Na tomto místě bude oficiální zadání vaší práce

- Toto zadání je podepsané děkanem a vedoucím katedry,
- musíte si ho vyzvednout na studiijním oddělení Katedry počítačů na Karlově náměstí,
- v jedné odevzdané práci bude originál tohoto zadání (originál zůstává po obhajobě na katedře),
- ve druhé bude na stejném místě neověřená kopie tohoto dokumentu (tato se vám vrátí po obhajobě).

Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering

Master's Thesis

Framework Squander Usage

Bc. Martin Kožený

Supervisor: Ing. Jiří Daněček

Study Programme: Electrical Engineering and Information Technology

Field of Study: Computer Science and Engineering

November 18, 2011

Aknowledgements

At this point I would like to thank supervisor of my thesis Mr Ing. Jiří Daněček and author of Squander framework Mr Aleksandar Milicevic for valuable comments and advices to this work.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.
I have no objection to usage of this work in compliance with the act §60 Zákon č. 121/2000Sb. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague, 30. 9. 2011 ....

Abstract

The aim of this work is to describe and study framework Squander developed on Massachusetts Institute of Technology by Mr. Aleksandar Milicevic. This framework brings into language Java another way of programming, which can improve effectivness of implementing and computation performance of the program.

Work shows how was this framework used for implementing set of algorithms, especially NP graph algorithms, and compares that implementation with common imperative way of programming.

Abstrakt

Abstrakt práce by měl velmi stručně vystihovat její podstatu. Tedy čím se práce zabývá a co je jejím výsledkem/přínosem.
Očekávají se cca 1 – 2 odstavce, maximálně půl stránky.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As was said in abstract, the main purpose of this thesis is to describe and use framework Squander developed at Massachusetts Institute of Technology (MIT) by Mr Aleksandar Milicevic. This framework brings to Java declarative constructs, which are useful for implementing programs that involve computations that are relatively easy to specify but hard to solve algorithmically. In such cases is better to use declarative constraints to naturally express the core of the computation, whereas imperative code is natural choice to read input parameters and setting up data structures for the computation. This is big advantage of this framework, that programmer can smoothly switch between declarative logical and imperative programming.

By ability of mixing imperative and declarative code can programmer easily express constraints of problem in terms of existing data structures and objects on the heap. Despite having overhead of encoding and decoding, it is surprising how is competitive Squander's SAT-based solution with specialized heuristic developed for concrete problem.

In first chapter I am going to describe basic principles of logical programming using language Prolog and in following one meaning of annotations, which are used to express logical constructions in framework Squander. Last chapters are devoted to framework itself and its comparison to common imperativ way of programming.

Chapter 2

Logical programming principles

The most known logical programming in the world is definitely Prolog and thereby we can describe properties of logical programming using this language. His name is derived from term PROgramming in LOGic and was developed for programming of symbolic computation. His success led to formation new discipline in mathematical information technology - **logical programming**.

Logical programming focus on description of relation's properties without need to know how to do that.

2.1 Logical programming paradigm

According to [5], logical programming differ from imperative languages in following points:

1. no assignment statement
2. no cycles, no branching
3. no flow control
4. object is marked as *variable*, which satisfies some set of conditions, that are being during computation more specified

Logical programming is based on following concepts:

1. declaring facts about objects and relations between objects
2. declaring valid rules about objects and relations between thmeselves and computing queries

In logical programming are *facts* unconditional commands and *rules* are conditional commands. Facts and rules are stored in one shared database. Language does not differ between program and data.

2.1.1 Facts

As is presented in [3], for expressing *facts* and *rules* are used *clauses.* Facts are used for expressing unconditionally true assertions and are clauses with defined headers, but with no body. Usual way to ilustrate how to composed facts are family relationships.

```
parent(david, john).
parent(john, jane).
parent(ann, jane).
parent(john, richard).
parent(ann, richard).
man(john).
man(richard).
womam(ann).
woman(jane).
```

Every clause declares concrete fact about relation. We can see that relation `parent`, e.g. `parent(john, jane).` is concrete *instance* of this relation for *objects* `john` and `jane`. After declaring those facts is possible to form queries concerning relation `parent`.

```
?-parent(john, jane).
```

Example above shows, how it is possible to ask, if John is parent to Jane. Because language has this fact recorded in its environment and answer is:

```
yes
```

Similarly can be query constructed on non-existing fact:

```
?-parent(john, emily).

no
```

Answer to this query is `no` of course. Query can be also composed in a way, that we want to get some object, which is with other object in required relation.

```
?-parent(X, jane).

X = john
```

Here it is also possible to get other possible solution, so we get one more positive answer:

```
X = ann
```

After that are all possible answers exhausted, so for the next command we get:

```
no
```

Now we try to express little bit complicated query: who is mother of Jane. This query is necessary construct from two suqueries. First we limit set of solution to Jane's parents. For that purpose we use query already shown above:

```
?-parent(X, jane).
```

In variable `X` is now stored every object, who has relation `parent` to object `jane`. In our case `john` and `ann`. Now we have to limit this set of results to object, which is declared in clause `women`, so we add:

```
?-parent(X, jane), women(X).
```

As a result of these clauses we get:

```
X = ann
```

and nothing more. In framework Squander are as facts used objects, that are declared in framework's rules (see subsection 2.1.2). There are no facts declared explicitly so we do not devote to them any more.

2.1.2 Rules

Expressing knowledge by facts cannot be always effective. Complexity of relationships in family expressed by unconditional commands would lead to big expansion of database. Despite having unlimited memory available, searching for relevant information would has been time consuming. For this reason Prolog provides conditional expressions - *rules*.

By investigating facts is possible to derived new rule based on logical or factual context. That knowledge allow us to express facts, which are not explicitly stored in database. Let us show it in following example:

```
mother(X, Y) :- parent(X, Y),woman(X).
```

Left side of rule expressed so called **head of rule** and right side **body of rule**. This rule express, that `X` is mother of `Y`. Rule is only labeled generalization of last example in previous subsection 2.1.1. In next example will be shown more complicated construct:

```
brother(X, Y) :- parent(O, Y),parent(O, X),man(X).
```

Expression above means, that `X` is brother of `Y` if exists at least one object `O`, which is common in relation `parent` for both `X` and `Y` assuming `X` is a man. Mathematical interpretation of teh rule is:

„For all arbitrary persons X and Y,
if some of the parents of X is O
and some of the parents of Y is O
and person X is a man,
then X is brother of Y.“

When calling this rule with following parameters:

```
brother(richard, Y).

Y = jane
```

Rules are main construct of framework Squander, because they defined state of object before and after computation, declares which object or object's properties can be modified etc. On the other hand these rules are not called as it is in Prolog, but they are declared as metadata for handling with objects. More about Squander's rules in chapter 4.

2.2 Terminology

As presents Kolář [4], when simplyfying in Prolog, we can say, that in every task appear *objects* and *relations*. Name of objects are called *terms* and name of relations are called *predicates*. Terms are analogous to arithmetic expressions, which point to the computed value, and predicates are analogous to name of procedures, which defines relationship between input parameters and output values, in imperative programming language.

There are two types of terms: *simple terms* consisting of constants (e.g. `ann`, `richard`,...) and *compound terms*. Compound terms are also called *structures* is every term containing simple or compound term.

In previous subsection 2.1.2, there are predicates `parent`, `man` and `woman` as names of three relations defined by program. Predicate with name `parent` is defined as a set: `{(john,jane),(ann,jane),(john,richard),(ann,richard)}`. Next predicate with name `man` is defined as a set: `{john,richard}` and finally predicate with name `woman` is defined as a set: `{ann,jane}`.

Finally there are three types of *formulas* in Prolog:

- *atomic* - basic formulas (e.g. `parent(john, jane).`, `parent(ann, jane).`)
- *conditional command* - implication constructs $A : -P_1, P_2, \ldots, P_n$ where $P_1, P_2, \ldots, P_n$ are atomic formulas (e.g. `brother(X, Y) :- parent(O, Y),parent(O, X),man(X).`)
- *target clauses* - query type ($? - C_1, C_2, \ldots, C_n$ where $C_1, C_2, \ldots, C_n$ are targets) (e.g. `?-parent(X, jane), women(X).`)

2.3 Evaluation in Prolog

Main difference between **procedural semantics** and **declarative semantics** is shown on clause below:

```
P :- Q, R.
```

where `P` and `Q` are arbitrary forms of terms. We can read this clause from declarative point of view:

- `P` is true, if `Q` and `R` are true
- from validity of `Q` and `R` follow `P`

From procedural point of view has clause different meaning:

- to solve problem `P`, it is necessary to solve **first** problem `Q` and **then** problem `R`
- to fulfill target `P`, it is necessary **first** to fulfil target `Q` and then fulfil target `R`

`Evaluation` of sequence of targets $G_1, G_2, ..., G_m$ in Prolog consists of following steps [4]:

- If is sequence of targets empty, evaluation **succeded**.
- For non-empty sequence of targets, operation *SEARCHING* is invoked.
- *SEARCHING*: Clauses of the program from top to bottom are searched till first clause C occurence, whose head is successfuly unified with target G_1. If such clause is not found, evaluation ended unsuccessfuly.

  If is found clause C in form

  $$H : -B_1, B_2, \ldots, B_n$$

  then all its variables are renamed such, that new form C' of clause C, which has no varibles in common with targets $G_1, G_2, ..., G_m$. C' has form:

  $$H' : -B'_1, B'_2, \ldots, B'_n$$

  G and H are unified by substition S. In sequence of targets is target G_1 replaced by body of clause C' such, that new seqence has form:

  $$B'_1, B'_2, \ldots, B'_n, G_2, \ldots, G_m$$

  If is C fact, then $n = 0$ and sequence of targets is shortened to $m - 1$ targets.

  Then substituion S is done in order to make new list of targets, so new form is:

  $$B''_1, B''_2, \ldots, B''_n, G'_2, \ldots, G'_m$$

By recursive invoication procedure `Evaluate` is evaulated this sequence of targets. If this evaluation ends successfuly, previous evaluation is treated as also successful. If this evaluation does not end successfuly, last sequence of targets is left and it is made return to operation *SEARCHING*, where is continued immediately behind clause C in order to find some next usable clause.

Let us show evaulation in following rules, which used facts declared in subsection 2.1.1:

```
ancestor(X, Y) :- parent(X, Y).
ancestor(X, Y) :- parent(X, Z), parent(Z, Y).
```

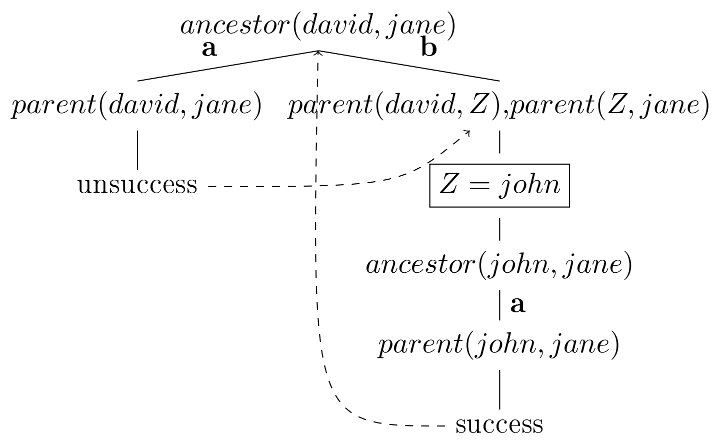


Figure 2.1: Expression evaluation in Prolog

First of the queries is answered unsuccessfuly when applied rule **a**, rule **b** is then successful.

Chapter 3

Meaning of annotations

3.1 Introduction

Annotations are tags that programmer insert into source code so that they can be processed by tools. The Java compiler understands a couple of annotations, but to go any further, you need to build your own processing tool or obtain a tool from a third party. Most common use of annotations are [2]:

- Automatic generation of auxiliary files, such as deployment descriptors or bean information classes.
- Automatic generation of code for testing, logging, transaction semantics, and so on.

In EJB 3.0 are annotations used in such sense, that a lot of repetitive code is automated by annotations.

3.2 Annotations as metadata

Metadata are data about data. In context of computer program, metadata are data about the code. Since Java 5.0 release, programmer can insert arbitrary data into his source code [6]. Annotations in Java is used like an *modifier*, placed before annotated item (it is a keyword similar to `public` or `static`). Annotations are used to annotate classes, fields or local variables and can be processed by tools that read them.

Each annotation is declared by annotation interface correspond to the element of the annotation.

Annotations do not directly affect program semantics, but they do affect the way programs are treated by tools and libraries, which can in turn affect the semantics of the running program. Annotations can be read from source files, class files, or reflectively at runtime.

3.3 Examples of use

3.3.1 Field validation

Annotation type declarations are similar to normal interface declarations. An at-sign (`@`) precedes the `interface` keyword. Each method declaration defines an *element* of the annotation type. Method declarations must not have any parameters or a `throws` clause. Return types are restricted to primitives, `String`, `Class`, `enums`, annotations, and arrays of the preceding types. Methods can have default values. Here is an example annotation type declaration:

```java
@Target(ElementType.FIELD)
public @interface Length {
    int max();
    int min();
}
```

Listing 3.1: Length annotation

Annotations can be annotatied itself. Such annotations are called *meta-annotations*. E.g. using (`@Target(ElementType.FIELD)`) indicates, that annotation type should be used to annotate only field declarations. This simple annotation is typically used for entity field e.g. of type `String` for permitted length of this field.

Once an annotation type is defined, you can use it to annotate declarations. An annotation is a special kind of modifier, and can be used anywhere that other modifiers (such as `public`, `static`, or `final`) can be used. By convention, annotations precede other modifiers. Annotations consist of an at-sign (`@`) followed by an annotation type and a parenthesized list of element-value pairs. The values must be compile-time constants. Here is a filed declaration with an annotation corresponding to the annotation type declared above:

```java
public class Role {
  @Length(max = 50)
  String name;

  @Length(max = 150)
  String description;
}
```

Listing 3.2: Class Role

Annotation type without any elements is called *marker* annotation type, e.g.:

```java
public @interface Entity{ }
```

Listing 3.3: Entity annotation

We can join these two types of annotation together in following example:

```java
@Entity
public class Role {
  @Length(max = 50)
  String name;

  @Length(max = 150)
  String description;
}
```

Listing 3.4: Entity Role

Another type of meta-annotation (`@Target(ElementType.METHOD)`) is presented below and indicates using annotation only for method declarations:

```java
@Target(ElementType.METHOD)
public @interface Test { }
```

Listing 3.5: Test annotation

In following example, there is updated entity shown above with tool for testing the annotations:

```java
@Entity
public class Role {
  @Length(max = 50)
  String name;

  @Length(max = 150)
  String description;

  @Test
  public void showMeTheFunny() {
    System.out.println("Here you have funny");
  }

  public void foo() {
    System.out.println("Foo");
  }

  public void bar() {
    System.out.println("Bar");
  }
}
```

Listing 3.6: Extended class Role

```java
public class RunTests {
   public static void main(String[] args) throws Exception {
      int passed = 0, failed = 0;
      for (Method m : Class.forName(args[0]).getMethods()) {
         if (m.isAnnotationPresent(Test.class)) {
            try {
               m.invoke(null);
                passed++;
            } catch (Throwable ex) {
                System.out.printf("Test %s failed: %s %n", m, ex.getCause());
                failed++;
            }
         }


      }
      System.out.printf("Passed: %d, Failed %d%n", passed, failed);

   }
}
```

Listing 3.7: Test of method annotation

```java
public class TestRole {
  public void test(Role role) throws Exception {
    for (Field f : Role.class.getFields()) {
      Annotation[] annotations = f.getDeclaredAnnotations();
      for (Annotation a : annotations) {
        if (a.annotationType().equals(Length.class)) {
          if (f.get(role) instanceof String && ((String)f.get(role)).length()
              > ((Length) a).max())
            throw new Exception(
                "Unacceptable length of field "+f.getName()+" of class Role");
        }
        ...
      }
    }
  }
}
```

Listing 3.8: Test of entity Role annotation

3.3.2 Annotating Event Handlers

In this subsection is shown example presented in [2]. When programmer have to use in his code action listeners, it leads to write a lot of repetitive code. Listeners are usually declared:

```java
myButton.addActionListener(new
  ActionListener()
  {
    public void actionPerformed(ActionEvent event)
    {
      doSomething();
    }
});
```

Listing 3.9: Action listener

Writing this boring code can be omitted by using annotations. Annotation will have form:

```java
@ActionListenerFor(source="myButton") void doSomething() { . . . }
```

Listing 3.10: Annotation for performing action

Instead of calling action listener is every method tagged with annotation.

```java
public class ButtonTest {
  public static void main(String[] args) {
    ButtonFrame frame = new ButtonFrame();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
  }
}
/**
 * A frame with a button panel
 */
class ButtonFrame extends JFrame {
  public ButtonFrame() {
    setTitle("ButtonTest");
    setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

    panel = new JPanel();
    add(panel);

    // create buttons
    yellowButton = new JButton("Yellow");
    blueButton = new JButton("Blue");
    redButton = new JButton("Red");

    // add buttons to panel
    panel.add(yellowButton);
    panel.add(blueButton);
    panel.add(redButton);

    ActionListenerInstaller.processAnnotations(this);
  }

  @ActionListenerFor(source = "yellowButton")
  public void yellowBackground() {
    panel.setBackground(Color.YELLOW);
  }

  @ActionListenerFor(source = "blueButton")
  public void blueBackground() {
    panel.setBackground(Color.BLUE);
  }

  @ActionListenerFor(source = "redButton")
  public void redBackground() {
    panel.setBackground(Color.RED);
  }

  public static final int DEFAULT_WIDTH = 300;
  public static final int DEFAULT_HEIGHT = 200;

  private JPanel panel;
  private JButton yellowButton;
  private JButton blueButton;
  private JButton redButton;
}
```

Listing 3.11: Use annotation for performing action

```
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface ActionListenerFor
{
  String source();
}
```

Listing 3.12: Declaration of annotation for performing action

We now need a mechanism to analyze them and install action listeners. That is the job of the `ActionListenerInstaller` class. The `ButtonFrame` constructor calls

```
ActionListenerInstaller.processAnnotations(this);
```

Listing 3.13: Action Installer invocation

The static `processAnnotations` method enumerates all methods of the object that it received. For each method, it gets the `ActionListenerFor` annotation object and processes it.

Here, we use the getAnnotation method that is defined in the `AnnotatedElement` interface. The classes `Method`, `Constructor`, `Field`, `Class`, and `Package` implement this interface. The name of the source field is stored in the annotation object. We retrieve it by calling the `source` method, and then look up the matching field.

```
String fieldName = a.source();
Field f = cl.getDeclaredField(fieldName);
```

Listing 3.14: Looking up for field

For each annotated method, we construct a proxy object that implements the `ActionListener` interface and whose `actionPerformed` method calls the annotated method. The details are not important. The key observation is that the functionality of the annotations was established by the `processAnnotations` method. In Example 3.15, the annotations were processed at run time. It would also have been possible to process them at the source level. A source code generator might have produced the code for adding the listeners. Alternatively, the annotations might have been processed at the bytecode level. A bytecode editor might have injected the calls to `addActionListener` into the frame constructor.

```java
public class ActionListenerInstaller {
  /**
   * Processes all ActionListenerFor annotations in the given object.
   *
   * @param obj
   *            an object whose methods may have ActionListenerFor annotations
   */
  public static void processAnnotations(Object obj) {
    try {
      Class cl = obj.getClass();
      for (Method m : cl.getDeclaredMethods()) {
        ActionListenerFor a = m.getAnnotation(ActionListenerFor.class);
        if (a != null) {
          Field f = cl.getDeclaredField(a.source());
          f.setAccessible(true);
          addListener(f.get(obj), obj, m);
        }
      }
    } catch (Exception e) {
      e.printStackTrace();
    }
  }

  /**
   * Adds an action listener that calls a given method.
   *
   * @param source
   *            the event source to which an action listener is added
   * @param param
   *            the implicit parameter of the method that the listener calls
   * @param m
   *            the method that the listener calls
   */
  public static void addListener(Object source, final Object param,
      final Method m) throws NoSuchMethodException,
      IllegalAccessException, InvocationTargetException {
    InvocationHandler handler = new InvocationHandler() {
      public Object invoke(Object proxy, Method mm, Object[] args)
          throws Throwable {
        return m.invoke(param);
      }
    };
    Object listener = Proxy.newProxyInstance(null,
        new Class[] { java.awt.event.ActionListener.class }, handler);
    Method adder = source.getClass().getMethod("addActionListener",
        ActionListener.class);
    adder.invoke(source, listener);
  }
}
```

Listing 3.15: Action Listener Installer class

Chapter 4

Framework description

4.1 Introduction

4.1.1 Overview

Squander is a framework providing unified environment for both declarative constraints and imperative statements in single program. This is very practical when implementing problems, which are easy to define but difficult to solve them algorithmically. In such cases, declarative constraints can be natural way to express problem, whereas imprative code is used for setting up the problem and data manipulation.

Thanks to ability of mixing imperative code with executable declarations, it is possible to express problem in terms of existing data structures and then run framework solver, which according to given constraints update the heap to reflect the solution.

Without having technology like this one, programmer has to translate his program to external solver, then run the solver and then again manually translate the solution back to the native programming language.

4.1.2 Architecture

We can divide running of Squander in following steps:

1. serialize heap into relations
2. translate specs and heap relations into Kodkod
3. translate relational into boolean logic
4. (if a solution is found) restore relations from boolean assignments (if a solution is found) restore field values from relations
5. (if a solution is found) restore the heap to reflect the solution

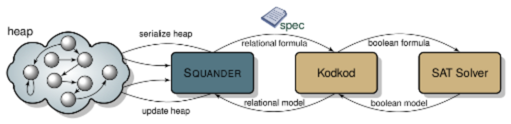


Figure 4.1: Architecture diagram

Further will be explained in deatil all of these steps and terms like Kodkod etc.

4.1.3 Applications

In this section will be briefly described applications of framework. Most typical applications are these:

- **solving hard constraint problems** - puzzles (solitaire, sudoku, n-queens,...), graph problems (traveling salesman problem, Hamiltonian path, general bisection breadth,...), schedulers, dependency managers,...
- **test input generation** - e.g. generate data structure instances that satisfy complex constraints
- **specification validation** - specifications can also contain errors, and the most intuitive way to test a specification would be to execute it on some concrete input and see if the result makes sense or not

- **runtime assertion checking** - check whether a given rich property holds at an arbitrary point during the execution of a program

On the other hand, framework has some limitations:

- **boundedness** - everything has to be bounded $\rightarrow$ framework cannot generate an arbitrary set of new objects (which may be needed to satisfy a specification); instead, the exact number of new objects of each class must be specified by the user
- **small integers** - integers must also be bounded to a small bitwidth (to make the solving tractable), which can occasionally cause subtle integer overflow bugs, which are typically hard to find
- **equality issues** - referential equality is used by default for all classes except for `String`, so it is impossible to write a spec that asserts that two objects are equal in the sense of Java `equals`
- **lack of support for higher-order expressions** - it is not possible to write a specification that says „find a path in the graph such that there is no other path in the graph longer than it“ and solve it with Squander; it is possible, however, to express and solve „find a path in the graph with at least k nodes“, which is computationally as hard as the previous problem, because a binary search can be used to efficiently find the maximum k for which a solution exists

4.2 Execution of Squander

Information about framework presented in this section are taken from [1], where is thre background of Squander described in detail.

4.2.1 Kodkod

Kodkod is a solver for relational logic. Kodkod requires bounded universe , a set of untyped relations, bounds for every relation and relation formula. Then translates given problem into boolean satisfiability problem (SAT) and applies of-the- shelf SAT solverto search for satysfiyng solution, which is reflected back if found.

When are the relations in Kodkod created, they are untyped, meaning that every relation can potentionally contain any tuple drawn from the finite universe. Actual set of tuples, which relation actually may contains is defined through Kodkod *bounds*:

- *lower bound* to define tuples, that relation **must** contain.
- *upper bounds* to define tuples, which relation **may** contain.

The size of these bounds has a big influence on search time - the fewer tuples are in the difference of lower and upper bound, the smaller search space is, the faster solving is.

4.2.2 JFSL

JSFL is formal lightweight specification for Java supporting relational and set algebra, as well as common Java operators. Using expressive power of relation algebra, JFSL makes it easy to succinctly and formally specify complex properties about Java programs such as method pre and postcondition, class invariants and so called *frame conditions*, which means portion of the heap, that is methos allow to modify. It also supports *specification fields* which can be useful for specifying abstract data types.

4.2.2.1 JFSL expressions

JFSL expressions are evaluated into relations. JFSL provides common algebra operators, together with interger and boolean operators. Some of them are shown in Table 4.1 and quantified operators, which are also provided by the framework, are shown in Table 4.2.

Operator	Description
`#`	set cardinality
`no`	„no“ multiplicity (empty)
`lone`	„lone“ mulitiplicity (zero or one)
`one`	„one“ multiplicity (exactly one)
`some`	„some“ multiplicity (one or more)
`!`	boolean negation
`-`	integer negation
`sum`	integer summation
`=>`	boolean implication
`<=>`	boolean equivalence („if and only if“)
`?`	if-then-else ternary operator (as in Java)
`@+`	relational union
`@-`	relational difference
`@&`	relational intersection

Table 4.1: Unary expressions supported by JFSL

Operator	Description	Usage
`all`	universal quantifier	`all x: T | P(x)`
`some`	existential quantifier	`some x: T | P(x)`
`sum`	integer summation	`sum i: int | a[i]`
`union`	set comprehension	`{x: T | P(x)}`

Table 4.2: Quantified expressions supported by JFSL

4.2.2.2 JFSL annotations

JFSL annotations are written as Java annotations and are for interaction between framework and programmer. Main components (JFSL annotations) are:

- `@Invariant` - this annotaion is attached to the classes and used to define, that condition must be fulfilled for the given class. That means, that this condition has to be fulfilled before and after execution.
- `@Requires` - this annotation specifies constraints before method invocation. The method is expected to execute correctly if only the precondition is satisfied immediately before method invocation. Class invariants are added immediately to the method preconditions.

- `@Ensures` - attached to the methods and used to specify constraints on the state, which have to be satisfied after method invocation. It means, that it captures all effect that method is expected to produce. Class invariants are implicitly added to method postconditions.
- `@Modifies` - this annotation is attached to the methods to specify frame conditions. Frame condition can hold up 4 different pieces of specification (syntax: `@Modifies ("f [s][l][u]")`). First, and the only one mandatory piece, is the name of modifiable field, `f`. It is optionally followed by *instance selector* `s`, followed by *lower bound* and the *upper bound*. Instance selector specifies instances, for which the field may be modified - if not specified, it is assumed „all". The lower bound contains concrete field values for some objects in the post-state. The upper bound holds the possible field values in the post-state.
- `@SpecField` - attached to the classes and used to define specification fields. Definition of type specification consists of type declaration, and optionally an abstract function. the abstraction function defines, how the field value is computed in terms of other fields. For example `@SpecField("x: one int| x = this.y - this.z")` defines a singleton integer field `x`, the value of which must be equal to the difference of `y` and `z`. Specifications fields are inherited from super-types and sub-types can override the abstraction function (by simply redefining it), a feature that is particularly useful for specifying abstract datatypes, such as Java collections.

In context of these specification, the goal is to execute a method, that makes modification of the portion of the heap (specified in frame condition), so that final state satisfies postcondition.

4.2.3 From object heap to relational logic

Execution of Squander begins in client's code when invoking method `Squander.exe()` involving following steps:

1. Assembling all revelant constraints from annotations, as well as class annotations corresponding to all invariant classes.
2. Construction relations representing objects and their fields in pre-state and adding additional relations for holding their values in the post-state, along with their Kodkod bounds
3. Parsing constraints and converting them to a single relational formula
4. If solution is found, translation of the Kodkod result objects into updates of Java heap state, by modification of the object fields

4.2.3.1 Traversing the heap

For discovering reachable portion of the heap, bread-first search algorithm is used, which started from the set of root objects and repeatedly visiting all children until all reachable objects have been visited. The most important part is how to enumerate childre, i.e. how to serialize a given object into a set of field values.

4.2.4 Relations and bounds

After traversing the heap, founding all reachable objects and discovered all classes/-fields reffered in the specification, we have enough information to construct the relation, that represent state of the heap.

The translation does not used all fields, but only those, whose are considered as *relevant fields* - those, who are explicitly mentioned in the specification (Squander's annotation). Similarly, not all reachable objects are needed; only objects reachable by following the relevant fields are included in the translation. These objects will be referred to as *literals*.

First we define a finite universe consisting of all literals, plus integers within the bound. For every literal, a unary relation is created.

For each Java type, one could either create a new relation (with appropriate bounds so that it contains the known literals), or one could construct a relational expression denoting the union of relations corresponding to all instance literals of that type.

For every field, a relation of type `fld`.*declType* → `fld`.*type* is created to hold assignments of field values to objects. If the field is modifiable (inferred from its mention in a `@Modifies` clause), an additional relation is created, with the suffix „pre“ appended to denote the pre-state value. Relations for unmodifiable fields are given an exact bound that reflects the current state of the heap. For the modifiable relations, the „pre“ relation is given the same exact bound, and the “post” relation is bounded so that it may contain any tuple permitted by the field's type. Local variables, such as `this`, `return`, and method arguments are treated similarly to literals.

Table 4.3 below summarizes how relations and bounds are created. Function `rel` takes a Java element and, depending whether the element is modifiable, returns either one or two relations. Function `bound` takes a Java element and its corresponding relation, and returns a bound for the relation. The `Bound` data type contains both lower and upper bounds. If only one expression is passed to its constructor (**B**), both bounds are set to that value. Helper functions `is_mod`, `is_post` and `fldval` are used to check whether a field is modifiable, to check whether a relation refers to the post-state, and to return a literal that corresponds to the value of a given field of a given literal, respectively.

```
rel :: Element → [Relation]

rel (Literal lit)          =   [R(lit.name, lit.type)]
rel (Type t)               =   $\bigcup_{lit<:t}$ rel lit
rel (Field fld)            =
  if is_ mod(fld)
      [R(fld.name, fld.declType →fld.type)] ++
      [R(fld.name + _pre, fld.declType →fld.type)]
    else
      [R(f.name, f.declType →f.type)]
rel (Local var)            =   [R(var.name, var.type)]

rel :: Element, Relation → [Bound]

bound (Literal lit) (Relation r)   =   B(lit)
bound (Field f) (Relation r)       =
    if is_ mod(fld) ∧ is_post(r)
      [B({}, ext(fld.declType×fld.type))]
    else
      [B($\bigcup_{lit<:Object}$ rel lit × fldval(lit,fld))]
bound (Return ret) (Relation r)    =   [B({}, ext(ret.type))]
bound (Local var) (Relation r)     =   [B(var)]

[Type] → Expression                    (hepler)

ext []                             =   {}
ext (t:[])                         =   $\bigcup_{lit<:t}$ lit
ext (t:xs)                         =   ext t × ext xs
```

Table 4.3: Translation of different Java constructs into relations (function `rel`) and bounds (function `bound`)

4.2.5 Example: translation of `BST.insert`

We will use for illustration of translation Binary Search Tree example. This BST has single root node. Every node contains integer key value and pointer to the left and right node. To verify validity of BST, some Squander's specifications are used (see Listing 4.1). `@SpecField` constraint declares, that set of BST nodes consist of root node and all nodes in left and right subtree of this root node except for `null` values. Another constraints represented by `@Invariant` annotation delares: (1) there are no loops in BST - using transitive closure to define, that actual node is not accessible transitively via field `parent` - and (2) and for every node n in the BST, the *key* of n is strictly greater, than all keys of all nodes in the left subtree and strictly lower than all keys of all nodes in its right subtree - there is used reflexive transitive closure operator (`.*`) to conveniently specify all reachable nodes starting from the root node.

```
@SpecField("this.nodes = this.root.*(left + right) - null")
public class BST {
  @Invariant({
      /*format tree*/"this !in this.^parent",
      /*left sorted*/"all x: this.left.*(left + right) - null | " +
          "x.key < this.key",
      /*right sorted*/"all x: this.right.*(left + right) - null | " +
          "x.key > this.key" })
  public static class Node {
    Node left, right, parent;
    int key;
  }

  private Node root;
}
```

Listing 4.1: Binary Search Tree declaration

Methods for modifying tree, `insert()` and `remove()` shown in Listing 4.2 are obvious. In order to insert node into tree, node with same key may not exist in the tree, and after the insertion, tree must contain the given node. All left and right pointers of nodes and pointer to root node are allowed to modify. Deletion is defined in similar way, BST must contain node with the same key as node to remove and after deletion this node may not exist in BST.

```
@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old (this.nodes + z)")
@Modifies("Node.left, Node.right, this.root")
public void insert(Node z) {
  Squander.exe(this, z);
}

@Requires("z in this.nodes")
@Ensures("this.nodes = @old(this.nodes) - z")
@Modifies("Node.left, Node.right, this.root")
public void remove(Node z) {
  Squander.exe(this, z);
}
```

Listing 4.2: Operations on BST

The resulting set of relations, shown in Table 4.4, are divided into three sections. Relations in the upper section are unary, unmodifiable relations, and represent objects on the heap. The middle section contains relations that are also unmodifiable, because they are used to either represent unmodifiable fields or values in the pre-state of modifiable fields. Finally, the relations in the bottom section represent the post-state of modifiable fields; these are the relations for which the solver will attempt to find appropriate values. By default, the lower bound is simply set to an empty set and the upper bound is the upper bound is set the extent of the field's type.

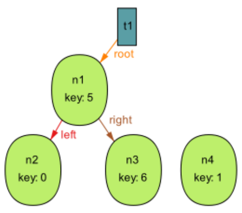


Figure 4.2: A snapshot of `t1.insert(t4)`

BST:	$\{t_1\}$
N1:	$\{n_1\}$
N2:	$\{n_1\}$
N3:	$\{n_1\}$
N4:	$\{n_1\}$
null:	$\{null\}$
BST_this:	$\{t_1\}$
z:	$\{n_4\}$
ints:	$\{0, 1, 5, 6\}$
key:	$\{(n_1 \to 5), (n_2 \to 0), (n_3 \to 6), (n_4 \to 1)\}$
root_pre:	$\{(t_1 \to n_1)\}$
nodes_pre:	$\{(t_1 \to n_1), (t_1 \to n_2), (t_1 \to n_3), (t_1 \to n_4)\}$
right_pre:	$\{(n_1 \to n_2), (n_2 \to 0), (n_3 \to null), (n_4 \to null)\}$
left_pre:	$\{(n_1 \to n_3), (n_2 \to 0), (n_3 \to null), (n_4 \to null)\}$
root:	$\{\}, \quad \{t_1\} \times \{n_1, n_2, n_3, n_4\}$
nodes:	$\{\}, \quad \{t_1\} \times \{n_1, n_2, n_3, n_4\}$
left:	$\{\}, \quad \{n_1, n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4\}$
right:	$\{\}, \quad \{n_1, n_2, n_3, n_4\} \times \{n_1, n_2, n_3, n_4\}$

Table 4.4: Translation of the heap from Figure 4.2

4.2.6 Tightening the bounds

By declaring as modifiable fields `left` and `right`, as in the specification in 4.1, we allow arbitrary modifications to the tree, as long as all constraints are satisfied. In effect, after the execution of the specification for the `insert()` method, the tree will contain all old nodes plus the new node, but the shape of the tree may randomly change.

If we want to change method specification so, that tree topology is preserved and new nodes can be append only to leaf nodes, we can do that by adding new clauses to the postcondition specification in sense, that left and right pointers of certain nodes must remain the same in the post state. This idea is completely right, but more efficient would to change the frame condition also for improving performance. Reducing search space by allowing modification only `left` and `right` pointers pointing to `null` value improve performance significantly.

Consider modified frame condition shown in Listng 4.3 specifying additional constraints on the modification of `left` and `right` fields. This frame condition shows idea specified in paragraph before and ensures, that all new nodes will be inserted at the leaf positions.

```
@Requires("z.key !in this.nodes.key")
@Ensures("this.nodes = @old ( this.nodes + z)")
@Modifies({"Node.left [{ n: this.nodes | n.left == null}]",
    "Node.right [{ n: this.nodes | n.right == null}]",
    "this.root"})
public void insert(Node z) {
  Squander.exe(this, z);
}
```

Listing 4.3: Modified frame condition on `insert()` method

With the list of modifiable objects, Squander modifies the bounds for the corresponding field relation so that the current field values of the objects not to be modified are included in the lower bound, thus forcing the value to stay the same in the post state. For the heap shown in Figure 4.2, the modifiable objects for the `left` field are `n2` and `n3`, because their left pointers are currently set to null. Similarly, for the `right` field, the modifiable objects are also `n2` and `n3`. The updated bounds for these two fields are shown in Table 4.5.

left:	$\{(n_1 \to n_2), (n_4 \to null)\}$,	$(n_1 \to n_2) \cup (n_4 \to null) \cup \{n_2, n_3\} \times \{n_1, n_2, n_3, n_4\}$
right:	$\{(n_1 \to n_3), (n_4 \to null)\}$,	$(n_1 \to n_3) \cup (n_4 \to null) \cup \{n_2, n_3\} \times \{n_1, n_2, n_3, n_4\}$

Table 4.5: The updated bounds for the `left` and `right` relations

4.2.7 Minimizing the universe size

For representing relations in Squander are used single sequential arrays indexed by Java integer. Maximal count of members in this array is `Integer.MAX_VALUE` which is 2147483647. In practice this can be a problem, because framework for relation of arity k, which has n atoms in the universe (later it will be described, what is exactly meant under this term), allocates matrix of size n^k (when ternary relation contains more than 1290 atoms). Heap with more than 1290 objects are not uncommon, so a simple translation described in 4.2.5 would not work. In the next subsection will be described a different translation technique called *KodkodPart*, which was developed to minimize number of atoms representing object heap.

4.2.7.1 *KodkodPart* translation

KodkodPart translation achieves a universe with fewer atoms by mapping Java objects to Kodkod atoms (not injective mapping). That means that multiple literals are mapped to a single atom, so that there will be fewer atoms than literals. The key requirements is, that there exists some inverse function, which maps back from the atoms literals.

Considering previous example with BST (see subsection 4.2.5), we can define domains $\mathcal{D}$, literals $\mathcal{L}$ and assignment literals to domains $\gamma : \mathcal{D} \rightarrow \mathcal{P}(\mathcal{L})$ for this example, which is summarized in Table 4.6.

$$\begin{aligned}
\mathcal{D} &= \{\texttt{BST}, \texttt{Node}, \texttt{Null}, \texttt{Integer}\} \\
\mathcal{L} &= \{bst_1, n_1, n_2, n_3, n_4, null, 0, 1, 5, 6\} \\
\gamma(\texttt{BST}) &= \{bst_1\} \\
\gamma(\texttt{Node}) &= \{n_1, n_2, n_3, n_4\} \\
\gamma(\texttt{Null}) &= \{null\} \\
\gamma(\texttt{Integer}) &= \{0, 1, 5, 6\}
\end{aligned}$$

Table 4.6: Summary of domains and instances for the `BST.insert` example

Recall that field types are represented as unions of base types (in this section also called *partitions*). For instance, the type of the field `BST.root` is `BST` → `Node` ∪ `Null`, because values of this field can be either instances of `Node` or the `null` constant. That means that all objects of class `Node` plus the constant `null` must be mapped to different atoms, so that it is possible to unambiguously restore the value of the field `root`. This is the basic idea behind the KodkodPart translation: *all literals within any given partition must be mapped to different atoms, whereas literals not belonging to a common partition may share atoms.* The inversion function can then work as follows: for a given atom, first select the correct partition based on the type of the field being restored, then unambiguously select the corresponding literal from that partition. To complete the example, the set of all unary types used in the specification for this example is:

$$\mathcal{T} = \{\texttt{BST}, \texttt{BST} \cup \texttt{Null}, \texttt{Node} \cup \texttt{Null}, \texttt{Null}, \texttt{Integer}\}$$

Set presented above is set of our partitions. A valid assignment of atoms to literals that uses only 5 atoms, as opposed to 10 which is how many the original translation would use, could be:

$bst_1 \to a_0$	$n_1 \to a_0$	$n_2 \to a_1$	$n_3 \to a_2$	$n_4 \to a_3$
$null \to a_4$	$0 \to a_0$	$1 \to a_1$	$5 \to a_2$	$6 \to a_3$

Limitation of this technique is when the class `Object` is used as a field type or anywhere in the specification. This lead toone big partition containing all literals (because every class is a subclass of `Object`), making the algorithm equivalent to the original translation.

4.2.7.2 Partitioning algorithm

For a given set of base domains $\mathcal{D}$, literals $\mathcal{L}$, and partitions $\mathcal{T}\left(\mathcal{T}=\mathcal{P}\left(\mathcal{D}\right)\right)$, and a given function $\gamma : \mathcal{D} \to \mathcal{P}\left(\mathcal{L}\right)$ that maps domains to their instance literals, this algorithm produces a set of atoms $\mathcal{A}$ and a function $\alpha : \mathcal{L} \to \mathcal{A}$, such that for every partition p, function α returns different values for all instance literals of p. Formally:

$$\left(\forall p \in \mathcal{T}\right)\left(\forall l_1, l_2 \in \psi\left(p\right)\right) l_1 \neq l_2 \Longrightarrow \alpha\left(l_1\right) \neq \alpha\left(l_2\right)$$

where ψ is a function that for a given partition returns a comprehension of all instance literals of all of its domains:

$$\psi : \mathcal{T} \to \mathcal{P}\left(\mathcal{L}\right); \;\; \psi\left(p\right) = \left\{\gamma\left(d\right) \mid d \in p\right\}$$

Obviously, a simple bijection would satisfy this specification, but such a solution wouldn't achieve its main goal, which is to minimize the number of atoms, because the number of atoms in this case would be exactly the same as the number of literals. In order to specify solutions that are actually useful, we are going to require the algorithm to produce a result such that the cardinality of $\mathcal{A}$ (i.e. the total number of atoms) is minimal.

It is not immediately clear what the minimal number of atoms ought to be. One might think that no more atoms are required than the number of instances in the largest partition. However, this is not always true. Consider the case shown in Figure 4.3. The largest partitions are `P1` and `P4`, both having 5 literals. On the other hand, any pair of the domains `B`, `C`, and `D` have a partition in common, even though there is no single partition containing them all. They thus form a strongly connected component, and their literals must differ.

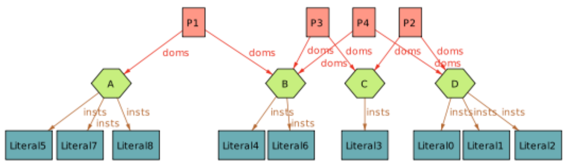


Figure 4.3: KodkodPart: an example where more than the number of literals of the largest partition is needed.

There are 6 literals in total in these three domains, so 5 atoms cannot be enough. As a conclusion, the minimal number of atoms is indeed the number of literals in the largest partitions, but only after all strongly connected domains have been merged into a single partition.

Luckily, cases like the one in Figure 4.3 never happen in Squander, so our implementation of the algorithm doesn't have to search for cliques and merge partitions. The reason this never happens is that domains are always Java classes, and partitions are types used to represent fields. A type of a field is a union type which includes the entire subclass hierarchy of the field's base type. For instance, if `C` and `D` are Java classes, `C` extends `D (C <: D)`, and some field has declared type `D`, then the type of the field (in the relational world) will be `D` ∪ `C` ∪ `Null`, meaning that `D` ∪ `Null` is never going to be used as a partition for anything.

In summary, the actual implementation inside Squander works as follows:

1. Dependencies between domains are computed. A domain depends on all domains with which it shares a partition. Let the function $\delta : \mathcal{D} \rightarrow \mathcal{P}(\mathcal{D})$ express this:

$$\delta(d) = \{d_1 \mid d_1 \neq d \ \wedge ((\exists p \in \mathcal{T})\, d_1 \in p)\}$$

2. The largest partition p_{max} is found such that

$$(\nexists p \in \mathcal{T})\, |\psi(p)| > |\psi(p_{max})|$$

3. For every literal l in $\psi(p_{max})$ an atom a is created, it is added to the universe $\mathcal{A}$ and assigned to l, such that $\alpha(l) = a$. From this point onwards, $\mathcal{A}$ is fixed.

4. For every other partition p iteratively, for all literals $l_p \in p$ that do not already have an atom assigned, a set of possible atoms $\mathcal{A}_{lp}$ is computed and the first value from this set is assigned to l_p. $\mathcal{A}_{lp}$ is computed when atoms corresponding to all literals of all dependent domains is subtracted from $\mathcal{A}$, i.e.:

$$\mathcal{A}_{lp} = \mathcal{A} \setminus \{\alpha(l) \mid l \in \mathcal{L}_d\}, \text{ where}$$

$$\mathcal{L}_d = \{\gamma(d) \mid d \in \mathcal{D}_d\}, \text{ where}$$

$$\mathcal{D}_d = \delta(d_l), \text{ where } d_l \in \mathcal{D} \wedge l_p \in \gamma(d_l)$$

4.3 Differences between Prolog and framework

Thing, which have both classical logical programming in Prolog and programming using framework Squander in common, is defining what to compute instead of how. Both programming paradigms use set of rules for its computation, but here the analogy ends.

If we study Prolog and Squander more in depth, we can see, that there is significant difference in declaring input data. Whereas Squander is strongly connected to its Java data structures, Prolog use set of facts, as was mentioned in chapter 2.

Both Prolog and Squander use set of rules for its computation. If we want to compute some variable in Prolog, language go recursively through all rules, matching the input using rewriting clauses, until all of them are replaced by valid facts and we do not care about order of evaluation (no flow control). Prolog uses three types of formula. *Atomic* formulas declares basic facts, *conditional commands* are used for implication constructs and finally, *target clauses* form query types.

To incorporate Squander into Java annotations are used. For solving some algorithm, annotation `@Requires`,`@Ensures` and `@Modifies` are mostly used. First one, `@Requires`, is used for declaring requirements for input data before method `Squander.exe()` is invoked. Annotation `@Ensures` defines how the data structures should look like after method execution. Last one, `@Modifies`, tells framework class objects or object's fields can be modified during execution. This is very important declaration for the performace of the computation in particular. If we define this annotation optimally, we minimize search space and speed up solving. Calculation itself is done by translating object heap into relations. Then are these relations transformed into SAT problem, which is then solved and the solution, if exists, is reflected back to the heap.

Chapter 5

Implementation of algorithms

5.1 Introduction

As was mentioned before, framework Squander is suitable for implementation of algorithms, which are easy to specify, but difficult to implement imperatively. For this reason I implemented algorithms belonging to NP problems. Problem is said that belongs to the set of NP problems if it is solveable in polynomial time on nondeterministic Turing machine. There is one more expression to explain for description of NP problems - *nondeterministic Turing machine.* First of all we have to define deterministic Turing machine, which for given set of rules prescribes at most one action to be performed for any given situation. Whereas nondetermistic Turing machine is machine, that for given set of rules prescribes more than one action. We can imagine, that in every step of computation is Turing machine cloned in nondeterministic Turing machine.

Most of the algorithm I tried to implement are NPO (especially graph algorithms), but there is a little problem with Squander. It does not support highr-order expressions (subsection 4.1.3). Unfortunately it is not possible to write a specification that says „find a path in the graph such that there is no other path in the graph longer than it“ and solve it with Squander; it is possible, however, to express and solve „find a path in the graph with at least k nodes“, which is computationally as hard as the previous problem, because a binary search can be used to efficiently find the maximum k for which a solution exists. Due that restriction are all of the optimizations algorithms implemented in such way, that there is some treshold level, which solution has to satisfy.

All in all, Squander should be able to solve all NP problems, because it transforms everything into SAT problem, which is NP-complete (Cook's theorem) problem and every NP problem is transferable into NP-complete problem.

5.2 Knapsack Problem

5.2.1 First version

At the beginning when I started „playing“ with Squander, I tried to implement optimization variant of well known algorithm Knapsack. This algorithm is usually introduced, that there is burglar with knapsack inside some estate and he has to steal as many things as capacity of his knapsack allowed to him with highest price.

This problem is also sometimes called the easiest one from NP-hard problems. If we try to solve it by brute force, we get an exponential time complexity. But there are more than one way how to solve this problem with different complexity. E.g. by using dynamic programming we get pseudo-polynomial time complexity, which means that time complexity is depends not only on input data but also on another variable.

By using Squander's method specification it was pretty easy to declare annotations for computation. Firstly it was necessary to prepare data structures. In this case I implement class `KnapSacck.java`, which has following properties:

- *integer array c* - costs of particular things in knapsack
- *integer array v* - weights of particular things in knapsack
- *integer n* - count of things in knapsack
- *integer capacity* - capacity of knapsack (measured in weight units)

That is all information we need to know about object `Knapsack`. Important thing is to define some treshold - in our case it will be total cost of things in knapsack, which we pass as parameter. Now we have to define some Squander's specification. Firstly we make a decision, if configuration of things will be returned as a new array or if we pass some array as a method parameter. Let us try to choose first option. This option is connected with annotation `@FreshObjects`, which is used to define, which type of object will be returned from `Squander.exe()` method, so one instance of the object `Integer` array should be created on the heap specifying which things are in the knapsack (value 1) and which not (value 0). After these steps, our method specification looks like this:

```java
@FreshObjects ( cls = Integer[].class , num = 1 )
public Integer [] solveKnapSackProblem(int minCost)
{
  return Squander.exe(this, minCost);
}
```

Listing 5.1: Declaration of return type of `Squander.exe()` method

After that we define some constraints, that help reduce search space. Question that we must ask is what objects or fields of which classes are allowed to be modified. Costs and weights are same during computation, similarly capacity and count of things in knapsack. The only structure that has to be modifiable is return array of `Integers`. According to Squander's documentation we have to declare via annotation `@modifies`, that both length and elements of returned array are modifiable:

```java
@Modifies ({ "return.length" , "return.elems" })
@FreshObjects ( cls = Integer[].class , num = 1 )
public Integer [] solveKnapSackProblem(int minCost)
{
  return Squander.exe(this, minCost);
}
```

Listing 5.2: Definition of modifiable objects on `Squander.exe()` method

The next step is to decide if we have some requirements on input data. We can see that this data are quite enough restricted by its own data type so we do not use annotation `@Requires`, which is used to define some prerequisities for the execution. Let us continue with specification what we expected as a result of computation.

So the last thing we have to specify is method output. At first come definition, that total weight of the all things in knapsack has to be smaller or equal to the knapsack capacity:

```java
@Ensures("(sum i: int | this.v[i]*return[i]) <= this.capacity")
@Modifies ({ "return.length" , "return.elems" })
@FreshObjects ( cls = Integer[].class , num = 1 )
public Integer [] solveKnapSackProblem(int minCost)
{
  return Squander.exe(this, minCost);
}
```

Listing 5.3: Declaration of knapsack capacity constraint on `Squander.exe()` method

Then it is necessary to specify, that total cost of things in knapsack has to be greater than varibale `minCost`, which is passed as method parameter:

```
@Ensures({"(sum i: int | this.v[i]*return[i]) <= this.capacity",
          "(sum i: int | this.c[i]*return[i]) > minCost"})
@Modifies ({ "return.length" , "return.elems" })
@FreshObjects ( cls = Integer[].class , num = 1 )
public Integer [] solveKnapSackProblem(int minCost)
{
  return Squander.exe(this, minCost);
}
```

Listing 5.4: Declaration of knapsack total cost constraint on `Squander.exe()` method

It is obvious, that Squander use keyword `this` in similar way as Java. Finally we make a specification, that length of return `Integer` array has to be `n` and that values of that arrays have to be from set $\{0, 1\}$:

```
@Ensures({ "return[int] in {0,1}",
           "return.length = this.n ",
           "(sum i: int | this.v[i]*return[i]) <= this.capacity",
           "(sum i: int | this.c[i]*return[i]) > minCost"
})
@Modifies ({ "return.length" , "return.elems" })
@FreshObjects ( cls = Integer[].class , num = 1 )
public Integer [] solveKnapSackProblem(int minCost)
{
  return Squander.exe(this, minCost);
}
```

Listing 5.5: Declaration of parameters of returned array on `Squander.exe()` method

Knapsack problem is now completely implemented but let us think about possible improvements. When thinking about improvement, we should concentrate on modifiable object, which is returned `Integer` array. In this case we get by with array of Boolean values, which shrinks variable domain as much as possible and limit the search space. We have declared constraint in `@Ensures` annotation, that values of returned array are in set $\{0, 1\}$, but this constraint does not limit search space. Due that change, we should also modify declaration of constraints describing minimal cost and capacity of knapsack. It is necessary also modify `@FreshObject` annotation to Boolean data type.

```
@Ensures({"return.length = this.n ",
    "(sum i: int | (return[i]?this.v[i]:0)) <= this.capacity",
    "(sum i: int | (return[i]?this.c[i]:0)) > minCost"
    })
@Modifies ({"return.length",
    "return.elems" })
@FreshObjects ( cls = Boolean[].class , num = 1 )
public Boolean [] solveKnapSackProblem(int minCost)
{
  return Squander.exe(this, minCost);
}
```

Listing 5.6: Improving implementation of Knapsack using Boolean data type

Now we have implement Knapsack problem in advanced way and it is time for testing, if it works correctly. During testing I used instaces for which I know the `bestPrice` solution and for a treshold `minCost` I used value equals to `bestPrice - 1`.

At first was the implementation tested on smaller instances consisting of 4 things. From 45 examples was one failed, when Squander returns configuration, which was overweight. Moreover when was minimal cost greater or eequal to 512, program fails on error
`Arity too large (3) for a universe of size 2055`. We know from chapter 4, subsection 4.2.7.1 that Squander used for representing relation array indexed by Java integer. Due to size of the universe $2055^3 = 8678316375$ which is greater than `Integer.MAX_VALUE` which is 2147483647 is Squander unable to represent relation. Why does it happen? If is in the specification `minCost` greater or equal to 512, Squander automatically uses 11 bits to represent `Integers`, which gives a scope $< -2^{10}, 2^{10}) = \{-1024, \ldots, 1023\} = 2048$ `Integers` and as was mentioned in subsection 4.2.7, these ternary relations cannot be mapped into `Integer` indexed array ($2048^3 > 2147483647$). Another problem comes in, when we variable `minCost` is set to 511. Then Squander uses bitwidth 10, which means $< -2^9, 2^9) = \{-512, \ldots, 511\}$ and that is why it is unable to find solution, which is 512 so not in the scope, and throws `noSolutionException`.

We can see from the measurement, that this implementation of knapsack using Squander is for this type of problem quite unsuitable.

5.2.2 Second version

In order to reduce search space, I tried to implement another version of knapsack problem. Let us think about things in knapsack as an entity, not as a configuration variable in array.

By using this way of thinking, we get reduced count of atoms in Squander, because now we have only one set of objects `Thing` but in the first version of implementation there were two arrays of the same data type (`Integer` array c and `Integer` array v same size as set of things we have now), which drastically increase number of atoms. Fields of entity `Thing` look like this:

- *integer c* - cost of actual thing
- *integer v* - weight of actual thing
- *boolean choosed* - symptom if is choosed
- *integer position* - this variable is used for sorting the output, because instances of `Thing` are in `java.util.Set` (for performance reasons)

For a specification of Squander, we create `Set` of objects `Thing`, allowing to modify only `boolean choosed` symptom. Summation logic is the same as in first version and variable `position` is used for sorting resulted set of object in order to have better interpretation of the solution.

In this example we use experience from the previous one and declare variable `minCost` as the real maximum cost of possible solution and write it to the specification (sum of cost should be equal to `minCost`), so best solution value will not disappear in higher `Integer` bitwidth. On the other hand, this implementation is not very usable, because we almost always do not know exact highest total price.

```java
@Ensures({"(sum e: result.elts | (e.choosed?e.v:0)) <= capacity",
          "(sum e: result.elts | (e.choosed?e.c:0)) = minCost"
        })
@Modifies ("Thing.choosed")
public void solveKnapSackProblem(Set<Thing> result, int n, int capacity, int
    minCost)
{
  Squander.exe(this, result, n, capacity , minCost);
}
```

Listing 5.7: Improving implementation of Knapsack using previous experince and reducing count of relations

This way of implementation is suitable for instances with more things in knapsack, which means higher value of variable `capacity` and higher value of variable `minCost`, because there are not so many relations created. Implementation was successfuly tested for 40 things in one instance at most.

5.3 General bisection breadth

This problem is taken from course *Parallel Systems and Algorithms* lectured at Faculty of Electrical Engeneering of Czech Technical University in Prague (CTU FEE). It is a graph algorithm and task for that algorithm is to find two groups of nodes - number of nodes in one group is exactly define - such, that count of edges connecting these two groups are minimal. Exact definition is:

„Find distribution set of nodes into to two disjoint groups X, Y such, that set X contains a nodes, set Y contains $n - a$ nodes and count of edges $\{u, v\}$, whose u is from X and v is from Y, is minimal.“

Because of using first order logic in Squander (4.1.3), it is necessary again to define some maximum treshold of counts of edges connecting groups of nodes. For generating graph was implement project called `Common graph utils`, which generates graph according to this rules:

1. generate and add n nodes to the graph
2. generate a random permutation of nodes and add edges between the neighboring nodes in the permutation, including the edge between the last and the first node. At this point, the graph contains a Hamiltonian cycle.
3. randomly choose a number between 30 and 90 percent of the maximum number of nodes $(n(n+1))$ and keep adding random edges until the number of edges in the graph is equal to the chosen number.
4. randomly choose a node and remove all its incoming edges. At this point, the graph still contains at least one Hamiltonian path, the one that starts from the node selected in this step.
5. if the goal is to generate graphs with no Hamiltonian paths, remove all outgoing edges of the node selected in the previous step.

This algorithm for generating graph is taken from Hamiltonian path problem, which will be introduced later, but we will use it for generating graphs for all graph algorithms presented in this thesis.

Regarding *General bisection breadth* algorithm, we will use these data structures and objects:

- *class Node* with property `value`, which symbolizes label of actual node
- *class Edge* with properties `src`, `dest` of type `Node` and `Integer` property cost, that will not be used in this algorithm at all
- *Set nodes* which will constain all nodes in the graph
- *Set edges* which will contain all edges in the graph
- *Set resultA* which will represent set of nodes of exactly defined size
- *integer a* representing exact size of set of nodes `resultA`
- *Set resultN* defining size of set containing rest nodes
- *integer n* defining total count of nodes in graph
- *integer treshold* representing maximum count of edges having source node in `resultA` and destination node in `resultN` and reversely
- *Set commonEdges* is help data structure containg edges starting from one set of nodes and ending in the other one

Again we started defining what we allow to modify and what are the prerequisities. The answer is easy, because all data structures we need to change are both main set of nodes `resultA` and `resultN` and set of `commonEdges` also with its length, because we want also to find set of fewer common edges than treshold is. After that we add constraint that variable `a`, which symbolizes count of nodes in set `resultA`, has to be lower or equal to total count of nodes `n`:

```
@Requires("a <= this.n")
@Modifies ({ "resultA.elts" , "resultN.elts", "commonEdges.elts",
             "commonEdges.length" })
public void solveZobecnenaBisekcniSirkaGrafuProblem(Set<Node> resultA,
        Set<Node> resultN, Set<Edge> commonEdges, int a, int treshold)
{
  Squander.exe(this, resultA, resultN, commonEdges, a, treshold);
}
```

Listing 5.8: Declaration of modifiable objects in General bisection breadth

Then we can add constraints about what we expect after execution by annotation `@Ensures`. This specification will contain information that both sets `resultA` and `resultN` are subsets of set of nodes in the graph, information that size of set `resultA` is `a` and size of set `resultN` is `n - a`:

```
@Requires("a <= this.n")
@Ensures ({"resultA.elts in this.nodes.elts" ,
           "#resultA.elts == a" ,
           "resultN.elts in this.nodes.elts" ,
           "#resultN.elts == (this.n - a)"})
@Modifies ({ "resultA.elts" , "resultN.elts", "commonEdges.elts",
             "commonEdges.length" })
public void solveZobecnenaBisekcniSirkaGrafuProblem(Set<Node> resultA,
        Set<Node> resultN, Set<Edge> commonEdges, int a, int treshold)
{
  Squander.exe(this, resultA, resultN, commonEdges, a, treshold);
}
```

Listing 5.9: Declaration of both sets of nodes

Finally we have to add specification that none of the nodes is at the same time in set `resultA` and in set `resultN` and specifications concerning set of `commonEdges`. First one is that count of `commonEdges` is lower or equal to `treshold` and that `commonEdges` are subset of set `edges`. Next step is to define that all egdes, whose source and destination node are in different set (`resultA` and `resultN`), are in set `commonEdges`:

```
@Requires("a <= this.n")
@Ensures ({"resultA.elts in this.nodes.elts" ,
           "#resultA.elts == a" ,
           "resultN.elts in this.nodes.elts" ,
           "#resultN.elts == (this.n - a)",
           "all a: resultA.elts | no n: resultN.elts |" +
           "a.value == n.value" ,
           "commonEdges.elts in this.edges.elts" ,
           "#commonEdges.elts <= treshold" ,
           "all e: this.edges.elts | " +
           "(((e.src in resultA.elts) && (e.dest in resultN.elts) || " +
           "(e.dest in resultA.elts) && (e.src in resultN.elts))?"+
           "(e in commonEdges.elts):(e !in commonEdges.elts))"})
@Modifies ({ "resultA.elts" , "resultN.elts", "commonEdges.elts",
             "commonEdges.length" })
public void solveZobecnenaBisekcniSirkaGrafuProblem(Set<Node> resultA,
        Set<Node> resultN, Set<Edge> commonEdges, int a, int treshold)
{
  Squander.exe(this, resultA, resultN, commonEdges, a, treshold);
}
```

Listing 5.10: Final version of implementation General bisection breadth algorithm

During testing testing this Squander implementation was able to deal with instances up to 30 nodes and never return bad solution on 50 instances.

5.4 L-dominant set of graph

5.4.1 First version

Idea of this algorithm was also taken as 5.3 from the same course at CTU FEE. Also this algorithm is graph a algorithm and main idea is to find set of nodes in the graph, which will covered with its neighborhood of defined breadth, whole graph. Again to be more precise, definition of algorithm is:

„For a given natural number $l \geq 0$ and node u of graph G is $l - negborhood$ of node u is set of all nodes of G in distance not more than l from node u, including node u itself. Then $l - dominant\ set\ of\ graph\ G$ is every set of nodes such, that union of their $l - neighborhood$ contains all nodes in G."

For this algorithm is necessary to have more specific data about the graph. At first we have to generate neighborhood of specified length of all nodes in the graph. Due generating this neighborhood I implemented *Breadth first search algorithm (BFS)* and extends data structure *Node* from previous algorithm 5.3 of information about its state during BFS traversing and its neighborhood. As was said before, Squander use first order logic so some treshold of size of result set has to be define. Data structures and objects used in this implementation:

- *class ExtendedNode* which extends *class Node* from 5.3 of enum field *state* (*FRESH, OPENED, CLOSED*) and of *Set neighborhood*
- *Set extendedNodes* containing all nodes in the graph
- *Set edges* containing all edges in the graph
- *integer l* defining $l - neighborhood$ of node
- *integer treshold* expressing maximum of nodes in result set

In our implementation we will return result set of nodes as an array of class `ExtendedNode`, which we allow to modify:

```java
@Modifies ({ "return.length" , "return.elems" })
@FreshObjects ( cls = ExtendedNode[].class , num = 1 )
public ExtendedNode [] solveLDominantniMnozinaGrafuProblem()
{
  return Squander.exe(this);
}
```

Listing 5.11: Declaration of modifiable objects in L-Dominant set of graph implementation

Now comes definition what we are expected to return. Definitely is return array subset of set `extendedNodes` and its length has to be shorter or equal to variable `treshold`:

```java
@Ensures ({"return[int] in this.extendedNodes.elts" ,
           "return.length <= this.treshold" ,
         })
@Modifies ({ "return.length" , "return.elems" })
@FreshObjects ( cls = ExtendedNode[].class , num = 1 )
public ExtendedNode [] solveLDominantniMnozinaGrafuProblem()
{
  return Squander.exe(this);
}
```

Listing 5.12: Defining result array in L-Dominant set of graph implementation

Next we define that all neighborhoods of all nodes in result array cover whole graph. Then we tell Squander that in result array must not be one node twice or more and all of the nodes in return result array are not in neighborhood themselves:

```java
@Ensures ({"return[int] in this.extendedNodes.elts" ,
           "return.length <= this.treshold" ,
           "return[int].neighborhood.elts == this.extendedNodes.elts",
           "all q1: return[int] | no q2: (return[int] - q1) |" +
           "q1.value == q2.value",
           "all q1: return[int] | all q2: (return[int] - q1) |" +
           " (all n: q2.neighborhood.elts | n.value!=q1.value)"
         })
@Modifies ({ "return.length" , "return.elems" })
@FreshObjects ( cls = ExtendedNode[].class , num = 1 )
public ExtendedNode [] solveLDominantniMnozinaGrafuProblem()
{
  return Squander.exe(this);
}
```

Listing 5.13: Final version of implementation L-Dominant set of graph algorithm

This implementation works with instances with up to 60 nodes. During testing appears little failure, that in some cases result array contains some nodes more than once despite constraint 5.14. I have fix this failure in second version of implementation 5.4.2.

```
@Ensures ( "all q1: return[int] | no q2: (return[int] - q1) |" +
           "q1.value == q2.value")
```

Listing 5.14: Constraint causing failure in implementation L-Dominant set of graph algorithm

5.4.2 Second version

This version of implementation algorithm L-dominant set of graph is modified from 5.4.1 in return type of result. Here is used instead of array a set of class `ExtendedNode`, so it is not necessary to check, whether result set contains concrete node or not, because from definition [9] is `java.util.Set` a collection, that contains no duplicate elements. The only thing I have needed for implementation is a help data structure `allExtendedNodes` for checking, whether all nodes in result set with its neighborhood covered whole graph with precondition of empty set 5.15.

```
@Requires("#this.allExtendedNodes.elts == 0")
@Ensures ({
    "return.elts in this.extendedNodes.elts" ,
    "return.length <= this.treshold" ,
    "return.elts.neighborhood.elts == this.extendedNodes.elts",
    "all e: return.elts | e.neighborhood.elts in this.allExtendedNodes.elts",
    "this.allExtendedNodes.elts == this.extendedNodes.elts",
    "all q1: return.elts | all q2: (return.elts - q1) |" +
    "(all n: q2.neighborhood.elts | n.value!=q1.value)"
    })
@Modifies ({ "return.elts", "this.allExtendedNodes.elts" })
@FreshObjects ( cls = Set.class, typeParams={ExtendedNode.class} , num = 1 )
public Set<ExtendedNode> solveLDominantniMnozinaGrafuProblem()
{
  return Squander.exe(this);
}
```

Listing 5.15: Implementation L-Dominant set of graph algorithm using set as a return type of result

5.5 Hamiltonian Path

Algorithm *Hamiltonian path* is quite well known. Core of this algorithm is to find path which covers all nodes in the graph and visit no node twice or more. It is last graph algorithm presented in this thesis using similar data structures as previous ones. Implementation of algorithm is taken from [1]. Also data structures and objects are similar as in previous ones:

- *class Node* with property `value`, which symbolizes label of actual node
- *class Edge* with properties `src`, `dest` of type `Node` and `Integer` property cost, that will not be used in this algorithm at all
- *Set nodes* containing all nodes in the graph
- *Set edges* containing all edges in the graph

Our implementation will return in new array of class `Edge` sequence of visited edges during traversing - to do that, `@FreshObjects` annotation is used. Then we allow to modify (via `@Modifies` annotation) elements of result array and its length (see Listing 5.16).

```
@Modifies ({ "return.length" , "return.elems" })
@FreshObjects ( cls = Edge[].class , num = 1 )
public Edge [] solveHamiltonianPath()
{
  return Squander.exe(this);
}
```

Listing 5.16: Declaration of modifiable objects and returned result in Hamiltonian path implementation

After that we add constraints defining, how resulted array of edges will looks like - `@Requires` annotation. Of course, firstly we tell Squander, that returned array is subset of set `edges`. Then we add similar constraint about nodes that all nodes (both source and destination) are equal to set of `nodes` and constraint that length of resulted array is count of `nodes -1`. Last constraint in `@Requires` annotation define, that for every edge in result set is destination node source node of following one:

```
@Ensures ({
    "return[int] in this.edges.elts" ,
    "return[int].(src + dest) = this.nodes.elts" ,
    "return.length = #this.nodes.elts - 1 " ,
    "all i: int | i >= 0 && i < return.length - 1 =>
    return[i].dest = return[i+1].src"
    })
@Modifies ({ "return.length" , "return.elems" })
@FreshObjects ( cls = Edge[].class , num = 1 )
public Edge [] solveHamiltonianPath()
{
  return Squander.exe(this);
}
```

Listing 5.17: Final implementation of Hamiltonian path algorithm

This implementation works fine for graphs with 15 nodes containing Hamiltonian path and for graphs with 10 nodes which do not contain Hamiltonian path.

5.6 Triangular solitaire

As last algorithm I tried to implement simple game called *Triangular solitaire*. Idea of that algorithm as idea of previous two graph algorithms (5.3 and 5.4) was taken from course *Parallel systems and algorithms* lectured at CTU FEE. Specification of the algorithm was finaly changed according to [7]. Target of this algortihm is to leave only one occupied position on the game field by doing allowed transitions. Allowed transition between two states of game field is to move move with choosed peg via another peg to position which is not occupied by the peg. Movements in actual state are available horizontally in both directions and vertically under angle 45 or 135 degrees, see following Figure 5.1.

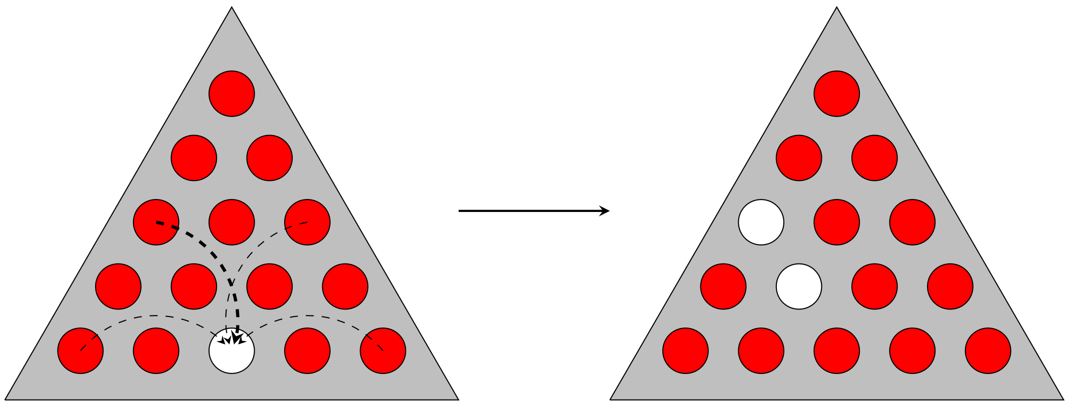


Figure 5.1: Initialized game field and possible transitions

In this Figure we can see except initialization state also example of one possible transition between two states. If it is target of the game to leave only one position occupied with peg, it is obvious that we need $N-1$ steps to do that, where N is count of positions in game field, because in every step we lose one peg ($N = \frac{n(n-+1)}{2}$, where n is length of side of the game field, it is simple summmation of arithmetic progression). Now we should declare, what we want to return from our program. Of course it will be sequence of states of game field, which lead to solution, specifiyng occupation of concrete cell. Let us declare data structures, which are necessary for computation.

First one will be definitely *class Cell* containing boolean field `occupied` and maybe some information about its position in the game field which is not very necessary for purpose of Squander computation. Next data structure we need is *class State* grouping individual cell together defining actual state of game field. *Class SolitaireSolver*, computing configuration of states, has following fields:

- *startState* defining inital state of the game field, where only one cell is not occupied
- *integer n* for defining size of side of game field - it has to be greater or equal to 5
- *integer array adj* containing type of adjacency (if exists) between cells, where index to that matrix is position of the cell in `cells` array ($left = 1, right = 2, top_{45} = 3, top_{135} = 4, bottom_{135} = 5, bottom_{45} = 6$)

Now let us do the specification of the main method computing desirable sequence of states. At first we will define some requirements on data structures we have. We only allow to modify field `occupied` of class `Cell`, but only on those instances, who are not representing `startState`, because this state has to be constant all the time (declared vis `@Modifies` annotation and optional instance selector in this annotation). Then we add constraint to object `startState`, that only one of the object in array `Cells` can be unoccupied. We declare this constraint using annotation `@Invariant` guarantees this constraint all the time (in this implementation will be enough to use annotation `@Required` when declaring this constraint, because we allow to modify only `Cell` objects in all states except for `startState`). Last requirement is that size of side of the triangle has to be greater or equal to 5.

```
@Invariant("one {i : {0 ... return[0].cells.length − 1} |"+
             "return[states.length −1].cells[i].occupied}")
@Requires("this.n >= 5")
@Modifies("Cell.occupied [{cell : Cell |"+
            "(all i:int|return[0].cells[i]!=cell)}]")
public State [] solveSolitergame(State [] states)
{
  return Squander.exe(this, new Object[] {states});
}
```

Listing 5.18: Defining requirements for Triangular solitaire algorithm

That were requirements for the algorithm. Now we have to add specifications about the output of the algorithm. Specifications about returned array of states are that all members of the result set are equal to all input states, because we have to pass „empty“ states as method parameter. Last specification told us, that only one cell of the last state is occupied. Specification is described in Listing 5.19.

```
@Invariant("one {i : {0 ... return[0].cells.length - 1} |"+
           "!return[states.length -1].cells[i].occupied}")
@Requires("this.n >= 5")
@Ensures({"return = states",
          "return[0] = this.startState",
          "one {i : {0 ... return[0].cells.length - 1} |"+
          "return[states.length -1].cells[i].occupied}"})
@Modifies("Cell.occupied [{cell : Cell |"+
          "(all i:int|return[0].cells[i]!=cell)}]")
public State [] solveSolitergame(State [] states)
{
  return Squander.exe(this, new Object[] {states});
}
```

Listing 5.19: Defining result set parameters for Triangular solitaire algorithm

Finally the most important thing is to add to method specification transition constraints between two following states. In that constraint we specify that in all states except last one exists three different cells such, that first one (`src`) and second one (`via`) is occupied and the last one (`dest`) is empty and that value for first and second one and for the second one and last one in the `adj` matrix is the same and nonzero (that means using positions of object cell in the `cells` array of actual state in `adj` matrix to get type of adjacency if exists). If everything what was mentioned is fulfilled, then is possible to make a transition such, that all cell's occupied values are copied from actual state to following one except for those three states mentioned above, which are copied inversively (`src`, `via` and `dest`) - see Listing 5.20.

This implementation works fine only for instances with up to 21 cells in game field (side length 6). For larger instances was timeout treshold exceeded.

```
@Invariant("one {i : {0 ... return[0].cells.length - 1} |"+
            "!return[states.length -1].cells[i].occupied}")
@Requires("this.n >= 5")
@Ensures({"return = states",
           "return[0] = this.startState",
           "one {i : {0 ... return[0].cells.length - 1} |"+
           "return[states.length -1].cells[i].occupied}",
           "all i: {1 ... return.length -1} |" +
           "(exists src, via, dest : {0 ... return[i-1].cells.length - 1} | " +
           "src!=via && src!=dest && via!=dest " +
           "&& (return[i-1].cells[src].occupied " +
           "&& return[i-1].cells[via].occupied "+
           "&& !return[i-1].cells[dest].occupied) " +
           "&& (this.adj[src][via] != 0) " +
           "&& (this.adj[via][dest] != 0) " +
           "&& (this.adj[src][via] = this.adj[via][dest]) " +
           "&& !return[i].cells[src].occupied " +
           "&& !return[i].cells[via].occupied " +
           "&& return[i].cells[dest].occupied "+
           "&& (all j:{0 ... return[i-1].cells.length - 1} |"+
           "(j!= src && j!= via && j!= dest)=>" +
           "return[i-1].cells[j].occupied=return[i].cells[j].occupied))"})
@Modifies("Cell.occupied [{cell : Cell |"+
           "(all i:int|return[0].cells[i]!=cell)}]")
public State [] solveSolitergame(State [] states)
{
  return Squander.exe(this, new Object[] {states});
}
```

Listing 5.20: Final implementation of Triangular solitaire algorithm

Chapter 6

Comparison

For meausurement of some metrics (execution time and memory analysis) I have tried to use Java profiler *Test & Performance Tools Platform (TPTP)*, which is Eclipse plugin profiling tool. This tool is highly integrated with Eclipse, so it allows profiling of applications running from within Eclipse - see Figure 6.1. This profiler had a big overhead when measuring Squnder implementation and this led to situation, that every instance in Squander reached time out. I have also tried another Eclipse profiler called *Colorer*, but developing of this profiler was cancelled in 2008 and also there was quite a lot of problems to use it in oldest available Eclipse Europa.

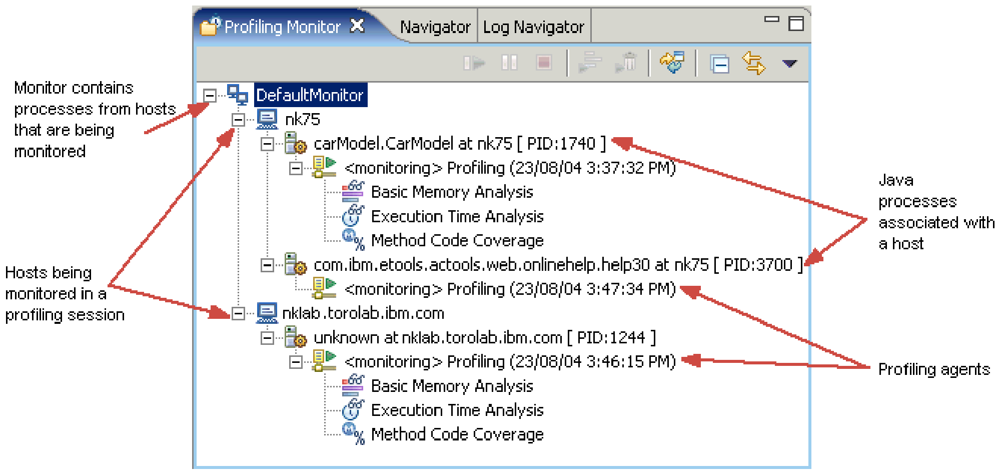
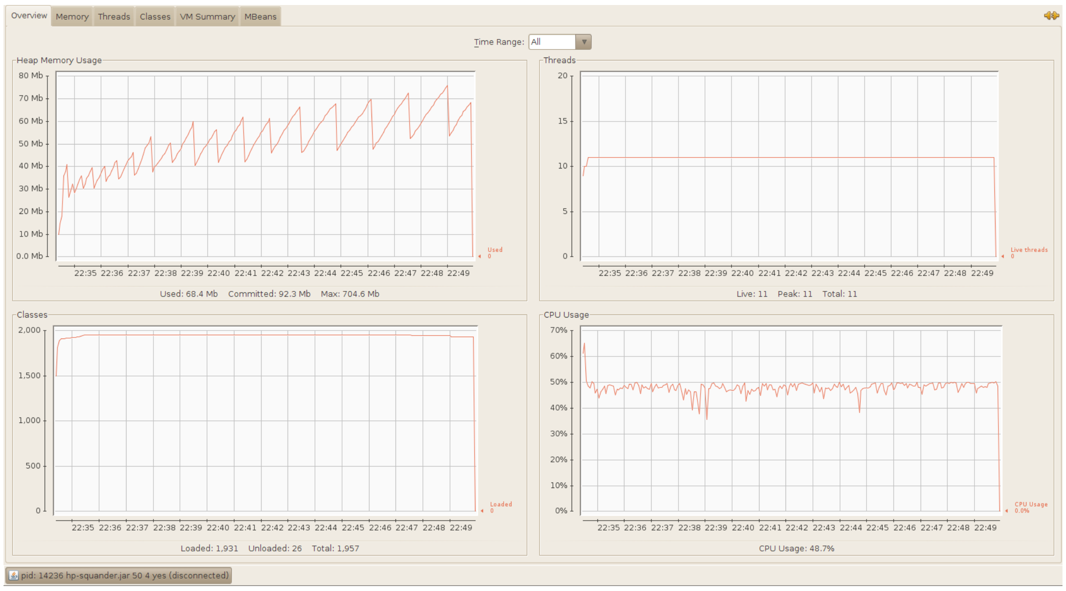

Profiling Monitor
Navigator
Log Navigator
DefaultMonitor
nk75
carModel.CarModel at nk75 [PID:1740]
<monitoring> Profiling (23/08/04 3:37:32 PM)
Basic Memory Analysis
Execution Time Analysis
Method Code Coverage
com.ibm.etools.actools.web.onlinehelp.help30 at nk75 [PID:3700]
<monitoring> Profiling (23/08/04 3:47:34 PM)
nklab.torolab.ibm.com
unknown at nklab.torolab.ibm.com [PID:1244]
<monitoring> Profiling (23/08/04 3:46:15 PM)
Basic Memory Analysis
Execution Time Analysis
Method Code Coverage
Monitor contains processes from hosts that are being monitored
Hosts being monitored in a profiling session
Java processes associated with a host
Profiling agents

Figure 6.1: TPTP profiling monitor

After that I decided to use tools provided by JVM, concretely *JConsole* (see Figure 6.2). This console can connect to any of running Java process and monitore its Heap memory usage, CPU usage, number of load classes or number of active threads. This data can be exported to csv file. It also provides some statistics like CPU time, compile time, which kind of garbage collector is used and how many collections were done etc.

For all measurement presented here, we had 1 warmup run and 5 test runs. The purpose of the initial warmup run was to eliminate possible „cold start“ effects, such as JVM initialization and class loading. In the subsequent 5 test runs, I executed the operation under test on 3 same instances of a given size, measured the total execution time, and reported the average. The timeout threshold was set to 25 minutes for all experiments, and the maximum Java heap size was left to the default value of 256MB. All measurements were run a Linux box, with Intel® Core™2 Duo CPU @ 2.1GHz, 4GB of RAM, running Ubuntu 10.04.



Used: 68.4 Mb Committed: 92.3 Mb Max: 704.6 Mb

Live: 11 Peak: 11 Total: 11

Loaded: 1,931 Unloaded: 26 Total: 1,957

CPU Usage: 48.7%

pid: 14236 hp-squander.jar 50 4 yes (disconnected)

Figure 6.2: Profiling using JConsole

6.1 Hamiltonian Path

I used for measurement of implementation of algorithm Hamiltonian Path Squnder implementation presented in 5.5. Impreative algorithm was implemented as simple backtracking. In main method for finding solution is algorithm iteratively going through matrix of followers row by row and when is resulted array of edges (length of this array is number of nodes - 1) empty, the actual egde is put at the beginning of the array. Another possibility is to put actual edge into resulted array to concrete position when this condition is satisfied: there is an edge between actual nodes in the iteration and in resulted array of edge does not appear source node of this edge. Rule that source and destination node af actual edge are different has to be fulfilled. After that if we are at the end of resulted array, then we are return back from the recursion, otherwise program is searching for the following edge in next recursive step. If program is returned from recursive step in depth n and none of the edges occupies actual position in resulted array, then is edge from recursive step $n - 1$ removed, because following edge for edge in depth $n - 1$ does not exist. Implementation of algorithm with comments is in Listing 6.1.

```java
public class GraphExecutor extends Graph {

  private Edge[] result;

  private int pointer;

  public GraphExecutor(int n, int maxGrade) {
    super(n, maxGrade);
    this.pointer = 0;
    this.result = new Edge[n - 1];
  }
  /*
   * Backtrack implementation
   */
  public void solveHamiltonianPathRecursively() {
    //iterate through matrix of followers
    for (int i = 0; i < this.n; i++) {
      for (int j = 0; j < this.n; j++) {
        //if there is a edge between nodes i,j and
        //these nodes are different and
        //recursion is at the beginning of resulted array or
        //is found node, that is not source or
        //destination node of any of edges in actual result array and
        //there is an edge between destination node j of last edge
        //in result array and the other one node i
        if (i != j
            && this.matrixOfFollowers[j][i]
            && ((this.result[0] == null) || (this.pointer > 0
                && this.result[this.pointer - 1].getDest()
                    .getValue() == j && doesResultNotContainNode(i)))) {
          //add edge to result array
          this.result[this.pointer] = findEdge(j, i);
          //Hamiltonian Path is complete
          if (this.pointer == this.n - 2)
            return;
          ++this.pointer;
          //recursively find following edge
          solveHamiltonianPathRecursively();
          //if following edge does not exist, return back
          if (this.pointer < this.n - 2
              || this.result[this.pointer] == null)
            this.result[--this.pointer] = null;
        }
      }
    }
  }
}
```

Listing 6.1: Imperative implementation of Hamiltonian Path algorithm

For testing both implementations I implemented simple graph generator according to the rules mentioned in 5.3 and add restriction of maximal grade of node in the graph. I made class `Graph` serizalizable in order to store it in some file and then load it into execution algorithm in order to run both implementations on the same data.

In my test I have measured execution time of imperative backtrack method and Squander implementation via method `getCurrentThreadCpuTime()` of `ThreadMXBean` interface. According to [8], a Java virtual machine has a single instance of the implementation class of this interface. This instance implementing this interface is an `MXBean` that can be obtained by calling the `ManagementFactory.getThreadMXBean()` method or from the platform `MBeanServer` method. This method returns the total CPU time for the current thread in nanoseconds. Results of the measurement are in Table 6.1.

From that table is obvious, that common imperative way of programming reached better results than Squander implementation. In that point it is important to note, that in execution time for Squander is included time for travesing heap, checking preconditions and creating bounds for the universe, ensuring post condition, translating to CNF and solving this CNF. We can also see, that bigger influence on the execution time has maximal grade of node in the graph than total number of nodes, mainly in imperative implementation, because it causes branching during backtracking.

Table 6.1: Execution times of Hamiltonian Path algorithm implementation in ms

	Number of nodes in the graph								
	40				50		60		
	max grade				max grade		max grade		
	3	4	5	6	3	4	2	3	4
Imperatively	180	4 220	39 910	772 860	120	60 390	10	440	780
Squander	3 200	33 510	130 800	**t/o**	3 270	704 230	3 540	6 380	**t/o**

	Number of nodes in the graph								
	70			80		90		100	
	max grade			max grade		max grade		max grade	
	2	3	4	2	3	2	3	2	3
Imperatively	30	3 940	**t/o**	20	14 920	70	173 090	40	**t/o**
Squander	3 940	5 100	**t/o**	4 060	70 860	5 080	**t/o**	4 700	**t/o**

As was said before, I used for tracking execution of algorithm JConsole, which started measuring on key press. When using JConsole for measuring *heap memory usage* and *cpu usage*, I have extracted data into csv file and use them for generating graphs in GNUPlot®, which is command-line driven graphing utility allowing to generate graphs into LaTeXsource code.

When looking at the graphs of heap memory usage of imperative implementation, we can see that there are some peaks in the graphs and those peak with biggest number of used memory symbolizes probably finding some solution in actual branch, which was not cut off, or some set of branches, that was not cut off or cut off in great depth before collection. In Squander implementation appears those peaks, when solving CNF and cleaning out clauses. Mainly is from graphs in selected tests obvious, that whereas imperative implementation uses around 10 Mb of memory (highest peak is 40 Mb), Squander allocates around 50 Mb of memory with highest peak 115 Mb.

Regarding graphs of CPU usage, Squander and also imperative implementation uses around 50 % of CPU except graph instance of 40 nodes, maximal node grade 5 when was used only 25 % of CPU computation performance in imperative implementation.

By using command line arguments `-verbose:gc -XX:+PrintGCTimeStamps` I got the information about time when was the garbage collector performed, how much of the memory was reclaimed including total available memory space and how long does it take (performing garbage collector is highlighted in graphs via green point). We can see that in Squander implementation is garbage collector performed every time, when cleaning clauses in CNF, which is quite often contrary to imperative implementation. In Table 6.2 are presented quotients of garbage collector activity to total execution time of both implementation. This table shows, that whereas in imperative implementation is garbage collector time one hundredth percent, in Squander implementation reaches sometimes one percent or more of execution time.

	Number of nodes in the graph		
	40	50	80
	max grade	max grade	max grade
	4	5	3
Imperatively	0.01% (0.005126 secs)	0.01% (0.006266 secs)	0.008% (0.00125 secs)
Squander	0.8% (1.054949 secs)	0.3% (1.908237 secs)	1.2% (0.851782 secs)

Table 6.2: Quotient of garbage collector activity to total execution time of Hamiltonian Path algorithm implementation

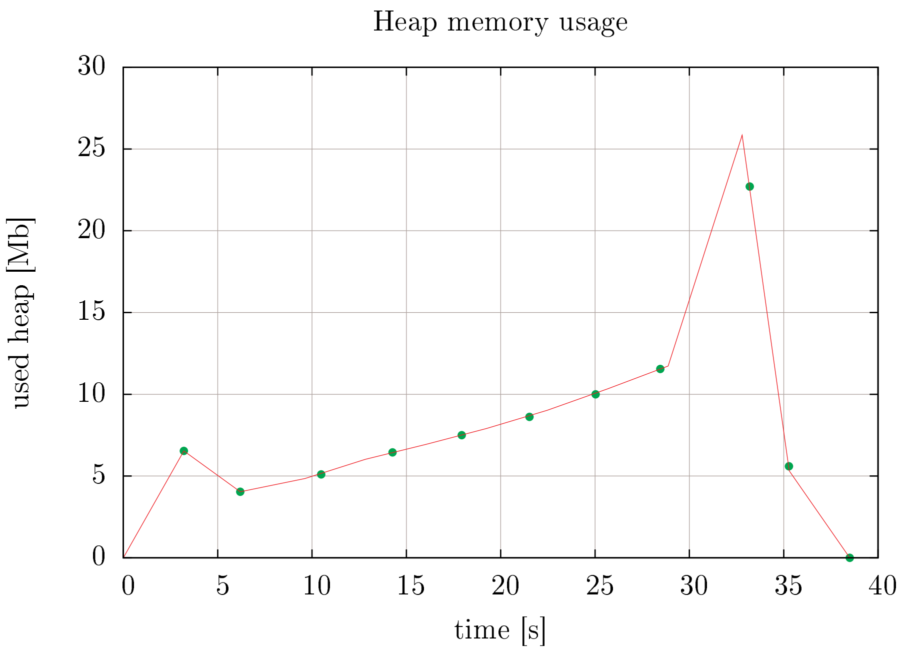


Figure 6.3: Heap memory usage in time - Hamiltonian Path imperative implementation for graph of 40 nodes, maximal node grade 5 with info (green points), when was GC performed

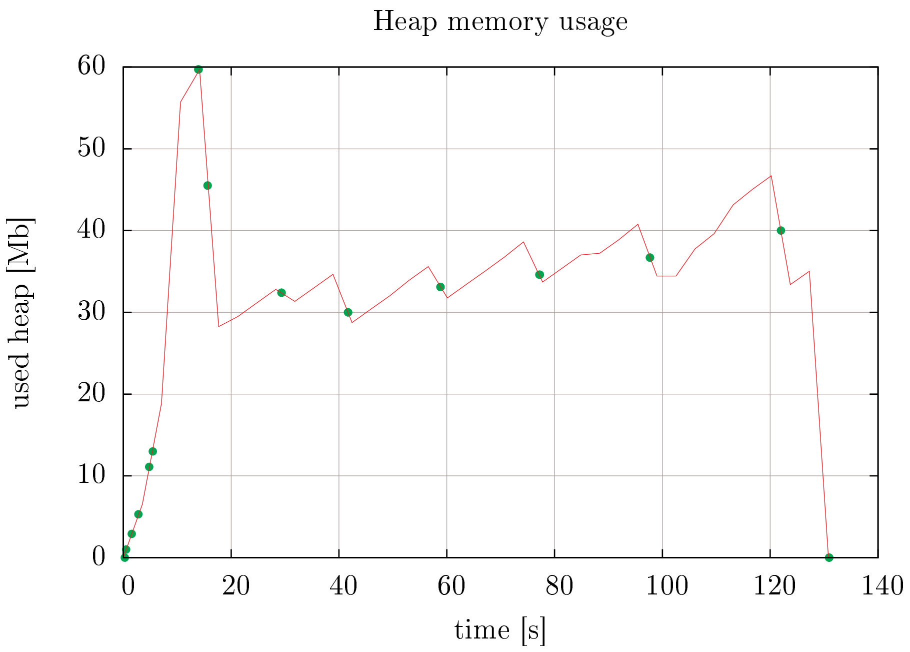


Figure 6.4: Heap memory usage in time - Hamiltonian Path Squander implementation for graph of 40 nodes, maximal node grade 5 with info (green points), when was GC performed

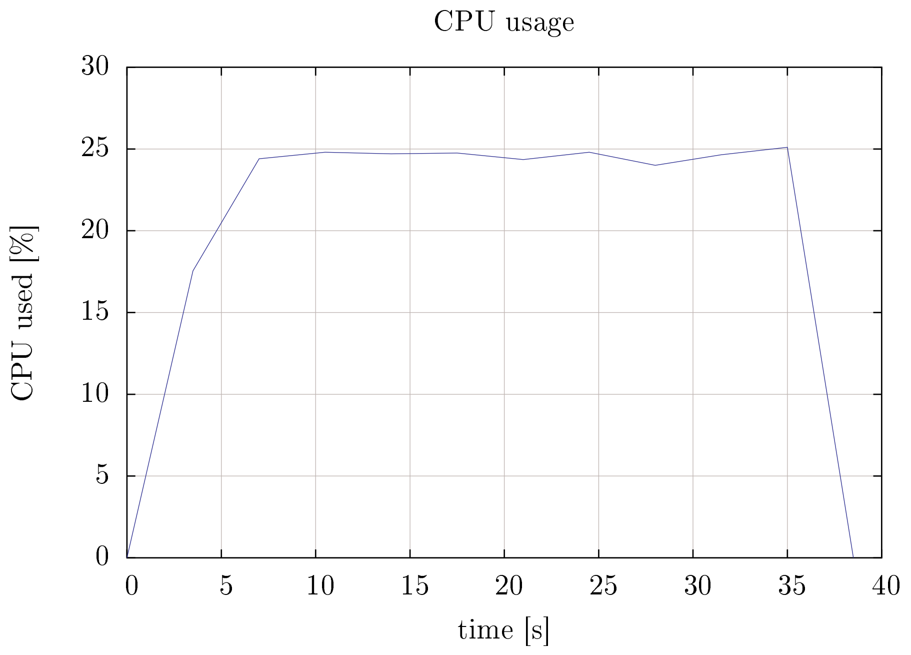


Figure 6.5: CPU usage in time - Hamiltonian Path imperative implementation for graph of 40 nodes and maximal node grade 5

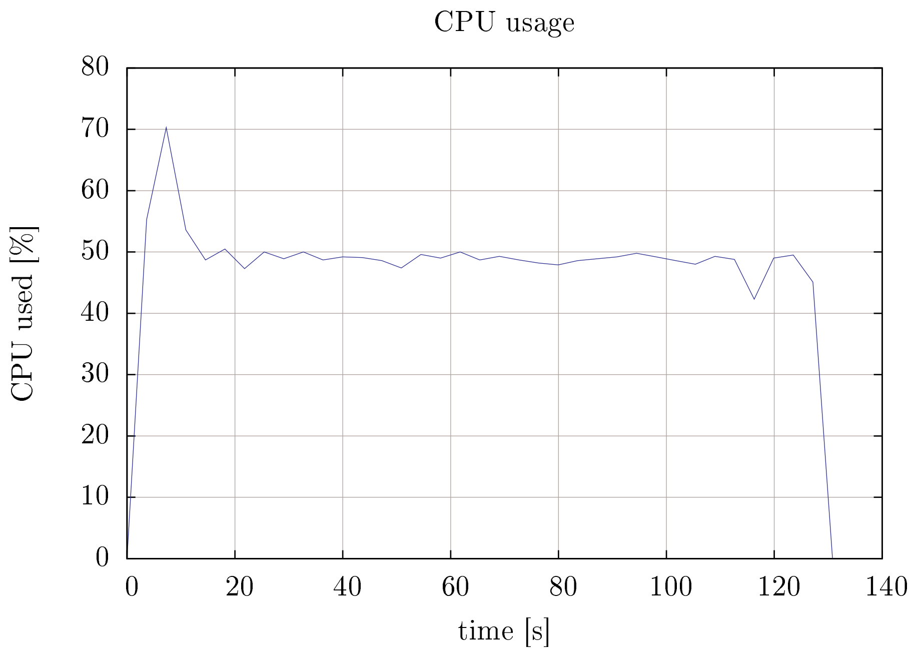


Figure 6.6: CPU usage in time - Hamiltonian Path Squander implementation for graph of 40 nodes and maximal node grade 5

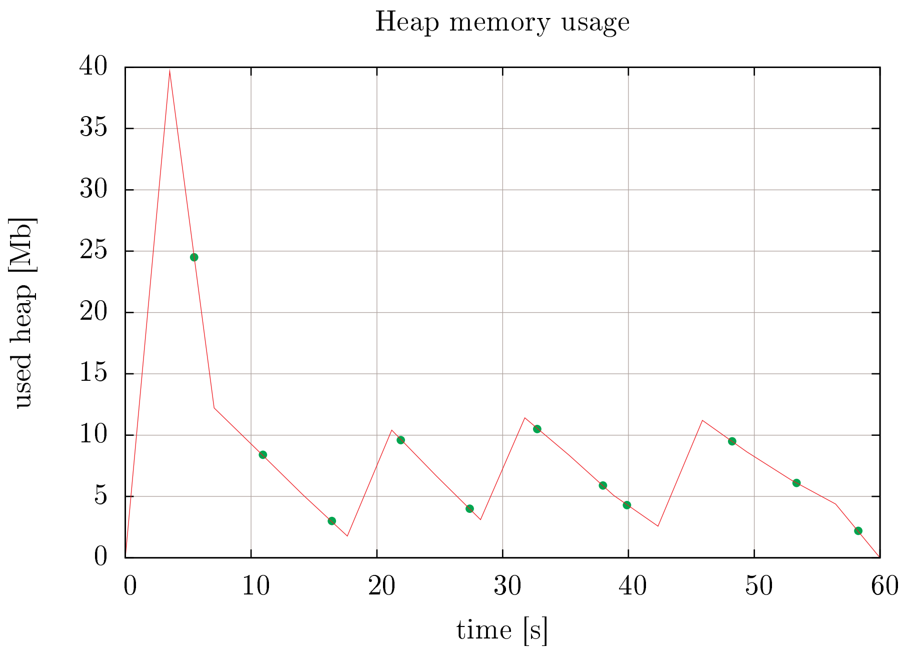


Figure 6.7: Heap memory usage in time - Hamiltonian Path imperative implementation for graph of 50 nodes, maximal node grade 4 with info (green points), when was GC performed

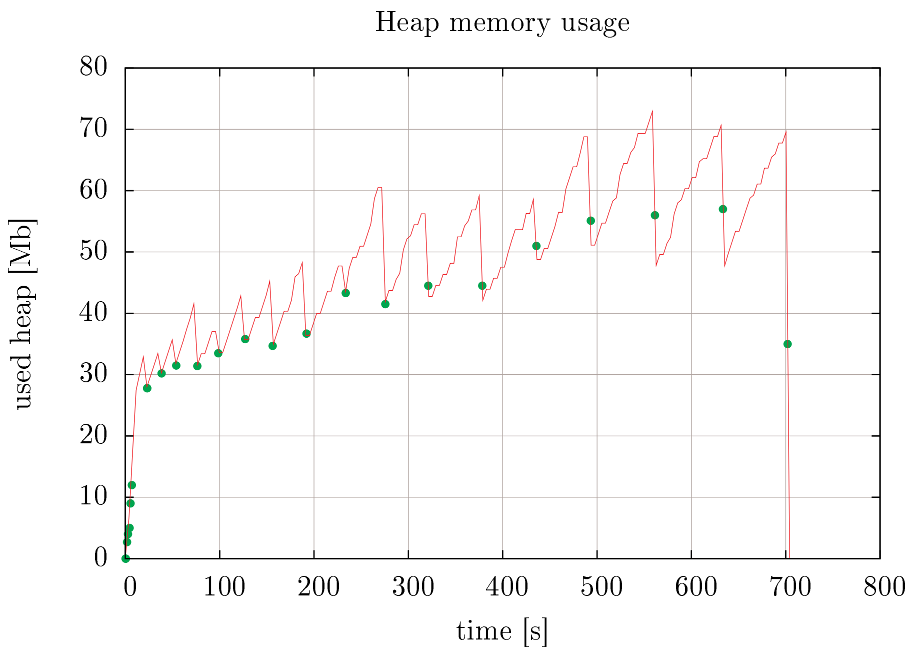


Figure 6.8: Heap memory usage in time - Hamiltonian Path Sqaunder implementation for graph of 50 nodes, maximal node grade 4 with info (green points), when was GC performed

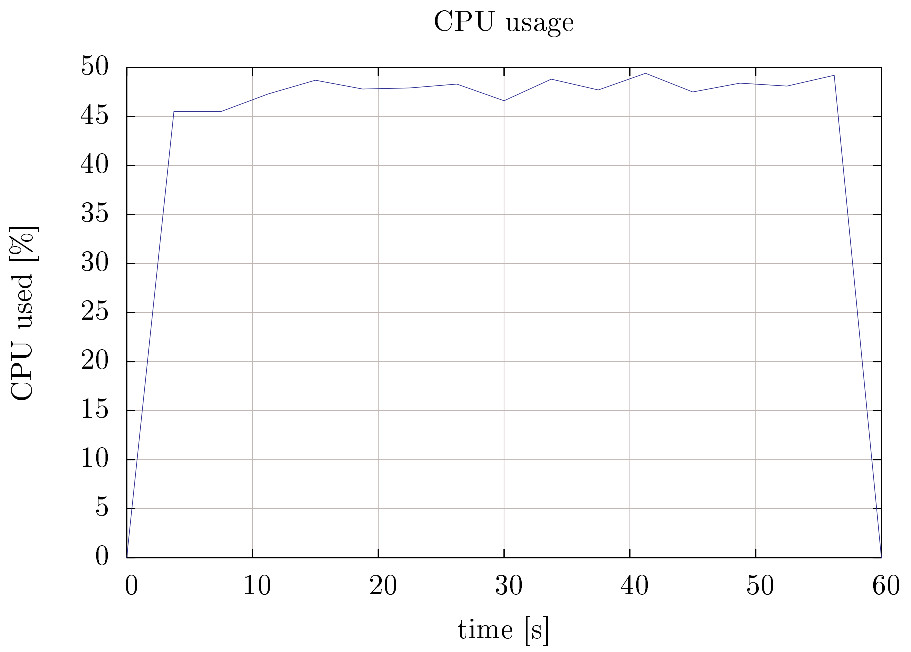


Figure 6.9: CPU usage in time - Hamiltonian Path imperative implementation for graph of 50 nodes and maximal node grade 4

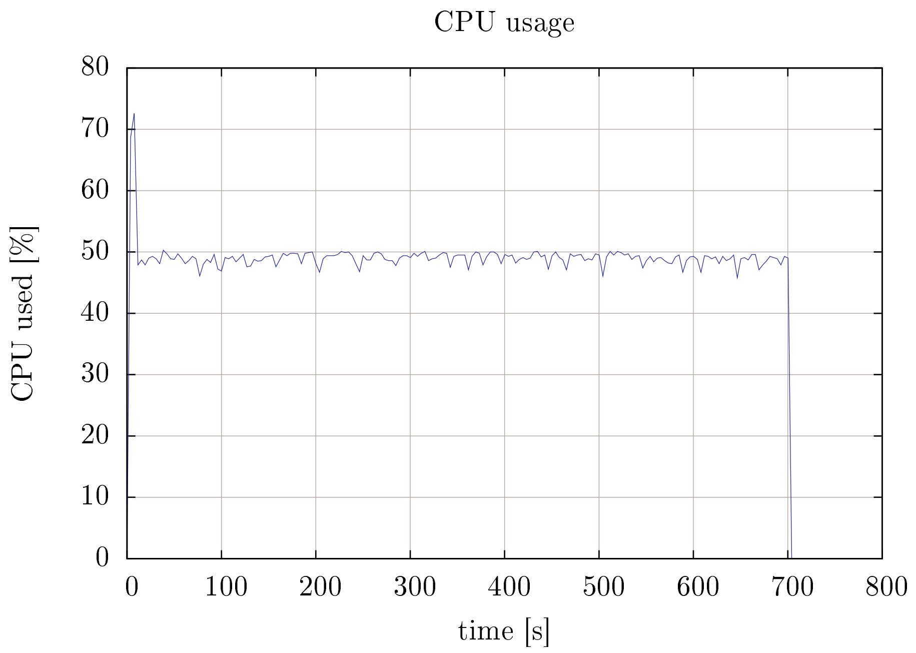


Figure 6.10: CPU usage in time - Hamiltonian Path Squander implementation for graph of 50 nodes and maximal node grade 4

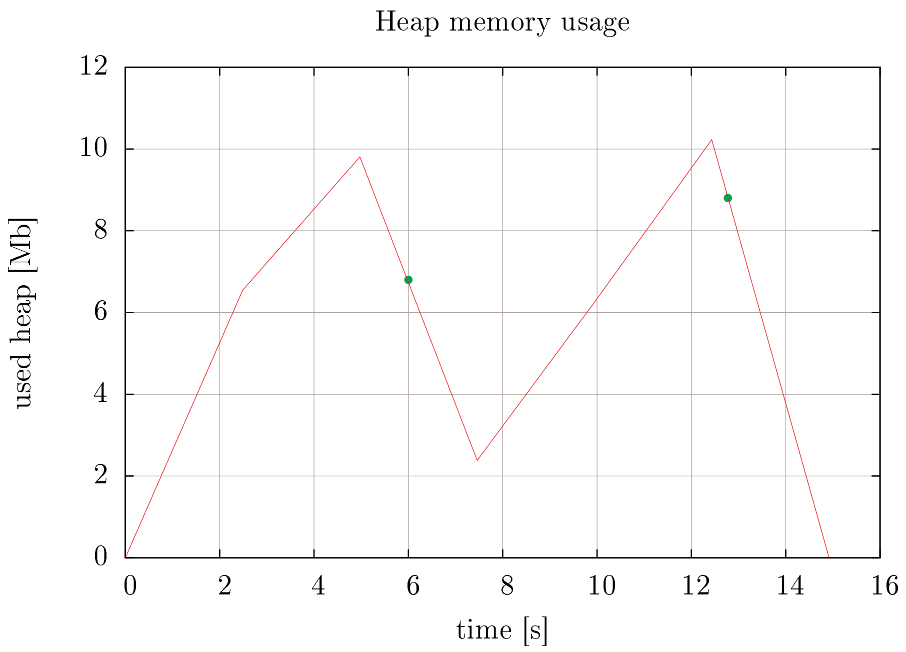


Figure 6.11: Heap memory usage in time - Hamiltonian Path imperative implementation for graph of 80 nodes, maximal node grade 3 with info (green points), when was GC performed

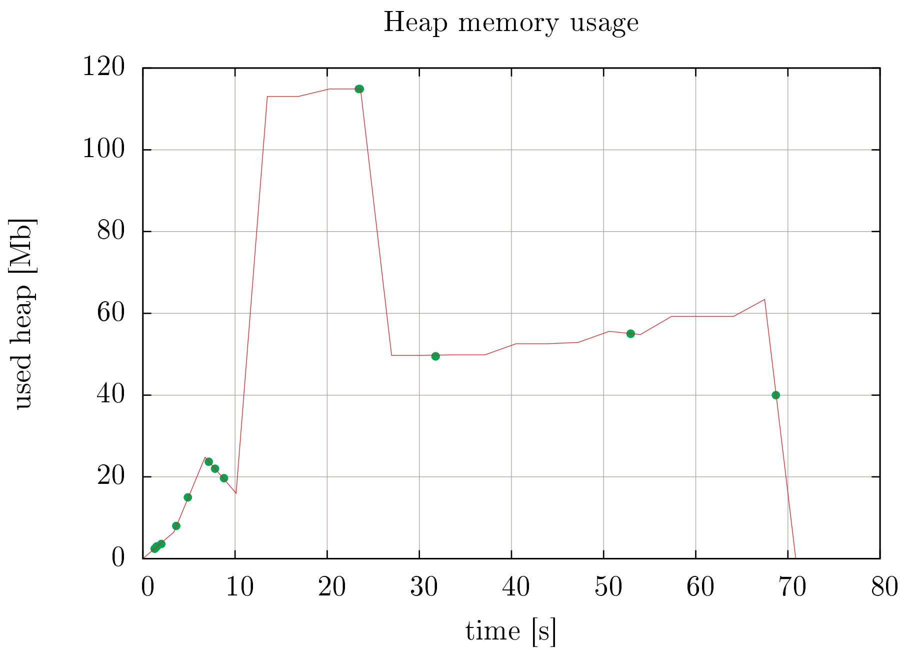


Figure 6.12: Heap memory usage in time - Hamiltonian Path Sqaunder implementation for graph of 80 nodes, maximal node grade 3 with info (green points), when was GC performed

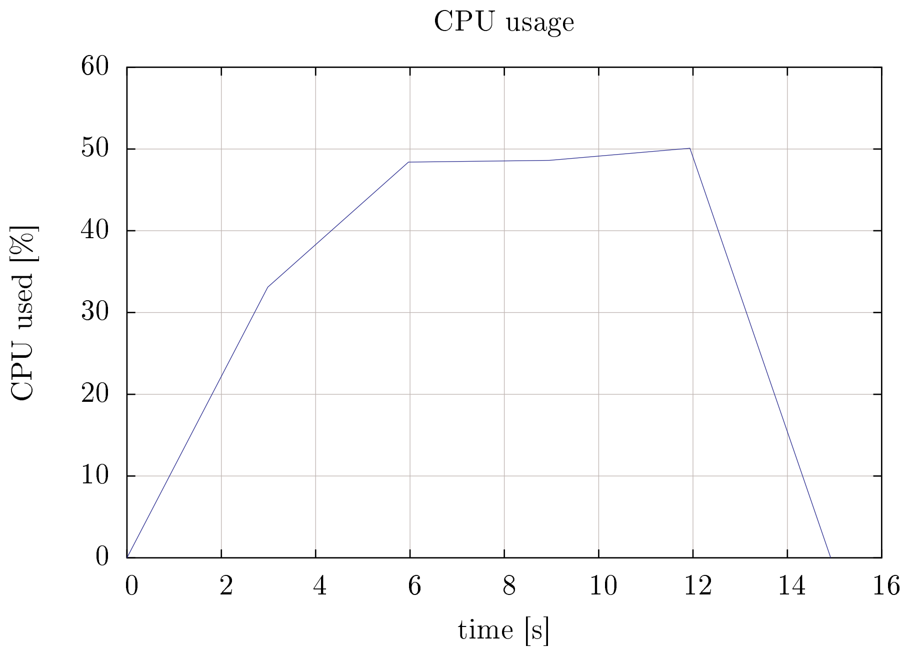


Figure 6.13: CPU usage in time - Hamiltonian Path imperative implementation for graph of 80 nodes and maximal node grade 3

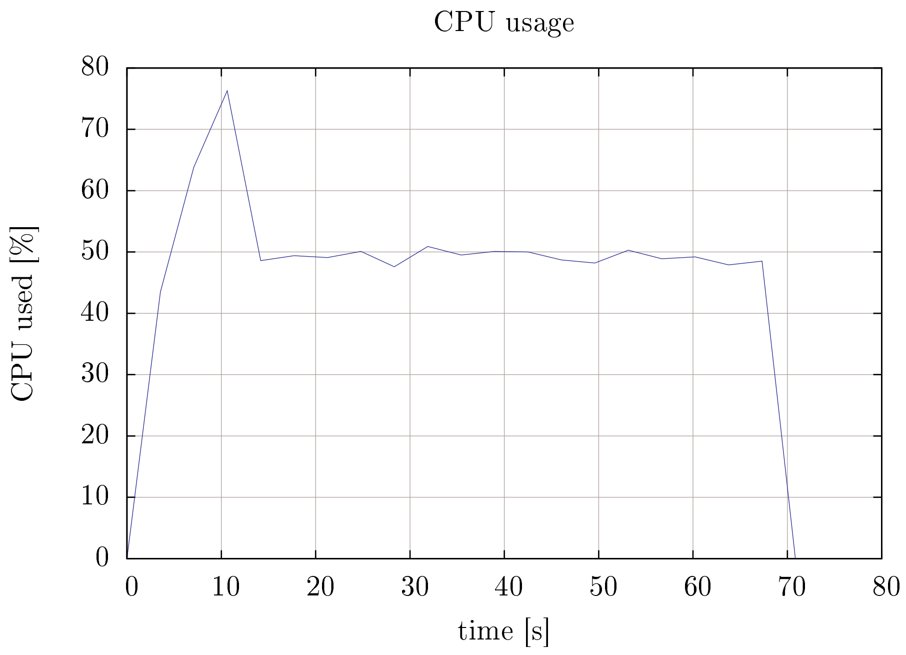


Figure 6.14: CPU usage in time - Hamiltonian Path Squander implementation for graph of 80 nodes and maximal node grade 3

6.2 Knapsack Problem

Imperative implementation of Knapsack algorithm can be done in many ways like brute force with or without branch & bounds criterium, dynamic programming or by some heuristic (e.g. quotient price to weight of actual thing), which could be sometimes inaccurate. I have choosed for those tests probably most common way of implementation by brute force branch & bounds depth first search (BB-DFS) - see Listings 6.2. Object `pw` is instance of class`PriceWeight` and holds actual weight and price for the configuration.

```java
private void countBruteForcePrice(int rank) {
  for (int i = rank; i < this.countOfObjects; i++) {
    putRestOfThingsIntoKnapsack(i);
    this.actualPrice = this.pw.getPrice();
    this.actualWeight = this.pw.getWeight();
    if (this.actualPrice < this.bruteForcePrice)
      return;
    this.configuration[i] = 1;
    recalculatePriceandWeightInBag();
    this.actualPrice = this.pw.getPrice();
    this.actualWeight = this.pw.getWeight();
    if (this.actualWeight <= this.maxWeight) {
      if (this.bruteForcePrice < this.actualPrice) {
        this.bruteForcePrice = this.actualPrice;
        this.bruteForceWeight = this.actualWeight;
        for (int j = 0; j < this.configuration.length; j++) {
          this.bestConfiguration[j] = this.configuration[j];
        }
        if (this.bruteForcePrice >= this.treshold)
          return;
      }
      countBruteForcePrice(i + 1);
    }
    this.configuration[i] = 0;
  }
  return;
}
```

Listing 6.2: Imperative implementation of Knapsack Problem algorithm

When measuring Knapsack implementation, it is important to emphasize, that execution time depends on resulted configuration of things in knapsack. What does it exactly means? E.g. when computing configuration fulfilling treshold condition on set of things, from which is in the resulted configuration only last thing, imperative algorithm implemented by branch and bounds-brute force has to search almost whole state space (except for branches, that are cut off) whereas Squander implementation started with empty configuration of things and test this configuration adding one thing after another when he finishes with all possibilities of actual number of things. Let us look at execution times in Table 6.3.

Table 6.3: Execution times of Knapsack algorithm implementation in ms for instances with „one last thing in solution“

	Number of things to choose from						
	25	27	30	32	35	37	40
Imperatively	8 220	35 130	300 910	1 314 750	10 952 150	**t/o**	**t/o**
Squander	1 440	1 440	1 460	1 640	1 530	3 010	3 130

That was extreme example of execution instance with „one last thing in solution“, but let us compare average time of execution on 50 instances - see Table 6.4. For imperative implementation is average execution time growing according to size of problem instance but this is not true for Squander.

Table 6.4: Average execution times of Knapsack algorithm implementation in ms

	Number of things to choose from						
	20	22	25	27	30	32	35
Imperatively	1.8	3.2	167.2	700.4	5 989.2	25 586.6	219 212.2
Squander	25 408.8	28 298.6	26 361.8	28 922.4	26 626.0	26 907.7	30 680.0

It appears that in Squander implementation does not depend on size of the instance so much. Let us look at following Table 6.5. From that table is obvious, that in Squander implementation depends execution time on number of choosed things in final solution more than on number of things to choose from. Table 6.3 gives us also information, that execution time of Squander implementation for instances with „one last thing in solution“ is little bit „suspiciously“ short.

Table 6.5: Dependency of execution times to number of things in final solution in Knapsack algorithm Squander implementation

	Number of things to choose from									
	20				27			32		
	number of ch. things				number of ch. things			number of ch. things		
	1	14	16	17	1	20	23	1	24	25
Time(ms)	1 060	7 970	27 520	29 130	1 550	28 700	30 110	1 460	27 600	29 940

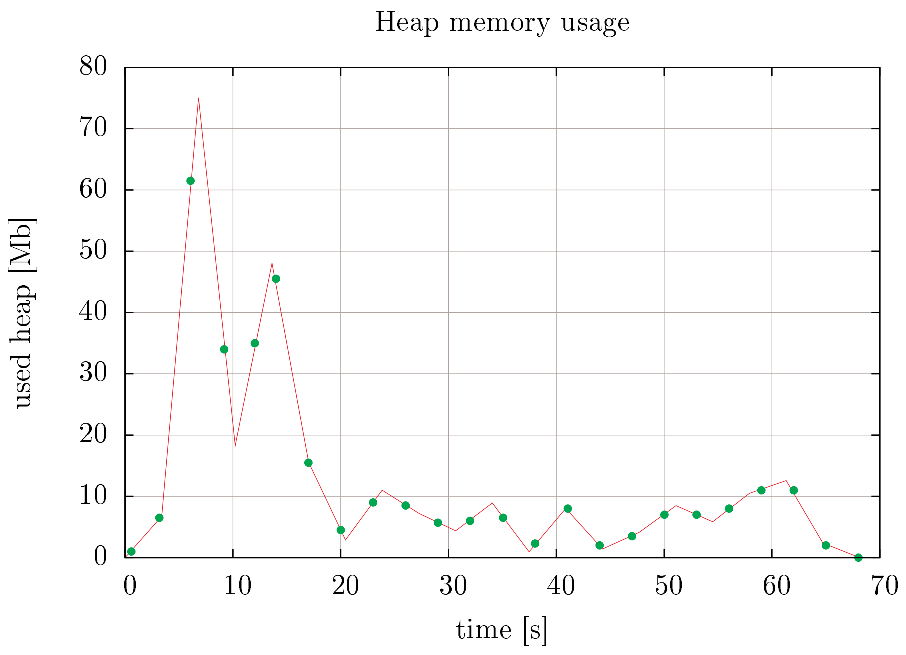


Figure 6.15: Heap memory usage in time - Hamiltonian Path imperative implementation for instance with 40 things and info (green points), when was GC performed

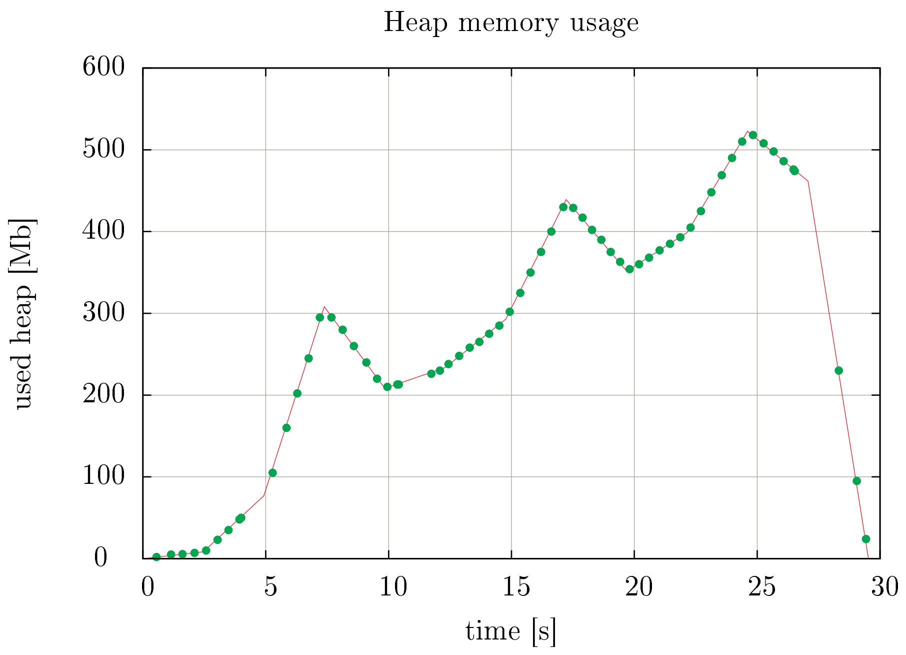


Figure 6.16: Heap memory usage in time - Knapsack Problem Squander implementation for instance with 40 things and info (green points), when was GC performed

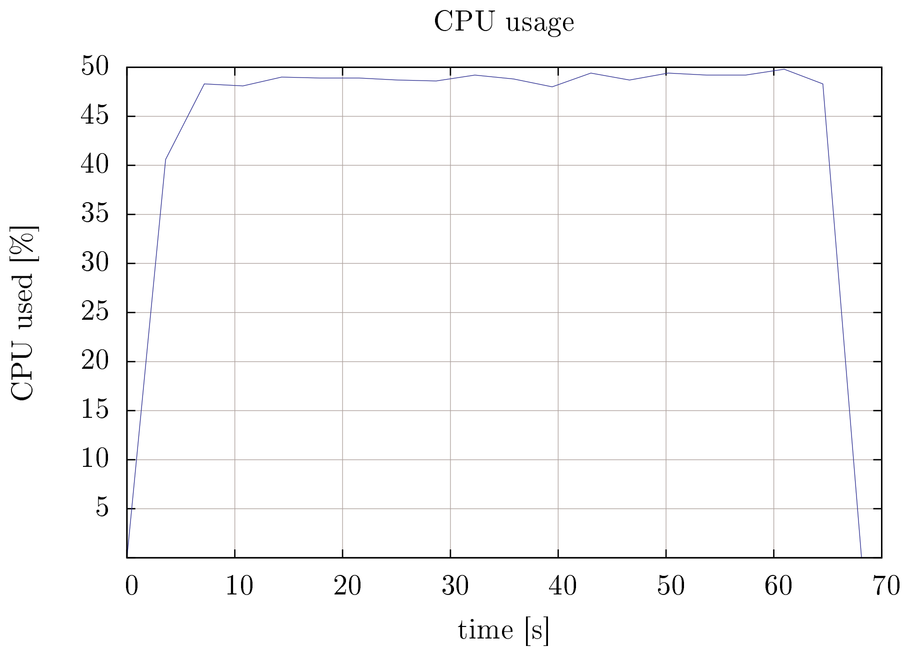


Figure 6.17: CPU usage in time - Knapsack Problem imperative implementation for instance with 40 things

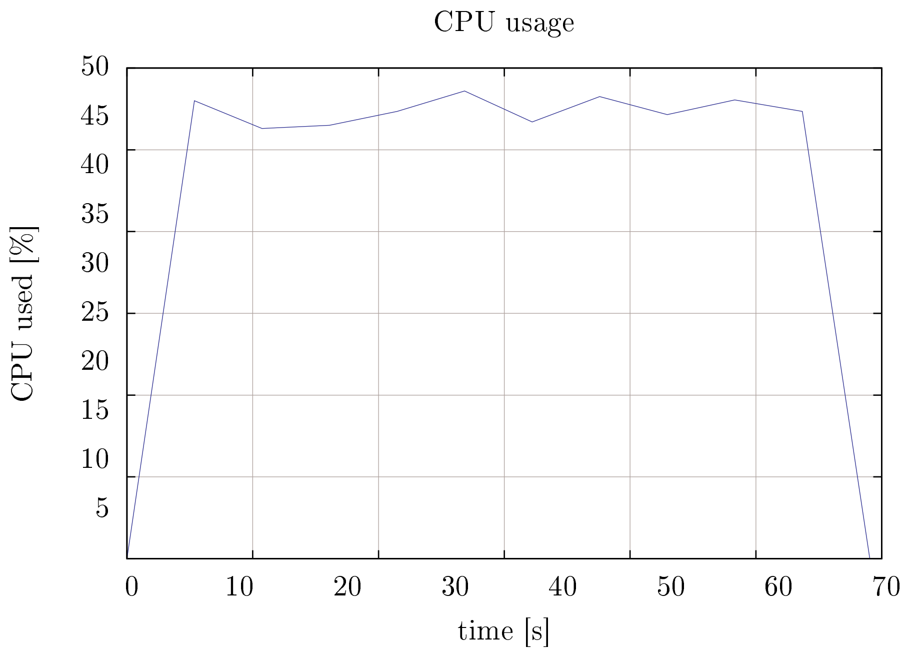


Figure 6.18: CPU usage in time - Hamiltonian Path Squander implementation for instance with 40 things

Now look at statistics about memory heap usage and CPU usage of both implementations of Knapsack Problem. CPU usage is for imperative implementation and Squander implementation little bit under level of 50 % almost all the time of execution, whereas heap memory uses Squander implementation much more than imperative implementation. In Squander implementation used memory reaches before the end of execution 520 Mb in constrast with imperative implementation, where is used 75 Mb of memory for heap at most. Garbage collector in Squander is also used many times - 2.6 % of execution time, whereas in imperative implementation it is 0.9 % in average! On the other hand Squander implementation holds maximum of the execution time for all instances around 30 seconds, but for some instances throws `outOfMemory Exception`, which is quite unpleasant.

6.3 L-dominant set of graph

Chapter 7

Conclusion

- Zhodnocení splnění cílů DP/BP a vlastního přínosu práce (při formulaci je třeba vzít v potaz zadání práce).
- Diskuse dalšího možného pokračování práce.

Bibliography

[1] Aleksandar Milicevic; MASSACHUSETTS INSTITUTE OF TECHNOLOGY, Department of Electrical Engineering and Computer Science; *Executable Specifications for Java Programs*, September 2010.

[2] Cay Horstmann and Gary Cornell; *Core Java*, 7th edition.

[3] Petr Jirků a kol.; *Programování v jazyku Prolog*, 1991.

[4] doc. RNDr. Josef Kolář, CSc.; *Jazyky pro umělou inteligenci*, 1994.

[5] Dr. Jaime Niño, Associate Professor of Computer Science; University of New Orleans; *Programming language structure - lectures*, 2011.

[6] Annotations;
`http://download.oracle.com/javase/1,5.0/docs/guide/language/annotations.html`

[7] Math is fun web;
`http://www.mathsisfun.com/games/triangle-peg-solitaire/index.html#`

[8] Oracle Java documentation - `ThreadMXBean`;
`http://download.oracle.com/javase/1,5,0/docs/api/`

[9] Oracle Java documentation - `java.util.Set`;
`http://download.oracle.com/javase/6/docs/api/java/util/Set.html`

Appendix A

Pokyny a návody k formátování textu práce

Tato příloha samozřejmě nebude součástí vaší práce. Slouží pouze jako příklad formátování textu.

Používat se dají všechny příkazy systému LaTeX. Existuje velké množství volně přístupné dokumentace, tutoriálů, příruček a dalších materiálů v elektronické podobě. Výchozím bodem, kromě Googlu, může být stránka CSTUG (Czech Tech Users Group) [**?**]. Tam najdete odkazy na další materiály. Vetšinou dostačující a přehledně organizovanou elektronikou dokumentaci najdete například na [**?**] nebo [**?**].

Existují i různé nadstavby nad systémy TeX a LaTeX, které výrazně usnadní psaní textu zejména začátečníkům. Velmi rozšířený v Linuxovém prostředí je systém Kile.

A.1 Vkládání obrázků

Obrázky se umísťují do plovoucího prostředí `figure`. Každý obrázek by měl obsahovat **název** (`\caption`) a **návěští** (`\label`). Použití příkazu pro vložení obrázku `\includegraphics` je podmíněno aktivací (načtením) balíku graphicx příkazem `\usepackage{graphicx}`.

Budete-li zdrojový text zpracovávat pomocí programu `pdflatex`, očekávají se obrázky s příponou `*.pdf`[1], použijete-li k formátování `latex`, očekávají se obrázky s příponou `*.eps`.[2]

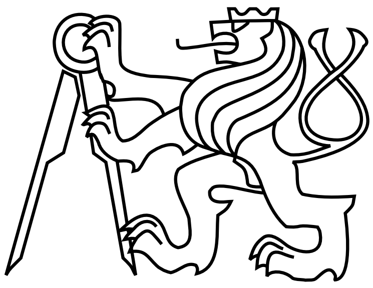


Figure A.1: Popiska obrázku

Příklad vložení obrázku:

```
\begin{figure}[h]
\begin{center}
\includegraphics[width=5cm]{figures/LogoCVUT}
\caption{Popiska obrazku}
\label{fig:logo}
\end{center}
\end{figure}
```

A.2 Kreslení obrázků

Zřejmě každý z vás má nějaký oblíbený nástroj pro tvorbu obrázků. Jde jen o to, abyste dokázali obrázek uložit v požadovaném formátu nebo jej do něj konvertovat (viz předchozí kapitola). Je zřejmě vhodné kreslit obrázky vektorově. Celkem oblíbený, na ovládání celkem jednoduchý a přitom dostatečně mocný je například program Inkscape.

Zde stojí za to upozornit na kreslící programe Ipe [**?**], který dokáže do obrázku vkládat komentáře přímo v latexovském formátu (vzroce, stejné fonty atd.). Podobné věci umí na Linuxové platformě nástroj Xfig.

Za pozornost ještě stojí schopnost editoru Ipe importovat obrázek (jpg nebo bitmap) a krelit do něj latexovské popisky a komentáře. Výsledek pak umí exportovat přímo do pdf.

[1] pdflatex umí také formáty PNG a JPG.

[2] Vzájemnou konverzi mezi snad všemi typy obrazku včetně změn vekostí a dalších vymožeností vám může zajistit balík ImageMagic (http://www.imagemagick.org/script/index.php). Je dostupný pod Linuxem, Mac OS i MS Windows. Důležité jsou zejména příkazy convert a identify.

DTD	**construction**	**elimination**
$\mid$	`in1|A|B a:sum A B`	`case([_:A]a)([_:B]a)ab:A`
	`in1|A|B b:sum A B`	`case([_:A]b)([_:B]b)ba:B`
$+$	`do_reg:A -> reg A`	`undo_reg:reg A -> A`
$*,?$	the same like $\mid$ and $+$	the same like $\mid$ and $+$
	with `emtpy_el:empty`	with `emtpy_el:empty`
R(a,b)	`make_R:A->B->R`	`a: R -> A`
		`b: R -> B`

Table A.1: Ukázka tabulky

A.3 Tabulky

Existuje více způsobů, jak sázet tabulky. Například je možno použít prostředí `table`, které je velmi podobné prostředí `figure`.

Zdrojový text tabulky A.1 vypadá takto:

```
\begin{table}
\begin{center}
\begin{tabular}{|c|l|l|}
\hline
\textbf{DTD} & \textbf{construction} & \textbf{elimination} \\
\hline
$\mid$ & \verb+in1|A|B a:sum A B+ & \verb+case([_:A]a)([_:B]a)ab:A+\\
&\verb+in1|A|B b:sum A B+ & \verb+case([_:A]b)([_:B]b)ba:B+\\
\hline
$+$&\verb+do_reg:A -> reg A+&\verb+undo_reg:reg A -> A+\\
\hline
$*,?$& the same like $\mid$ and $+$ & the same like $\mid$ and $+$\\
& with \verb+emtpy_el:empty+ & with \verb+emtpy_el:empty+\\
\hline
R(a,b) & \verb+make_R:A->B->R+ & \verb+a: R -> A+\\
 & & \verb+b: R -> B+\\
\hline
\end{tabular}
\end{center}
\caption{Ukázka tabulky}
\label{tab:tab1}
\end{table}
\begin{table}
```

A.4 Odkazy v textu

A.4.1 Odkazy na literaturu

Jsou realizovány příkazem `\cite{odkaz}`.

Seznam literatury je dobré zapsat do samostatného souboru a ten pak zpracovat programem bibtex (viz soubor `reference.bib`). Zdrojový soubor pro `bibtex` vypadá například takto:

```
@Article{Chen01,
  author  = "Yong-Sheng Chen and Yi-Ping Hung and Chiou-Shann Fuh",
  title   = "Fast Block Matching Algorithm Based on
             the Winner-Update Strategy",
  journal = "IEEE Transactions On Image Processing",
  pages   = "1212--1222",
  volume  =  10,
  number  =   8,
  year    = 2001,
}

@Misc{latexdocweb,
  author  = "",
  title   = "{\LaTeX} --- online manuál",
  note    = "\verb|http://www.cstug.cz/latex/lm/frames.html|",
  year    = "",
}
...
```

Pozor: Sazba názvů odkazů je dána BibTEX stylem (`\bibliographystyle{abbrv}`). BibTEX tedy obvykle vysází velké pouze počáteční písmeno z názvu zdroje, ostatní písmena zůstanou malá bez ohledu na to, jak je napíšete. Přesněji řečeno, styl může zvolit pro každý typ publikace jiné konverze. Pro časopisecké články třeba výše uvedené, jiné pro monografie (u nich často bývá naopak velikost písmen zachována).

Pokud chcete BibTEXu napovědět, která písmena nechat bez konverzí (viz `title = "{\LaTeX} --- online manuál"` v předchozím příkladu), je nutné příslušné písmeno (zde celé makro) uzavřít do složených závorek. Pro přehlednost je proto vhodné celé parametry uzavírat do uvozovek (`author = "..."`), nikoliv do složených závorek.

Odkazy na literaturu ve zdrojovém textu se pak zapisují:

```
Podívejte se na \cite{Chen01},
další detaily najdete na \cite{latexdocweb}
```

Vazbu mezi soubory `*.tex` a `*.bib` zajistíte příkazem `\bibliography{}` v souboru `*.tex`. V našem případě tedy zdrojový dokument `thesis.tex` obsahuje příkaz `\bibliography{reference}`.

Zpracování zdrojového textu s odkazy se provede postupným voláním programů `pdflatex <soubor>` (případně `latex <soubor>`), `bibtex <soubor>` a opět `pdflatex <soubor>`.[3]

Níže uvedený příklad je převzat z dříve existujících pokynů studentům, kteří dělají svou diplomovou nebo bakalářskou práci v Grafické skupině.[4] Zde se praví:

```
...
j) Seznam literatury a dalších použitých pramenů, odkazy na WWW stránky, ...
 Pozor na to, že na veškeré uvedené prameny se musíte v textu práce
 odkazovat -- [1].
Pramen, na který neodkazujete, vypadá, že jste ho vlastně nepotřebovali
a je uveden jen do počtu. Příklad citace knihy [1], článku v časopise [2],
stati ve sborníku [3] a html odkazu [4]:
[1] J. Žára, B. Beneš;, and P. Felkel.
     Moderní počítačová grafika. Computer Press s.r.o, Brno, 1 edition, 1998.
     (in Czech).
[2] P. Slavík. Grammars and Rewriting Systems as Models for Graphical User
     Interfaces. Cognitive Systems, 4(4--3):381--399, 1997.
[3] M. Haindl, Š. Kment, and P. Slavík. Virtual Information Systems.
     In WSCG'2000 -- Short communication papers, pages 22--27, Pilsen, 2000.
     University of West Bohemia.
[4] Knihovna grafické skupiny katedry počítačů:
     http://www.cgg.cvut.cz/Bib/library/
```

[3]První volání `pdflatex` vytvoří soubor s koncovkou `*.aux`, který je vstupem pro program `bibtex`, pak je potřeba znovu zavolat program `pdflatex` (`latex`), který tentokrát zpracuje soubory s příponami `.aux` a `.tex`. Informaci o případných nevyřešených odkazech (cross-reference) vidíte přímo při zpracovávání zdrojového souboru příkazem `pdflatex`. Program `pdflatex` (`latex`) lze volat vícekrát, pokud stále vidíte nevyřešené závislosti.

[4]Několikrát jsem byl upozorněn, že web s těmito pokyny byl zrušen, proto jej zde přímo necituji. Nicméně příklad sám o sobě dokumentuje obecně přijímaný konsensus ohledně citací v bakalářských a diplomových pracích na KP.

... abychom výše citované odkazy skutečně našli v (automaticky generovaném) seznamu literatury tohoto textu, musíme je nyní alespoň jednou citovat: Kniha [?], článek v časopisu [?], příspěvek na konferenci [?], www odkaz [?].

Ještě přidáme další ukázku citací online zdrojů podle české normy. Odkaz na wiki o frameworcich [?] a ORM [?]. Použití viz soubor `reference.bib`. V seznamu literatury by nyní měly být živé odkazy na zdroje. V `reference.bib` je zcela nový typ publikace. Detaily dohledal a dodal Petr Dlouhý v dubnu 2010. Podrobnosti najdete ve zdrojovém souboru tohoto textu v komentáři u příkazu `\thebibliography`.

A.4.2 Odkazy na obrázky, tabulky a kapitoly

- Označení místa v textu, na které chcete později čtenáře práce odkázat, se provede příkazem `\label{navesti}`. Lze použít v prostředích `figure` a `table`, ale též za názvem kapitoly nebo podkapitoly.
- Na návěští se odkážeme příkazem `\ref{navesti}` nebo `\pageref{navesti}`.

A.5 Rovnice, centrovaná, číslovaná matematika

Jednoduchý matematický výraz zapsaný přímo do textu se vysází pomocí prostředí `math`, resp. zkrácený zápis pomocí uzavření textu rovnice mezi znaky `$`.

Kód `$ S = \pi * r^2 $` bude vysázen takto: $S = \pi * r^2$.

Pokud chcete nečíslované rovnice, ale umístěné centrovaně na samostatné řádky, pak lze použít prostředí `displaymath`, resp. zkrácený zápis pomocí uzavření textu rovnice mezi znaky `$$`. Zdrojový kód: `|$$ S = \pi * r^2 $$|` bude pak vysázen takto:

$$S = \pi * r^2$$

Chcete-li mít rovnice číslované, je třeba použít prostředí `eqation`. Kód:

```
\begin{equation}
  S = \pi * r^2
\end{equation}

\begin{equation}
  V = \pi * r^3
\end{equation}
```

je potom vysázen takto:

$$S = \pi * r^2 \tag{A.1}$$

$$V = \pi * r^3 \tag{A.2}$$

A.6 Kódy programu

Chceme-li vysázet například část zdrojového kódu programu (bez formátování), hodí se prostředí `verbatim`:

```
         (* nickname2 *)
Lego> Refine in1
             (do_reg (nickname1 h));
Refine by  in1 (do_reg (nickname1 h))
   ?4 : pcdata
   ?5 : pcdata
          (* surname2 *)
Lego> Refine surname1 h;
Refine by  surname1 h
   ?5 : pcdata
          (* email2 *)
Lego> Refine undo_reg (email1 h);
Refine by  undo_reg (email1 h)
*** QED ***
```

A.7 Další poznámky

A.7.1 České uvozovky

V souboru `k336_thesis_macros.tex` je příkaz `\uv{}` pro sázení českých uvozovek. „Text uzavřený do českých uvozovek.“

Appendix B

Seznam použitých zkratek

2D	Two-Dimensional
ABN	Abstract Boolean Networks
ASIC	Application-Specific Integrated Circuit

⋮

Appendix C

UML diagramy

Tato příloha není povinná a zřejmě se neobjeví v každé práci. Máte-li ale větší množství podobných diagramů popisujících systém, není nutné všechny umísťovat do hlavního textu, zvláště pokud by to snižovalo jeho čitelnost.

Appendix D

Instalační a uživatelská příručka

Tato příloha velmi žádoucí zejména u softwarových implementačních prací.

Appendix E

Obsah přiloženého CD

Tato příloha je povinná pro každou práci. Každá práce musí totiž obsahovat přiložené CD. Viz dále.

Může vypadat například takto. Váš seznam samozřejmě bude odpovídat typu vaší práce. (viz [**?**]):

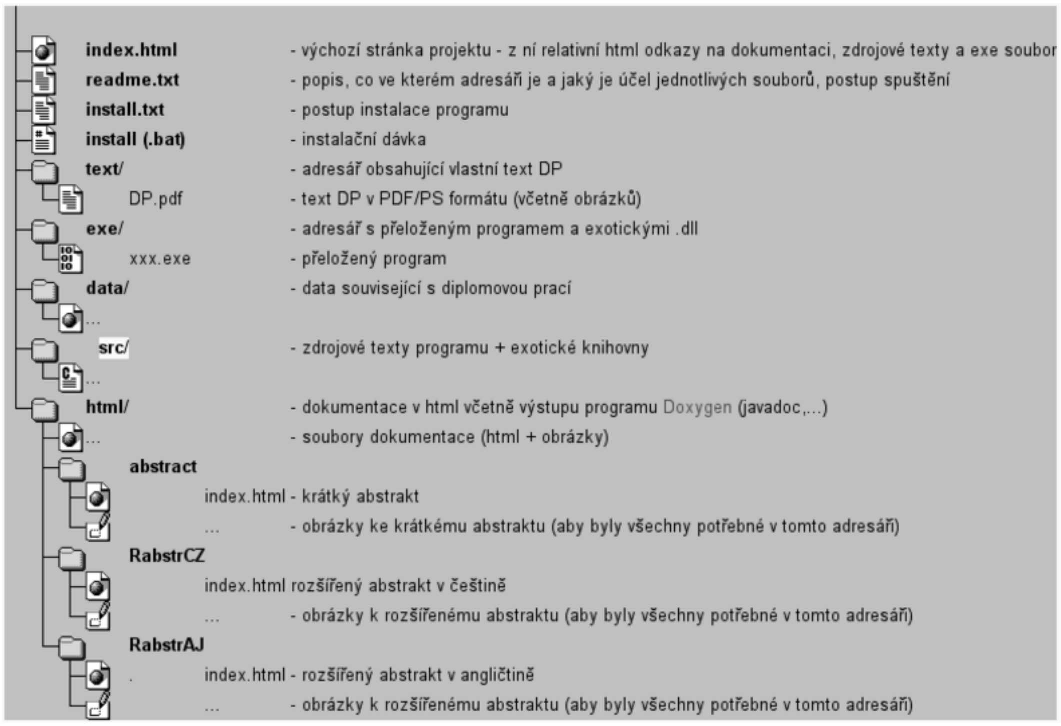
Na GNU/Linuxu si strukturu přiloženého CD můžete snadno vyrobit příkazem:

```
$ tree . >tree.txt
```

Ve vzniklém souboru pak stačí pouze doplnit komentáře.

Z **README.TXT** (případne index.html apod.) musí být rovněž zřejmé, jak programy instalovat, spouštět a jaké požadavky mají tyto programy na hardware.

Adresář **text** musí obsahovat soubor s vlastním textem práce v PDF nebo PS formátu, který bude později použit pro prezentaci diplomové práce na WWW.



```
index.html          - výchozí stránka projektu - z ní relativní html odkazy na dokumentaci, zdrojové texty a exe soubor
readme.txt          - popis, co ve kterém adresáři je a jaký je účel jednotlivých souborů, postup spuštění
install.txt         - postup instalace programu
install (.bat)      - instalační dávka
text/               - adresář obsahující vlastní text DP
    DP.pdf          - text DP v PDF/PS formátu (včetně obrázků)
exe/                - adresář s přeloženým programem a exotickými .dll
    xxx.exe         - přeložený program
data/               - data související s diplomovou prací
  ...
src/                - zdrojové texty programu + exotické knihovny
  ...
html/               - dokumentace v html včetně výstupu programu Doxygen (javadoc,...)
  ...               - soubory dokumentace (html + obrázky)
    abstract
        index.html  - krátký abstrakt
        ...         - obrázky ke krátkému abstraktu (aby byly všechny potřebné v tomto adresáři)
    RabstrCZ
        index.html rozšířený abstrakt v češtině
        ...         - obrázky k rozšířenému abstraktu (aby byly všechny potřebné v tomto adresáři)
    RabstrAJ
      . index.html  - rozšířený abstrakt v angličtině
        ...         - obrázky k rozšířenému abstraktu (aby byly všechny potřebné v tomto adresáři)
```

Figure E.1: Seznam přiloženého CD — příklad