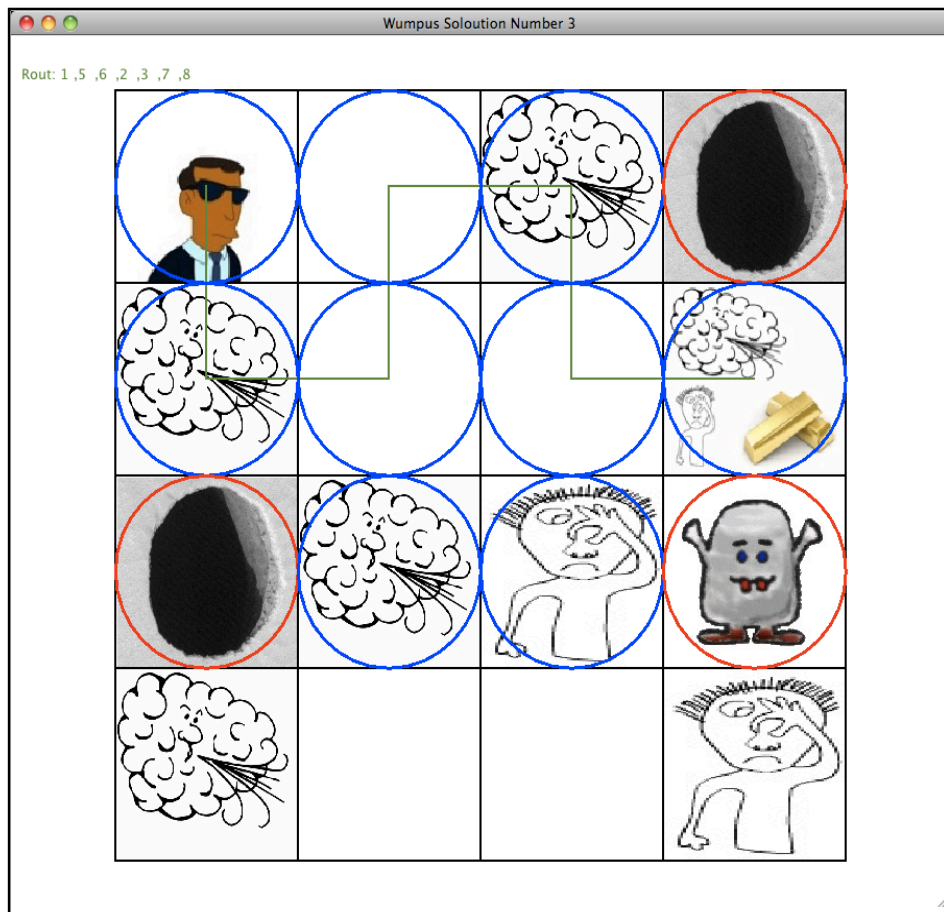
```
Found Gold in Feld8
Soloution 5 -- Rout: 1 5 6 10 11 7 8
```

Results:

```
Danger: (feld9 feld4 feld12)
Safe : (feld1 feld10 feld11 feld2 feld3 feld5 feld6 feld7 feld8)
Breeze: (feld10 feld8 feld5 feld3 feld13)
Stench: (feld11 feld8 feld16)
Glitter: (feld8)
PIT : (feld4 feld9)
Wumpus: (feld12)
Gold: (feld8)
```

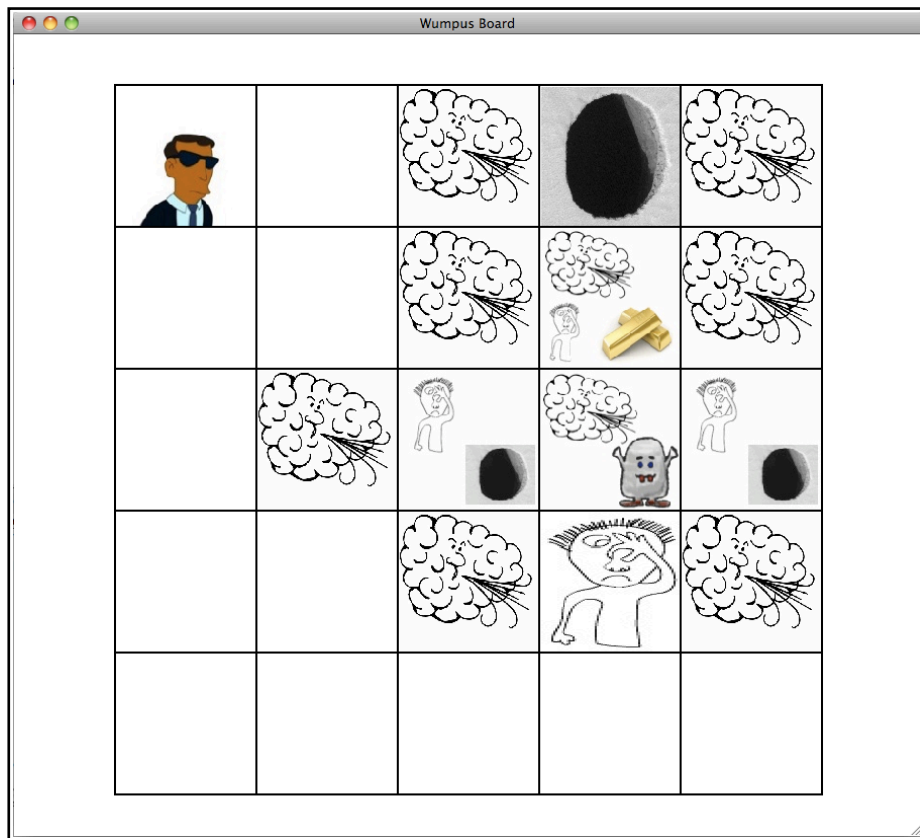
Number of soloutions found: 5







In this example a 5x5 game board is shown. The program can make no judgment, if the field which contains *gold*, is *safe* or not and therefore, it won't go in it and can't therefore find the *gold* on the board.



Wumpus GENERATED : 14  
Gold GENERATED : 9  
PIT GENERATED : 4  
PIT GENERATED : 13  
PIT GENERATED : 15

Solving:

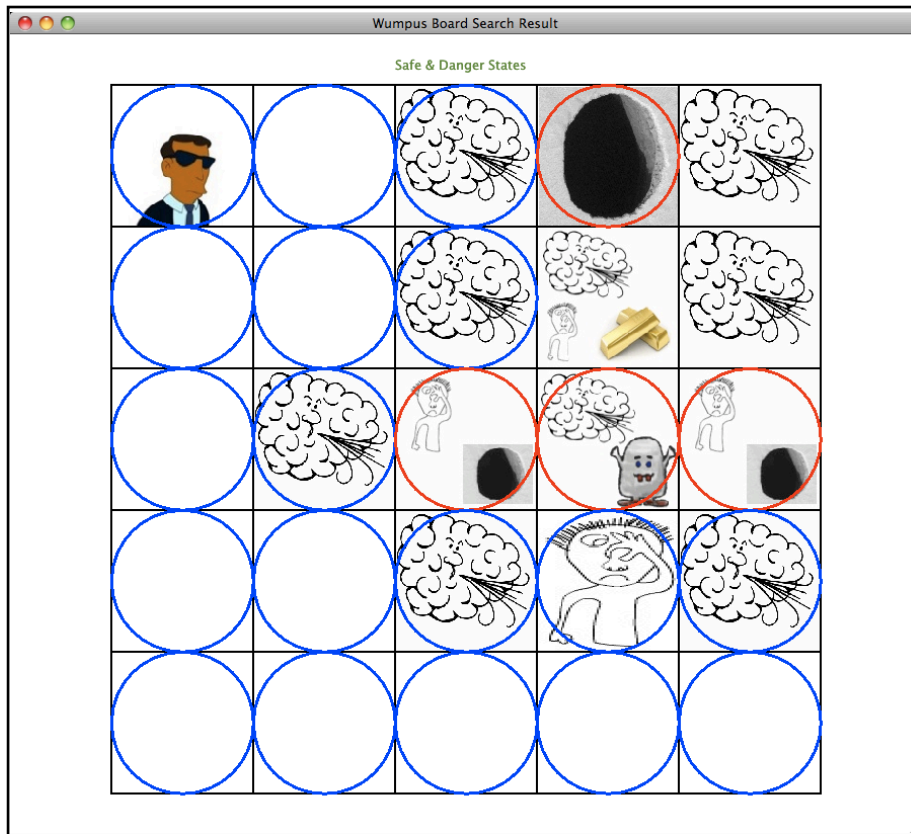
```
Taxonomy: ((top nil (breeze danger field gold safe stench)) (breeze (top) (bottom)) (corner (field)
(bottom)) (danger (top) (pit wumpus)) (edge (field) (bottom)) (field (top) (corner edge middle))
(glitter (gold) (bottom)) (gold (top) (glitter)) (middle (field) (bottom)) (pit (danger) (bottom))
(safe (top) (bottom)) (stench (top) (bottom)) (wumpus (danger) (bottom)) (bottom (breeze corner edge
glitter middle pit safe stench wumpus) nil))
```

Results:

```
Danger: (feld4 feld15 feld14 feld13)
Safe : (feld1 feld11 feld12 feld16 feld17 feld18 feld19 feld2 feld20 feld21 feld22 feld23 feld24
feld25 feld3 feld6 feld7 feld8)
Breeze: (feld20 feld18 feld12 feld8 feld3 feld5 feld10 feld9 feld14)
Stench: (feld19 feld15 feld13 feld9)
Glitter: nil
PIT : (feld4 feld15 feld13)
Wumpus: (feld14)
Gold: nil
```

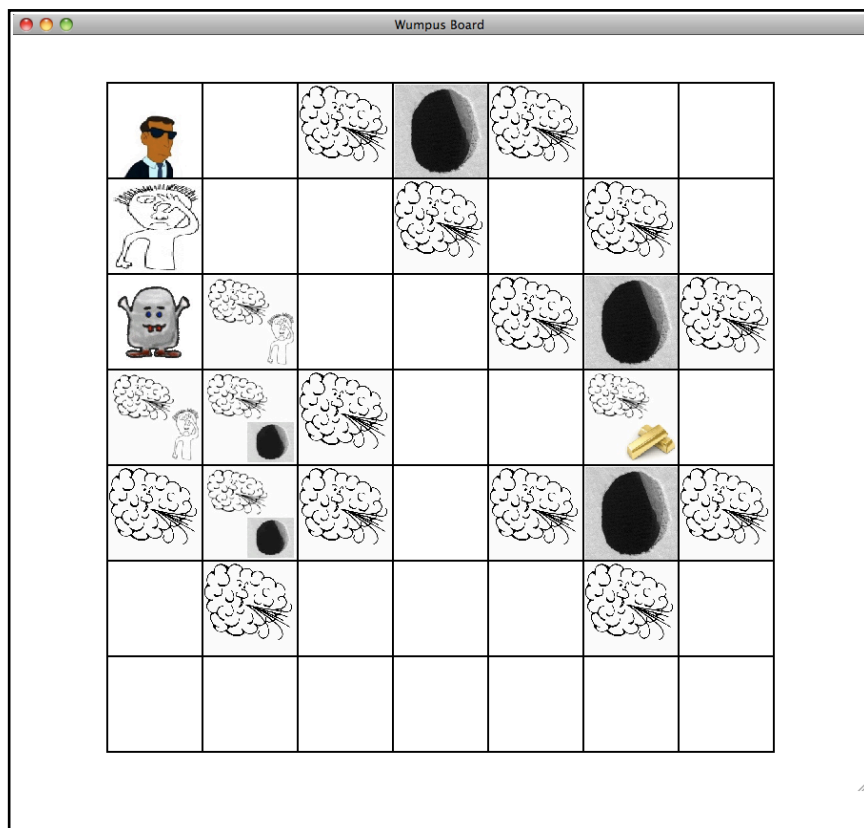
Number of solutions found: 0





### Example 6.3

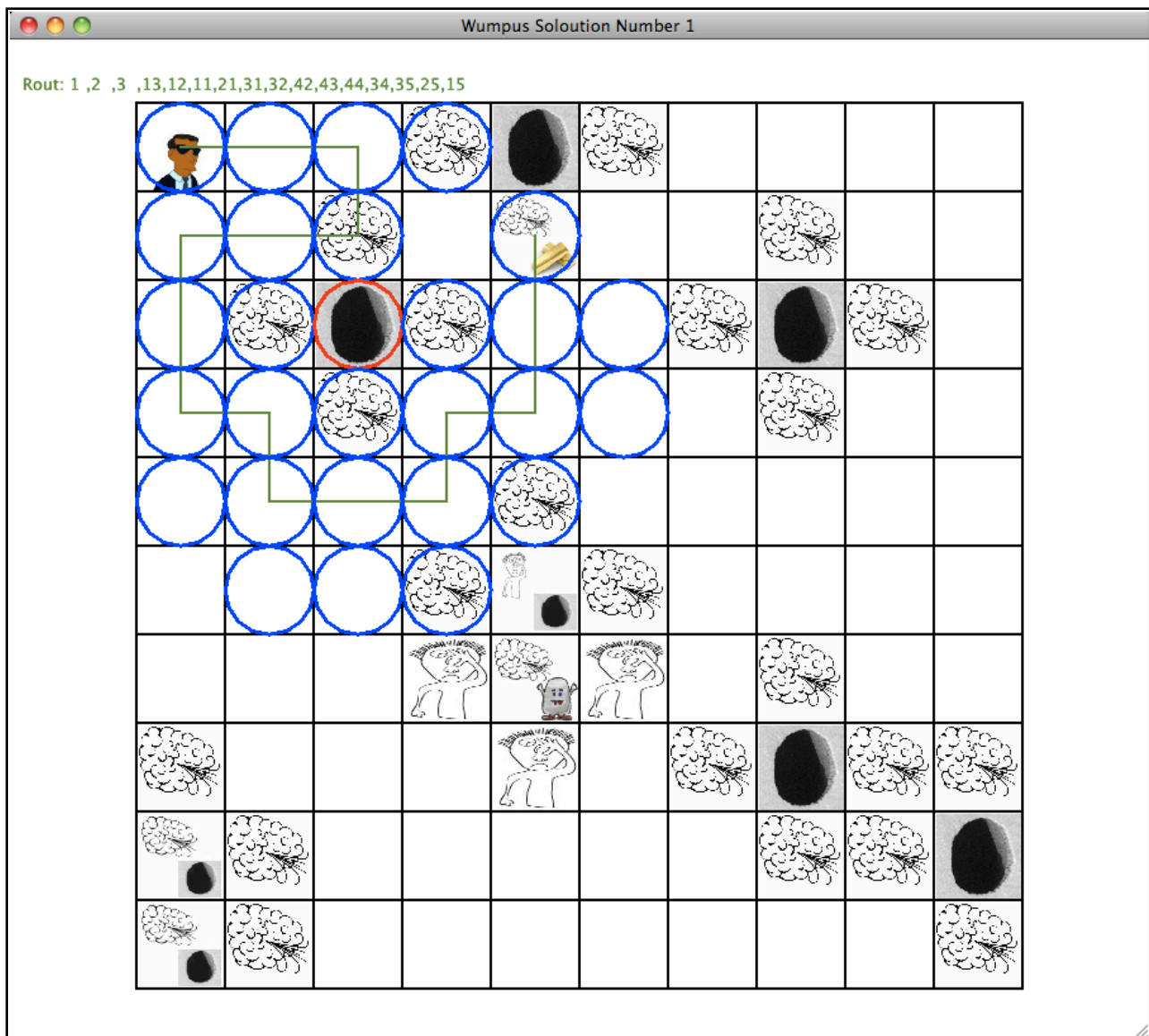
In this example a 7x7 game board is shown with some of the solutions found for this example.





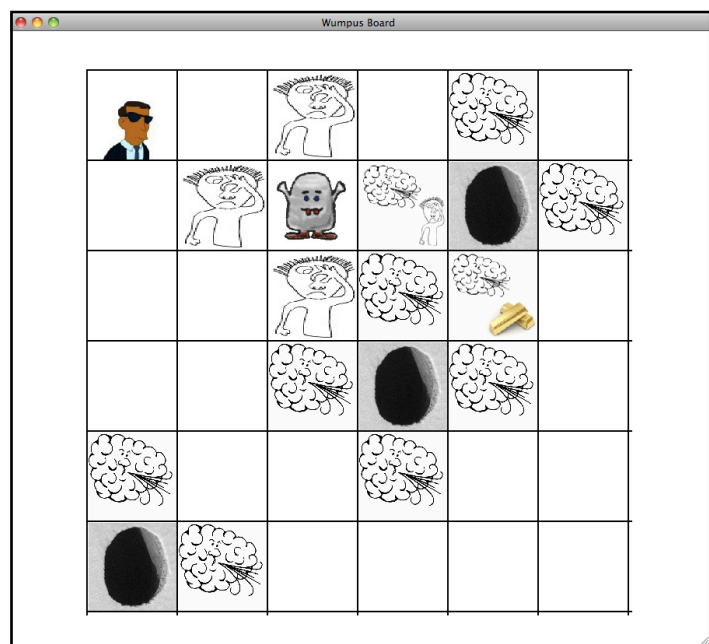






## Example 6.5

For this example the program has founded **884** solutions.



# Chapter 7

---

## Evaluation

---

In this chapter the evaluation of the algorithm will be discussed.

## 7.1 Size Evaluation

It is obvious that in order to evaluate the algorithm, it has to be considered that the time the algorithm needs to solve a game board depends on the:

- Size of the game board
- Position of the gold, wumpus and pits and the distance between them

The time that the algorithm needs to solve a wumpus problem increases by the increase of the size of the board. In the following two different evaluation will be shown. First the time and number of solution founded for different boards with the same size and then the average duration that the algorithm approximately needs, in order to solve game board each different board size.

```
18 Solutions ---- The process took approximately: 4.17075      seconds
0 Solutions ---- The process took approximately: 0.55011594    seconds
11 Solutions ---- The process took approximately: 2.254913      seconds
0 Solutions ---- The process took approximately: 0.372992      seconds
14 Solutions ---- The process took approximately: 2.356473      seconds
11 Solutions ---- The process took approximately: 2.3729122     seconds
0 Solutions ---- The process took approximately: 0.245242      seconds
0 Solutions ---- The process took approximately: 3.949567      seconds
0 Solutions ---- The process took approximately: 1.593254      seconds
6 Solutions ---- The process took approximately: 1.788405      seconds
9 Solutions ---- The process took approximately: 1.58146       seconds
0 Solutions ---- The process took approximately: 1.123217      seconds
...
...
```

### Approximately time to solve different 4x4 game boards

```
0 Solutions ---- The process took approximately: 3.075479      seconds
0 Solutions ---- The process took approximately: 0.604698      seconds
0 Solutions ---- The process took approximately: 0.423785      seconds
30 Solutions ---- The process took approximately: 11.933191     seconds
28 Solutions ---- The process took approximately: 50.311344     seconds
0 Solutions ---- The process took approximately: 0.69356406     seconds
0 Solutions ---- The process took approximately: 0.340978      seconds
12 Solutions ---- The process took approximately: 4.711925     seconds
77 Solutions ---- The process took approximately: 78.9322      seconds
9 Solutions ---- The process took approximately: 0.828152      seconds
0 Solutions ---- The process took approximately: 1.1631111     seconds
0 Solutions ---- The process took approximately: 2.8967628     seconds
6 Solutions ---- The process took approximately: 2.117228      seconds
0 Solutions ---- The process took approximately: 0.47151002     seconds
...
...
```

### Approximately time to solve different 5x5 game boards

```

23 Solutions ---- The process took approximately: 3.4381 seconds
0 Solutions ---- The process took approximately: 2.34 seconds
11 Solutions ---- The process took approximately: 5.7345 seconds
0 Solutions ---- The process took approximately: 0.24211 seconds
4 Solutions ---- The process took approximately: 2.32467 seconds
0 Solutions ---- The process took approximately: 12.24284 seconds
6 Solutions ---- The process took approximately: 5.6445 seconds
0 Solutions ---- The process took approximately: 0.5355 seconds
14 Solutions ---- The process took approximately: 2.6642 seconds
91 Solutions ---- The process took approximately: 456.439 seconds
0 Solutions ---- The process took approximately: 1.884 seconds
...
...

```

### Approximately time to solve different 6x6 game boards

## 7.2 General Evaluation

Another way to show the quality of the algorithm is to show the average time that the algorithm needs to solve a wumpus board for different sizes. The table below shows the average time for some board sizes:

Size	Time(sec)
4	1.8632
5	2.845
6	3.705742

# Chapter 8

---

## Conclusion

---

## 8.1 Summary

As discussed in the pervious chapter, it has been shown that the wumpus worlds can be solved with description logics and with the help of the axioms we defined in the TBox and the algorithm that I have designed, it was possible to get different solutions for different boards, if any solutions for them existed.

By the evaluation it was observed that the time the algorithm needs to solve the problem depends on the size of the board and the position of the elements specially *gold* on the board. For some problems the algorithm could not find any solution, due to the fact that the gold is surrounded by *danger* elements.

The only difficulty that I had during this work was first, understanding the open world assumption and matching it to the logic defined in the TBox. the other problem was after updating the OS of my MAC computer to Snow Leopard where as the result some inconsistency between Java and Racer occurred which I have solved this problem by a simple *System* command in Java.

## 8.2 Outlook

The next step developing this work could be changing the board games from a static board to dynamic, so that as the agent moves through the board, the position of the *gold*, *wumpus* and *pits* also change dynamically.

An another option is developing 2 agents which both try to find the *gold* and the agent that find the *gold* quicker, is then the winner.

Another option is defining many agents for each board elements, which each tries to make its win maximum and the win of other agents minimum.

# Chapter 9




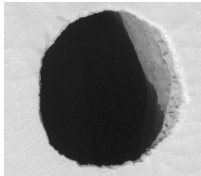


---

## Definitions

---



## Boards Signs

Symbol	Description
	Gold
	Breeze
	Stench
	Pit
	Wumpus
	Agent

# Chapter 10

---

## References

---

1. **The Description Logic Handbook : Theory, Implementation, and Application,**  
**Baader, Franz (c 2007)**
2. **Script of the lecture “Application Logics”, Prof. Dr. Ralf Möller**
3. **[RacerPro Reference Manual, Version 1.9.2, Racer Systems GmbH & Co. KG](#)**
4. **[RacerPro User's Guide Version 1.9.2, Racer Systems GmbH & Co. KG](#)**

Special thanks to:

Prof. Dr. Ralf Möller  
Sebastian Wandelt

---