Search and Conquer (*SaC*)
— Java Library for Searching Graphs and Game Trees

User Guide

Przemysław Klęsk and Marcin Korzeń

Faculty of Computer Science and Information Technology
West Pomeranian University of Technology
Żołnierska 49, 71-210 Szczecin, Poland

INNOWACYJNA GOSPODARKA
NARODOWA STRATEGIA SPÓJNOŚCI

CEWI
Platforma Informatyczna

UNIA EUROPEJSKA
EUROPEJSKI FUNDUSZ
ROZWOJU REGIONALNEGO

Contents

Preface

Search algorithms constitute a significant part in studies on artificial intelligence. From a historical perspective, *pathfinding* and *chess* come to mind as probably two most natural examples where search algorithms have been applicable with success. These examples are also good representatives for the two general groups of search problems that the *SaC* library attempts to solve, namely:

1. **discrete (combinatorial) optimization problems**, where a problem can be represented as a *graph of states*, and finding its solution brings down to searching;

2. **two-person games problems**, where one looks for the best move or decision in a game or a situation of conflict, which can be represented as a *game tree*.

Fig. 1[1] illustratively depicts the two groups.

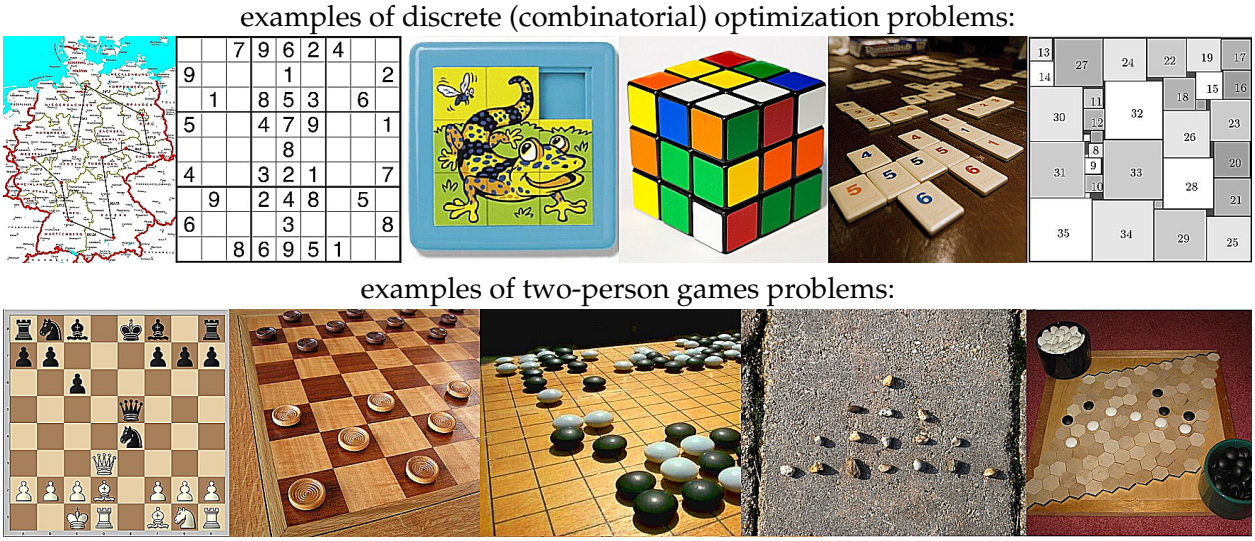


Figure 1: Illustration of two groups of search problems tackled by *SaC*.

As regards the first group, on one hand it includes various types of geographical or physical graphs where some object can move from one location to another. One may think here of examples

[1]Images acquired from the *Google Images* search engine.

like: shortest path finding (for different means of transport), routing problems (internet routing, vehicle routing), traversing mazes, etc. On the other hand, this group includes also a variety of abstract graphs related to puzzles or riddles, combinatorial by nature, where by certain *manipulations* we can change the state of some object. We would like to discover a sequence of such manipulations which brings the object to a solution state with some desired properties. One may think here of many recreation examples like: Rubik's cube, sliding puzzle, sudoku, solitaires; but also of more practical problems in technology like: spatial packing problems (2D or 3D), optimal material cutting, arranging schedules, resource planning, etc.

As regards the second group, the straightforward examples are mind games e.g.: chess, checkers, GO, Hex, Nim, tic-tac-toe, and many others. Beside those, one may indicate some examples taken from the game theory, like the famous prisonner's dilemma, but also: bargaining problems, political conflicts, competition wars between companies, etc.

In games, a player at a given game position (a state) typically has some number of possible *moves* at disposal which transfer the game to new positions. In each of these positions the opponent has a number of counter-moves, and this scheme continues. A tree structure representing the game arises in a natural manner. Due to the exponential growth of game trees, computer programs are in practice limited to going over only a small portion of a whole tree. Usually, the analysis must be ended up after just several levels of depth. Then, by assigning some numerical evaluations to terminal positions — being the consequences of moves made at the root of the tree — the algorithm is able to indicate the most promising move. It is worth to remark that not only mind games can be analyzed in this manner. Many computer games, usually strategy games, but possibly even arcade ones or shooters, can be subjected to that scheme if only rules of a game are well defined and a finite set of moves can be formulated at decision moments.

Aim and purposes

In the programmer's eye search algorithms share many common elements, despite some differences (usually minor ones). Therefore, the main idea behind the *Search and Conquer (SaC)* library was to propose a set of interfaces and classes — the API — which would unify and facilitate the way search algorithms are carried out. Java was the selected language for the library. Although is it easy to look up on the Internet examples of implementations (also in Java) of some search algorithms, these examples are typically disconnected from one another, and coded with the focus on a specific problem, so with little generality.

The intention of authors was to design the *SaC* library in such a manner that its future users, not necessarily skilled programmers, would be able to 'hook up' with ease their own specific problems to the library. By that we mailny mean that to formulate and to solve a particular search problem requires possibly very little of the programming effort on the user's part.

Currrently, in *SaC* there are nine implementations, ready to be executed, of well known search algorithms — six for graphs (*Breadth-first search*, *Depth-first search*, *Dijkstra's algorithm*, *Best-first search*, *A**, *IDA**) and three for games (*Min-Max*, *alpha-beta pruning*, *Scout*). At disposal are also some extras (configuration options, selection of data structures involved) that lead to variations of the default search performance. Additionally, the library comes with a set of examples included in

the `sac.examples` package. Among them there are console solvers for: sliding puzzle, traveling salesman problem, sudoku; and two-person game applications (with a simple GUI) for: checkers and Nim.

The central 'actor' in *SaC* is the `sac.State` interface. It represents an *abstract state* in some graph or some game tree. The rest of the object oriented API is built up around this interface.

From the user's perspective, in order to execute a search algorithm it is sufficient to describe the particular search problem by defining — i.e. providing implementations for — the following elements.

1. **Generation of descendants** — *What new states (direct descendants) can be generated from a given state?*
2. **Identification** — *What identifiers (string or integer representations) can be assigned to states, so that the same state is not visited multiple times unnecessarily?*
3. **Termination** — *Is given state a terminal? I.e. a solution state (graphs) or a win state (game trees).*

3′. **Heuristics (optional)** — *An estimation how far a state is from the solution (graphs), or an evaluation whether the state represents some advantage for the maximizing or the minimizing player (game trees).*

In *SaC*, suitable places are prepared for the above purposes. We have assigned the number 3′ to the last element — heuristics, since it can be viewed as an extension of the element no. 3 — termination.

We see three potential groups of users that might be interested in *SaC*. Firstly — people, not necessarily computer scientists, needing to solve some engineering or technological optimization problem. Secondly — computer games programmers, in particular programmers of games for mobile devices (the choice of Java may be a friendly factor). Thirdly — academic communities, using the mentioned algorithms in research or didactics.

Acknownledgements

The *SaC* library has been developped as a subproject within a larger academic project named TEWI (Polish abbrevation for *Telekomunikacja Edukacja Wiedza Innowacje*, translatable as: Telecommuniaction Education Knowledge Innovations), financed by the European Fund for Regional Development (POIG.02.03.00-00-028/09), see `http://tewi.p.lodz.pl`.

The goal of TEWI was to create a web platform gathering open source software projects, from various domains, developped by Polish academic communities. The project has been conducted by a consortium managed by the Łódź University of Technology, and involved the following partners: Białystok Technical University, Warsaw University of Technology, Gdańsk University of Technology, Łódź Univeristy of Computer Sciences and Skills, Cracow University of Technology, West Pomeranian University of Technology in Szczecin, Polish-Japanese Institute of Information Technology.

As regards the *SaC* project itself, it has been programmed in the Java language, using the Java SE 7 (1.7.0) version. Java is a product and a trademark of the Oracle company (`http://www.oracle.com`).

The authors would like to thank the creators of the following projects and libraries that contributed to *SaC*:

- *Graphviz* — a free software for automatic graph drawing, see `http://www.graphviz.org`. In *SaC*, we enable a possibility to generate output text files, compliant with the Graphviz language. The files contain represenations of graphs or trees that were searched by a particular algorithm. Having such a file and the Graphviz software installed, the user can produce visualizations using one of Graphviz engines like `dot`, `neato`, etc. We used Graphviz in its 2.38 version. All graph / tree illustrations presented throughout this user guide have been produced using Graphviz;

- *jFreeChart* — a free library which allows to generate and display charts from the Java code, see `http://www.jfree.org/jfreechart`. In *SaC*, we take advantage of jFreeChart in two contexts: (1) when displaying graphical monitors showing the progress of an ongoing graph search procedure, (2) when producing bar charts or xy plots based on statistics collected from some large experiment (e.g. a comparison of multiple algorithms or heuristics on multiple instances of some search problem). We used jFreeChart in its 1.0.14 version (`jfreechart-1.0.14.jar`);

- *jCommons* — a project offering generally reusable Java components `http://www.jcommons.sourceforge.net`. In *SaC*, the jCommons library is needed as a prerequisite for jFreeChart. We used jCommons in its 1.0.17 version (`jcommon-1.0.14.jar`).

- *SWT (Standard Widget Toolkit)* — an open source widget toolkit for Java designed to provide efficient, portable access to the user-interface facilities of the operating systems on which it is implemented, see `http://www.eclipse.org/swt`. In *SaC*, the SWT is used in simple demo-AIs built for the games of checkers and Nim.

Chapter 1

Getting started

1.1 Installation and requirements

Since *SaC* is a Java library, placed in a .jar file, there is no explicit installation procedure whatsoever. In other words, the user is supposed to simply download the library and attach it to his Java project, where he intends to use *SaC* and its searching capabilities.

This user guide document was written along with the first version of *SaC* distribution, numbered as 1.0.3. The `sac-1.0.3-dist-bundle.zip` file (to be downloaded) encapsulates the whole distribition, and contains the following elements.
The library files:

- **`sac-1.0.3.jar` — the main *SaC* library file**,
- `jfreechart-1.0.14.jar` — a prerequisite library,
- `jcommon-1.0.14.jar` — a prerequisite library,
- `swt.jar` — a prerequisite library (64 bit version),

the documantation (javadoc) and source files:

- `sac-1.0.3-javadoc.zip`,
- `sac-1.0.3-src.zip`,

the UML class diagram — a high-level view on *SaC*'s main elements:

- `sac-1.0.3-uml.pdf`,

examples (templates) of configuration files:

- `graph_configurator_example.properties`,
- `game_configurator_example.properties`,

and finally, a few .bat files (for Windows) executing examples of *SaC* ready-made solvers / programmes:

- `run_sudoku.bat`,
- `run_slidingpuzzle.bat`,
- `run_tsp.bat`,
- `run_rectpacking.bat`,
- `run_checkers.bat`,
- `run_nim.bat`.

1.2 "Hello world" for a graph

Suppose we have at disposal the following Java class to represent directed graphs with weights (the class is not related to *SaC* itself).

```
1  public class DirectedGraph {
2
3      private double[][] costs;
4      private int goal;
5
6      public DirectedGraph(int howManyNodes, int goal) {
7          costs = new double[howManyNodes][howManyNodes];
8          for (int i = 0; i < howManyNodes; i++)
9              for (int j = 0; j < howManyNodes; j++)
10                 costs[i][j] = Double.POSITIVE_INFINITY;
11         this.goal = goal;
12     }
13
14     public void addEdge(int i, int j, double cost) {
15         costs[i][j] = cost;
16     }
17
18     public double[][] getCosts() {
19         return costs;
20     }
21
22     public int getGoal() {
23         return goal;
24     }
25 }
```

It uses a two-dimensional array (`double[][] costs`) to store costs of transitions between particular nodes — in other words to model the edges. The symbolic infinity (`Double.POSITIVE_INFINITY`) indicates the lack of an edge. The index of the target node to be reached is kept in the field named `goal`. Obviously, the presented implementation is just an example and many other possibilities may come to mind. Up to now *SaC* is not involved anyhow.

Now, consider a graph created by the code below and depicted in Fig. 1.1. Suppose we would like to be able to search this graph (or other graphs alike, stored as `DirectedGraph`) using *SaC*.

```
1  DirectedGraph myGraph = new DirectedGraph(7, 6);
2  myGraph.addEdge(0, 1, 3.0);
3  myGraph.addEdge(0, 2, 1.0);
4  myGraph.addEdge(0, 5, 2.5);
5  myGraph.addEdge(1, 3, 2.0);
6  myGraph.addEdge(1, 4, 1.5);
7  myGraph.addEdge(2, 1, 1.0);
8  myGraph.addEdge(2, 4, 3.0);
9  myGraph.addEdge(3, 7, 1.0);
10 myGraph.addEdge(4, 7, 2.0);
11 myGraph.addEdge(5, 6, 4.0);
12 myGraph.addEdge(6, 7, 0.5);
```

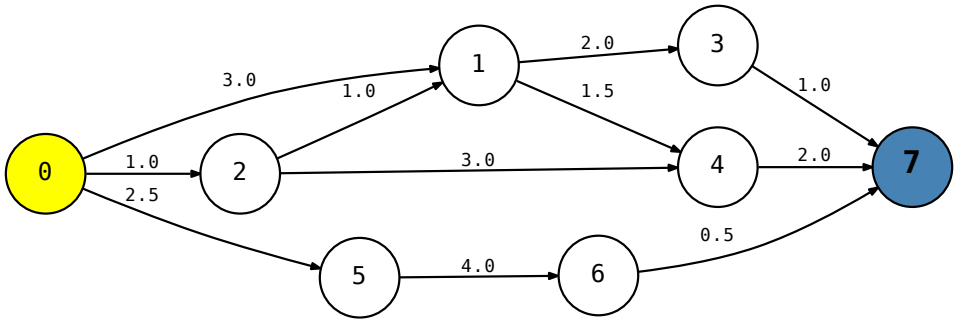


Figure 1.1: "Hello world" graph to be travalled from node 0 (initial) to node 7 (goal) .

To apply *SaC* and its searching capabilities we prepare the following class.

```
1  public class HelloWorldGraphState extends GraphStateImpl {
2
3      public static DirectedGraph dg = null;
4      private int i; // current node
5
6      public HelloWorldGraphState(int i) {
7          this.i = i;
8      }
9
10     @Override
11     public List<GraphState> generateChildren() {
12         List<GraphState> children = new ArrayList<GraphState>();
13         double[][] costs = dg.getCosts();
14         for (int j = 0; j < costs.length; j++)
15             if (costs[i][j] < Double.POSITIVE_INFINITY)
16                 children.add(new HelloWorldGraphState(j));
17         return children;
18     }
19
20     @Override
21     public int hashCode() {
22         return i;
23     }
24
25     @Override
26     public boolean isSolution() {
27         return (i == dg.getGoal());
28     }
29
30     @Override
31     public String toString() {
32         return Integer.toString(i);
33     }
34
```

```
35      static {
36          setGFunction(new StateFunction() {
37
38              @Override
39              public double calculate(State state) {
40                  HelloWorldGraphState hwgs = (HelloWorldGraphState) state;
41                  HelloWorldGraphState parent = (HelloWorldGraphState) hwgs.getParent();
42                  return (parent == null) ? 0.0 : parent.getG() + dg.getCosts()[parent.i][hwgs.i];
43              }
44          }
45          );
46      }
47  }
```

A reader's implementation could differ in details but in general the following rules have to be followed. Firstly, the class should extend the `sac.game.GraphStateImpl` class which is a suitable abstraction for states takinng part in graph searches. Secondly, the three vital elements should be described: (1) **generation of descendants** — by overriding the `generateChildren()` method, (2) **identification** of states — in our example this has been done by an override of the `hashCode()` method (it comes from the `java.lang.Object` class), (3) **termination** — by overriding the `isSolution()` method.

Additionally, a function object called `StateFunction` is attached to our class by means of the `setGFunction(...)` static method. In fact, this fragment is not a must in general, but in our example it is needed to take the costs of edges into account. Without this fragment all transitions would be by default treated as being of cost 1 (one manipulation / move). The attached object is equipped with a `calculate(...)` method. It is meant to return the cost paid by travelling from the initial state up to the current state. The name `setGFunction(...)` is driven by a naming convention known from graph search algorithms, in particular the A^* algorithm. Commonly, the travelled cost from the initial state up to a state s is denoted by $g(s)$, and the estimated cost from s to the goal state by $h(s)$. The understanding of these details is not important right now and will be discussed later in section 4.1.

Now, by executing the following `main(...)` method

```
1  HelloWorldGraphState.dg = myGraph; // static reference to graph from previous listing
2
3  GraphSearchAlgorithm algorithm = new Dijkstra(new HelloWorldGraphState(0));
4  algorithm.execute();
5  HelloWorldGraphState solution = (HelloWorldGraphState) algorithm.getSolutions().get(0);
6
7  System.out.println("SOLUTION: " + solution);
8  System.out.println("PATH: " + solution.getPath());
9  System.out.println("PATH COST: " + solution.getG());
```

one can see the search results related to the Dijkstra's algorithm:

```
SOLUTION: 7
PATH: [0, 2, 1, 3, 7]
PATH COST: 5.0
```

The reader can check that the path shown is in fact the path of lowest cost (the shortest path).

The *search graph* generated by *SaC* during our exemplary execution is presented in Fig. 1.2 (the

visualization was produced by *Graphviz* from an input file generated from *SaC*'s API). Throughout this user guide, the coloring of our illustrations for graph searches is as follows: yellow denotes the initial state, red denotes states that were generated but not visited, green denotes visited states

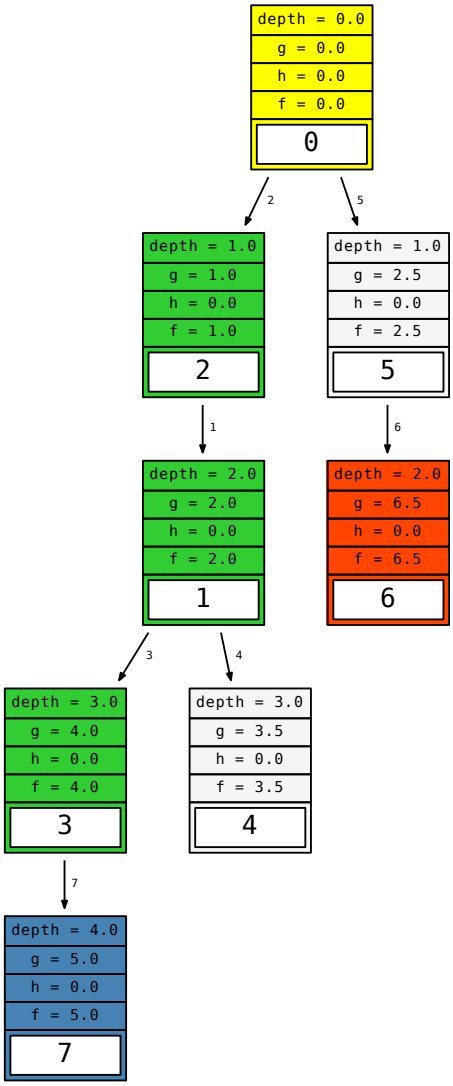


Figure 1.2: Search graph produced by *SaC* using Dijkstra's algorithm for the "hello world" exemplary graph from Fig. 1.1.

.

lying on the path to the solution, light gray denotes visited states not on the path, blue indicates the goal (solution) state. In every box, displayed is an information about: the depth of a state, its traveled cost g, estimation of its remaining cost h (0 by defeault when no heuristics is defined), and its representation in the inner white rectangle.

The reader might have noticed the lack of the $0 \rightarrow 1$ link in the search graph from Fig. 1.2 although such a transition is possible according to our directed graph under search (Fig. 1.1). The explanation is as follows — as the search procedure was progressing it did at first generate the state 1 as one of descandants of state 0, but later it discovered that it is cheaper to reach the state 1 going via 2 (path $0 \rightarrow 2 \rightarrow 1$), therefore finally, the state 2 was memorized as the 'best parent' for state 1. Probably, this is a good place to point out the distinction between a graph subject to searching and a *search graph*. The first pertains to the structure, possibly with cycles and possibly infinite, describing some real-world object or phenomenon that we analyze by searching. The latter describes the way the actual search procedure was progressing[1]. Our illustrations of search graphs in this user guide can be regarded as images or snaphots of the search procedure at its stoppage moment.

How would a breadth-first search procedure do on our example (instead of Dijkstra's algorithm)? The necessary modification is in the line instantiating the algorithm (line 2):

```
2 ...
3 GraphSearchAlgorithm algorithm = new BreadthFirstSearch(new HelloWorldGraphState(0));
4 ...
```

with the rest of the code unchanged. It results in the following output:

```
SOLUTION: 7
PATH: [0, 1, 3, 7]
PATH COST: 6.0
```

and the search graph illustrated in Fig. 1.3. One can recognize that the breadth-first search procedure neglects the transition costs and is guided only by depths of states (i.e. the number of hops from the initial state). The resulting path is therefore of greater cost than before.

As the reader might have noted our exemplary `HelloWorldGraphState` class kept a static reference to the directed graph object. The reference was static because it was common to all `HelloWorldGraphState` objects and could be shared. This reference was needed for the purpose of generation of descendants — we accessed it in the `generateChildren()` method to know what are the edges outgoing from the current node (encapsulated by the search state). We should explain that such a feature — memorization of the whole graph under search — will not be always present when using *SaC*. Very often the programmer will be able to construct the descendants solely on the basis of the information present in the current state (e.g. possible manipulations for the current arrangment of a sliding puzzle or Rubik's cube, etc.). The reader should understand that graphs under search can be very large, often too large to be memorized or even infinite.

[1]Since the procedure suitably avoids the cycles a *search graph* is also often referred to as a *search tree*.

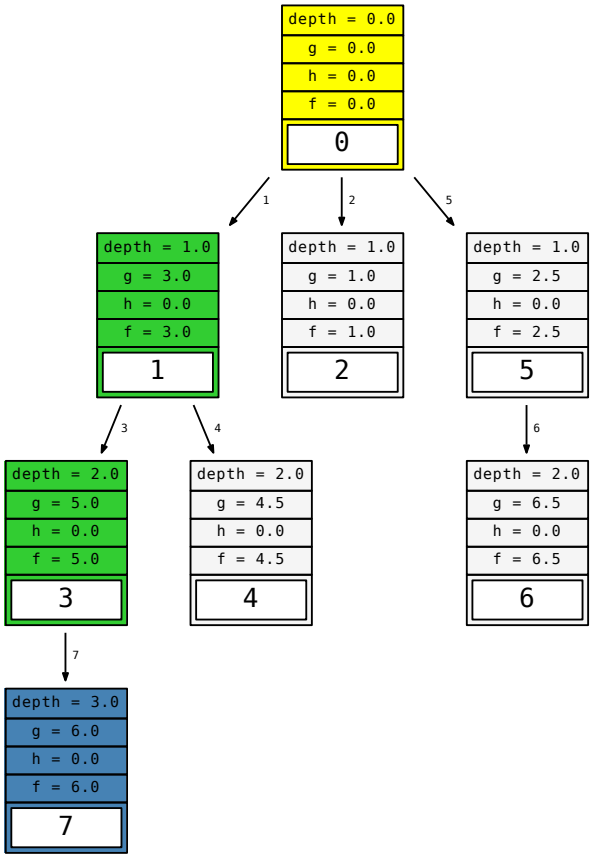


Figure 1.3: Search graph produced by *SaC* using Breadth-first search algorithm for the "hello world" exemplary graph from Fig. 1.1.

.

Let us now play with a larger scale example, depicted in Fig. 1.4. We generate a graph with 100 nodes placed on a plane within a square of side 100. The initial node is placed in the top-left corner, the goal node in the bottom-right corner. The rest of nodes is placed randomly according to the uniform distribution within the square. Edges between nodes are also generated on random but satisfy the following constraints. The number of edges is approximately 10% of the total number of edges in a completely connected graph. Two nodes can be connected only if the distance between them along a straight line is not greater than 20% of the square diagonal. In other words we allow only fairly close nodes to be connected — the graph can be regarded as a simulation of a map. The cost of an edge is equal to the aforementioned distance plus some random $\epsilon > 0$ (uniformly

distributed) proportional to that distance but not greater than 10% of it. Fig. 1.4 shows the graph together with the shortest path. The reader may zoom in the figure to see more details.

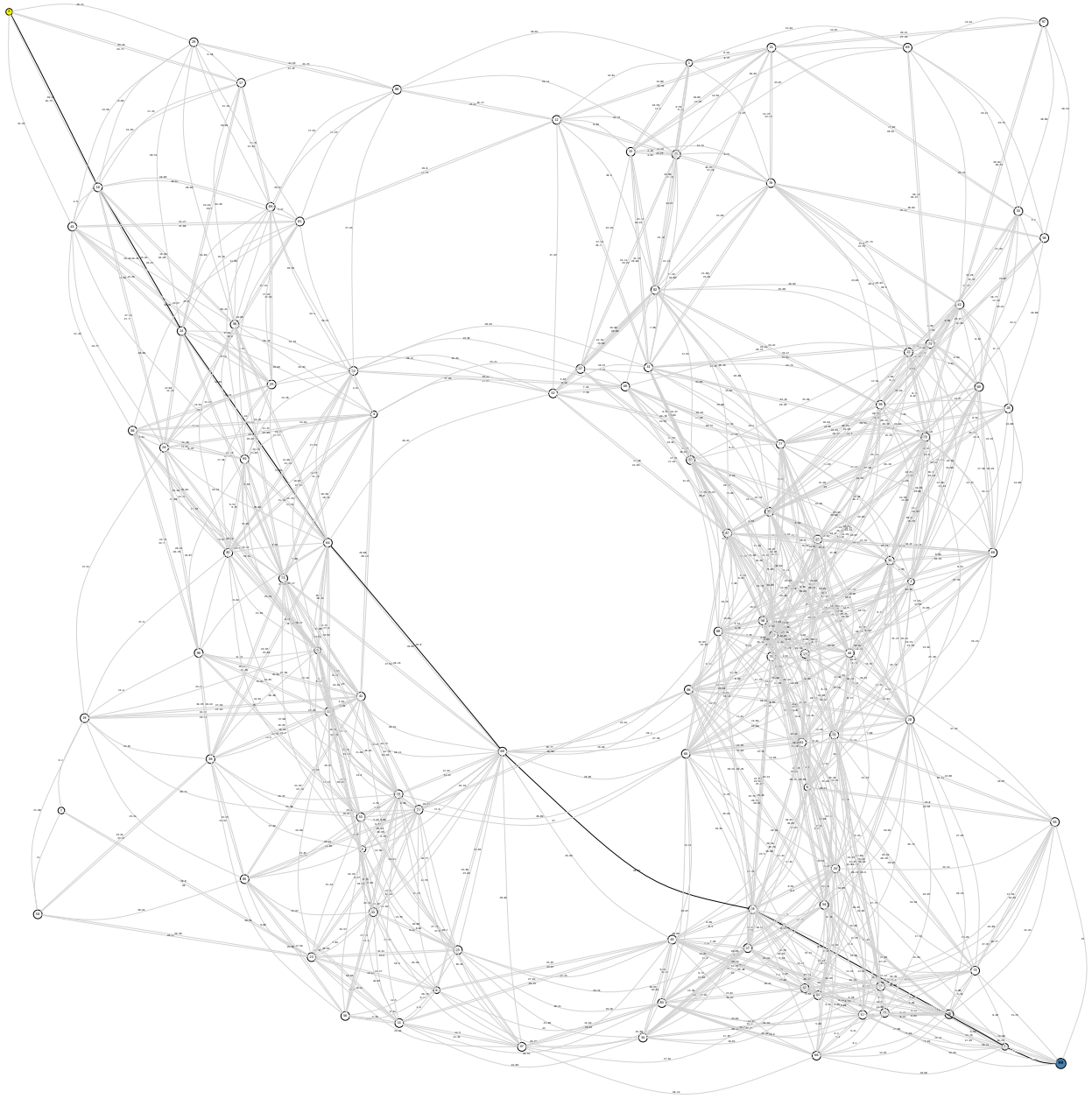


Figure 1.4: Graph with 100 nodes generated on random: initial node 0 (yellow), goal node 99 (blue). Shortest path marked as bolder black.

.

To store our graph we have introduced a new class `DirectedGraphWith2DCoordinates`, shown

below, as an extension of `DirectedGraph`. Again, we remark this class is not related to *SaC* itself.

```
1  public class DirectedGraphWith2DCoordinates extends DirectedGraph {
2
3      private double[][] coordinates;
4
5      public DirectedGraphWith2DCoordinates(int howManyNodes, int goal) {
6          super(howManyNodes, goal);
7          coordinates = new double[howManyNodes][2];
8      }
9
10     public void addCoordinates(int i, double x, double y) {
11         coordinates[i][0] = x;
12         coordinates[i][1] = y;
13     }
14
15     public double[][] getCoordinates() {
16         return coordinates;
17     }
18 }
```

The code to generate the actual instance of our graph (from Fig. 1.4) is listed below. In the code we use a `java.util.Random` object as the random generator and we force the seed of randomization (1234). This allows to generate exactly the same graph on every run of the program.

```
1  int n = 100;
2  DirectedGraphWith2DCoordinates myGraph = new DirectedGraphWith2DCoordinates(n, n - 1);
3
4  double side = 100.0;
5  myGraph.addCoordinates(0, 0.0, side); //initial
6  myGraph.addCoordinates(n - 1, side, 0.0); //goal
7
8  Random random = new Random(1234); // java.util.Random, imposed randomization seed: 1234
9  for (int i = 1; i < n - 1; i++)
10     myGraph.addCoordinates(i, side * random.nextDouble(), side * random.nextDouble());
11
12 int e = (int) Math.round(0.1 * n * (n - 1)); // 10 % of a complete graph
13 double maxDistance = 0.2 * side * Math.sqrt(2.0); // max distance between two nodes to be
       connected
14 for (int k = 0; k < e; k++) {
15     int i, j;
16     double distance;
17     do {
18         i = random.nextInt(n);
19         j = random.nextInt(n);
20         distance = Math.sqrt(
21                 Math.pow(myGraph.coordinates[i][0] - myGraph.coordinates[j][0], 2)
22               + Math.pow(myGraph.coordinates[i][1] - myGraph.coordinates[j][1], 2)
23               );
24
25     } while ((i == j) || (myGraph.getCosts()[i][j] < Double.POSITIVE_INFINITY) || (distance >
           maxDistance));
26
27     double epsilon = random.nextDouble() * 0.1 * distance;
28     myGraph.addEdge(i, j, distance + epsilon);
29 }
```

To search our graph with Dijkstra's algorithm we prepare the following program and we display a bit more information to the console than before. Note that we again use the same state class — `HelloWorldGraphState`.

```
1  HelloWorldGraphState.dg = myGraph; // our random graph (map) with 100 nodes
2
3  GraphSearchAlgorithm algorithm = new Dijkstra(new HelloWorldGraphState(0));
4  algorithm.execute();
5  HelloWorldGraphState solution = (HelloWorldGraphState) algorithm.getSolutions().get(0);
6
7  System.out.println("SOLUTION: " + solution);
8  System.out.println("PATH: " + solution.getPath());
9  System.out.println("PATH COST: " + solution.getG());
10 System.out.println("DURATION [ms]: " + algorithm.getDurationTime());
11 System.out.println("CLOSED: " + algorithm.getClosedStatesCount());
12 System.out.println("OPEN: " + algorithm.getOpenSet().size());
```

We execute the program and obtain the following output in the console:

```
SOLUTION: 99
PATH: [0, 18, 14, 64, 60, 10, 5, 99]
PATH COST: 149.51694106202785
DURATION [ms]: 11
CLOSED: 100
OPEN: 0
```

Without going too much into details, let us explain that `algorithm.getClosedStatesCount()` returns the number of states visited during the search, whereas `algorithm.getOpenSet().size()` returns the number of states that were generated but not visited because the algorithm was stopped when reaching a solution.

Perhaps some readers may know that Dijkstra's algorithm belongs to so called *uninformed* search methods. The algorithm takes into account only the travelled (experienced) cost $g(s)$ while traversing a graph and does not use any apriori information or heuristics to guide the ongoing search more economically to the goal. Another famous algorithm called A^* performs an *informed* search. It uses the sum $g(s) + h(s)$ to decide about the order in which states are visited. We remind that the heuristic summand $h(s)$ is an estimate on the cost remaining to the goal (in fact a lower bound on exact remaining cost). Let us try out A^* on our last example.

The necessary modification (line 2):

```
2  ...
3  GraphSearchAlgorithm algorithm = new AStar(new HelloWorldGraphState(0));
4  ...
```

leads to the following output

```
SOLUTION: 99
PATH: [0, 18, 14, 64, 60, 10, 5, 99]
PATH COST: 149.51694106202785
DURATION [ms]: 10
CLOSED: 100
OPEN: 0
```

We note that beside a tiny change in duration time nothing else changed. The number of visited states is still the same — 100. Our algorithm worked exactly as Dijktra's algorithm. What is wrong? The heuristic function (the additional information) has not been defined. We need to go back to our `HelloWorldGraphState` class and add the following fragment to the static block:

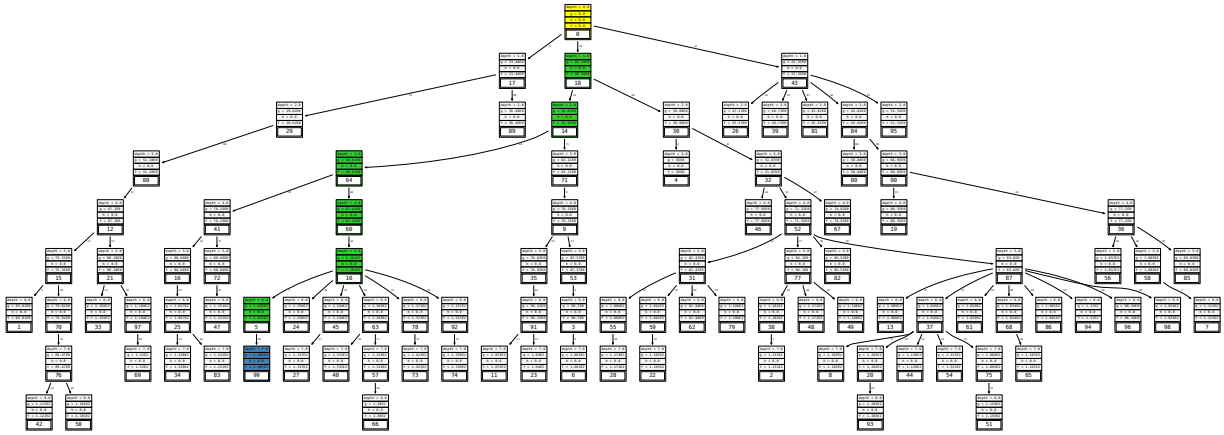
```
1  public class HelloWorldGraphState extends GraphStateImpl {
2
3      ...
4
5      static {
6          ...
7
8          setHFunction(new StateFunction() {
9
10             @Override
11             public double calculate(State state) {
12                 HelloWorldGraphState hwgs = (HelloWorldGraphState) state;
13                 DirectedGraphWith2DCoordinates dg2D = (DirectedGraphWith2DCoordinates)
                       HelloWorldGraphState.dg;
14                 return Math.sqrt(
15                           Math.pow(dg2D.getCoordinates()[hwgs.i][0] - dg2D.getCoordinates()[dg2D.
                                 getGoal()][0], 2)
16                         + Math.pow(dg2D.getCoordinates()[hwgs.i][1] - dg2D.getCoordinates()[dg2D.
                             getGoal()][1], 2)
17                         );
18             }
19         });
20     }
21 }
```

By doing so we define the heuristics which estimates the cost remaining to the goal as the distance along a straight line (Euclidean distance). We are able to calculate such distance because we know the coordinates of the goal. Clearly, some obstacles might occur along the way, but still this guiding information is better than none. Below we show the new output for A^* (after the heuristic function has been attached to the class):

```
SOLUTION: 99
PATH: [0, 18, 14, 64, 60, 10, 5, 99]
PATH COST: 149.51694106202785
DURATION [ms]: 3
CLOSED: 18
OPEN: 38
```

As one can note the algorithm finds the same optimal path but the number of visited (closed) states is now significantly smaller. Search graphs produced by both algorithms are shown in Fig. 1.5.

(a) search graph using Dijkstra's algorithm:



(b) search graph using A^* algorithm:

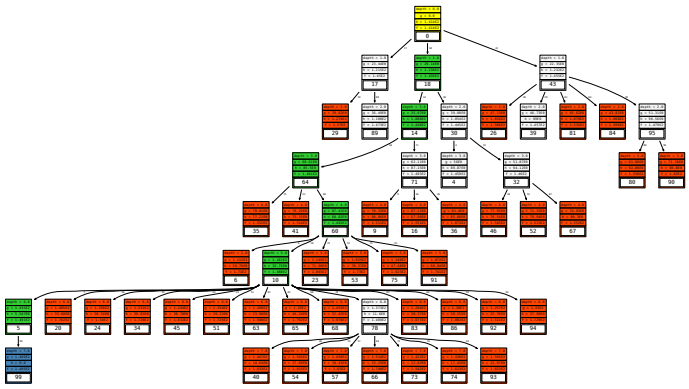


Figure 1.5: Search graphs produced by Dijkstra's (a) and A^* (b) algorithms for the random graph with 100 nodes from Fig. 1.4.

1.3 "Hello world" for a game

Imagine the following simple game. There are n stones on a table. Players make their moves interchangeably by taking away either one or two stones. A player is not allowed to take two stones when exactly two are left. The player left with the last stone loses.

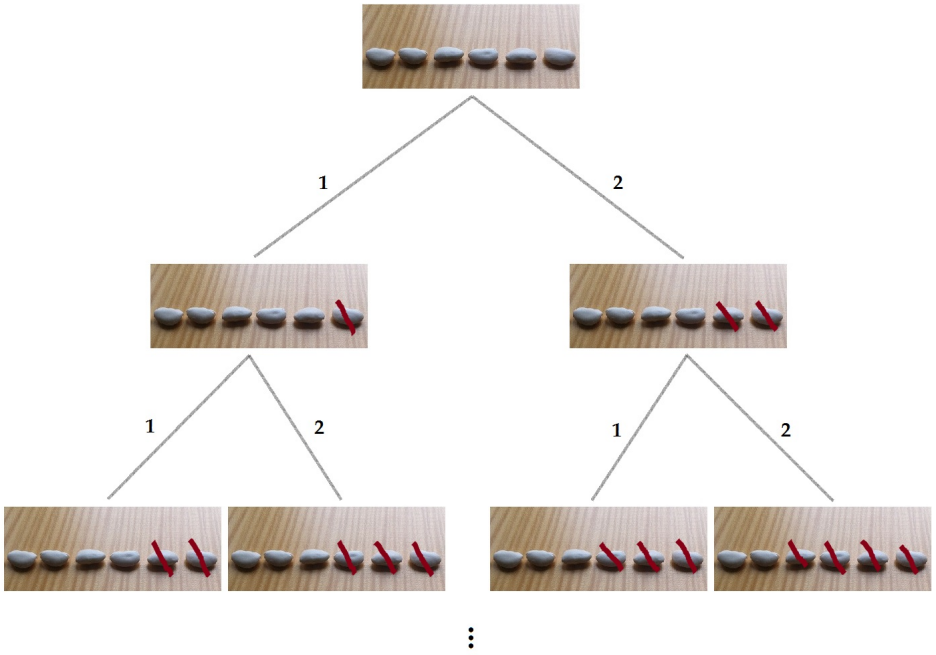


Figure 1.6: Exemplary game — "take one or two stones away". Player left with the last stone loses.

Some readers probably know that this kind of game can be easily solved via dynamic programming. When $n = 1$ the value of the game for the current player is a loss (L). Therfore, for $n = 2$ and $n = 3$ the game value is a win (W), since the player to move can tranfer both these cases to $n = 1$ by taking one or two stones respectively. For $n = 4$ the game value is a loss again, and the pattern $LWWLWWLWW\ldots$ emerges. Therefore, it is easy to tell the game value and the correct move for any n. If n can be expressed as $n = 3m$ or $n = 3m - 1$ (for some natural m), then it is a win for the player to move and he must simply transfer the game into $n = 3m - 2$.

Suppose however, that we do not know the dynamic programming approach, and we would like to find the solution of our game for some n using *SaC*. Here is how.

```
1  public class HelloWorldGameState extends GameStateImpl {
2
3      private int n;
4
5      public HelloWorldGameState(int n) {
6          this.n = n;
7      }
8
9      @Override
10     public List<GameState> generateChildren() {
11         List<GameState> children = new LinkedList<GameState>();
12         for (int i = 1; i <= 2 && i < n; i++) {
13             HelloWorldGameState child = new HelloWorldGameState(n - i);
14             child.setMoveName(Integer.toString(i));
15             child.setMaximizingTurnNow(!isMaximizingTurnNow());
16             children.add(child);
17         }
18         return children;
19     }
20
21     @Override
22     public int hashCode() {
23         int[] pair = {n, (isMaximizingTurnNow() ? 1 : -1)};
24         return Arrays.hashCode(pair);
25     }
26
27     static {
28         setHFunction(new StateFunction() {
29             @Override
30             public double calculate(State state) {
31                 HelloWorldGameState hwgs = (HelloWorldGameState) state;
32                 if (hwgs.n == 1)
33                     return (hwgs.isMaximizingTurnNow()) ? Double.NEGATIVE_INFINITY : Double.
                           POSITIVE_INFINITY;
34                 return 0.0;
35             }
36         });
37     }
38 }
```

Firstly, our class should extend the `sac.game.GameStateImpl` class which is a suitable abstraction for states taking part in game tree searches. Secondly, the three vital elements should be described: (1) **generation of descendants** — by overriding the `generateChildren()` method, (2) **identification** of states — in our example this has been done by an override of the `hashCode()` method, which comes from the `java.lang.Object` class, and we produce the hashcode from a pair of information (number of stones n, whose turn it is) uniquely describing our states, (3) **termination (or heuristics)** — this has been done by attaching via the `setHFunction(...)` static method a suitable function object to our class. This function is equipped with a `calculate(...)` method, and in our example it returns either $\pm\infty$ when the last stone is reached or 0 (no evaluation) otherwise.

Now, by executing the following `main(...)` method

```
1 GameSearchAlgorithm algorithm = new MinMax(new HelloWorldGameState(6));
2 algorithm.execute();
3 System.out.println("MOVES SCORES: " + algorithm.getMovesScores());
```

one obtains the answer for $n = 6$, being:

```
MOVES SCORES: {2=1.498077612385263E308, 1=-1.3482698511467367E308}
```

The large positive value $\approx 1.498 \cdot 10^{308}$ is the *SaC*'s way to indicate a win for the maximizing (initial) player if he starts by taking two stones. The large negative value $\approx -1.348 \cdot 10^{308}$ indicates a loss for the player if he starts by taking one stone.

We should shortly explain that in game searches, in order to represent wins or losses *SaC* returns numbers of order 10^{308} (close to the `Double.MAX_VALUE` value) rather than symbolic infinities — `Double.NEGATIVE_INFINITY` or `Double.POSITIVE_INFINITY` (as the code from the example might suggest). The returned numbers show how quickly a win or a loss is attained after a particular move. A larger absolute value indicates a win or a loss after fewer moves. This can be regarded as 'grading of infinities', which *SaC* calculates on the fly, and is explained in more detail in chapter 4 — "Searching game trees" (in the API section)[2].

The *search tree* generated by *SaC* during the example from above is presented in Fig. 1.7 (the visualization was produced by *Graphviz* from an input file generated from *SaC*'s API). Throughout this user guide, the coloring of our illustrations for game searches is as follows: yellow denotes the initial state, light gray denotes visited regular states, dark gray indicates non-win terminal states, blue indicates win terminal states (a win for either player), dark red indicates states for which the game value or a bound on that value was read as a ready result (because these states had occurred before) from the so called transposition table[3] , light red indicates the so called cutoff states which could not affect the game value, green indicates states residing along the principal variation[4]. In every box, displayed is an information about: depth of a state, its heuristic value h, its game value v (also known as the minimax value) assigned by the search procedure, and its representation.

The reader might be tempted now to check if the search would also work correctly for some larger n, e.g. $n = 100$. The answer unfortunately is *no*. At least not without some intervention by the user. The reason is that for games, *SaC*, like all search engines, has some depth limit (search horizon), by default set to 3.5 i.e. seven *half-moves*[5]. Therefore, win states located beyond the search horizon are not seen by the algorithm. Be reminded that in our exemplary implementation we return zero for non-win states (line no. 34 of the `HelloWorldGameState` class). Actually, with depth limit set to seven half-moves, already for $n = 9$ there will exist one win state beyond the search horizon. To demonstrate more of such states, in Fig. 1.8 we depict the game tree produced by *SaC* for $n = 12$. In the bottom left corner one can see several states, marked with dark gray, that are

[2]To be precise, *SaC* actually may return `Double.NEGATIVE_INFINITY` or `Double.POSITIVE_INFINITY`. This happens only in the case when a win or a loss is predicted to come immediately — that is in one half-move.

[3]The notion and mechanisms of the transposition table will explained later in section 4.2.5.

[4]Principal variation is a line of play during which both players make optimal moves with respect to the current search horizon.

[5]In the game related nomenclature, a single move made by one player is called a *half-move* or a *ply*, and is counted as 1/2. A full move is considered to be done after both players have made their half-moves.

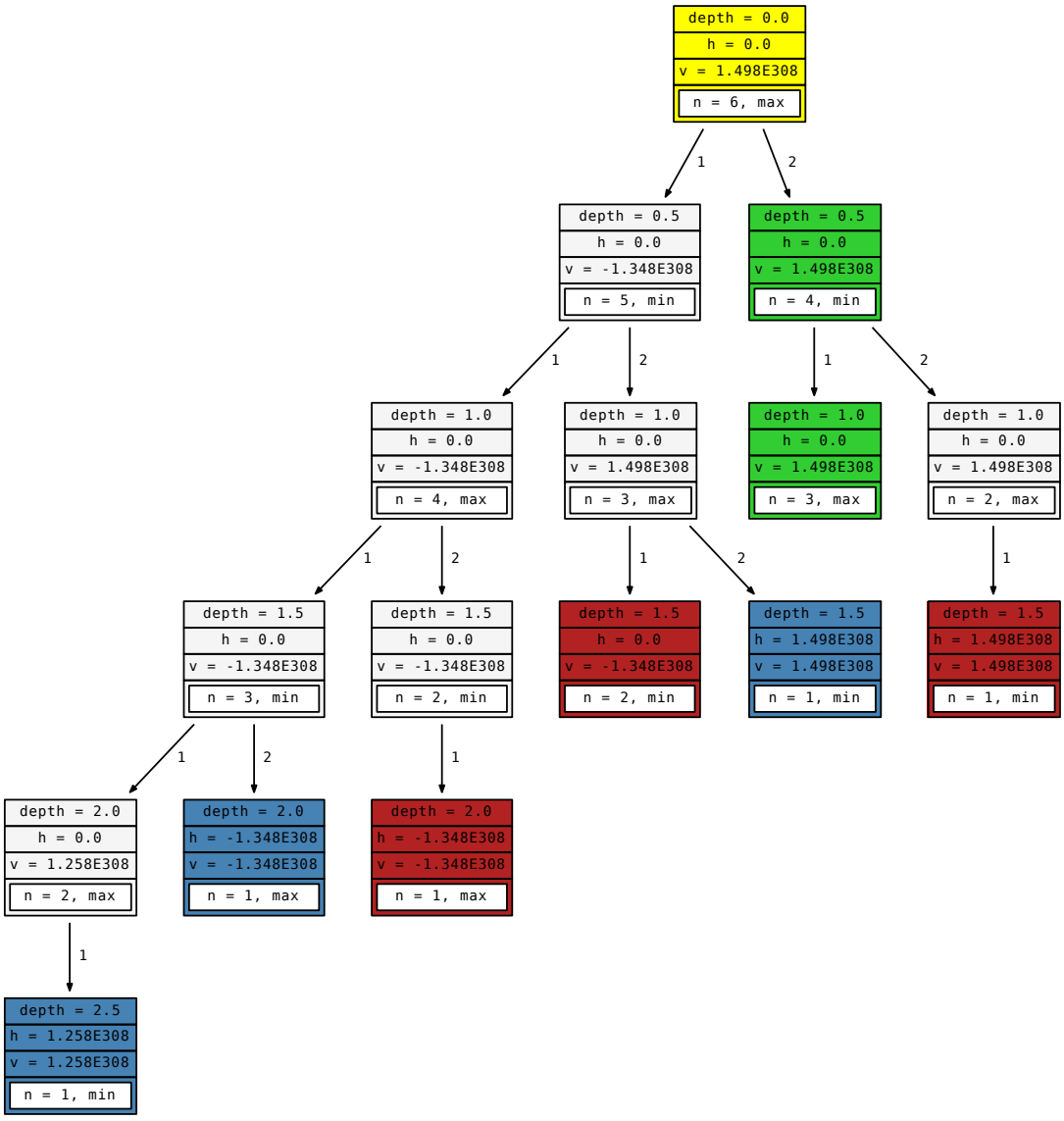


Figure 1.7: "Hello world" example for $n = 6$ — game tree searched by *SaC*.

terminals due to depth limit reached, and were assigned a zero by our implementation as their heuristic game evaluation. We mention that the program response in that case would be

```
MOVES SCORES: {2=1.1556598724114885E308, 1=0.0}
```

and the score 0.0 is a consequence of a too shallow search subtree.

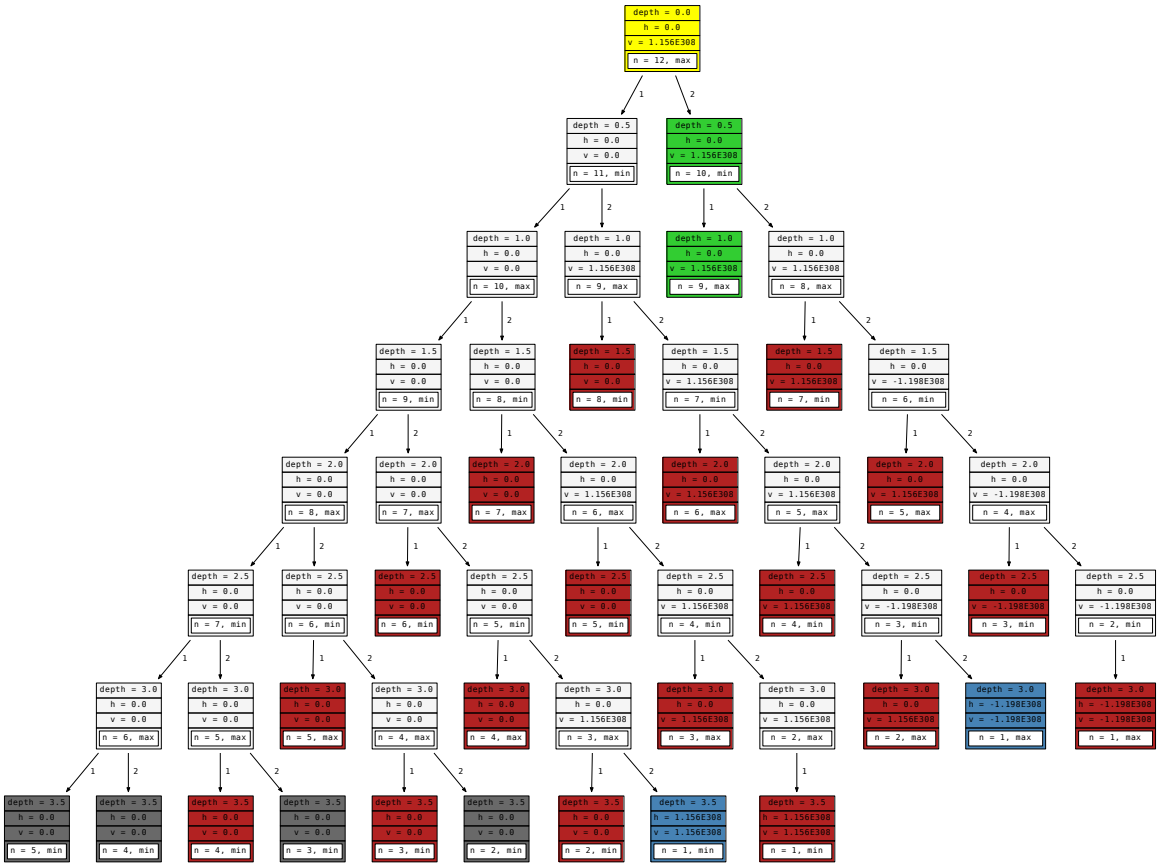


Figure 1.8: "Hello world" example for $n = 12$ — game tree searched by *SaC*. Terminal non-win states (dark gray) visible in the bottom left corner due to a default depth limit set to 3.5.

To solve the case of $n = 100$, a more 'aware' user can encourage *SaC* to search deeper. A sample code to do so is as follows.

```
1 public static void main(String[] args) {
2     GameSearchConfigurator configurator = new GameSearchConfigurator();
3     configurator.setDepthLimit(49.5);
4     GameSearchAlgorithm algorithm = new MinMax(new HelloWorldGameState(6), configurator);
5     algorithm.execute();
6     System.out.println("MOVES␣SCORES:␣" + algorithm.getMovesScores());
7     System.out.println("VISITED:␣" + algorithm.getClosedStatesCount());
8     System.out.println("DURATION␣[ms]:␣" + algorithm.getDurationTime());
9 }
```

And, the information displayed to the console is

```
MOVES SCORES: {2=-9.260843422017989E307, 1=-9.260843422017989E307}
VISITED: 2550
DURATION TIME [ms]: 70
```

showing the correct response that the initial player loses regardless of his moves after the optimal play by his opponent. Fig. 1.9 illustrates the tree associated to that search in a simpler format (we encourage the reader to zoom in the figure multiple times to see the coloring of states better).

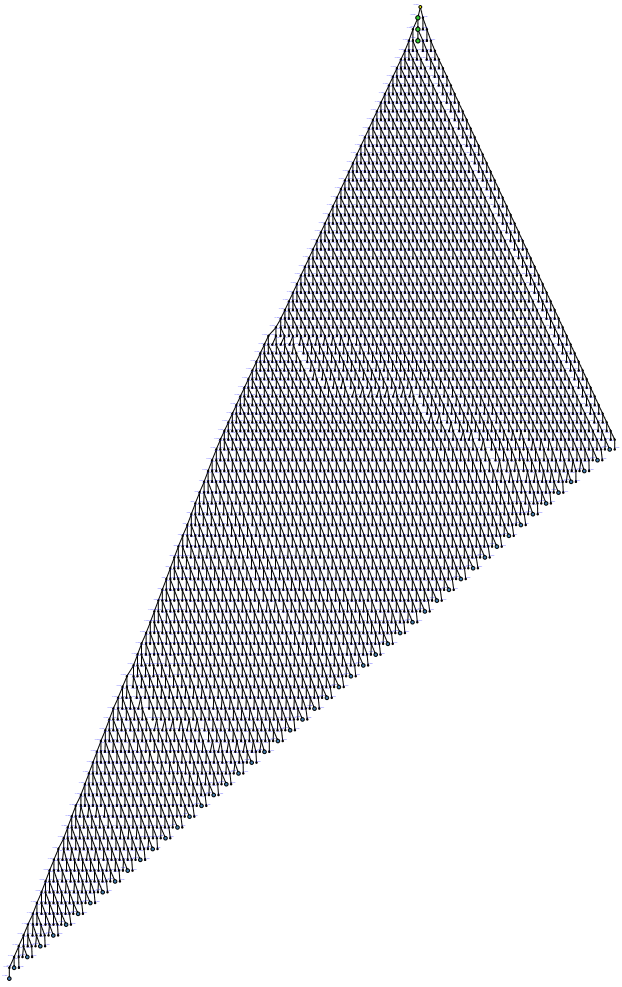


Figure 1.9: "Hello world" example for $n = 100$ — game tree searched by *SaC* with the depth limit explicitly set to 49.5 (99 half-moves).

Chapter 2

State abstraction

In this section we present some of the API related to the *state* as a top-level abstraction for both graph states and game states.

In *SaC*, introduced is an interface `sac.State` and its default implementation `sac.StateImpl`. To give the reader an overview we start by showing the source code[1] of the interface, then we make some comments on particular elements. Since most of methods in the `sac.StateImpl` implementation are straightforward, hereby we show only the more important excerpts of it. For full code the reader is addressed to Appendix 6.1.

```
1  package sac;
2
3  import java.util.List;
4  import sac.graphviz.Graphvizable;
5
6  public interface State extends Comparable<State>, Graphvizable {
7
8      public Identifier getIdentifier();
9      public void refreshIdentifier();
10     public State getParent();
11     public void setParent(State parent);
12     public List<? extends State> getChildren();
13     public double getDepth();
14     public void setDepth(double depth);
15     public List<? extends State> getPath();
16     public List<String> getMovesAlongPath();
17     public List<? extends State> generateChildren();
18     public double getH();
19     public void setH(Double h);
20     public void refreshH();
21     public String getMoveName();
22     public void setMoveName(String moveName);
23     public void refresh();
24 }
```

[1] The listing is in its full, but without javadocs. The javadocs can be found within the library both in the actual source files and in the generated HTML format.

2.1 Identifiers

The first element of interest is the *identifier* (`sac.Identifier`). Identifiers are meant to be representations of states within search procedures. If possible, they should be unique representations, but more importantly they should be fast representations (i.e. quickly computable). In *SaC*, two scenarios are allowed: (1) indentifiers are *integers* calculated via the `int hashCode()` method — the default scenario, (2) indentifiers are *strings* calculated via the `toString()` method.

As the reader may know, both `hashCode()` and `toString()` are standard Java methods included in the most elementary class `java.lang.Object` and are meant to be overriden by the user if needed. In other words, in *SaC* we let the user decide which scenario he prefers and we expect him to implement at least one of the above two methods. The `sac.Identifier` type should be viewed as a wrapper for this mechanism, regardless of the chosen scenario.

We remark, that typically in practice we decide for string identifiers when we absolutely need to guarantee the uniqueness of identifiers in given problem. Yet, one should be aware that comparisons of long strings may be computationally expensive. On the other hand, we decide for integer identifiers (hashcodes) when we care more about the speed of execution, and simultenously we agree to accept a tiny risk of a situation when conflicting hashcodes might occur, i.e. the same hashcodes for two different states.

The source code of the `sac.Identifier` is as follows.

```
1  package sac;
2
3  public final class Identifier implements Comparable<Identifier> {
4
5      private static IdentifierType type = IdentifierType.HASH_CODE;
6      private Object id = null;
7
8      public Identifier(State state) {
9          id = (type == IdentifierType.HASH_CODE) ? Integer.valueOf(state.hashCode()) : state.
               toString();
10     }
11
12     public static final IdentifierType getType() {
13         return type;
14     }
15
16     public static final void setType(IdentifierType type) {
17         Identifier.type = type;
18     }
19
20     public int compareTo(Identifier otherIdentifier) {
21         if (type == IdentifierType.HASH_CODE)
22             return ((Integer) id).compareTo((Integer) otherIdentifier.id);
23         else
24             return ((String) id).compareTo((String) otherIdentifier.id);
25     }
26
27     public boolean equals(Object otherIdentifier) {
28         Identifier otherIdentifier2 = (Identifier) otherIdentifier;
29         if (type == IdentifierType.HASH_CODE)
30             return ((Integer) id).equals((Integer) otherIdentifier2.id);
31         else
```

```
32              return ((String) id).equals((String) otherIdentifier2.id);
33      }
34
35      public String toString() {
36          return id.toString();
37      }
38
39      public int hashCode() {
40          int result = (type == IdentifierType.HASH_CODE) ? ((Integer) id).intValue() : ((String)
                id).hashCode();
41          return result;
42      }
43  }
```

As one may note the `sac.Identifier` class is basically a wrapper around its inner field: `private Object id`. It is interpreted either as an integer or a string depending on a global static setting (`private static IdentifierType type`). On the usage level, this setting can be changed either directly from the code via the static `setType(...)` method, or by means of so called *configurators*, which are discussed in further sections.

2.2 Parent – children bindings

The `sac.State` interface contains a getter `getParent()` and a setter `setParent(...)` to respectively retrieve or impose a reference to the parent state for a given state. These two methods are meant mainly for the purposes of the core of *SaC*, and in practice are called seldom by an end user. In fact, we discourage the user to call the setter on his own because a careless usage might potentially result in an incorrect behaviour of a search procedure. As a convention, we use the `null` result of a getter when a state is the root state.

As regards descendant states (children) there is only a getter present — `getChildren()`. It is meant to return a list of references to children. We should remark however that by default *SaC* does *not* memorize these references for memory saving reasons. Hence, by default an empty list is returned. We explain that a link from a child to its parent (accessible via `getParent()`) is always memorized, but the opposite direction links from a parent to its children are redundant. In practice, children objects during a search procedure are created by the *SaC* itself via the `generateChildren()` callback method executed on a parent. The children are suitably visited[2] later on. Again, this default *SaC* behaviour (related to not-memorizing references from parents to children) can be explicitly changed by a *configurator* object, but in practice such a change is needed only in the case when the user is interested in drawing graphs by means of Graphviz, and is not needed for the search itself.

A vital element is the mentioned `generateChildren()` callback method. This method must inevitably be implemented by the user and should return a list of direct descendants (children) for given state. In such an implementation the user is expected to:

- create a local list,

[2]In graph searches, children states (non-visited so far) are placed in a queue at this stage and polled from it later. In game searches, a recursive call on a child state is made by the core of *SaC*. In either case, to correctly carry out the search procedure there is no need to memorize references from a parent to all its children.

- build the children according to the rules / nature of the specific problem,
- give names to the moves (or manipulations) which lead to the creation of particular children (via `setMoveName(...)`),
- add the children to the local list,
- return the list.

Such proceedings have been already demonstrated in the "hello world" examples.

On the other hand, the user is *not* expected to attach anyhow his local list to the parent object (and in fact should not do it, having in mind the remarks from the previous paragraph). The core of *SaC* awaits just for the result of the `generateChildren()` callback method and proceeds with the result approprietely further on.

2.3 Depth of state, path from root to given state

The `sac.State` interface allows the user to ask for the depth of a state by the `getDepth()` getter. Depth is understood as the number of moves performed to transfer the root state into the given state. In particular, the depth of the root state is zero. Although the API makes also the setter available — `setDepth(...)` — the user is discouraged to use it explicitly[3].

Additionally, the user is allowed to ask for the *path* (sequence of states or moves) from the root state up to the given state. Depending on the need, this can be done either by the `getPath()` method (returns the sequence of states) or by the `getMovesAlongPath()` method (returns the sequence of move names). We should remark that the path as a sequence of states includes the root state in the front and therefore is longer by one element comparing to the sequence of move names.

2.4 Heuristics

The interpretation of a heuristic value assigned to a state is different for graphs and different for game trees. When searching graphs, the heuristics is a non-negative estimation of the distance remaining to the solution[4]. When searching game trees, the heuristics is an evaluation of the game position represented by a given state[5]. In that case positive values indicate some advantage for the maximizing player and negative values indicate some advantage for the minimizing player. By default, the player who starts the game is regarded as the maximizing player in *SaC*.

[3] For analogical reasons as in the case of `setParent(...)`.

[4] In fact it should be either an exact value or a lower bound on that distance, otherwise the heuristic is *inadmissible* and might lead to non-optimal paths to the solution.

[5] Such an evaluation is often impresice and reflects human experience and knowledge about the game. For example in chess, the heuristics may involve materialistic elements (number and quality of pieces remaining on the board), but also positional elements like: advantage in the board center, structure of pawns, activeness of pieces, pass pawns, king safety, etc.

The API provides both the getter `getH()` and the setter `setH(...)` for the heuristics. Again (as in former contexts), we discourage the user to use the setter explicitly, it is meant only for the purposes of the core of *SaC*.

To define how the heuristics should be calculated for given search problem, the user is supposed to provide an implementation of the `sac.StateFunction` class. Its default form in *SaC* is presented below and it returns zero via the `calculate(...)` method regardless of the state passed as an argument. The user should therefore appropriately populate the body of this method.

```
1  package sac;
2
3  public class StateFunction {
4
5      public StateFunction() {
6      }
7
8      public double calculate(State state) {
9          return 0.0;
10     }
11 }
```

In particular, in the context of game trees we encourage the user to return `Double.POSITIVE_INFINITY` and `Double.NEGATIVE_INFINITY` constants as a representation of a win respectively for the maximizing and the minimizing player[6].

When the user-defined function is ready, it should be instantiated as an object — the calculator of heuristics — and attached to the state class by means of the static method `setHFunction(StateFunction hFunction)` (it comes from the `sac.StateImpl` class). Note that `setHFunction(...)` is static, therefore the imposed function object will also be static — common to all the states.

There are two programmistic ways the user may create and attach his function object. The first way is to create a separate named class (thus a separate source file) being an extension of the `sac.StateFunction`, e.g. in the following manner

```
1  public class MyHeuristics extends StateFunction {
2
3      public double calculate(State state) {
4          //suitable calculations
5      }
6  }
```

and then to attach an instance of this class by placing the following static block in the state class.

```
1  public class MyState extends GraphStateImpl { // or extends GameStateImpl
2      ...
3      static {
4          setHFunction(new MyHeuristics());
```

[6] A side remark: in the context of graph searching, the `Double.POSITIVE_INFINITY` constant might theoretically be also of some use, e.g. for generated states that do not satisfy certain constraints for given problem. Such states with infinite heuristics would be then placed at the very end of the queue. However, one may argue about the sense of keeping them in the queue at all. Usually, it is better to simply discard (ignore) such states i.e. not to return them among results of the `generateChildren()` method.

```
    }
    ...
}
```

The second way is to do it a bit simpler, namely by the Java mechanism of an anonymous class (so without a separate source file), as shown below.

```
public class MyState extends GraphStateImpl { // or extends GameStateImpl
    ...
    static {
        setHFunction(new StateFunction() {
            @Override
            public double calculate(State state) {
                //suitable calculations
            }
        });
    }
    ...
}
```

2.5 Refresh methods

In the listing of the `sac.State` interface from the beginning of this chapter one can notice three refresh methods: `refreshIdentifier()`, `refreshH()`, `refresh()`. When and why are they needed?

Firstly, we need to explain that both the identifier and the value of heuristics are designed in *SaC* in such a manner so that they are computed possibly only *once*. By that, *SaC* tries avoid multiple and often expensive computations of the same results[7]. The typical scenario is the following. The computation of either the identifier or the heuristics is postponed until the first call to the suitable getter is made, i.e. `getIdentifier()` or `getH()` respectively. More precisely, after a state object is constructed (instantiated) its internal variables meant to keep the identifier and the heuristics remain set to `null` for some time. When the first call to a particular getter is made, the suitable value is computed according to the recipe provided by the programmer and is stored internally (in the place of the `null`). After that, for each successive call those memorized values are served immediately, not recomputed.

As already said, the scenario described above is typical and covers the most common use cases in *SaC*. However, there exist certain situations when explicit refreshes (forcing of recomputation) of identifier or heuristic value are necessary. Such situations usually occur e.g. when the same (working) reference in RAM memory pointing to some state is used multiple times to initiate the search procedure, but the content of this reference gets changed from one search to another. A good example of that can be a chess playing program. Within its main loop, meant to handle the ongoing game and to display it to the screen, the program may use the same working reference to a state object representing the current position on the board. After a move by

[7] As an example one may think of the Traveling Salesman Problem, described in the section 3.3, in which the heuristics can be based on the concept of Minimum Spanning Tree (MST). To generate a MST one needs to perform an additional algorithm (e.g. Kruskal's or Prim's algorithm). Therefore, it would be a clear waste to do it multiple times unnecessarily within a search procedure.

the human player is made, the program applies the move to that state by some suitable method, e.g. `currentState.makeMove("e2:e4")`, and we would like to use the modified state as the starting point for a *SaC* search procedure. Since the state may now contain old values of the identifier and the heuristics (from before the move) the programmer ought to refresh such a state. Otherwise, a search might work incorrectly. The aforementioned methods are therefore meant to explicitly force the recomputation of: the identifier (`refreshIdentifier()`), the heuristics (`refreshH()`), or both (`refresh()`).

Chapter 3

Searching graphs

This chapter is devoted to the part of *SaC* library responsible for searching graphs. We begin the chapter with a section reminding several well known graph search algorithms. The reader familiar with these algorithms can move over to subsequent sections. The second section discusses *SaC*'s API related to graph searching. The last one presents four graph-related examples in which we have applied *SaC* for demonstration, namely: sliding puzzle, Traveling Salaseman Problem, and sudoku.

3.1 Algorithms

Most of graph search algorithms can be conveniently formulated with the use of two sets (collections) of states — named by convention as the *Open* set and the *Closed* set. At a given stage of a running algorithm, the *Closed* set contains states that have already been visited (and turned out not to be a solution), whereas the *Open* set contains states waiting to be potentially visited. Awaiting states have been generated as descendants (graph neighbours) from the previously visited ones.

Depending on the type and purpose of the search algorithm, *Open* and *Closed* sets are implemented via different data structures, affecting their behaviour and performance. In particular, one can note that decisive about the algorithm type is the order according to which generated states are taken (and removed) from the *Open* set for further processing.

3.1.1 Breadth-first search, Depth-first search

It is difficult to point out original authors of breadth or depth-first searching techniques. In fact, they are commonly treated as simple non-informative techniques to traverse a graph, rather than actual search methods which should be informative (guided by some useful information). Historically, it is likely that the very first version of a Depth-first search (DFS) was investigated in the 19th century by a French mathematician Charles Pierre Trémaux as a strategy for solving mazes.

As the naming suggests, in the breadth-first approach the algorithm must first vist all states with depth d before it can proceed with states at depth $d+1$. In a sense contrarily, in the depth-first

approach the algorithm must not visit any unvisited state at depth d if there exist some generated and unvisited states with depth $d + 1$. We present both these algorithms in the subsequent pseudocodes. In fact, the majority of the code remains the same, the only difference is in the order in which successive states are removed (polled) from the *Open* queue (line no. 6).

Alg. 1 Breadth-first search

```
1:  procedure BreadthFirstSearch(s_0)                          ▷ initial state s_0
2:      Closed := ∅                                            ▷ empty set of visited states
3:      set reference from s_0 to its parent to null
4:      Open := {s_0}                                          ▷ queue of states to be visited
5:      while Open ≠ ∅ do
6:          remove from Open the state s with the smallest depth   ▷ 'poll' operation
7:          if s is the goal state then return s               ▷ solution found
8:          generate descandants {t} of s
9:          for all t do
10:             if t ∉ Closed and t ∉ Open then add t to Open
11:         add s to Closed
12:     return null                                            ▷ no solution found
```

Alg. 2 Depth-first search

```
1:  procedure DepthFirstSearch(s_0)                            ▷ initial state s_0
2:      Closed := ∅                                            ▷ empty set of visited states
3:      set reference from s_0 to its parent to null
4:      Open := {s_0}                                          ▷ queue of states to be visited
5:      while Open ≠ ∅ do
6:          remove from Open the state s with the largest depth    ▷ 'poll' operation
7:          if s is the goal state then return s               ▷ solution found
8:          generate descandants {t} of s
9:          for all t do
10:             if t ∉ Closed and t ∉ Open then add t to Open
11:         add s to Closed
12:     return null                                            ▷ no solution found
```

In the pseudocodes we implicitly assume that all states are aware of their depths (on the programmistic level, each state object is equipped with an integer depth field). Once a descendant t is generated from s, the depth of t becomes equal to the depth of s plus one.

Due to the wanted order, the *Open* set can in practice be implemented as a simple FIFO (First In First Out) collection (an ordinary queue) for the breadth-first case, and as a LIFO (Last In First Out) collection (a stack) for the depth-first case. Yet, for large graphs both FIFO and LIFO data structures may become unefficient. We discuss more efficient data structures, better suited for more advanced algorithms and larger graphs to be searched, in the successive sections.

3.1.2 Dijkstra's algorithm

The famous shortest path finding algorithm, known popularly as Dijkstra's algorithm, was conceived by Edsgar Dijkstra in 1956 and published in the work (Dijkstra, 1959). Originally, the algorithm has been formulated so that it allows to find *all* shortest paths between an initial graph node (state) and *all* the remaining nodes (a.k.a. single source all shortest paths). This explains why the algorithm does *not* use any heuristics that would reduce the search time by estimating the remaining distance to the goal, since there exist no single goal.

On the other hand, the algorithm can easily be modified so that it stops sooner (before shortest paths to all nodes are establised) — at the moment it reaches a particular node (state) distinguished as the goal. We use this idea in our presentation of the algorithm. In fact, the reader shall see later that Dijkstra's algorithm, formulated as the shortest path finder for a single goal, can be viewed as a special case of the A^* algorithm, described later[1].

Let $g(s)$ denote the exact cost (or distance) for traveling from the initial state to the state s. Let $\Delta(s \to t)$ denote the cost of transition from s to t, where t is a direct descendant (neighbour) of s. Then, the Dijkstra's algorithm can be formulated as follows.

Alg. 3 Dijkstra's algorithm

```
1:  procedure DIJKSTRA(s_0)                                   ▷ initial state s_0
2:      Closed := ∅                                           ▷ empty set of visited states
3:      g(s_0) := 0                                           ▷ distance covered from start
4:      set reference from s_0 to its parent to null
5:      Open := {s_0}                                         ▷ queue of states to be visited
6:      while Open ≠ ∅ do
7:          remove from Open the state s with the smallest g(s)   ▷ 'poll' operation
8:          if s is the goal state then return s              ▷ solution found
9:          generate descandants {t} of s
10:         for all t do
11:             if t ∈ Closed then continue                   ▷ t already visited
12:             g(t) := g(s) + Δ(s → t)
13:             set reference from t to its parent to s
14:             if t ∉ Open then
15:                 add t to Open
16:             else
17:                 if new g(t) is smaller than value known so far then
18:                     replace t in Open with the new one
19:                     update position of t in Open
20:         add s to Closed
21:     return null                                           ▷ no solution found
```

[1] If the heuristic summand of the ordering function used in the A^* algorithm is forced to be always zero, the A^* algorithm becomes the Dijkstra's algorithm.

It is worth to remark that by means of update steps — $g(t) := g(s) + \Delta(s \to t)$ — the g cost can be regarded as a cost observed along the way, which makes it exact.

In practice on the programmistic level, the *Open* set in Dijkstra's algorithm is commonly implemented as a *priority queue*. It is a data structure typically based on a *binary heap*[2], which helps to efficiently retrieve the successive states from the queue preserving the order induced by the g function. Addition of an element (state) into the priority queue is of logarithmic complexity — $O(\log_2 n)$, where n stands for the number of elements in the queue. Removal of the minimal element from the head of the queue (poll operation) is also $O(\log_2 n)$ complex. The latter can be explained by the fact that although the minimal element can be quickly peeked at in $O(1)$ time, as it is the first element in the heap, after this element is removed the heap must be rearranged[3] which requires logarithmic time. As regards the 'replace' or 'contains' operation performed on a priority queue (needed in lines no. 14–19 in the algorithm), they are unfortunately of linear time — $O(n)$ — at least for typical priority queue implementations in existing programming languages. In *SaC*, we offer some improvements with respect to these operations by applying auxiliary data structures. This topic is discussed further in the API-related section 3.2 for graphs.

As regards implementations of the *Closed* set, one should mostly care about fast performance of 'contains' and 'add' operations. Therefore, typically *Closed* sets are implemented as *hash maps* having constant or amortized-constant complexity — $O(1)$. The hash maps are fast at the cost of RAM memory consumption. This might become a problem when very large graphs are to be searched. In *SaC*, we offer an alternative possibility to use a *sorted tree* as a *Closed* set, which allows for some memory savings but implies logarithmic complexity — $O(\log_2 n)$.

3.1.3 Best-first search (BFS)

Pearl (1984) proposed a search approach in which the algorithm expands always the most promising (the 'best') state in the first order. The evaluation how promising a state s is, is performed by a so called heuristic function $h(s)$. This function can be constructed differently, and in general may depend on: the information contained in s itself, the information collected along the path from the initial state up to s, some general knowledge about the problem and the desired properties of the solution state. By convention $h(s)$ is defined as non-negative function and its smaller values suggest closeness of s to the goal state. Informally, $h(s)$ can be therefore treated as a distance function.

In general, Best-first search algorithms and heuristics applied in them are designed to quickly reach the goal state by *any* path. Therefore, they do not focus on optimizing the path anyhow. In

[2]Binary heaps can be regarded as binary trees, commonly implemented using dynamically extensible arrays. An element at index i in the array has its children under indices $2i + 1$ and $2i + 2$ (with indexing starting from zero). Oppositely, the parent of an i-th element is under the index $\lfloor (i - 1)/2 \rfloor$. There are two types of heaps: max-heaps and min-heaps depending on the wanted order to be imposed. For Dijkstra's algorithm a min-heap is used, satisfying the following condition: $g(s_1) \leq g(s_2)$ and $g(s_1) \leq g(s_3)$, for all s_1, s_2, s_3 where s_2, s_3 are children elements (in the heap's array) for the element s_1.

[3]The rearrangement of the heap is carried out by placing temporarily the last element from the underlying array in the place of the first element, and by moving this element downwards (swapping it with one of its children) as long as the heap condition is not met. The number of swaps is at most equal to the height of the tree represented by the heap, thus approximately $O(\log_2 n)$.

other words by using the best-first approach we do not care how long the path is, neither in terms of the number of states (manipulations) along it, nor its cost. In fact, the notion of cost of the path (e.g. as it was defined by the g function for Dijkstra's algorithm) does not exist in the BFS.

Alg. 4 Best-first search

```
 1: procedure BestFirstSearch(s_0)                          ▷ initial state s_0
 2:     Closed := ∅                                         ▷ empty set of visited states
 3:     calculate h(s_0)                                    ▷ heuristics according to provided recipe
 4:     set reference from s_0 to its parent to null
 5:     Open := {s_0}                                       ▷ queue of states to be visited
 6:     while Open ≠ ∅ do
 7:         remove from Open the state s with the smallest h(s)   ▷ 'poll' operation
 8:         if s is the goal state then return s            ▷ solution found
 9:         generate descandants {t} of s
10:         for all t do
11:             if t ∈ Closed then continue                 ▷ t already visited
12:             calculate h(t)
13:             set reference from t to its parent to s
14:             if t ∉ Open then
15:                 add t to Open
16:             else
17:                 if new h(t) is smaller than value known so far then
18:                     replace t in Open with the new one
19:                     update position of t in Open
20:         add s to Closed
21:     return null                                         ▷ no solution found
```

A comment should be made on the fragment of the algorithm (lines 16–19), where it discovers, for a descendant state t, that another instance of that state already exists in the *Open* set, and then checks if the newly calculated $h(t)$ is smaller (better) than the value known so far. A question can be posed: can the heuristic function return different values for two instances of the same state? In the general case, the answer is 'yes' — e.g. if the h function is not solely the function of a state, but also depends on some information collected along the path. Nevertheless, most commonly in practice the BFS applications use heuristic functions being constant for multiple instances of the same state.

3.1.4 A^*

The A^* algorithm was proposed by Haart, Nilsson and Raphael in (Hart, Nilsson and Raphael, 1968; Hart, Nilsson and Raphael, 1972). Informally, the algorithm can be regarded as a combination of Dijkstra's and BFS algorithms (or a more general variant of them). It is because the algorithm uses both the exact cost function g and the heuristic cost function h to decide about the order in

which states are visited.

More strictly, let the evaluation function (deciding about the retrieval order from the *Open* set) for a state s be of form:

$$f(s) = g(s) + h(s), \tag{3.1}$$

where $g(s)$ is the exact cost or distance (observed along the way) for traveling from the initial state to the state s, and $h(s)$ is a heuristic estimation of the cost remaining to the goal state. Because the summand $h(s)$ is a heuristic function, the whole $f(s)$ function can also be regarded as a heuristics. Beneath, we present the pseudocode of the algorithm.

Alg. 5 A^*

```
 1: procedure AStar(s_0)                                   ▷ initial state s_0
 2:     Closed := ∅                                        ▷ empty set of visited states
 3:     g(s_0) := 0                                        ▷ distance covered from start
 4:     calculate h(s_0)                                   ▷ heuristics according to provided recipe
 5:     f(s_0) := g(s_0) + h(s_0)                          ▷ sum deciding about order of Open queue
 6:     set reference from s_0 to its parent to null
 7:     Open := {s_0}                                      ▷ queue of states to be visited
 8:     while Open ≠ ∅ do
 9:         remove from Open the state s with the smallest f(s)   ▷ 'poll' operation
10:         if s is the goal state then return s           ▷ solution found
11:         generate descandants {t} of s
12:         for all t do
13:             if t ∈ Closed then continue                ▷ t already visited
14:             g(t) := g(s) + Δ(s → t)
15:             calculate h(t)
16:             f(t) := g(t) + h(t)
17:             set reference from t to its parent to s
18:             if t ∉ Open then
19:                 add t to Open
20:             else
21:                 if new f(t) is smaller than value known so far then
22:                     replace t in Open with the new one
23:                     update position of t in Open
24:         add s to Closed
25:     return null                                        ▷ no solution found
```

For the shortest (smallest cost) path finding applications, it is crucial that the h function is a so called ***admissible heuristics***. This means that h must not overestimate the unknown true cost remaining to the goal state. In other words h should be a lower bound on the remaining cost.

If the heuristics is admissible and the A^* algorithm finds a goal state than it is guaranteed that this state is the optimal solution — has the shortest path. *Proof*: Imagine the A^* algorithm stops at a certain point (line no. 10) and returns a state s^* with the cost value $g(s^*)$. Obviously, $h(s^*) = 0$,

since s^* satisfied the stop condition. We know that the *Open* queue preserves the non-decreasing order of removal with respect to f function for successive states. Therefore, it is known that all states $s \in Open$, remaining in the queue at the stop moment, satisfy the condition $f(s) \geq f(s^*)$. Two cases should now be considered. (1) If some state s satisfies the stop condition, then $h(s) = 0$ but $g(s) \geq g(s^*)$, since $f(s) \geq f(s^*)$. In other words s is also the goal state but with the cost of its path not cheaper than the cost of path for s^*. (2) If some state s does not satisfy the stop condition, but can potentially lead to the goal state later and we have that $g(s) \leq g(s^*)$, then the final path will for sure be not cheaper than the path for s^*, because the lower bound $h(s) > 0$ on the remaining true cost indicates that $g(s) + h(s) \geq g(s^*)$, again because $f(s) \geq f(s^*)$. ■

An auxiliary notion of *monotnous heuristics* is also useful in this context. A heuristics is said to be monotnous if for all pairs s, t, where t is a descendant of s, the following condition is satisfied:

$$f(s) \leqslant f(t), \tag{3.2}$$

which can also be rewritten as

$$\begin{aligned} g(s) + h(s) &\leqslant g(t) + h(t), \\ h(s) &\leqslant g(t) - g(s) + h(t), \\ h(s) &\leqslant \Delta(s \to t) + h(t). \end{aligned} \tag{3.3}$$

The last inequality can be restated as follows: the heuristic value at s must not be greater than the cost of transition from s to t plus the heuristic value at t. The inequality can become equality only in cases when one travels towards the goal along a straight line (in the sense of metrics associated with the given graph). Moreover, if a heuristics is monotonous then it is also admissible.

Historically, the algorithm was first named as A algorithm. In the notational sense, this was related to the fact that if we consider two heuristics h and h^*, where h^* is marked with a star to denote an optimal heuristics (returning exact costs remaining to the goal), then the algorithm using h^* can also be called optimal and denoted as A^*. Moreover, the optimality of such an algorithm is twofold. Firstly, it performs best (visits the fewest states) among all A algorithms. Secondly, it performs not worse than *all* other graph search algorithms (not necessarily from the family of A algorithms) which are worse or equally well informed in the sense of information stored in h^*.

3.1.5 IDA^*

For some problems where the search graph is very large, the A^* algorithm may run into RAM memory consumption troubles. The number of states in both *Open* and *Closed* sets might potentially exhaust the whole RAM.

Iterative Deepening A^* (IDA^*) proposed by Korf (1985) can be viewed as a variant of A^* with low memory consumption. The IDA^* does not keep record of visited states, i.e. does not use the *Closed* set. Also, it typically keeps in memory only the states residing on the path it currently pursues. Depending on the algorithm formulation, it either uses only a small *Open* set (non recursive formulation with a main while loop) or does not use the *Open* set at all (recursive formulation). Obviously, the low memory usage is not for free — the IDA^* is slower than A^* because it visits many states multiple times.

The idea behind the *IDA** can be sketched as follows. First, the algorithm uses the heuristic value calculated for the initial state $h(s_0)$ to establish the search horizon $H = f(s_0) = 0 + h(s_0)$. Then, the algorithm follows different search paths outgoing from the initial state in a depth-first manner. If it reaches a goal state within the search horizon (i.e. if the observed cost g for the reached state is less or equal to H) then it returns the goal and the associated path. Whenever the algorithm reaches a state outside the search horizon, then the state is not pursued further (its descendants are not generated), but the algorithm may use the observed cost for this state and its heuristics to establish a new search horizon H'. More precisely the new search horizon is derived as:

$$H' = \min_{\{s:\, g(s)>H\}} f(s). \tag{3.4}$$

Finally, when all paths reaching outside the current horizon H are exhausted then the algorithm deepens the search horizon by the substitution $H := H'$, and the whole process is repeated.

We now present the pseudocodes for two *IDA** formulations — a recursive one and a non-recursive (with a main while loop).

Alg. 6 Recursive *IDA**

```
1:  procedure RecursiveIterativeDeepeningAStar(s_0)          ▷ initial state s_0
2:      g(s_0) := 0                                          ▷ distance covered from start
3:      calculate h(s_0)                                     ▷ heuristics according to provided recipe
4:      f(s_0) := g(s_0) + h(s_0)                            ▷ sum deciding about order of Open queue
5:      set reference from s_0 to its parent to null
6:      H := f(s_0)                                          ▷ first search horizon
7:      while true do
8:          (s, H') :=Search(s_0, H)
9:          if s ≠null then return s                         ▷ solution found
10:         if H' = ∞ then return null                       ▷ no solution found
11:         H := H'
12: procedure Search(s, H)                                   ▷ initial state s_0
13:     if f(s) > H then return (null, f(s))
14:     if s is the goal state then return (s, g(s))         ▷ solution found
15:     H' := ∞
16:     generate descandants {t} of s
17:     for all t do
18:         g(t) := g(s) + Δ(s → t)
19:         f(t) := g(t) + h(t)
20:         (u, H'') :=Search(t, H)
21:         if u ≠null then return (u, g(u))                 ▷ solution found
22:         H' := min{H', H''}                               ▷ deepening the horizon
        return (null, H')
```

Alg. 7 *IDA**

```
1:  procedure IterativeDeepeningAStar(s_0)                          ▷ initial state s_0
2:      g(s_0) := 0                                                 ▷ distance covered from start
3:      calculate h(s_0)                                            ▷ heuristics according to provided recipe
4:      f(s_0) := g(s_0) + h(s_0)                                   ▷ sum deciding about order of Open queue
5:      set reference from s_0 to its parent to null
6:      Open := {s_0}                                               ▷ queue of states to be visited
7:      H := f(s_0)                                                 ▷ first search horizon
8:      H' := ∞                                                     ▷ next search horizon
9:      while Open ≠ ∅ do
10:         remove from Open the state s with the smallest f(s)     ▷ 'poll' operation
11:         if g(s) > H then
12:             H' := min{H', f(s)}
13:             if Open = ∅ then
14:                 H := H'                                         ▷ deepening the horizon
15:                 H' := ∞
16:                 Open := {s_0}
17:             continue
18:         if s is the goal state then return s                    ▷ solution found
19:         generate descandants {t} of s
20:         for all t do
21:             g(t) := g(s) + Δ(s → t)
22:             calculate h(t)
23:             f(t) := g(t) + h(t)
24:             set reference from t to its parent to s
25:             if t ∉ Open then
26:                 add t to Open
27:             else
28:                 if new f(t) is smaller than value known so far then
29:                     replace t in Open with the new one
30:                     update position of t in Open
31:     return null                                                 ▷ no solution found
```

3.2 API

3.2.1 Graph state abstraction

In *SaC*'s API for searching graphs, the main 'actor' is the `sac.graph.GraphState` interface being an extension of the `sac.State` discussed earlier in Chapter 2. Below, we present the code listing for the `sac.graph.GraphState` interface and then some excerpts from its default implementation — `sac.graph.GraphStateImpl`. For full code of the implementation the reader is addressed to Appendix 6.2.

```java
package sac.graph;

import java.util.List;

import sac.State;

public interface GraphState extends State {

    @Override
    public GraphState getParent();
    @Override
    public List<GraphState> getChildren();
    @Override
    public List<GraphState> getPath();
    @Override
    public List<GraphState> generateChildren();

    public double getG();
    public double getF();
    public boolean isSolution();
    public void refreshCosts();
}
```

The first thing to note is that the `GraphState` interface reformulates the signatures of four basic methods related to the parent – children binding and path tracking. The difference (of cosmetic nature) is in the types returned, which now become `GraphState` or `List<GraphState>`, as the resulting objects should be treated as graph states (rather then general states).

The new elements in the interface (with respect to its ancestor `sac.State`) are cost related methods: `getG()`, `getF()`, `refreshCosts()`, and the `isSolution()` method. As regards the cost related methods the relevant excerpts from their implementation are listed below.

```java
public abstract class GraphStateImpl extends StateImpl implements GraphState {

    ...

    protected Double g = 0.0;
    protected Double f = null;
    protected static StateFunction gFunction;

    public static class GFunction extends StateFunction {
        @Override
        public double calculate(State state) {
            return (state.getParent() == null) ? 0.0 : ((GraphState) state.getParent()).getG() +
                1.0;
        }
    }

    public final static void setGFunction(StateFunction gFunction) {
        GraphStateImpl.gFunction = gFunction;
    }

    @Override
    public final double getG() {
        if (g == null)
            g = Double.valueOf(gFunction.calculate(this));
        return g;
```

```
25      }
26
27      @Override
28      public final double getF() {
29          if (f == null)
30              f = Double.valueOf(getG() + getH());
31          return f;
32      }
33
34      @Override
35      public final void refreshCosts() {
36          g = Double.valueOf(gFunction.calculate(this));
37          h = Double.valueOf(hFunction.calculate(this));
38          f = Double.valueOf(getG() + getH());
39      }
40
41      static {
42          gFunction = new GFunction();
43      }
44  }
```

As one may note, graph states are by default equipped with a function of observed cost (g function) which treats each transition from a parent to its child as being of cost 1. This default behavior is often valid for many graph problems where our objective is to minimize the number of moves or manipulations leading from the initial state to the goal state (e.g. sliding puzzle, Rubik's cube, simple non-weighed graphs). This assumption makes the g function behave identically as depth. If it is user's wish to change this default behaviour for a given problem, he must provide an implementation of a `StateFunction` and attach statically it to his class (being an extension of the `GraphState` class) by the `setGFunction(...)` method.

It is worth to remark that the calculations of g and f functions are postponed until the very first calls of respective getters — `getG()` and `getF()`. It is the same design idea as was discussed earlier in the context of identifiers and heuristics (h function). And its purpose is to minimize the number of actual calculations. The `refreshCosts()` method is meant to be called by search algorithms, typically just after the moment where descendant states are generated and then processed one after another. Explicit user calls to this method are in practice not needed (exceptions are the presentation-related or online game playing situations discussed already in Section 2.5).

As regards the `isSolution()` method, the user should provide its implementation in compliance with the nature of his search problem. When a state represents the goal state (satisfies the desired terminal properties) then the `isSolution()` method should naturally return `true` (otherwise — `false`).

3.2.2 General graph search algorithm

In this section we discuss *SaC*'s internal mechanisms related to the general graph searching procedure. This procedure is represented by an abstract class: `sac.graph.GraphSearchAlgorithm`. Obviously, the end user of *SaC* library does not have to be aware of core-level intricacies. Instead, in practice he can limit himself to instantiating a specific search algorithm (e.g.: `new Dijkstra(...)` or `new AStar(...)`, etc.) and running it for his particular problem. Therefore, we recommend the reading of subsequent contents only to readers really interested in low level details of *SaC*,

perhaps the readers intending to extend the library in the future with new algorithms or data structures.

The listing below demonstrates the most important parts of the `GraphSearchAlgorithm` class. For clarity, some less interesting parts have been skipped. The class is designed to work as a general and common (model) procedure for searching graphs. As the reader shall see later, the subclasses representing specific search algorithms (e.g. `BreadthFirstSearch`, `DepthFirstSearch`, `Dijsktra`, etc.) are, in the programmistic sense, very light extensions of the `GraphSearchAlgorithm` class. In fact, those light extensions only redefine the order according to which states are removed from the *Open* set[4].

```
1  package sac.graph;
2
3  ...
4
5  public abstract class GraphSearchAlgorithm extends SearchAlgorithm {
6
7      protected GraphState initial = null;
8      protected OpenSet openSet = null;
9      protected ClosedSet closedSet = null;
10
11     protected GraphSearchConfigurator configurator = null;
12
13     protected List<GraphState> solutions = null;
14     protected int step = 0;
15     protected GraphState current = null;
16     protected GraphState bestSoFar = null;
17
18
19     public GraphSearchAlgorithm(GraphState initial, GraphSearchConfigurator configurator) {
20         ...
21     }
22
23     protected void setupOpenAndClosedSets(Comparator<GraphState> openSetComparator) {
24         ...
25     }
26
27     protected void reset() {
28         ...
29     }
30
31     ... // getters, setters
32
33     @Override
34     public void execute() {
35         ... // initialization of data structures and monitoring thread
36         doExecute(); // actual search start
37         ... // stoppage of monitoring threads
38     }
39
40     protected void doExecute() {
41         ... // the body of the main search procedure, presented later on
42     }
43 }
```

[4]Exception to this rule is the `IterativeDeepeningAStar` class, which additionally overrides the main search procedure — `execute()`.

We now concentrate on the `doExecute()` method. It can be viewed as a counterpart of algorithmic pseudocodes discussed earlier in Section 4.1. Here is the full listing of the method.

```
1  protected void doExecute() {
2      startTime = System.currentTimeMillis();
3      if (initial == null)
4          return;
5      openSet.add(initial);
6      step = 0;
7      while (!openSet.isEmpty()) {
8          step++;
9
10         // time limit check
11         if (configurator.getTimeLimit() < Long.MAX_VALUE) {
12             long currentTime = System.currentTimeMillis();
13             if (currentTime - startTime > configurator.getTimeLimit()) {
14                 endTime = System.currentTimeMillis();
15                 break;
16             }
17         }
18
19         // poll current best from queue
20         current = openSet.poll();
21
22         // putting current to closed set
23         if (configurator.isClosedSetOn())
24             closedSet.put(current);
25
26         // keeping best so far
27         if ((initial.getH() > 0) && ((bestSoFar == null) || (current.getH() < bestSoFar.getH())))
28             bestSoFar = current;
29
30         // solution check
31         boolean isSolution = current.isSolution();
32
33         // registering solution
34         if (isSolution) {
35             if (solutions.isEmpty())
36                 bestSoFar = current;
37             solutions.add(current);
38             if (configurator.getWantedNumberOfSolutions() == solutions.size())
39                 break;
40         }
41
42         // generating children
43         List<GraphState> children = current.generateChildren();
44
45         // iterating over children
46         for (GraphState child : children) {
47
48             // check if child was closed
49             boolean closedSetContains = (configurator.isClosedSetOn()) ? closedSet.contains(child
                   ) : false;
50
51             if (!closedSetContains) { // child not in closed set
52
53                 // set child -> parent link and depth
54                 child.setParent(current);
55                 child.setDepth(current.getDepth() + 1);
56
```

```
57                  // update scores g, h, f
58                  child.refreshCosts();
59
60                  // check if child is in open set
61                  boolean openSetContains = openSet.contains(child);
62
63                  if (!openSetContains) {
64                      // add child reference to parent
65                      if (configurator.isParentsMemorizingChildren())
66                          current.getChildren().add(child);
67
68                      // add child to open set
69                      openSet.add(child);
70
71                  } else {
72                      // getting reference to child existing in open set
73                      GraphState existingChild = openSet.get(child);
74
75                      // replacing, if new child better than existing
76                      if (openSet.getComparator().compare(child, existingChild) < 0) {
77                          openSet.replace(existingChild, child);
78
79                          // add child reference to parent (better child)
80                          if (configurator.isParentsMemorizingChildren())
81                              current.getChildren().add(child);
82
83                          // removing from some other parent reference to worse existing child
84                          existingChild.getParent().getChildren().remove(existingChild);
85                      }
86                  }
87              }
88          }
89      }
90
91      endTime = System.currentTimeMillis();
92  }
```

Apart from several auxiliary operations (time limit checking, memorizing the solutions, memorizing the best state so far) the important steps involved in the above procedure proceed as in most of algorithms discussed earlier.

The two crucial places to note are invocations of callback methods: `current.isSolution()` and `current.generateChildren()`. Their implementations must be provided by the user. Note also that found solutions are not returned directly from the procedure, but memorized in the internal field (`List<GraphState> solutions`) of the class. Once the wanted number of solutions is reached (by default it is 1), the procedure is stopped.

The `doExecute()` method is not called directly by the user (it is under `protected` scope). Instead, the user calls a wrapper method `execute()`. Its full listing is as follows.

```
1   public void execute() {
2       reset();
3
4       GraphSearchMonitor monitor = null;
5       Thread monitorThread = null;
6
7       if (configurator.isMonitorOn()) {
8           // starting monitor thread
```

```
9           try {
10              Constructor<GraphSearchMonitor> constructor = (Constructor<GraphSearchMonitor>) Class
                    .forName(configurator.getMonitorClassName())
11                      .getConstructor(GraphSearchAlgorithm.class, Long.TYPE);
12              monitor = (GraphSearchMonitor) constructor.newInstance(this, configurator.
                    getMonitorRefreshTime());
13          } catch (Exception e) {
14              monitor = new ConsoleGraphSearchMonitor(this, configurator.getMonitorRefreshTime());
15              e.printStackTrace();
16          }
17
18          monitorThread = new Thread(monitor);
19          monitorThread.start();
20      }
21
22      doExecute(); // actual search start
23
24      if (configurator.isMonitorOn()) {
25          monitor.stop();
26          // waiting for monitor thread to stop
27          while (true) {
28              if (!monitorThread.isAlive())
29                  break;
30              try {
31                  Thread.sleep(100);
32              } catch (InterruptedException ie) {
33              }
34          }
35      }
36  }
```

As one can note, the overhead around the actual `doExecute()` call boils down to: resetting the algorithm and operations on the monitoring thread. As regards the latter, the monitoring thread allows the user to watch the progress of a graph search, e.g.: number of states in *Open* and *Closed* sets, number of solutions found so far, RAM usage, etc. There are two monitor classes provided with *SaC*: `sac.graph.ConsoleGraphSearchMonitor` and `sac.graph.GraphicalGraphSearchMonitor`. Their common abstraction is a superclass named: `sac.graph.ConsoleGraphSearchMonitor` — the type actually used in the code. More details on monitors is given later in Section 5.3. Before starting a graph search, the suitable monitor is brought to life (using information from the configurator object) and triggered as a side thread. After the `doExecute()` method is finished, the monitor thread is stopped.

As regards the `reset()` method, it involves the following operations: (1) setting up the wanted type for state identifiers (hash codes or strings) from the configurator object[5], (2) refreshing the initial object (in particular: detaching the list of its descendants present after the previous search), (3) clearing the solutions list, (4) nulling auxiliary references to current and best states, (5) setting up data structures for *Open* and *Closed* sets. Below we show the listing of the `reset()` method.

```
1   protected void reset() {
2       Identifier.setType(this.configurator.getIdentifierType());
3
4       if (initial != null) {
```

[5]Its content might have changed since the last search execution.

```
5           initial.refresh();
6           initial.getChildren().clear();
7           initial.refreshCosts();
8       }
9
10      solutions.clear();
11      bestSoFar = null;
12      current = null;
13
14      setupOpenAndClosedSets(openSet.getComparator());
15  }
```

The `setupOpenAndClosedSets(...)` method inspects the information stored in the configurator object and instantiates, by means of the Java reflection mechanism, suitable implementations for *Open* and *Closed* sets (before the search is started). The listing of the method is as follows.

```
1   protected void setupOpenAndClosedSets(Comparator<GraphState> openSetComparator) {
2       // open set
3       try {
4           Constructor<OpenSet> constructor = (Constructor<OpenSet>) Class.forName(configurator.
                getOpenSetClassName()).getConstructor(Comparator.class);
5           this.openSet = (OpenSet) constructor.newInstance(openSetComparator);
6       } catch (Exception e) {
7           this.openSet = new OpenSetAsPriorityQueueFastContainsFastReplace(openSetComparator);
8           e.printStackTrace();
9       }
10
11      // closed set
12      if (configurator.isClosedSetOn())
13          try {
14              Constructor<ClosedSet> constructor = (Constructor<ClosedSet>) Class.forName(
                    configurator.getClosedSetClassName()).getConstructor();
15              this.closedSet = (ClosedSet) constructor.newInstance();
16          } catch (Exception e) {
17              this.closedSet = new ClosedSetAsHashMap();
18              e.printStackTrace();
19          }
20  }
```

This method is also of importance in the context of defining specific search algorithms (subclasses of the general and abstract `GraphSearchAlgorithm`), which is discussed in the subsequent section.

3.2.3 Specific graph search algorithms

As one may have noted, the argument passed to the `setupOpenAndClosedSets(...)` method is a comparator `Comparator<GraphState> openSetComparator`. It decides about the states retrieval order associated with the *Open* set. Both the `setupOpenAndClosedSets(...)` method and the mentioned comparator play an important role in defining the specific search algorithms. This can be best depicted by the following example.

```
1   package sac.graph;
2
3   import java.util.Comparator;
4
```

```
5  public class AStar extends GraphSearchAlgorithm {
6
7      public AStar(GraphState initial, GraphSearchConfigurator configurator) {
8          super(initial, configurator);
9          setupOpenAndClosedSets(new AStarComparator());
10     }
11
12     public AStar(GraphState initial) {
13         this(initial, null);
14     }
15
16     public AStar() {
17         this(null, null);
18     }
19
20     private class AStarComparator implements Comparator<GraphState> {
21         @Override
22         public int compare(GraphState gs1, GraphState gs2) {
23             double difference = gs1.getF() - gs2.getF();
24             if (difference == 0.0) {
25                 return gs1.getIdentifier().compareTo(gs2.getIdentifier());
26             } else {
27                 return (difference > 0.0) ? 1 : -1;
28             }
29         }
30     }
31 }
```

The presented class is an actual definition of the A^* algorithm in *SaC*. As one can see the code is very short. The class extends the `GraphSearchAlgorithm` abstraction, and in its main constructor it instantiates a suitable comparator for the *Open* set. For convenience, the comparator is defined as an inner class (bottom part of the code). In the example, the comparator works as it should be working for the case of A^* algorithm — f function values are used for comparing states. In the case of a tie, the comparison of identifiers decides. In other words, states with equal f values are sorted lexicographically.

The proceedings described above are common for all specific graph search algorithms. To demonstrate it, we present another example, this time of the `BestFirstSearch` class. The programmistic scheme is repeated, and the resulting code is very short again.

```
1  package sac.graph;
2
3  import java.util.Comparator;
4
5  public class BestFirstSearch extends GraphSearchAlgorithm {
6
7      public BestFirstSearch(GraphState initial, GraphSearchConfigurator configurator) {
8          super(initial, configurator);
9          setupOpenAndClosedSets(new BestFirstSearchComparator());
10     }
11
12     public BestFirstSearch(GraphState initial) {
13         this(initial, null);
14     }
15
16     public BestFirstSearch() {
17         this(null, null);
```

```
18      }
19
20      private class BestFirstSearchComparator implements Comparator<GraphState> {
21
22          @Override
23          public int compare(GraphState gs1, GraphState gs2) {
24              double difference = gs1.getH() - gs2.getH();
25              if (difference == 0.0) {
26                  return gs1.getIdentifier().compareTo(gs2.getIdentifier());
27              } else {
28                  return (difference > 0.0) ? 1 : -1;
29              }
30          }
31      }
32  }
```

3.2.4 Variants of *Open* and *Closed* sets

In *SaC*, actual implementations of *Open* or *Closed* sets are separated, in the programmistic sense, from their abstractions — `sac.graph.OpenSet` and `sac.graph.ClosedSet` interfaces, respectively. The listings below present the methods offered by these interfaces.

```
1   package sac.graph;
2
3   import java.util.Comparator;
4
5   public interface OpenSet {
6       public void add(GraphState graphState);
7       public GraphState poll();
8       public GraphState peek();
9       public boolean contains(GraphState graphState);
10      public void replace(GraphState graphState, GraphState replacer);
11      public GraphState get(GraphState graphState);
12      public int size();
13      public boolean isEmpty();
14      public void clear();
15      public Comparator<GraphState> getComparator();
16  }
```

```
1   package sac.graph;
2
3   public interface ClosedSet {
4       public boolean contains(GraphState graphState);
5       public GraphState get(GraphState graphState);
6       public void put(GraphState graphState);
7       public void remove(GraphState graphState);
8       public int size();
9       public boolean isEmpty();
10      public void clear();
11  }
```

Currently in *SaC*, available are three variants (implementations) of the *Open* set and two variants of the *Closed* set, as listed in tables 3.1 and 3.2. The variants differ with respect to the

computational complexity of operations involved. Faster variants are naturally more memory consuming.

Table 3.1: Available variants (implementations) of the *Open* set and their computational complexities.

class name (in `sac.graph` package)	`poll`	`contains/get`	`add`	`replace`
`OpenSetAsPriorityQueue`	$O(\log_2 n)$	$O(n)$	$O(\log_2 n)$	$O(n)$
`OpenSetAsPriorityQueueFastContains`	$O(\log_2 n)$	$O(1)$	$O(\log_2 n)$	$O(n)$
`OpenSetAsPriorityQueueFastContainsFastReplace`	$O(\log_2 n)$	$O(1)$	$O(\log_2 n)$	$O(\log_2 n)$

Table 3.2: Available variants (implementations) of the *Closed* set and their computational complexities.

nazwa klasy (w `sac.graph`)	`contains/get`	`put`
`ClosedSetAsTreeMap`	$O(\log_2 n)$	$O(\log_2 n)$
`ClosedSetAsHashMap`	$O(1)$	$O(1)$

As regards the *Open* set, the first implementation listed in Table 3.1 is `OpenSetAsPriorityQueue`. It is based on a standard priority queue class — `java.util.PriorityQueue` — offered within J2SE. This class in turn is underneath based on a *binary heap* data structure. It is very likely that the `java.util.PriorityQueue` class would be a programmer's choice if a search algorithm was to be implemented in Java from scratch. Although `poll` and `add` operations are of logarithmic complexity — $O(\log_2 n)$ — for this class, the `contains` operation is linear — $O(n)$. The reason is that to look for a particular object within a binary heap (`contains` operation), one must sequentially traverse all the objects it contains. As regards the `replace` operation, it is worth to remark that a suitable method for that purpose is *not* offered directly by J2SE in the `java.util.PriorityQueue` class. In order to do a replacement one must combine several methods. First, find the object by iterating over all objects in the queue (linear time), then remove it (logarithmic time), and finally insert its replacement into the heap (logarithmic time). Therefore, the finding part is the slowest, which makes the complexity of the whole operation $O(n)$.

The second variant of the *Open* set is the `OpenSetAsPriorityQueueFastContains` class. It is faster from the first variant with respect to the `contains` operation. On the low level, apart from a standard priority queue we have equipped this class with an auxiliary *hash map* (`java.util.HashMap`). The hash map memorizes the pairs <`Identifier, GraphState`>, where the `Identifier` works as a key and the reference to `GraphState` works as a value in the map. Every time a state is being added to the *Open* set, it is added both the priority queue and the hash map. Owing to the latter, one is able later to quickly check, in $O(1)$ time, if a state with a particular identifier is present in *Open*. Morover, since identifiers are coupled with references to states, one can also quickly check if the value of a suitable comparing function (g, f, h, depth, etc.) in the existing state is better than for the state currently processed. By that one knows if a `replace` operation is necessary.

The third variant of the *Open* set is the class: `OpenSetAsPriorityQueueFastContainsFastReplace`. This is the variant turned on by default. The class uses a custom implementation of the binary heap, provided by the authors of *SaC*. The heap is implemented as a *dynamic array* data structure

(`java.util.ArrayList`). As an auxiliary structure, a hash map is again involved. Firstly, this hash map makes the `contains` operation $O(1)$ fast (as in the former variant). But secondly (and more importantly), this hash map allows also to have a quick access to array *indices* of states contained in the heap. The following code excerpt shows how this is achieved.

```
1  package sac.graph;
2
3  ...
4
5  public class OpenSetAsPriorityQueueFastContainsFastReplace extends OpenSetImpl {
6
7      private ArrayList<GraphState> binaryHeap;
8      private Map<Identifier, MapEntry> map;
9
10     private class MapEntry {
11         private GraphState graphState;
12         private int binaryHeapIndex;
13         ...
14     }
15
16     public OpenSetAsPriorityQueueFastContainsFastReplace(Comparator<GraphState> comparator) {
17         super(comparator);
18         binaryHeap = new ArrayList<GraphState>(1024 * 1024);
19         map = new HashMap<Identifier, MapEntry>(1024 * 1024, (float) 0.75);
20     }
21
22
23     @Override
24     public void replace(GraphState graphState, GraphState replacer) {
25         Identifier identifier = graphState.getIdentifier();
26         Integer index = map.get(identifier).getBinaryHeapIndex();
27         if (index == null)
28             return;
29         binaryHeap.set(index.intValue(), replacer);
30         map.get(identifier).setGraphState(replacer);
31         reheapUp(index);
32     }
33
34     protected void reheapUp(int childIndex) {
35         if (childIndex == 0)
36             return; // stop of recursion
37         int parentIndex = (childIndex - 1) / 2;
38         GraphState parent = binaryHeap.get(parentIndex);
39         GraphState child = binaryHeap.get(childIndex);
40         if (comparator.compare(parent, child) > 0) { // comparator comes from OpenSetImpl
41             binaryHeap.set(parentIndex, child);
42             binaryHeap.set(childIndex, parent);
43             map.get(child.getIdentifier()).setBinaryHeapIndex(parentIndex);
44             map.get(parent.getIdentifier()).setBinaryHeapIndex(childIndex);
45             reheapUp(parentIndex);
46         }
47     }
48
49     ...
50 }
```

When a replacement is to be done, the suitable index is read from the map and the replacing (better) state is inserted into the heap related array under that index. Then, the heap has to be

reorganized upwards — `reheapUp(...)` method — starting on from the new state, so that the heap condition is satisfied. This is done in $O(\log_2 n)$ time.

We now move on to discuss *Closed* set variants.

Currently in *SaC*, available are two *Closed* set variants. The default one is represented by the class: `ClosedSetAsHashMap`. As the name suggests, it is based on a standard (J2SE) hash map — `java.util.HashMap` — allowing for $O(1)$ complexity of all operations one needs in graph searching: `put`, `get`, `contains`. The trade-off is obviously in high memory consumption. By default, an instantiation of the `ClosedSetAsHashMap` class, creates underneath a map with an initial capacity of $\approx 8 \cdot 10^6$ entries, and the *load factor* of 0.75 (as shown in the listing below).

```
1  package sac.graph;
2
3  ...
4
5  public class ClosedSetAsHashMap implements ClosedSet {
6
7      private Map<Identifier, GraphState> map;
8
9      public ClosedSetAsHashMap() {
10         this.map = new HashMap<Identifier, GraphState>(8 * 1024 * 1024, (float) 0.75);
11     }
12
13     ...
14 }
```

The second *Closed* set variant — `ClosedSetAsTreeMap` — has been provided in *SaC* with an intention to mitigate some of memory consumption problems one can come accross more quickly when using a slightly faster hash map based variant. The `ClosedSetAsTreeMap` implementation is based on the J2SE's `java.util.TreeMap` class, which in turn underneath is based on the *red–black tree* data structure[6]. The red-black tree guarantees the complexity of all important (for graph searching) operations — `put`, `get`, `contains` — to be $O(\log_2 n)$.

[6]Red–black trees are self-balancing binary search trees. The idea is due to Bayer (1972) who originally named the structure as *symmetric binary B-Tree*. The contemporary modern name is due to the paper (Guibas and Sedgewick, 1978). Balance of the tree is preserved by painting each tree node with one of two colors (by convention red and black) and imposing certain properties on occurrence of the colors. The properties are designed in such a manner that tree rearrangements and repaintings are carried out efficiently — in $O(\log_2 n)$ time. The tree is typically not perfectly balanaced, but guarantees that the distance from the root to the furthest leaf is at maximum twice as long as the distance to the closest leaf. This makes the complexity of all crucial operations proportional to the height of the tree, thus also $O(\log_2 n)$

3.2.5 Configuration options for searching graphs

On several occasions we have mentioned the usage of a `GraphSearchConfigurator` object. Below, we present a brief code listing of this class, with all configuration options and their default values. We purposely leave the javadocs present in the listing to make the meaning of options more clear. Every configuration option can be accessed by a suitable getter or setter (that are skipped in the listing).

```
1  public class GraphSearchConfigurator {
2
3      /**
4       * Identifier type for states. By default: HASH_CODE.
5       */
6      private IdentifierType identifierType = IdentifierType.HASH_CODE;
7
8      /**
9       * Class name for open set. By default: sac.graph.OpenSetSacFast.
10      */
11     private String openSetClassName = OpenSetAsPriorityQueueFastContainsFastReplace.class.getName
           ();
12
13     /**
14      * Is closed set on. By default: true. Closed set can be off when the search space is a tree
            (not a graph with cycles).
15      */
16     private boolean closedSetOn = true;
17
18     /**
19      * Class name for closed set. By default: sac.graph.ClosedSetAsHashMap.
20      */
21     private String closedSetClassName = ClosedSetAsHashMap.class.getName();
22
23     /**
24      * Do parents memorize references to their children. Set to false for lower memory usage (
            WARNING: in that case drawing graph via GraphViz is impossible). By default: false.
25      */
26     private boolean parentsMemorizingChildren = false;
27
28     /**
29      * Wanted number of solutions. By default: 1.
30      */
31     private int wantedNumberOfSolutions = 1;
32
33     /**
34      * Time limit in milliseconds. By default: 'infinity' in long type (Long.MAX_VALUE).
35      */
36     private long timeLimit = Long.MAX_VALUE;
37
38     /**
39      * Is monitor on. By default: false;
40      */
41     private boolean monitorOn = false;
42
43     /**
44      * Class name for monitor. By default: sac.graph.DefaultConsoleMonitor.
45      */
46     private String monitorClassName = ConsoleGraphSearchMonitor.class.getName();
47
```

```
48      /**
49       * Monitor time period. By default: 1000 ms.
50       */
51      private long monitorRefreshTime = 1000;
52
53      public GraphSearchConfigurator() {
54      }
55
56      public GraphSearchConfigurator(String propertiesFilePath) throws Exception {
57          ...
58      }
59
60      ... // getters, setters
61  }
```

A configurator can be instatiated either by a default constructor with no arguments — `GraphSearchConfigurator()`, or from a properties text file — `GraphSearchConfigurator(String propertiesFilePath)`. The names of options in the properties file are uppercased versions of field names, and with underscores separating successive words. Below, we show a possible exemplary contents of a configuration `.properties` file with some non-default values.

```
1   #Example of graph configurator settings
2
3   identifierType=STRING
4   openSetClassName=sac.graph.OpenSetAsPriorityQueueFastContains
5   closedSetOn=true
6   closedSetClassName=sac.graph.ClosedSetAsHashMap
7   parentsMemorizingChildren=false
8   wantedNumberOfSolutions=5
9   timeLimit=Long.MAX_VALUE
10  monitorOn=false
11  monitorClassName=sac.graph.GraphicalGraphSearchMonitor
12  monitorRefreshTime=250
```

3.3 Examples

3.3.1 Sliding puzzle

The sliding puzzle is a one-player recreation invented by Noyes Chapman in 1880. The player is given a board containing flat pieces — tiles — commonly with a picture or numbers drawn on them. In the initial state the pieces are shuffled. There is an empty space in the place of one of the tiles and the player can slide adjacent tiles into the empty space. The goal is to bring the board back to its original arrangement. In contrary to human players, from a computer solver we typically require also that the found sequence of moves is minimal.

Most commonly the boards with sliding puzzles are square grids, in general of size $n \times n$, see Fig. 3.1[7]. Due to one tile taken away, the puzzle is also known as $(n^2 - 1)$-puzzle. Eight-puzzles are quite easily solvable by children, but some fifteen-puzzle instances can already be out of reach for adults, especially if the minimal path to solution should be found.

[7] Images aquired from *Google Images* search engine.

Figure 3.1: Sliding puzzle boards.

Implementation of sliding puzzle state

Below we show an implementation of a sliding puzzle state using *SaC*. We represent the puzzle as a one-dimensional array of bytes and implement the required routines: generation of decendants — `generateChildren()`, identification — `hashCode()`, termination — `isSolution()`. For convenience, we introduce a copying constructor and some helper methods: to get possible moves, to make a move, and to shuffle the puzzle. In the ending static block we attach the heuristics to the class — `HFunctionLinearConflicts`, which is discussed a bit later on.

```
1  public class SlidingPuzzle extends GraphStateImpl {
2
3      protected static byte n;
4      protected static byte N; // N = n * n
5      protected byte emptyIndex;
6      protected byte[] board;
7
8      public SlidingPuzzle(int n) {
9          SlidingPuzzle.n = (byte) n;
10         SlidingPuzzle.N = (byte) (n * n);
11         board = new byte[N];
12         for (byte i = 0; i < N; i++)
13             board[i] = i;
14         emptyIndex = 0;
15     }
16
17     public SlidingPuzzle(SlidingPuzzle parent) {
18         board = new byte[N];
19         for (byte i = 0; i < N; i++)
20             board[i] = parent.board[i];
21         emptyIndex = parent.emptyIndex;
22     }
23
```

```
24      public LinkedList<Byte> getPossibleMoves() {
25          LinkedList<Byte> list = new LinkedList<Byte>();
26          if ((emptyIndex % n) + 1 < n)
27              list.add((byte) (emptyIndex + 1));
28          if ((emptyIndex % n) - 1 >= 0)
29              list.add((byte) (emptyIndex - 1));
30          if (emptyIndex + n < N)
31              list.add((byte) (emptyIndex + n));
32          if (emptyIndex - n >= 0)
33              list.add((byte) (emptyIndex - n));
34          return list;
35      }
36
37      private void makeMove(byte newEmptyIndex) {
38          board[emptyIndex] = board[newEmptyIndex];
39          board[newEmptyIndex] = 0;
40          emptyIndex = newEmptyIndex;
41      }
42
43      public void shuffle(int numberOfMoves) {
44          Random randi = new Random();
45          for (int i = 0; i < numberOfMoves; i++) {
46              List<Byte> moves = getPossibleMoves();
47              int index = randi.nextInt(moves.size());
48              byte mov = moves.get(index);
49              makeMove(mov);
50          }
51      }
52
53      @Override
54      public List<GraphState> generateChildren() {
55          List<GraphState> list = new LinkedList<GraphState>();
56          Iterator<Byte> it = getPossibleMoves().listIterator();
57          while (it.hasNext()) {
58              SlidingPuzzle child = new SlidingPuzzle(this);
59              child.makeMove(it.next());
60              String moveName = "D";
61              if (this.emptyIndex - 1 == child.emptyIndex)
62                  moveName = "L";
63              else if (this.emptyIndex + 1 == child.emptyIndex)
64                  moveName = "R";
65              else if (this.emptyIndex - n == child.emptyIndex)
66                  moveName = "U";
67              child.setMoveName(moveName);
68              list.add(child);
69          }
70          return list;
71      }
72
73      @Override
74      public int hashCode() {
75          return Arrays.hashCode(board);
76      }
77
78      public String toString() {
79          StringBuilder stringBuilder = new StringBuilder();
80          StringBuilder line = new StringBuilder();
81          final int cellSize = 5;
82          for (int i = 0; i < n * cellSize + 1; i++)
83              line.append("-");
84
```

```
85          stringBuilder.append(line).append("\n");
86          int k = 0;
87          for (int i = 0; i < n; i++) {
88              for (int j = 0; j < n; j++) {
89                  int boardAtK = board[k];
90                  stringBuilder.append(String.format("|%1$3d␣", boardAtK));
91                  k++;
92              }
93              stringBuilder.append("|\n").append(line);
94              if (i < n - 1)
95                  stringBuilder.append("\n");
96          }
97
98          return stringBuilder.toString();
99      }
100
101     @Override
102     public boolean isSolution() {
103         for (byte i = 0; i < N; i++)
104             if (board[i] != i)
105                 return false;
106         return true;
107     }
108
109     static {
110         setHFunction(new HFunctionLinearConflicts());
111     }
112 }
```

Example of an easy eight-puzzle

The program beneath solves the following puzzle: $(0, 1, 5, 8, 2, 7, 4, 3, 6)$. The puzzle was created by shuffling (starting from a correctly arranged board) with one hundred random moves, but turns out to be an easy one to solve.

```
1  SlidingPuzzle slidingPuzzle = new SlidingPuzzle(3);
2  slidingPuzzle.shuffle(100);
3  System.out.println("SLIDING␣PUZZLE␣TO␣SOLVE:\n" + slidingPuzzle);
4
5  GraphSearchAlgorithm algorithm = new AStar(sp);
6  algorithm.execute();
7  SlidingPuzzle solution = (SlidingPuzzle) algorithm.getSolutions().get(0);
8
9  System.out.println("SOLUTION:\n" + solution);
10 System.out.println("PATH␣LENGTH:␣" + solution.getPath().size());
11 System.out.println("MOVES␣ALONG␣PATH:␣" + solution.getMovesAlongPath());
12 System.out.println("CLOSED␣STATES:␣" + algorithm.getClosedStatesCount());
13 System.out.println("OPEN␣STATES:␣" + algorithm.getOpenSet().size());
14 System.out.println("DURATION␣TIME:␣" + algorithm.getDurationTime() + "␣ms");
```

The program outputs the following result to the console and the graph searched is depicted in Fig. 3.2.

```
SLIDING PUZZLE TO SOLVE:
----------------
|  0 |  1 |  5 |
```

```
----------------
|  8 |  2 |  7 |
----------------
|  4 |  3 |  6 |
----------------
SOLUTION:
----------------
|  0 |  1 |  2 |
----------------
|  3 |  4 |  5 |
----------------
|  6 |  7 |  8 |
----------------
PATH LENGTH: 17
SEQUENCE OF MOVES: [R, D, L, D, R, R, U, L, L, D, R, R, U, U, L, L]
CLOSED STATES: 26
OPEN STATES: 17
DURATION TIME: 0 ms
```

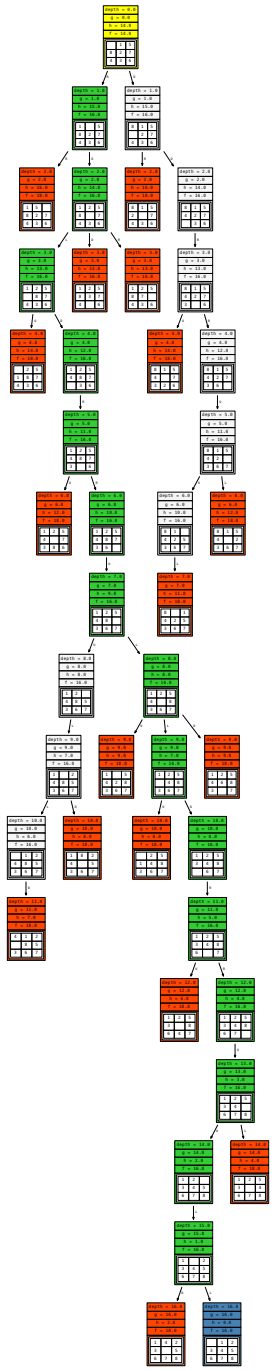


Figure 3.2: Graph searched by *SaC* using "linear conflicts" heuristics for an eight-puzzle $(0, 1, 5, 8, 2, 7, 4, 3, 6)$.

Heuristics for sliding puzzle

An insightful reading on how to construct heuristics in general can be found in (Hansson, Mayer and Yung, 1985). The authors analyze in particular the sliding puzzle problem and describe three heuristics for it: "Misplaced tiles", "Manhattan", "Manhattan + linear conflicts", together with proofs of their admissibility. Below, we explain them in brief and show their implementations in *SaC*.

"Misplaced tiles" This heuristics is a primitive one. It boils down to counting cells (tiles) that are in a wrong place, except for the empty cell. This can be regarded as a relaxed model of the sliding puzzle in which we ignore the actual rules and we are able to freely pick up cells and put them down into the right location, as if each such manipulation — "move" — was of cost 1. Clearly the maximum value of this heuristics is $n^2 - 1$, which can be a significant underestimation of the true minimum cost. Here is an implementation.

```
1  public class HFunctionMisplacedTiles extends StateFunction {
2
3      @Override
4      public double calculate(State state) {
5          SlidingPuzzle slidingPuzzle = (SlidingPuzzle) state;
6          double h = 0.0;
7          for (int i = 0; i < slidingPuzzle.board.length; i++) {
8              if ((i != slidingPuzzle.emptyIndex) && (slidingPuzzle.board[i] != i))
9                  h += 1.0;
10         }
11         return h;
12     }
13 }
```

"Manhattan" This is a popularly known heuristics. It calculates the sum of cell distances, in Manhattan metrics, to their correct location (the distance for the empty cell is excluded). In terms of relaxed models, this can be regarded as a variant of the sliding puzzle in which we do not have to make moves using the empty cell, but rather each cell can move independently. While traveling the tiles may freely stack on one another.

```
1  public class HFunctionManhattan extends StateFunction {
2
3      @Override
4      public double calculate(State state) {
5          SlidingPuzzle slidingPuzzle = (SlidingPuzzle) state;
6          double h = 0.0;
7          for (int i = 0; i < slidingPuzzle.board.length; i++) {
8              if (i != slidingPuzzle.emptyIndex)
9                  h += manhattan(slidingPuzzle, i);
10         }
11         return h;
12     }
13
14     protected int manhattan(SlidingPuzzle slidingPuzzle, int index) {
15         int n = SlidingPuzzle.n;
```

```
16          int i1 = slidingPuzzle.board[index] / n;
17          int j1 = slidingPuzzle.board[index] % n;
18          int i2 = index / n;
19          int j2 = index % n;
20          return Math.abs(i1 - i2) + Math.abs(j1 - j2);
21      }
22  }
```

"Manhattan + linear conflicts" This is the most advanced heuristics of the three. Apart from the Manhattan distances summand it also includes the second summand being the number of so called linear conflicts multiplied by two. What is a linear conflict? Imagine that a first row of a fifteen-puzzle is as follows: $1, 2, 3, 0$. In this row the tiles are not in their correct locations, the sum of Manhattan distances is 3, but there is no linear conflict, since by shifting the 0 element to the left three times the row gets correctly arranged. Now consider the following row: $2, 1, 3, 0$. The sum of Manhattan distances is again 3, but there exist a linear conflict, since 1 is after 2. In consequence, one of these elements at some point during the actual solving will have to be brought down to the second row, and after some time, brought back up. That is why each linear conflict requires at least two additional moves. Linear conflicts shoud be detected both in rows and columns. Here is an implementation.

```
1   public class HFunctionLinearConflicts extends HFunctionManhattan {
2
3       @Override
4       public double calculate(State state) {
5           SlidingPuzzle slidingPuzzle = (SlidingPuzzle) state;
6           return super.calculate(state) + linearConflicts(slidingPuzzle);
7       }
8
9       protected int linearConflicts(SlidingPuzzle slidingPuzzle) {
10          int n = SlidingPuzzle.n;
11          byte[] table = slidingPuzzle.board;
12
13          int h = 0;
14
15          int[] group = new int[n];
16          int[] conflicts = new int[n];
17
18          // rows
19          for (int i = 0; i < n; i++) {
20              for (int j = 0; j < n; j++)
21                  group[j] = table[i * n + j];
22
23              for (int j = 0; j < n - 1; j++) {
24                  if ((group[j] / n != i) && (group[j] > 0)) // is this row the goal row for group[
                        j]
25                      conflicts[j] = 0;
26                  else {
27                      for (int k = j + 1; k < n; k++) {
28                          if ((group[k] / n == i) && (group[k] > 0) && (group[j] > group[k]))
29                              conflicts[j]++;
30                      }
31                  }
32              }
```

```
33
34              // while there remain some positive conflicts[j]
35              while (true) {
36                  int max = Integer.MIN_VALUE;
37                  int jMax = -1;
38                  for (int j = 0; j < n - 1; j++)
39                      if (conflicts[j] > max) {
40                          max = conflicts[j];
41                          jMax = j;
42                      }
43                  if (max <= 0)
44                      break;
45                  conflicts[jMax] = 0;
46                  for (int k = jMax + 1; k < n; k++)
47                      if ((group[k] / n == i) && (group[jMax] > group[k])) {
48                          h += 2.0;
49                          conflicts[k]--;
50                      }
51              }
52          }
53
54          // columns
55          for (int i = 0; i < n; i++) {
56              for (int j = 0; j < n; j++)
57                  group[j] = table[j * n + i];
58
59              for (int j = 0; j < n - 1; j++) {
60                  if ((group[j] % n != i) && (group[j] > 0)) // is this column the goal row for
                        group[j]
61                      conflicts[j] = 0;
62                  else {
63                      for (int k = j + 1; k < n; k++) {
64                          if ((group[k] % n == i) && (group[k] > 0) && (group[j] > group[k]))
65                              conflicts[j]++;
66                      }
67                  }
68              }
69
70              // while there remain some positive conflicts[j]
71              while (true) {
72                  int max = Integer.MIN_VALUE;
73                  int jMax = -1;
74                  for (int j = 0; j < n - 1; j++)
75                      if (conflicts[j] > max) {
76                          max = conflicts[j];
77                          jMax = j;
78                      }
79
80                  if (max <= 0)
81                      break;
82                  conflicts[jMax] = 0;
83                  for (int k = jMax + 1; k < n; k++)
84                      if ((group[k] % n == i) && (group[jMax] > group[k])) {
85                          h += 2.0;
86                          conflicts[k]--;
87                      }
88              }
89          }
90
91          return h;
92      }
```

```
93 }
```

Comparison of heuristics — an experiment

In *SaC*, one can quite conveniently carry out a comparison of heuristics for the same fixed problem, iterating over them in a loop. Here is a sample code.

```
1 StateFunction[] heuristics = {new HFunctionMisplacedTiles(), new HFunctionManhattan(), new
      HFunctionLinearConflicts()};
2 for (StateFunction h : heuristics) {
3     SlidingPuzzle.setHFunction(h);
4     GraphSearchAlgorithm algorithm = new AStar(slidingPuzzle); // slidingPuzzle object defined
          earlier
5     algorithm.execute();
6     SlidingPuzzle solution = (SlidingPuzzle) algorithm.getSolutions().get(0);
7
8     // printing information to the console
9 }
```

Beneath, we show two outputs of such comparisons. The first is for the original example with $n = 3$ — puzzle $(0,1,5,8,2,7,4,3,6)$, and the second is for the $n = 4$ case — puzzle $(0,1,7,10,6,5,3,2,12,4,14,11,9,13,8,15)$.

```
SLIDING PUZZLE TO SOLVE:
----------------
|  0 |  1 |  5 |
----------------
|  8 |  2 |  7 |
----------------
|  4 |  3 |  6 |
----------------
***
HEURISTICS: sac.examples.slidingpuzzle.HFunctionMisplacedTiles
PATH LENGTH: 17
MOVES ALONG PATH: [R, D, L, D, R, R, U, L, L, D, R, R, U, U, L, L]
CLOSED STATES: 440
OPEN STATES: 268
DURATION TIME: 30 ms
***
HEURISTICS: sac.examples.slidingpuzzle.HFunctionManhattan
PATH LENGTH: 17
MOVES ALONG PATH: [R, D, L, D, R, R, U, L, L, D, R, R, U, U, L, L]
CLOSED STATES: 38
OPEN STATES: 26
DURATION TIME: 0 ms
***
HEURISTICS: sac.examples.slidingpuzzle.HFunctionLinearConflicts
PATH LENGTH: 17
MOVES ALONG PATH: [R, D, L, D, R, R, U, L, L, D, R, R, U, U, L, L]
CLOSED STATES: 26
OPEN STATES: 17
DURATION TIME: 0 ms

SLIDING PUZZLE TO SOLVE:
---------------------
|  0 |  1 |  7 | 10 |
```

```
---------------------
|  6 |  5 |  3 |  2 |
---------------------
| 12 |  4 | 14 | 11 |
---------------------
|  9 | 13 |  8 | 15 |
---------------------
***
HEURISTICS: sac.examples.slidingpuzzle.HFunctionMisplacedTiles
PATH LENGTH: 31
MOVES ALONG PATH: [R, R, R, D, L, D, D, L, L, U, R, R, U, U, L, D, L, D, R, D, R, U, U, U, R, D, L, L, U, L]
CLOSED STATES: 719791
OPEN STATES: 682571
DURATION TIME: 11258 ms
***
HEURISTICS: sac.examples.slidingpuzzle.HFunctionManhattan
PATH LENGTH: 31
MOVES ALONG PATH: [R, R, R, D, L, D, D, L, L, U, R, R, U, U, L, D, L, D, R, D, R, U, U, U, R, D, L, L, U, L]
CLOSED STATES: 2129
OPEN STATES: 2100
DURATION TIME: 28 ms
***
HEURISTICS: sac.examples.slidingpuzzle.HFunctionLinearConflicts
PATH LENGTH: 31
MOVES ALONG PATH: [R, R, R, D, L, D, D, L, L, U, R, R, U, U, L, D, L, D, R, D, R, U, U, U, R, D, L, L, U, L]
CLOSED STATES: 919
OPEN STATES: 942
DURATION TIME: 38 ms
```

Obviously, regardless of the heuristics applied we obtain solution paths of the same length. But, one can clearly note that better heuristics, i.e. more tight lower bounds on the true cost, lead to smaller graphs being searched — fewer closed and open states. The graphs from both examples are depicted in figures 3.3 (tree-like layout) and 3.4 (star-like layout).

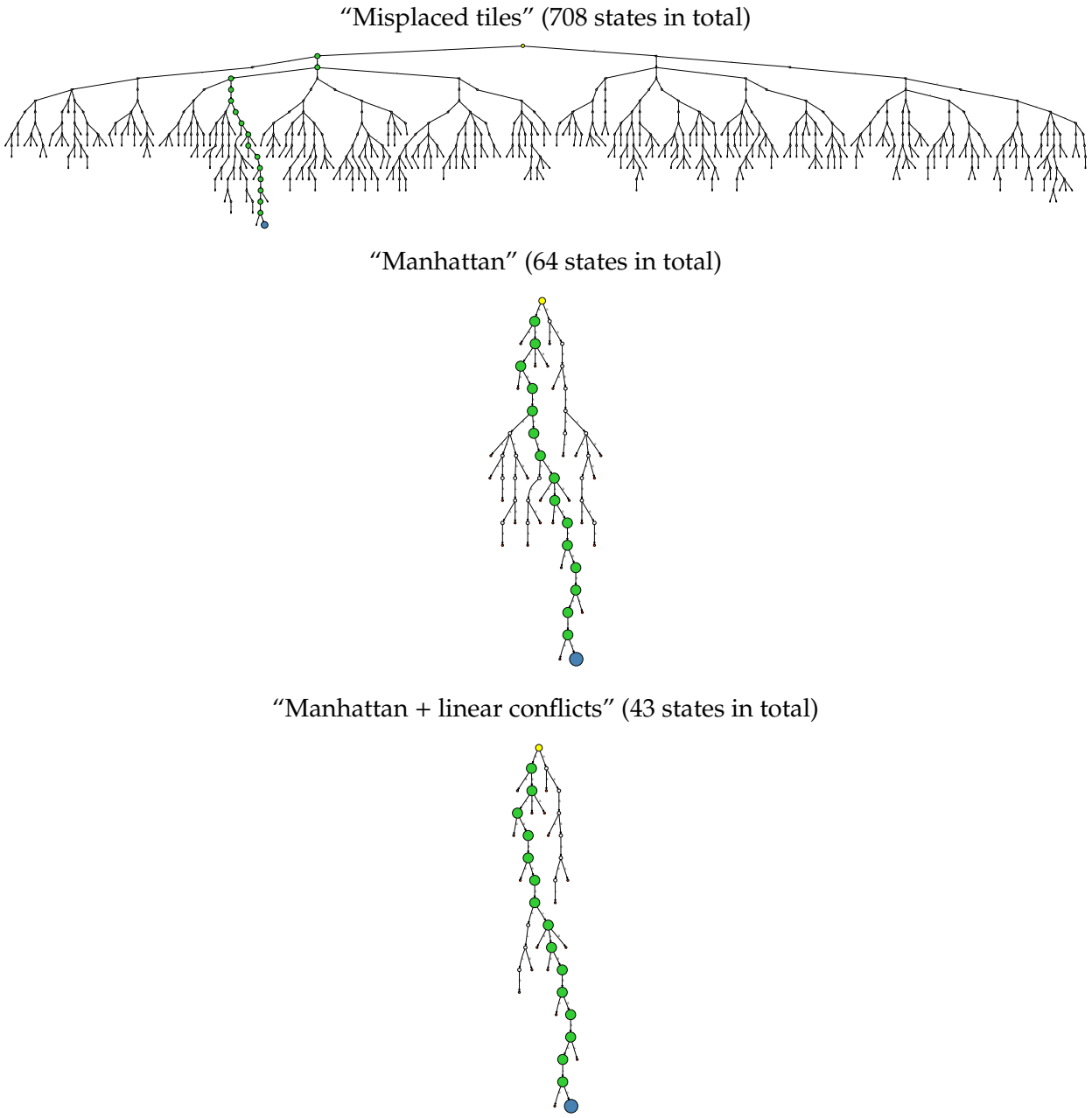
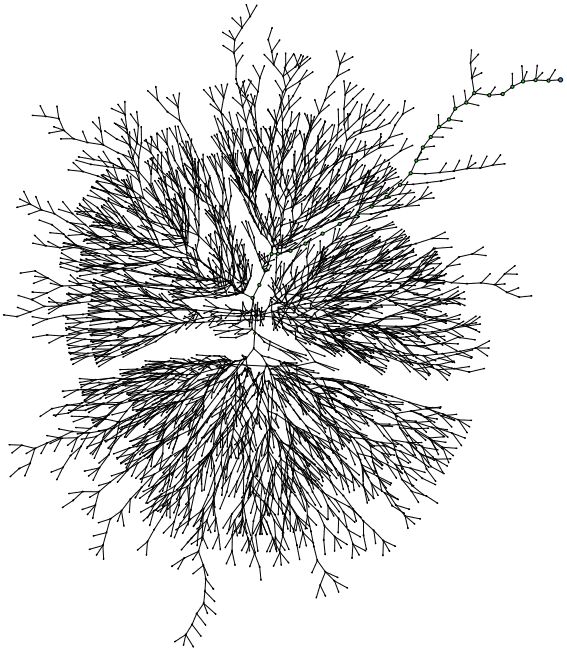


Figure 3.3: Graphs searched by *SaC* using different heuristics for the same initial eight-puzzle: $(0, 1, 5, 8, 2, 7, 4, 3, 6)$.

"Manhattan" (4 229 states in total)



"Manhattan + linear conflicts" (1 861 states in total)

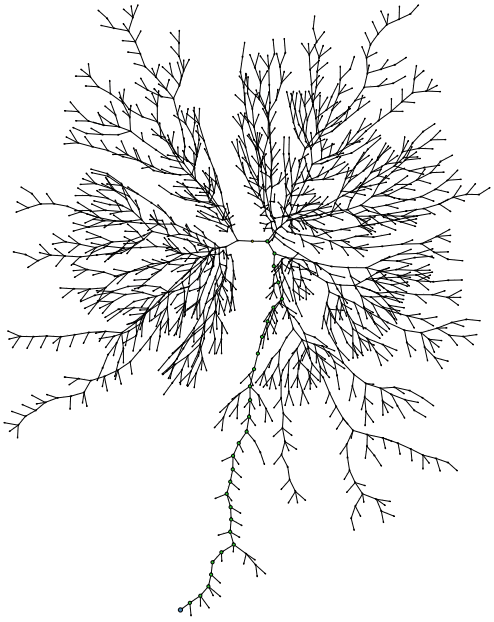


Figure 3.4: Graphs searched by *SaC* using different heuristics for the same initial fifteen-puzzle: $(0, 1, 7, 10, 6, 5, 3, 2, 12, 4, 14, 11, 9, 13, 8, 15)$. "Misplaced tiles" heuristics was omitted due to too large graph. Visualizations were rendered by the `neato` layouter in *Graphviz*.

Some harder fifteen-puzzles

In this section we show some results obtained using *SaC* for harder sliding puzzle examples. These examples are taken as a short excerpt from the work (Hansson et al., 1985). We put away the "Misplaced tiles" heuristics from considerations as being practically useless. Experiments are carried out using two algorithms A^* and IDA^*.

In the table 3.3 presented are five selected instances of a fifteen-puzzle. The numbering shown in the left-most column corresponds to the numbering in (Hansson et al., 1985). The instances are ordered top down from the easiest one (i.e. requiring the least of time to be solved) towards the hardest one. Before executing the search we explicitly imposed the RAM memory limit on the JVM to be 2 GB. This was meant to demonstrate that only two first of the examples allow us to solve them by the A^* algorithm being more memory consuming than IDA^*, which does not keep record of closed set. We should mention that we have used default *SaC* configuration settings which make the *Open* set to be implemented by `sac.graph.OpenSetAsPriorityQueueFastContainsFastReplace` and the closed set by `sac.graph.ClosedSetAsHashMap` — both fastest but the most memory consuming structures. Obviously the duration time of A^* for the examples it managed to solve was shorter than in the case of IDA^*.

no.	initial state	path length	IDA^* closed	IDA^* time [s]	A^* closed and open	A^* time [s]
85	$4,7,13,10,1,2,9,6,12,8,14,5,3,0,11,15$	44	$1.5 \cdot 10^7$	46.7	$1.7 \cdot 10^5$, $1.6 \cdot 10^5$	2.8
5	$4,7,14,13,10,3,9,12,11,5,6,15,1,2,8,0$	56	$2.6 \cdot 10^7$	60.5	$1.6 \cdot 10^6$, $1.4 \cdot 10^6$	34.4
2	$13,5,4,10,9,12,8,14,2,3,7,1,0,15,11,6$	55	$3.8 \cdot 10^7$	85.6	$2.6 \cdot 10^6$, $2.1 \cdot 10^6$	77.1
54	$12,11,0,8,10,2,13,15,5,4,7,3,6,9,14,1$	56	$1.9 \cdot 10^8$	442.5	out of RAM (2 GB) at: $2.7 \cdot 10^6$, $2.3 \cdot 10^6$	—
1	$14,13,15,7,11,12,9,5,6,0,2,1,4,8,10,3$	57	$2.5 \cdot 10^8$	634.8	out of RAM (2 GB) at: $2.3 \cdot 10^6$, $2.0 \cdot 10^6$	—

Table 3.3: Performance of A^* and IDA^* algorithms for five selected instances of fifteen-puzzle taken from (Hansson et al., 1985).

Sliding puzzle console solver

Along with the distribution of the *SaC* library comes a console solver dedicated for the sliding puzzle. The solver can be accessed by the included `run_slidingpuzzle.bat` file (or directly by the `sac.examples.slidingpuzzle.ConsoleSolver` class). A default execution triggered via

```
java -Xmx2048M -cp "sac-1.0.1.jar;jfreechart-1.0.14.jar;jcommon-1.0.17.jar" sac.examples.slidingpuzzle.ConsoleSolver
```

(the line from the .bat file) produces the following output to the screen with help information and a default eight puzzle solved:

```
SLIDING PUZZLE SOLVER
-----------------------------------------------------------------------------------------------------------------
PARAMETERS:
```

```
-sp - input path to text file (one line, comma-separated) with sliding puzzle to be solved ('0' assumed as an empty tile)
-a - full class name of graph search algorithm to be used (deafault: sac.graph.AStar)
-h - full class name of heuristic function be used (deafault: sac.examples.slidingpuzzle.HFunctionLinearConflicts)
-c - input path to .properties file with configuration settings for search process
-g - output path to .dot file in Graphviz format, representing graph that was searched
-gWithContent - true/false flag stating if points in Graphviz graph should be drawn with a content or no
-------------------------------------------------------------------------------------------------------------------------
DEFAULT SLIDING PUZZLE: '0,3,2,4,7,8,1,5,6'.
SLIDING PUZZLE TO SOLVE:
---------------
|  0 |  3 |  2 |
---------------
|  4 |  7 |  8 |
---------------
|  1 |  5 |  6 |
---------------
ALGORITHM: sac.graph.AStar.
HEURISTICS: sac.examples.slidingpuzzle.HFunctionLinearConflicts.
SOLVING...
DURATION TIME: 10 ms.
CLOSED STATES: 45.
OPEN STATES: 33.
SOLUTION:
---------------
|  0 |  1 |  2 |
---------------
|  3 |  4 |  5 |
---------------
|  6 |  7 |  8 |
---------------
PATH LENGTH (INCLUDING TERMINAL STATES): 17.
PATH AS SEQUENCE OF MOVES: [D, R, D, R, U, L, L, D, R, U, U, L, D, R, U, L].
ALL DONE.
```

3.3.2 Traveling Salesman Problem

The Traveling Salesman Problem (TSP) is one of the most recognizable problems within computer science. The problem can be formulated as follows: *Given is a set of places with connections (roads) between them. A salesman departs from one of the places (distinguished as the starting point), must visit all the places and return to the place of origin. The salesman is allowed to visit each place only once. The goal is to find the shortest route for the a saleseman.* In graph terms, the TSP can be reformulated as the problem of finding a Hamiltonian cycle with the smallest cost for the given graph.

In general, the TSP is NP-hard, although some of its large instances have been successfully solved (e.g. a TSP tour covering all 24 978 towns in Sweden[8]). One may note that for n places and a complete graph of connections, a naive approach checking exhaustively all possible routes must perform $(n-1)!$ iterations, which is exponential with respect to n. It should also be noted that a greedy approach, constisting in visting the nearest place at every step, leads in general to non-optimal solutions. In particular, for TSPs with places distributed randomly and uniformly on a plane (Euclidean TSP) the greedy approach returns on average routes longer by approximately 25% than an optimal route (Johnson and McGeoch, 1997). On the other hand, it is also possible to construct such 'malicious' examples for which the greedy approach leads to the worst routes (Gutin, Yeo and Zverovich, 2002).

The exemplary implementation of a TSP solver in *SaC*, described in the following section, is based on a well known and simple heuristics using the concept of a *minimum spanning tree* (MST). For a survey on more advanced and recent methods related to the TSP the reader is addressed e.g. to the following works: (Rego, Gamboa, Glover and Osterman, 2011; Kaplan, Shafrir, Lewenstein

[8]The project was carried out by a research team directed by David Applegate from AT&T Labs, `http://www.math.uwaterloo.ca/tsp/sweden/`.

and Sviridenko, 2003).

Implementation of TSP state

Our solver is restricted to TSPs related to symmetric graphs with complete sets of connections. Before presenting the actual state implementation, we first show some code excerpts of auxiliary base classes, namely: `sac.examples.tsp.Place`, `sac.examples.tsp.Connection` and `sac.examples.tsp.Map`. For full versions the reader is addressed to source codes of the library.

A place is represented by its x, y coordinates and an id number (for convenience). It also contains a set of references to the connections outgoing to other places. In the excerpt below we omit getters and setters (marked by `...`).

```
1  public class Place implements Comparable<Place> {
2
3      private int id;
4      private double x;
5      private double y;
6      private SortedSet<Connection> connections = null;
7
8      public Place(int id, double x, double y) {
9          this.id = id;
10         this.x = x;
11         this.y = y;
12         connections = new TreeSet<Connection>();
13     }
14
15     @Override
16     public int compareTo(Place otherPlace) {
17         return id - otherPlace.id;
18     }
19
20     @Override
21     public boolean equals(Object otherPlace) {
22         Place otherPlace2 = (Place) otherPlace;
23         return (compareTo(otherPlace2) == 0);
24     }
25
26     ...
27 }
```

A connection — an edge in our graph — is defined by a pair of places and is equipped with its cost precalculated prior to the instantiation. We keep the two references to places always sorted according to their ids. Again, in the excerpt we omit getters and setters.

```
1  public class Connection implements Comparable<Connection> {
2
3      private Place place1 = null;
4      private Place place2 = null;
5      private double cost;
6
7      public Connection(Place place1, Place place2, double cost) {
8          if (place1.compareTo(place2) < 0) {
9              this.place1 = place1;
10             this.place2 = place2;
```

```
        } else {
            this.place1 = place2;
            this.place2 = place1;
        }
        this.cost = cost;
    }

    @Override
    public String toString() {
        return "(" + place1.getId() + "," + place2.getId() + ")";
    }

    @Override
    public int compareTo(Connection otherConnection) {
        int place1IdDifference = place1.getId() - otherConnection.getPlace1().getId();
        if (place1IdDifference == 0)
            return place2.getId() - otherConnection.getPlace2().getId();
        return place1IdDifference;
    }

    @Override
    public boolean equals(Object otherConnection) {
        Connection otherConnection2 = (Connection) otherConnection;
        return (compareTo(otherConnection2) == 0);
    }

    ...
}
```

A map object represents the world (the graph) for our salesman. It stores references to places and connections in data structures convenient for the future search. One of the class constructors creates a set of n places, distributed randomly within a unit square, together with all their connections. The place with id 1 is treated as the starting point. In the code excerpt below we omit: getters, setters, a constructor and auxiliary methods for populating the map with a specific TSP from a text file, and the code related to drawing operations (visualization purposes).

```
public class Map {

    private SortedMap<Integer, Place> places = null;
    private Place startPlace = null;
    private SortedSet<Connection> connections = null;

    private static final double MIN_X = 0.0;
    private static final double MAX_X = 1.0;
    private static final double MIN_Y = 0.0;
    private static final double MAX_Y = 1.0;

    public Map(int n) {
        calculateDrawingConstants();

        places = new TreeMap<Integer, Place>();
        connections = new TreeSet<Connection>();

        for (int i = 1; i <= n; i++) {
            Place place = new Place(i, MIN_X + Math.random() * (MAX_X - MIN_X), MIN_Y + Math.
                random() * (MAX_Y - MIN_Y));
            places.put(i, place);
```

```
21          }
22
23          calculateCosts();
24
25          startPlace = places.get(1); // first place as start place
26      }
27
28      private void calculateCosts() {
29          int n = places.size();
30          for (int i = 1; i <= n; i++)
31              for (int j = i + 1; j <= n; j++) {
32                  Place place1 = places.get(i);
33                  Place place2 = places.get(j);
34                  double cost = Math.sqrt((place1.getX() - place2.getX()) * (place1.getX() - place2
                        .getX()) + (place1.getY() - place2.getY())
35                          * (place1.getY() - place2.getY()));
36                  Connection connection = new Connection(place1, place2, cost);
37                  connections.add(connection);
38                  place1.getConnections().add(connection);
39                  place2.getConnections().add(connection);
40              }
41      }
42
43      @Override
44      public String toString() {
45          StringBuilder result = new StringBuilder("");
46          int i = 0;
47          for (java.util.Map.Entry<Integer, Place> entry : places.entrySet()) {
48              Integer id = entry.getKey();
49              Place place = entry.getValue();
50              result.append(id + ": (" + place.getX() + ", " + place.getY() + ")");
51              if ((++i) < places.size())
52                  result.append("\n");
53          }
54          return result.toString();
55      }
56
57      ...
58  }
```

Now, we can move on to our implementation of a TSP state itself. In the approach we decided for, states represent partial routes of the salesman. For example, for the case of $n = 5$ places, the initial state is the place with id 1, its direct descendants are states: $1 \to 2$, $1 \to 3$, $1 \to 4$, $1 \to 5$, and after creation they are suitably placed in the *Open* queue by a search algorithm. Suppose that $1 \to 4$ turns out to be the state of lowest partial cost[9] and is polled from the queue in the first order. Its descandants are states: $1 \to 4 \to 2$, $1 \to 4 \to 3$, $1 \to 4 \to 5$. And the scheme above continues until the search algorithm polls from the queue a terminal state i.e. a route which uses all the places and comes back to the place of origin. We remark that the first such route polled from the queue is guaranteed to be the optimal route[10].

In the following code listing, we present the most important excerpts of our TSP state class

[9] In the case of Dijkstra algorithm, the partial cost is just the g summand i.e. just the exact known cost of the route already travelled through. In the case of A^*, the cost is defined by the $g + h$ sum, i.e. the known cost as before plus an optimistic estimate on the remaining cost.

[10] Provided that cost functions g and h are correctly defined by the programmer.

(`sac.examples.tsp.TravelingSalesmanProblem`). We leave some of the comments present in the code, shown for clarification of some variables or operations.

```
1  public class TravelingSalesmanProblem extends GraphStateImpl {
2
3      /**
4       * Reference to the map, common for all states.
5       */
6      public static Map map = null;
7
8      private List<Connection> route = null;
9      private List<Integer> routeAsIds = null;
10     private Place currentPlace = null;
11     private SortedSet<Place> remaining = null;
12
13     /**
14      * Minimum spanning tree built on remaining places.
15      */
16     private MinimumSpanningTree mst = null;
17
18     /**
19      * Best (cheapest) connection from the current place to some place in the minimum spanning
            tree of remaining places.
20      */
21     private Connection bestConnectionToMST = null;
22
23     public TravelingSalesmanProblem(Map map) {
24         TravelingSalesmanProblem.map = map;
25         route = new ArrayList<Connection>();
26         currentPlace = map.getStartPlace();
27         routeAsIds = new ArrayList<Integer>();
28         routeAsIds.add(currentPlace.getId());
29         remaining = new TreeSet<Place>();
30         remaining.addAll(map.getPlaces().values());
31     }
32
33     public TravelingSalesmanProblem(TravelingSalesmanProblem parent) {
34         route = new ArrayList<Connection>();
35         for (Connection connection : parent.route)
36             route.add(connection);
37         routeAsIds = new ArrayList<Integer>();
38         for (Integer id : parent.routeAsIds)
39             routeAsIds.add(id);
40         currentPlace = parent.currentPlace;
41         remaining = new TreeSet<Place>();
42         remaining.addAll(parent.remaining);
43     }
44
45     @Override
46     public List<GraphState> generateChildren() {
47         List<GraphState> children = new LinkedList<GraphState>();
48         for (Connection connection : currentPlace.getConnections()) {
49             Place otherPlace = (connection.getPlace1().equals(currentPlace)) ? connection.
                   getPlace2() : connection.getPlace1();
50             if (remaining.contains(otherPlace)) {
51                 if ((otherPlace.equals(map.getStartPlace())) && (remaining.size() > 1))
52                     continue;
53                 TravelingSalesmanProblem child = new TravelingSalesmanProblem(this);
54                 child.route.add(connection);
55                 child.routeAsIds.add(otherPlace.getId());
```

```
56                  child.currentPlace = otherPlace;
57                  child.remaining.remove(otherPlace);
58                  child.setMoveName(String.valueOf(otherPlace.getId()));
59                  children.add(child);
60              }
61          }
62          return children;
63      }
64
65      @Override
66      public boolean isSolution() {
67          return remaining.isEmpty();
68      }
69
70      @Override
71      public String toString() {
72          return routeAsIds.toString();
73      }
74
75      @Override
76      public int hashCode() {
77          return routeAsIds.hashCode();
78      }
79
80      static {
81          setHFunction(new StateFunction() {
82              @Override
83              public double calculate(State state) {
84                  TravelingSalesmanProblem tsp = (TravelingSalesmanProblem) state;
85
86                  // checking if parent's MST can be used (specifically: if the place to
87                  // which parent's bestConnectionToMST leads does not fork further)
88                  if (tsp.parent != null) {
89                      TravelingSalesmanProblem tspParent = (TravelingSalesmanProblem) tsp.parent;
90                      Connection singleConnection = tspParent.mst.isPlaceWithSingleConnection(tsp.
                            currentPlace);
91                      if (singleConnection != null) {
92                          tsp.mst = new MinimumSpanningTree(tspParent.mst);
93                          tsp.mst.getConnections().remove(singleConnection);
94                          tsp.mst.setCost(tsp.mst.getCost() - singleConnection.getCost());
95                          tsp.bestConnectionToMST = singleConnection;
96                          return tsp.mst.getCost() + tsp.bestConnectionToMST.getCost();
97                      }
98                  }
99                  // construction of new MST required
100                 tsp.mst = new MinimumSpanningTree(tsp.remaining);
101                 tsp.bestConnectionToMST = null;
102                 double bestCost = Double.POSITIVE_INFINITY;
103                 for (Connection connection : tsp.currentPlace.getConnections()) {
104                     Place otherPlace = (connection.getPlace1() == tsp.currentPlace) ? connection.
                            getPlace2() : connection.getPlace1();
105                     if ((tsp.remaining.contains(otherPlace)) && (connection.getCost() < bestCost)
                            ) {
106                         bestCost = connection.getCost();
107                         tsp.bestConnectionToMST = connection;
108                     }
109                 }
110                 double cost = tsp.mst.getCost();
111                 if (tsp.bestConnectionToMST != null)
112                     cost += bestCost;
113                 return cost;
```

```
114             }
115         });
116
117         setGFunction(new StateFunction() {
118             @Override
119             public double calculate(State state) {
120                 TravelingSalesmanProblem tsp = (TravelingSalesmanProblem) state;
121                 return (tsp.parent == null) ? 0.0 : (tsp.getParent()).getG() + tsp.route.get(tsp.
                        route.size() - 1).getCost();
122             }
123         });
124     }
125
126     ...
127 }
```

As regards the fields in our `TravelingSalesmanProblem` class, it contains: a static reference to the map object, a collection of non-static fields describing the particular state, a minimum spanning tree object meant for the heuristics calculation purposes, and a separate reference to the best (cheapest) connection from the current place to some place within the MST.

On can notice that there is a certain redundancy in the description of the state, since we represent the route as a list of succesive connections (`List<Connection route`) and simultaneously also as a list of ids of visited places (`List<Integer> routeAsIds`). Moreover, we keep a sorted set of remaining places (`SortedSet<Place> remaining`), which is also redundant, since such a set could be always uniquely derived from the route and the set of all places (kept inside the map). Nevertheless, such redundacy in terms of memory consumption is later compensated by speed gains in operations like: descendants generations, construction of MST, or identification of a state via `toString()` or `hashCode()` methods.

As regards the generation of descandants — the `generateChildren()` method — it boils down to: copying the parent state, iterating over connections outgoing from the current place and appending a new place to the route provided that this place has not been already visited. After that, the reference to the current place is set to the new place.

As for the cost functions g and h, we implement both of them as anonymous functions and attach them statically to the class via `setGFunction(...)` and `setHFunction(...)` respectively. The g function is straightforward and boils down to adding up the cost of the last connection to the g cost of the parent. The h function is a bit more complex. First of all we should remark that the MST reference in a descendant state is `null` at the start, and the calculation of MST is postponed until the first call to the `getH()` method[11] is made. Then, we check wether the MST of the parrent can be fairly cheeply reused or if a new MST should be derived. In doing so, we simply check whether the last place in the route so far (current place in our descendant) has only one edge within the parent's MST. In other words we need to know if there is no fork from that place within the MST. If there is no such fork, we can reuse this MST and modify it only slightly i.e. the single egde is deleted from the new MST and it (this edge) becomes the best connection from the current state to the modified MST. Otherwise, if a fork exists, we rebuild the MST from the scratch. The implementation of the MST related class (`sac.examples.tsp.MinimumSpanningTree`) applies Kruskal's algorithm (Kruskal, 1956) and is included in Appendix 6.5.

[11] Hence, underneath it is also the first call to the `hFuntion.calculate()` method.

Examples of small and easy TSPs

The following code example generates and solves an easy Euclidean TSP with $n = 5$ places distributed randomly within a unit square.

```
Map map = new Map(5);
TravelingSalesmanProblem tsp = new TravelingSalesmanProblem(map);
System.out.println("TSP TO SOLVE:\n" + map);

GraphSearchAlgorithm algorithm = new AStar(tsp);
algorithm.execute();

System.out.println("CLOSED STATES: " + algorithm.getClosedStatesCount());
System.out.println("OPEN STATES: " + algorithm.getOpenSet().size());
System.out.println("DURATION TIME: " + algorithm.getDurationTime() + " ms");

if (algorithm.getSolutions().isEmpty()) {
    System.out.println("NO SOLUTIONS FOUND.");
    System.out.println("BEST STATE SO FAR: ");
    System.out.println(algorithm.getBestSoFar());
} else {
    TravelingSalesmanProblem solution = (TravelingSalesmanProblem) algorithm.getSolutions().get
        (0);
    System.out.println("SOLUTION: " + solution);
    System.out.println("PATH COST (LENGTH): " + solution.getG());
}
```

The program outputs the following result to the console and the graph searched is depicted in Fig. 3.5. In the figure, every graph state contains a visualization of the partial route travelled so far (marked in black) and the minimum spanning tree (marked in gray) related to the calculation of heuristics. It is possible to note that the algorithm reaches in fact two solutions, because in a non-directed graph a reversal of an optimal route leads to an equivalent optimal route as well.

```
TSP TO SOLVE:
1: (0.12251284631678094, 0.380209913097437)
2: (0.5639280517556917, 0.5560186588425798)
3: (0.6089976670947047, 0.17022778070498457)
4: (0.7890152412835078, 0.9022516218739999)
5: (0.2753596011637117, 0.5962434229024552)
CLOSED STATES: 12
OPEN STATES: 12
DURATION TIME: 4 ms
SOLUTION: [1, 3, 2, 4, 5, 1]
PATH COST (LENGTH): 2.1937848431352442
```

In Fig. 3.6 we present three TSP examples for $n = 5$, $n = 10$, and $n = 15$. The figure shows the console output of a TSP solving program and the resulting routes. Search graphs related to these examples are shown on subsequent pages, in figures 3.7–3.9.

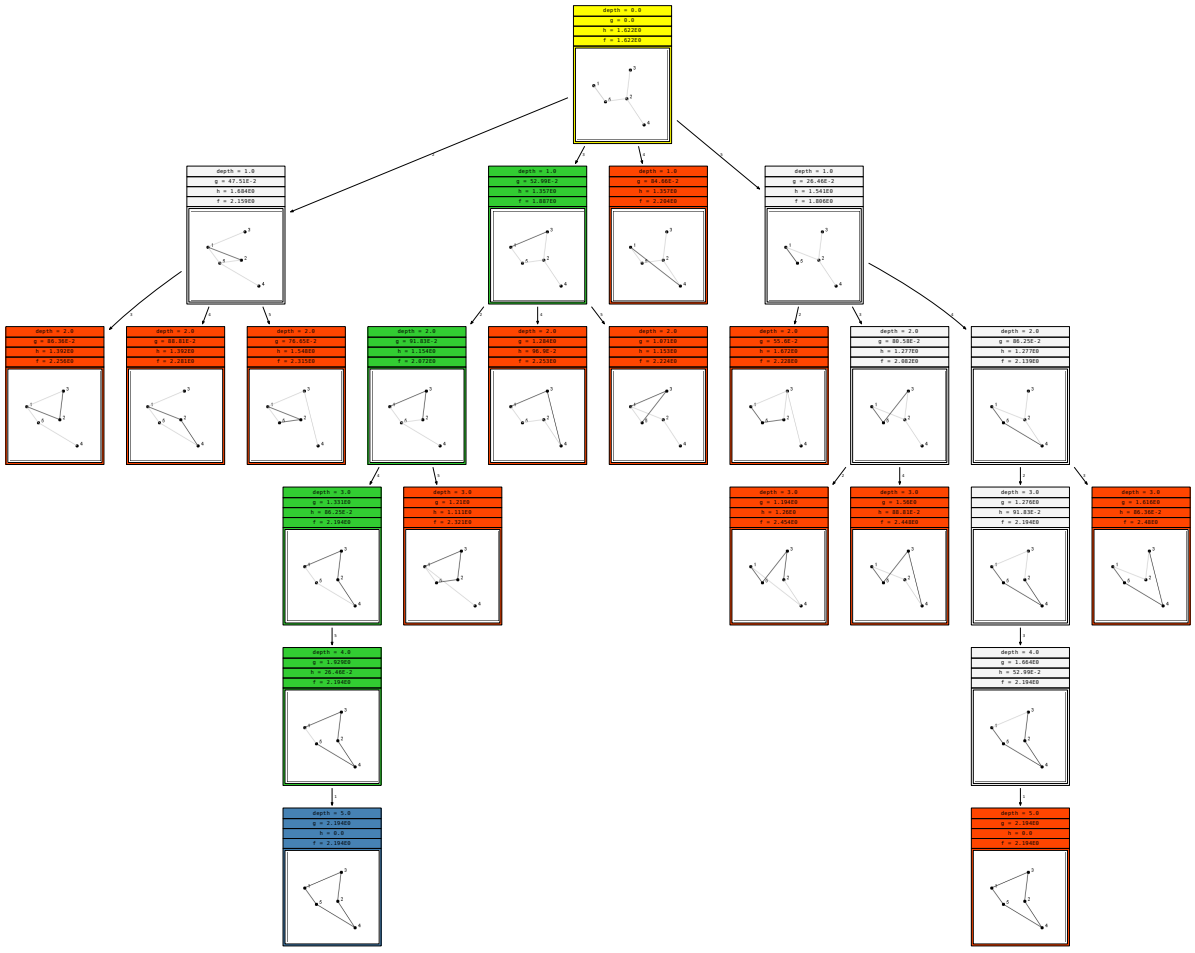
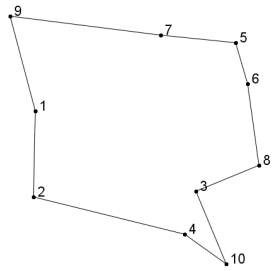


Figure 3.5: Graph searched by *SaC* using MST-based heuristics for a Euclidean Traveling Salesman Problem with $n = 5$ places distributed randomly within a unit square.

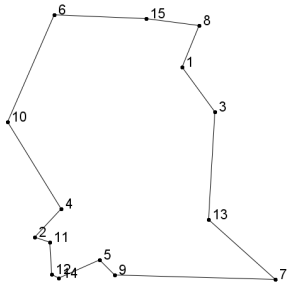
Example 1 ($n = 10$)

```
TSP TO SOLVE:
1: (0.12721669119887158, 0.37344395814939046)
2: (0.1216036308133 73, 0.6769233524617521)
3: (0.6967743393401367, 0.6558920453521531)
4: (0.6567692230822297, 0.8073145753237446)
5: (0.8377899934797072, 0.13271896685690432)
6: (0.8792651552573485, 0.276021081084061 76)
7: (0.5705158967610642, 0.104689835886705 22)
8: (0.9198623796845153, 0.563743371717250 1)
9: (0.036877836919299245, 0.036900909230807 69)
10: (0.803098627387394, 0.9124602486409994)
CLOSED STATES: 30
OPEN STATES: 107
DURATION TIME: 43 ms
SOLUTION: [1, 9, 7, 5, 6, 8, 3, 10, 4, 2, 1]
PATH COST (LENGTH): 3.1485171705107526
```



Example 2 ($n = 15$)

```
TSP TO SOLVE:
1: (0.6465487963847636, 0.21821665354693065)
2: (0.12410081933969885, 0.8205066551015285)
3: (0.7636609394148224, 0.3769891636455803)
4: (0.2196605421495409, 0.721007130156528 3)
5: (0.355011642392267 57, 0.9028959080059216)
6: (0.194091455028749 14, 0.0325702294277431 25)
7: (0.977839796679355, 0.9712745046770875)
8: (0.7073474430257418, 0.07126970719532189)
9: (0.4076402270626023, 0.9546082439383254)
10: (0.029324526390854255, 0.4126818021669526)
11: (0.1794318696639896, 0.8393016210197272)
12: (0.18437221955272642, 0.9521910807090714)
13: (0.740701851836031, 0.7595720414974898)
14: (0.209876821548100 83, 0.9667661669912148)
15: (0.520866086630205 8, 0.047291493246816696)
CLOSED STATES: 200
OPEN STATES: 1146
DURATION TIME: 200 ms
SOLUTION: [1, 8, 15, 6, 10, 4, 2, 11, 12, 14, 5, 9, 7, 13, 3, 1]
PATH COST (LENGTH): 3.4907876232208066
```



Example 3 ($n = 20$)

```
TSP TO SOLVE:
1: (0.08186119552601445, 0.49089277702354817)
2: (0.05155073744417993, 0.08268872589515641)
3: (0.2885260246313 4314, 0.009126346941663699)
4: (0.8064223640909227, 0.7637184456373667)
5: (0.7628990893773238, 0.6979541317575202)
6: (0.586002368788608, 0.6044459621574342)
7: (0.8093879023036267, 0.130396111377294 12)
8: (0.176434677162695 45, 0.004464542854964004)
9: (0.4294707658409983, 0.805746750096812)
10: (0.289736925590794 67, 0.2680530377656897)
11: (0.9846547483004421, 0.8696582199140886)
12: (0.022887988275901 66, 0.009918798528826045)
13: (0.9391576678180453, 0.2995144653344426)
14: (0.317492518781660 85, 0.098573855155170 92)
15: (0.394381242770310 5, 0.21271979616530157)
16: (0.986092942508290 5, 0.7153770321657219)
17: (0.874312163385920 9, 0.8979076761069746)
18: (0.321394896019261 97, 0.87778415943177)
19: (0.224308203366414, 0.017709126233366 29)
20: (0.911117207089934 7, 0.952104717842952 4)
CLOSED STATES: 148
OPEN STATES: 1751
DURATION TIME: 305 ms
SOLUTION: [1, 18, 9, 6, 5, 4, 17, 20, 11, 16, 13, 7, 15, 10, 14, 3, 19, 8, 12, 2, 1]
PATH COST (LENGTH): 3.7940773777347 47
```

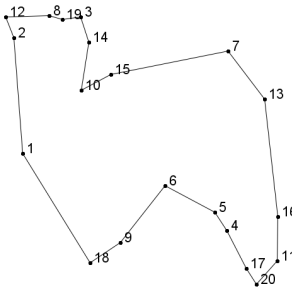


Figure 3.6: Several TSP examples solved by *SaC*.

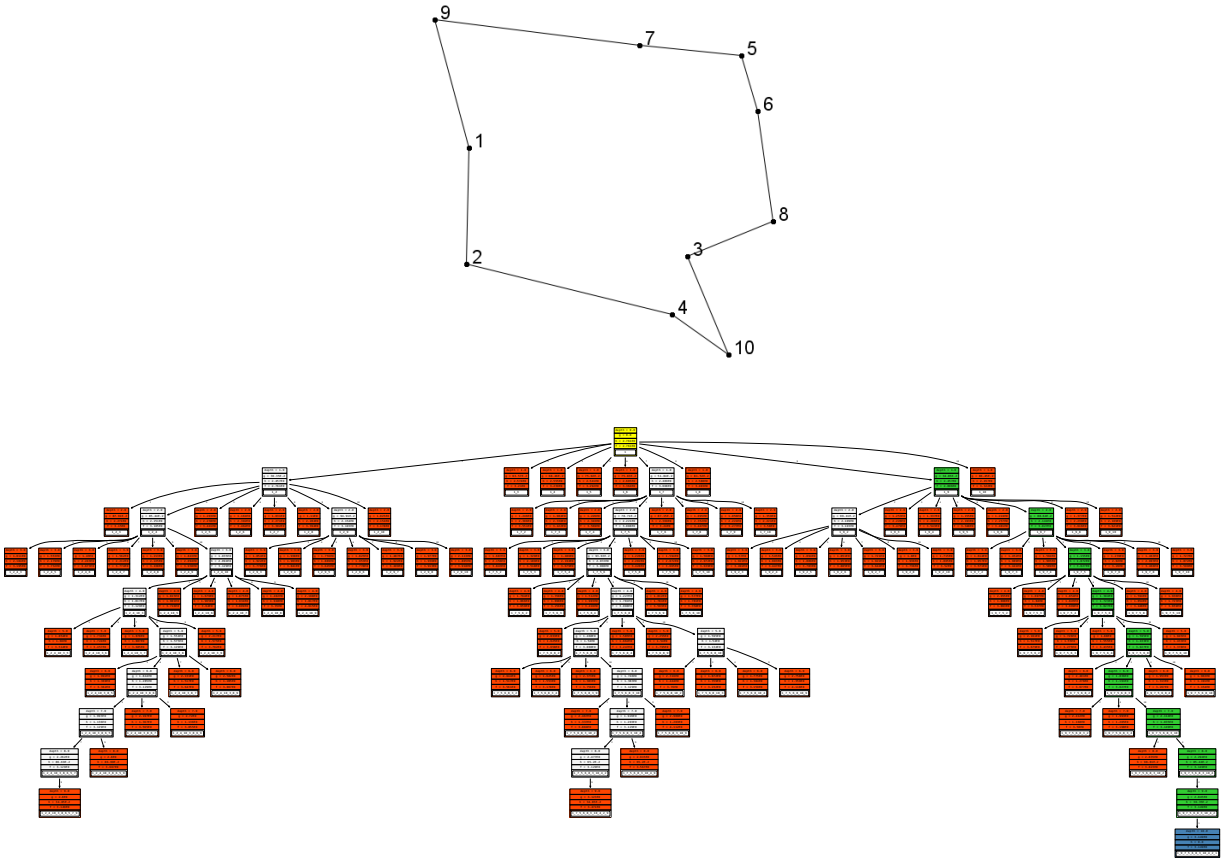


Figure 3.7: Solution and search graph for the TSP problem from example 1 in Fig. 3.6

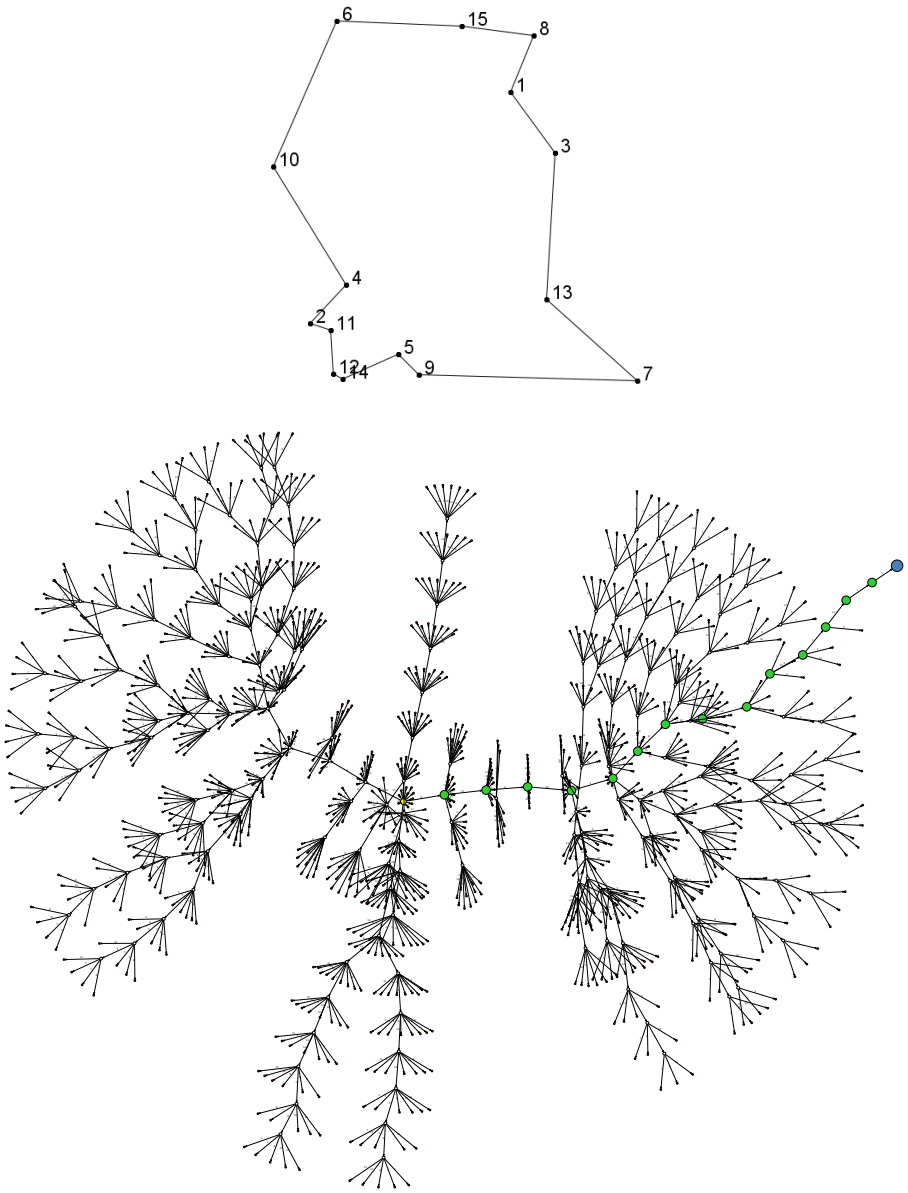


Figure 3.8: Solution and search graph for the TSP problem from example 2 in Fig. 3.6

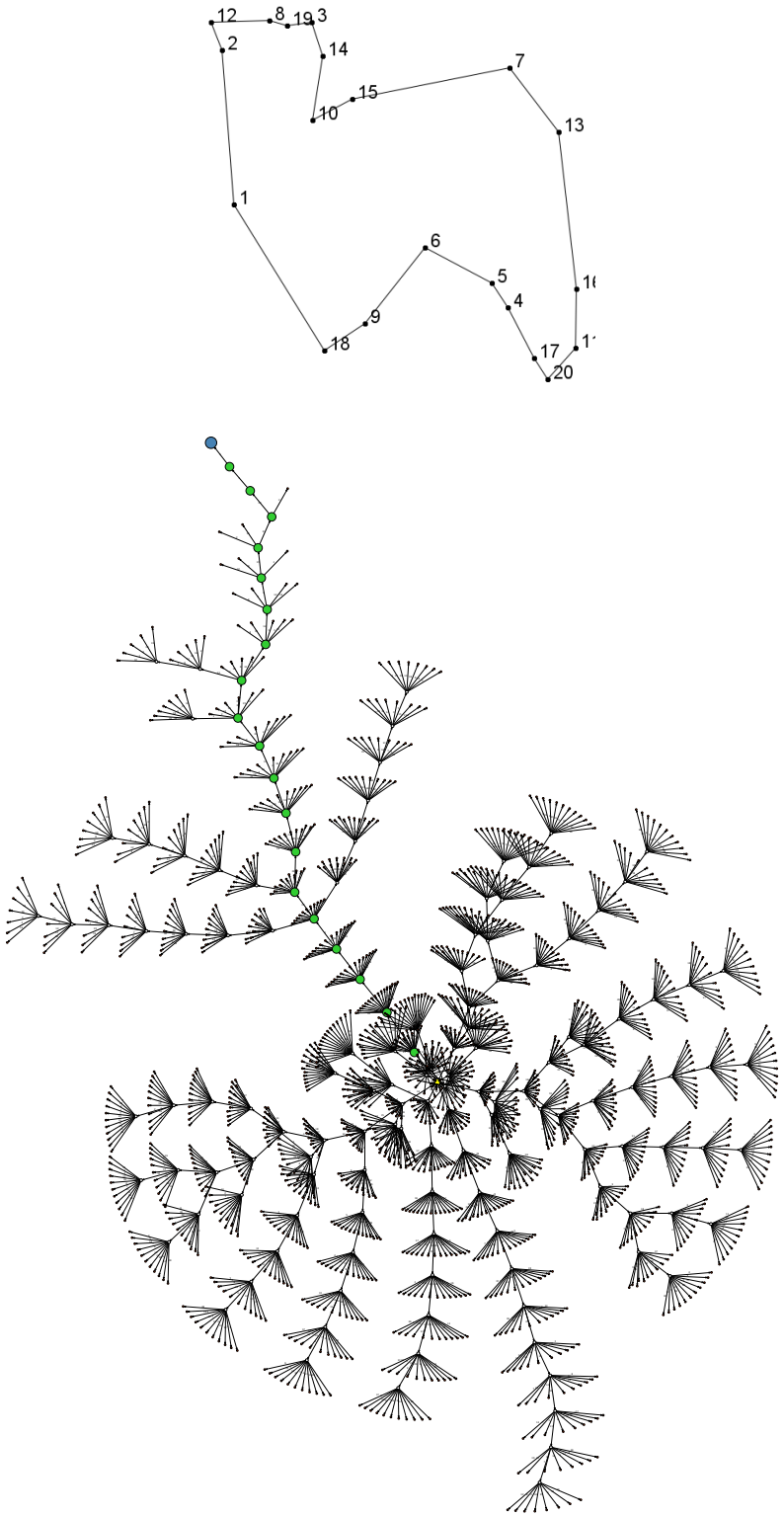


Figure 3.9: Solution and search graph for the TSP problem from example 3 in Fig. 3.6

Simple statistics for random Euclidean TSPs

Examples from the previous subsection were fairly simple. In fact, we selected them out because the A^* algorithm generated quite small search graphs for them — the number of states present at the stop moment (in both *Open* and *Closed* sets) was smaller than 2 000 in all cases. These graphs were convenient for us for the purpose of visualization.

In this subsection we would like to give the reader a feeling on the actual difficulty that particular initial values of n create for the search algorithm. For that purpose we arranged a batch experiment consisting of multiple random TSPs. 20 repetitions were done for each $n = 5, 10, 15, 20$. The code below represents this experiment. Additionally, it demonstrates the usage of `sac.stats.Stats` class, convenient for batch experiments of that type. Results produced by the presented program are gathered in Table. 3.4.

More information on how to register statistics and monitor search procedures within *SaC* can be found in Chapter 5.

```
1  Stats stats = new Stats();
2  for (int n = 5; n <= 20; n += 5) {
3      for (int r = 0; r < 20; r++) {
4          Map map = new Map(n);
5          TravelingSalesmanProblem tsp = new TravelingSalesmanProblem(map);
6
7          GraphSearchAlgorithm algorithm = new AStar(tsp);
8          algorithm.execute();
9
10         stats.addEntries(algorithm, n, r);
11     }
12 }
13
14 for (int n = 5; n <= 20; n += 5) {
15     System.out.println("***");
16     System.out.println("n = " + n);
17     System.out.println("MEAN CLOSED STATES: " + stats.mean(StatsCategory.
           GRAPH_SEARCH_CLOSED_STATES.toString(), n, null));
18     System.out.println("MEAN OPEN STATES: " + stats.mean(StatsCategory.GRAPH_SEARCH_OPEN_STATES.
           toString(), n, null));
19     System.out.println("MEAN DURATION TIME: " + stats.mean(StatsCategory.
           GRAPH_SEARCH_DURATION_TIME.toString(), n, null));
20     System.out.println("MEAN PATH G: " + stats.mean(StatsCategory.GRAPH_SEARCH_PATH_G.toString(),
            n, null));
21 }
```

Table 3.4: Simple statistics observed in 20 repetitions for random Euclidean TSPs (within a unit square).

n	repetitions	mean closed states	mean open states	mean duration time	mean path cost
5	20	18.7	16.4	3.1 ms	1.98
10	20	194.4	721.4	24.5 ms	2.79
15	20	3 893.7	24 756.0	472.6 ms	3.37
20	20	25 643.9	247 673.7	9 742.3 ms	3.91

TSP console solver

Along with the distribution of the *SaC* library comes a console solver dedicated for the Traveling Salesman Problem. The solver can be accessed by the included `run_tsp.bat` file (or directly by the `sac.examples.tsp.ConsoleSolver` class). A default execution triggered via

```
java -Xmx2048M -cp "sac-1.0.1.jar;jfreechart-1.0.14.jar;jcommon-1.0.17.jar" sac.examples.tsp.ConsoleSolver
```

(the line from the .bat file) generates a random TSP for $n = 10$ and produces the following output to the screen with help information available:

```
TRAVELING SALESMAN PROBLEM (TSP) SOLVER
-----------------------------------------------------------------------------------------------------------------------
PARAMETERS:
-tsp - input path to text file (places in successive lines written as: x, y) with a TSP to be solved
-a - full class name of graph search algorithm to be used (deafault: sac.graph.AStar)
-tspImage - output path to .gif file representing the TSP to be solved)
-tspSolutionImage - output path to .gif file representing the solution of TSP)
-c - input path to .properties file with configuration settings for search process
-g - output path to .dot file in Graphviz format representing graph that was searched)
-gWithContent - true/false flag stating if points in Graphviz graph should be drawn with a content or no
-----------------------------------------------------------------------------------------------------------------------
PREPARING A RANDOM TSP
TSP TO SOLVE:
1: (0.23837959212004678, 0.42856952786747304)
2: (0.04856303446645194, 0.14510993730937916)
3: (0.8481119838802103, 0.8476036286011716)
4: (0.5134840439526505, 0.380365517342815 76)
5: (0.8938906752066433, 0.45893894608984886)
6: (0.9912720715320988, 0.5512600244991797)
7: (0.3081039830474124, 0.49977790528024346)
8: (0.061756899494391115, 0.473908767794066)
9: (0.4568222308553671, 0.8739890958612715)
10: (0.9215007375728422, 0.42934196019090054)
ALGORITHM: sac.graph.AStar
SOLVING...
DURATION TIME: 110 ms.
CLOSED STATES: 222.
OPEN STATES: 717.
SOLUTION:
[1, 8, 2, 4, 5, 10, 6, 3, 9, 7, 1]
PATH COST (LENGTH): 2.8254791884954225
ALL DONE.
```

A specific (non-random) TSP problem can be given as input by providing a text file via `-tsp` parameter. In such a file the coordinates of places should be specified in successive lines with x and y values separated by a comma. Note that the console solver rescales original coordinates to fit the places in the unit square.

3.3.3 Sudoku

Sudoku is a number-placement puzzle. In the most common setup, the sudoku board is a 9×9 grid of cells with distinguished 9 subsquares inside, each being a 3×3 grid. Initially, the board is partially filled with numbers. The goal of the solver is to fill in the empty cells in such a manner that all rows, all columns, and all subsquares contain all the numbers from the set $\{1, 2, \ldots, 9\}$. It is commonly agreed that a well-posed initial sudoku should lead to a unique solution. Sudoku can be regarded as a special case of *latin square*[12] with an additional constraint imposed on the subsquares.

Fig. 3.10 shows an example of the sudoku puzzle.

[12]Latin square is a $n \times n$ grid filled with n symbols. Each symbol must occur exactly once in each row and each column.

	2		8	1		7	4	
7					3	1		
	9				2	8		5
		9		4			8	7
4			2		8			3
1	6			3		2		
3		2	7				6	
		5	6					8
	7	6		5	1		9	

→

5	2	**3**	8	1	**6**	7	4	**9**
7	**8**	**4**	**5**	**9**	3	1	**2**	**6**
6	9	**1**	**4**	**7**	2	8	**3**	5
2	**3**	9	**1**	4	**5**	**6**	8	7
4	**5**	**7**	2	**6**	8	**9**	**1**	3
1	6	**8**	**9**	3	**7**	2	**5**	**4**
3	**4**	2	7	**8**	**9**	**5**	6	**1**
9	**1**	5	6	**2**	**4**	**3**	**7**	8
8	7	6	**3**	5	1	**4**	9	**2**

Figure 3.10: Example of sudoku puzzle — initial board shown on the left-hand-side and its solution on the right-hand-side (numbers filled in to form a solution are marked in red).

Sudoku was popularized by a Japanese company Nikoli in late 1980s. Some sources (Shortz, 2005; Wikipedia, 2014) suggest that the puzzle was originally and anonymously proposed by Howard Garns, a freelance puzzle designer from Indiana, and published first in 1979 in Dell Magazines under the name *Number Place*.

A generalized sudoku can be defined as follows. Given is an initial $n^2 \times n^2$ grid of cells, containing n^2 subsquares (each of size $n \times n$). The grid is partially filled with numbers. The goal is to fill the missing cells in such a manner that all rows, all columns, and all subsquares contain all the numbers from the set $\{1, 2, \ldots, n^2\}$.

Implementation of sudoku state

Below we show a sudoku implementation using *SaC*. The code is almost in its full, omitted are import statements, and the `toGraphvizLabel()` method. The sudoku board is represented as a two-dimensional array of bytes. The value of zero represents and empty cell to be filled in. The subsquare side length n is denoted in the code by `n`, and the whole board side length n^2 is denoted by `N`. Each sudoku state 'knows' its number of empty cells (`emptyCells`). It also keeps some auxiliary fields with information about: the sum of remainining possibilities over all cells, and the cell with the minimum number of remaining possibilities (this cell is used later on to generate descendants). The remaining possibilities for an (i, j) cell are the ones that remain after excluding the numbers occuring in the i-th row, the j-th column, and the subsquare that (i, j) cell belongs to.

```
1  public class Sudoku extends GraphStateImpl {
2
3      protected static byte n;
4      protected static byte N; // n * n
5      protected byte[][] board = null;
6
7      protected int emptyCells; // number of empty cells
8      protected int sumRemainingPossibilities; // sum of remaining possibilities in all cells.
9      protected int minRemainingPossibilities; // minimum number of remaining possibilities in some
           cell
10     protected int minI = -1; // row position of the cell with mimum number of possibilities
11     protected int minJ = -1; // column position of the cell with mimum number of possibilities
```

```
12      protected SortedSet<Byte> possibilities = null; // remaining possibilities in the 'minimum
            cell'
13
14      public Sudoku(Sudoku parent) {
15          board = new byte[N][N];
16          emptyCells = parent.emptyCells;
17          for (int i = 0; i < N; i++)
18              for (int j = 0; j < N; j++) {
19                  board[i][j] = parent.board[i][j];
20              }
21          possibilities = new TreeSet<Byte>();
22      }
23
24      public Sudoku(int n) {
25          Sudoku.n = (byte) n;
26          N = (byte) (n * n);
27          board = new byte[N][N];
28          emptyCells = N * N;
29          possibilities = new TreeSet<Byte>();
30      }
31
32      public Sudoku(int n, String sudokuAsCommaSeparatedString) {
33          Sudoku.n = (byte) n;
34          N = (byte) (n * n);
35          board = new byte[N][N];
36
37          emptyCells = N * N;
38
39          StringTokenizer tokenizer = new StringTokenizer(sudokuAsCommaSeparatedString, ",");
40          int z = 0;
41          while (tokenizer.hasMoreElements()) {
42              byte number = Byte.valueOf((String) tokenizer.nextElement());
43              int i = z / N;
44              int j = z % N;
45              board[i][j] = number;
46              if (number > 0) {
47                  emptyCells--;
48              }
49              z++;
50          }
51          possibilities = new TreeSet<Byte>();
52      }
53
54      @Override
55      public List<GraphState> generateChildren() {
56          List<GraphState> children = new LinkedList<GraphState>();
57          double theH = getH(); // call made in order to do pre-calculations for heuristics and to
58                                // discover minI, minJ
59                                // (if not present so far)
60          if (minRemainingPossibilities == 0)
61              return children; // discrepancy or solution
62          if (theH > 0) {
63              for (byte possibility : possibilities) {
64                  Sudoku child = new Sudoku(this);
65                  child.board[minI][minJ] = possibility;
66                  if (child.isAdmissible(minI, minJ)) {
67                      child.emptyCells = emptyCells - 1;
68                      child.setMoveName("(" + (minI + 1) + "," + (minJ + 1) + "):=" + possibility);
69                      children.add(child);
70                  }
71              }
```

```
72          }
73          return children;
74      }
75
76      @Override
77      public boolean isSolution() {
78          return (emptyCells == 0);
79      }
80
81      @Override
82      public int hashCode() {
83          byte[] linearBoard = new byte[N * N];
84
85          for (int i = 0; i < N; i++)
86              System.arraycopy(board[i], 0, linearBoard, i * N, N);
87
88          return Arrays.hashCode(linearBoard);
89      }
90
91      @Override
92      public String toString() {
93          StringBuilder builder = new StringBuilder();
94          for (int i = 0; i < N; i++) {
95              for (int j = 0; j < N; j++) {
96                  builder.append(board[i][j]);
97                  if (j < N - 1)
98                      builder.append(",");
99              }
100             if (i < N - 1)
101                 builder.append("\n");
102         }
103         return builder.toString();
104     }
105
106     protected boolean isAdmissible(int i, int j) {
107         List<Byte> groupUnderCheck = new ArrayList<Byte>();
108
109         // square around (i, j)
110         int minI = (i / n) * n;
111         int minJ = (j / n) * n;
112         for (int ii = minI; ii < minI + n; ii++)
113             for (int jj = minJ; jj < minJ + n; jj++)
114                 if (board[ii][jj] > 0)
115                     groupUnderCheck.add(Byte.valueOf(board[ii][jj]));
116         if (!isGroupAdmissible(groupUnderCheck))
117                 return false;
118         groupUnderCheck.clear();
119
120         // i-th row
121         for (int jj = 0; jj < N; jj++)
122             if (board[i][jj] > 0)
123                 groupUnderCheck.add(Byte.valueOf(board[i][jj]));
124         if (!isGroupAdmissible(groupUnderCheck))
125             return false;
126         groupUnderCheck.clear();
127
128         // j-th column
129         for (int ii = 0; ii < N; ii++)
130             if (board[ii][j] > 0)
131                 groupUnderCheck.add(Byte.valueOf(board[ii][j]));
132         if (!isGroupAdmissible(groupUnderCheck))
```

```
133             return false;
134
135         return true;
136     }
137
138     protected boolean isGroupAdmissible(List<Byte> group) {
139         if (group.size() == 0)
140             return true; // empty group is implied by all zeros in it
141         boolean[] visited = new boolean[N];
142         for (int i = 0; i < N; i++)
143             visited[i] = false;
144         for (Byte element : group)
145             if (visited[element.byteValue() - 1])
146                 return false;
147             else
148                 visited[element.byteValue() - 1] = true;
149
150         return true;
151     }
152
153     protected void precalculateForHeuristics() {
154         minRemainingPossibilities = Integer.MAX_VALUE;
155         sumRemainingPossibilities = 0;
156         for (int i = 0; i < N; i++)
157             for (int j = 0; j < N; j++) {
158                 if (board[i][j] > 0)
159                     continue;
160                 SortedSet<Byte> remaining = remainingPossibilities(i, j);
161                 int remainingSize = remaining.size();
162                 sumRemainingPossibilities += remainingSize;
163                 if (remainingSize < minRemainingPossibilities) {
164                     minRemainingPossibilities = remainingSize;
165                     minI = i;
166                     minJ = j;
167                     possibilities = remaining;
168                     if (minRemainingPossibilities == 0)
169                         return;
170                 }
171             }
172     }
173
174     private SortedSet<Byte> remainingPossibilities(int i, int j) {
175         SortedSet<Byte> remaining = new TreeSet<Byte>();
176         for (int k = 1; k <= N; k++)
177             remaining.add((byte) k);
178
179         // removing from remaining numbers existing in i-th row and j-th column
180         for (int k = 0; k < N; k++) {
181             remaining.remove(board[i][k]);
182             remaining.remove(board[k][j]);
183         }
184
185         // removing number as a possibility from the square i, j belongs to
186         int iMin = (i / n) * n;
187         int iMax = iMin + n;
188         int jMin = (j / n) * n;
189         int jMax = jMin + n;
190         for (int k = iMin; k < iMax; k++)
191             for (int l = jMin; l < jMax; l++)
192                 remaining.remove(board[k][l]);
193
```

```
194         return remaining;
195     }
196
197     static {
198         setHFunction(new HFunctionSumRemainingPossibilities());
199     }
200 }
```

The implementation presented contains three constructors — a copying constructor, a constructor of an empty board (populated with zeros), and a constructor from a comma separated string with numbers. It also contains suitable routines like: `hashCode()`, `toString()`, `isSolution()`, `generateChildren()`.

As regards the generation of descendants, it is always carried out with respect to some cell containing the minimum number of remaining possibilities. If there is more than one such cell in the board, the most top-left one is used. Information about the 'minimum cell' (its position, the actual remaining possibilities, and their number) are precalculated in the private `precalculateForHeuristics()` method.

In our implementation a sudoku can have at most n^2 descendants if all $\{1, 2, \ldots, n^2\}$ remaining possibilities are valid. We remark that our process of descendants generation is in fact a guessing process. Although we do not introduce immediate conflicts (duplications of some number in a row, a column, or a subsquare) by excluding the numbers present in the row, the column, and the subsquare the chosen 'minimum cell' belongs to, it is possible that such conflicts may occur later. In other words we make the search procedure follow the paths for all guesses that seem currently valid. We implement the `isAdmissible(...)` method to check if a sudoku is legal or not after a new number has been inserted into it. If there exist a duplication this method returns a `false`. Each potential descendant is first checked using the `isAdmissible(...)` method before being added to the actual list of descendants. Therefore, it is possible that a sudoku state (reached within the search procedure) may have as few as zero descendants. This means that in fact we have reached a contradictory state and the search algorithm shall abandon the current search path and move to another one by polling the next state from the *Open* queue.

By default, our sudoku implementation is equipped with a heuristic function represented by the class `HFunctionSumRemainingPossibilities`. It is attached by the `setHFunction(...)` static call. This heuristics boils down to calculating the number of remaining possibilities in every cell of the board and returning the sum. In other words, we say that a sudoku is closer to the goal state the smaller that sum is. On the other hand, we shall show that it is also possible to successfully use a different and trivial heuristic function — `HFunctionEmptyCells` — yielding the number of empty cells in the board. In that case, we say a sudoku is closer to the goal state the fewer empty cells it contains. We should mention that since sudoku is a placement puzzle we do not have to care about minimizing the path (the number of moves / manipulations leading to the solution). In fact, from the very start we know what number of cells must be populated. From the perspective of search algorithms, this means we do not have to take the travalled cost g into account, and this type of puzzles is typically well tackled by *Best-first search* algorithms or *Depth-first-search*[13].

[13] A^* or Dijsktra's algorithm would in this case degenerate in a sense to *Breadth-first search* and would only slow down the performance.

Moreover, we do not have to worry about the admissibility of the heuristics (since the g summand is neglected).

Below, we include the codes of the two heuristics described above.

```
public class HFunctionSumRemainingPossibilities extends StateFunction {

    @Override
    public double calculate(State state) {
            Sudoku sudoku = (Sudoku) state;
            sudoku.precalculateForHeuristics();
            return ((sudoku.minRemainingPossibilities == 0) && (sudoku.emptyCells > 0)) ? Double.
                POSITIVE_INFINITY : sudoku.sumRemainingPossibilities;
    }
}
```

```
public class HFunctionEmptyCells extends StateFunction {

    @Override
    public double calculate(State state) {
            Sudoku sudoku = (Sudoku) state;
            sudoku.precalculateForHeuristics();
            return ((sudoku.minRemainingPossibilities == 0) && (sudoku.emptyCells > 0)) ? Double.
                POSITIVE_INFINITY : sudoku.emptyCells;
    }
}
```

Example of a 'not so easy' sudoku

We start with an example of a seemingly easy sudoku. It contains 54 givens and therefore merely 27 empty cells (typical sudokus have fever than 30 givens). In particular, the top three rows in the example are already filled in completely. Yet, our *SaC* solver (based on a *Best-first search* algorithm) follows several stray leads before finding the correct path leading to the solution. Our run took 70 ms. Beneath, we present the source code and the program output.

```
Sudoku sudoku = new Sudoku(3,
        "8,5,4,2,1,9,7,6,3," +
        "3,9,7,8,6,5,4,2,1," +
        "2,6,1,4,7,3,9,8,5," +
        "0,8,0,0,0,0,0,9,0," +
        "0,4,0,5,3,8,0,7,0," +
        "0,3,0,0,0,0,0,5,0," +
        "9,2,6,3,8,4,5,1,7," +
        "5,0,3,7,0,0,0,4,8," +
        "4,7,8,0,0,1,0,0,0");
System.out.println("SUDOKU TO SOLVE:\n" + sudoku);

GraphSearchAlgorithm algorithm = new BestFirstSearch(sudoku);
algorithm.execute();

System.out.println("SOLUTION:\n" + algorithm.getSolutions().get(0));
System.out.println("---");
System.out.println("CLOSED STATES: " + algorithm.getClosedStatesCount() + ".");
```

```
19 System.out.println("OPEN STATES: " + algorithm.getOpenSet().size() + ".");
20 System.out.println("DURATION TIME " +  algorithm.getDurationTime() + " ms.");
```

```
SUDOKU TO SOLVE:
8,5,4,2,1,9,7,6,3
3,9,7,8,6,5,4,2,1
2,6,1,4,7,3,9,8,5
0,8,0,0,0,0,0,9,0
0,4,0,5,3,8,0,7,0
0,3,0,0,0,0,0,5,0
9,2,6,3,8,4,5,1,7
5,0,3,7,0,0,0,4,8
4,7,8,0,0,1,0,0,0
SOLUTION:
8,5,4,2,1,9,7,6,3
3,9,7,8,6,5,4,2,1
2,6,1,4,7,3,9,8,5
7,8,5,1,2,6,3,9,4
6,4,9,5,3,8,1,7,2
1,3,2,9,4,7,8,5,6
9,2,6,3,8,4,5,1,7
5,1,3,7,9,2,6,4,8
4,7,8,6,5,1,2,3,9
---
CLOSED STATES: 54.
OPEN STATES: 5.
DURATION TIME 70 ms.
```

Fig. 3.11 shows the graph searched for the sudoku from our first example. The heuristic function used in the search was "sum of remaining possibilities" (`HFunctionSumRemainingPossibilities` class). As usual, we encourage the reader to zoom in the figure to see more details. One can note that the first two passes were driven by forced moves — there exsisted cells where exactly one remaining possibility was left. After that, the search procedure reached a state where the cell $(4, 3)$ had two remaining possibilities $\{2, 5\}$ and it was the 'minimum cell'. Two descendant states with each of these two possibilities inserted to the board were generated and added to the *Open* queue. Interestingly, the descendant related to the first possibility was superficially more attractive since its heuristic score was $h = 57$, as opposed to $h = 63$ for the second possibility. In other words, placing the 2 in the cell $(4, 3)$ (first possibility) eliminated more remaining possibilities from the whole board than if the 5 was placed (second possibility). Therefore, the algorithm followed the first possibility in the first order. This later turned out to be leading to a contradiction ($h = \infty$). As one can note from the figure, there were more such forking points where the guesswork driven by our heuristics lead the procedure astray.

How would the second heuristic function — "empty cells" (`HFunctionEmptyCells`) — cope with the same example? Below, we first show the code excerpt demonstrating how one can switch to another heuristics (the rest of the code is as before), then we show the program output. Obviously, the procedure reaches the same solution but one can note that now a few more states are visited. We remark that this fact should not be treated as a proof that the "empty cell" heuristics is worse in general. Later, we shall show a counter example.

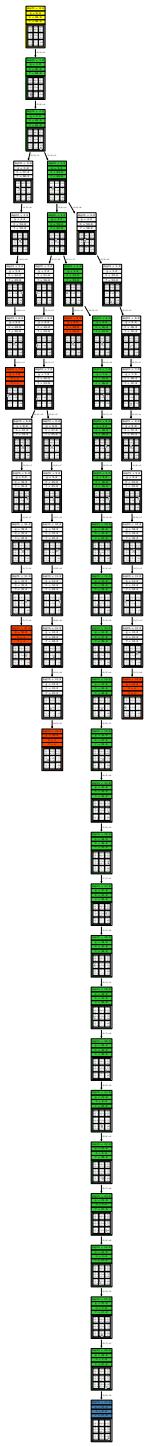


Figure 3.11: Graph searched by *SaC* using "sum of remaining possibilities" heuristics for a "not so easy sudoku".

```
1 ...
2 GraphSearchAlgorithm algorithm = new BestFirstSearch(sudoku);
3 Sudoku.setHFunction(new HFunctionEmptyCells());
4 algorithm.execute();
5 ...
```

```
SUDOKU TO SOLVE:
8,5,4,2,1,9,7,6,3
3,9,7,8,6,5,4,2,1
2,6,1,4,7,3,9,8,5
0,8,0,0,0,0,0,9,0
0,4,0,5,3,8,0,7,0
0,3,0,0,0,0,0,5,0
9,2,6,3,8,4,5,1,7
5,0,3,7,0,0,0,4,8
4,7,8,0,0,1,0,0,0
SOLUTION:
8,5,4,2,1,9,7,6,3
3,9,7,8,6,5,4,2,1
2,6,1,4,7,3,9,8,5
7,8,5,1,2,6,3,9,4
6,4,9,5,3,8,1,7,2
1,3,2,9,4,7,8,5,6
9,2,6,3,8,4,5,1,7
5,1,3,7,9,2,6,4,8
4,7,8,6,5,1,2,3,9
---
CLOSED STATES: 65.
OPEN STATES: 6.
DURATION TIME 70 ms.
```

Easy sudoku

Let us now switch to a truly easy sudoku that one might have come accross in a newspaper. It has 36 givens, so fewer than in the previous example, but the search procedure leads directly (along one path) to the solution — there are no false leads. Below, we show the output of the related program and the graph search in Fig. 3.12 using "sum of remaining possibilities" as heuristics. Not to consume too much space the figure was rotated by 90° because it represents a single path.

```
SUDOKU TO SOLVE:
0,2,0,8,1,0,7,4,0
7,0,0,0,0,3,1,0,0
0,9,0,0,0,2,8,0,5
0,0,9,0,4,0,0,8,7
4,0,0,2,0,8,0,0,3
1,6,0,0,3,0,2,0,0
3,0,2,7,0,0,0,6,0
0,0,5,6,0,0,0,0,8
0,7,6,0,5,1,0,9,0
SOLUTION:
5,2,3,8,1,6,7,4,9
7,8,4,5,9,3,1,2,6
6,9,1,4,7,2,8,3,5
2,3,9,1,4,5,6,8,7
4,5,7,2,6,8,9,1,3
1,6,8,9,3,7,2,5,4
3,4,2,7,8,9,5,6,1
9,1,5,6,2,4,3,7,8
8,7,6,3,5,1,4,9,2
---
CLOSED STATES: 46.
OPEN STATES: 0.
DURATION TIME 63 ms.
```

Figure 3.12: Graph searched by *SaC* using "sum of remaining possibilities" heuristics for an easy sudoku.

Now, we again try the second heuristics ("empty cells") on the same initial sudoku. Below is the program output. As on can see the number of states residing in *Closed* and *Open* sets at the stoppage moment are exactly the same. Therefore, both heuristics perfom equally for this particular example.

```
SOLUTION:
5,2,3,8,1,6,7,4,9
7,8,4,5,9,3,1,2,6
6,9,1,4,7,2,8,3,5
2,3,9,1,4,5,6,8,7
4,5,7,2,6,8,9,1,3
1,6,8,9,3,7,2,5,4
3,4,2,7,8,9,5,6,1
9,1,5,6,2,4,3,7,8
8,7,6,3,5,1,4,9,2
---
```

```
CLOSED STATES: 46.
OPEN STATES: 0.
DURATION TIME 62 ms.
```

'Qassim Hamza' sudoku

The following sudoku, known as 'Qassim Hamza', is regarded as a very hard for human. There are 22 givens arranged diagonally in subsquares to make things harder. The solving process requires several guesses to be made. Beneath, we show the outputs of two runs of our solver, respectively for "sum of remaining possibilitiess" and "empty cells" heuristics. Both executions last less than a second, but this time the trivial "empty cells" heuristics turns out to be cheaper. Moreover, the difference in the number of visited and generated states is significant. Fig. 3.13 demonstrates the smaller of the two graphs searched (this time we display the nodes with no contents).

```
SUDOKU TO SOLVE:
0,0,0,7,0,0,8,0,0
0,0,0,0,4,0,0,3,0
0,0,0,0,0,9,0,0,1
6,0,0,5,0,0,0,0,0
0,1,0,0,3,0,0,4,0
0,0,5,0,0,1,0,0,7
5,0,0,2,0,0,6,0,0
0,3,0,0,8,0,0,9,0
0,0,7,0,0,0,0,0,2
---
HEURISTICS: sac.examples.sudoku.HFunctionSumRemainingPossibilities.
SOLUTION :
3,2,9,7,1,6,8,5,4
1,7,6,8,4,5,2,3,9
4,5,8,3,2,9,7,6,1
6,4,3,5,7,2,9,1,8
7,1,2,9,3,8,5,4,6
8,9,5,4,6,1,3,2,7
5,8,1,2,9,4,6,7,3
2,3,4,6,8,7,1,9,5
9,6,7,1,5,3,4,8,2

CLOSED STATES: 5267.
OPEN STATES: 452.
DURATION TIME 593 ms.
---
HEURISTICS: sac.examples.sudoku.HFunctionEmptyCells.
SOLUTION:
3,2,9,7,1,6,8,5,4
1,7,6,8,4,5,2,3,9
4,5,8,3,2,9,7,6,1
6,4,3,5,7,2,9,1,8
7,1,2,9,3,8,5,4,6
8,9,5,4,6,1,3,2,7
5,8,1,2,9,4,6,7,3
2,3,4,6,8,7,1,9,5
9,6,7,1,5,3,4,8,2

CLOSED STATES: 525.
OPEN STATES: 40.
DURATION TIME 171 ms.
```

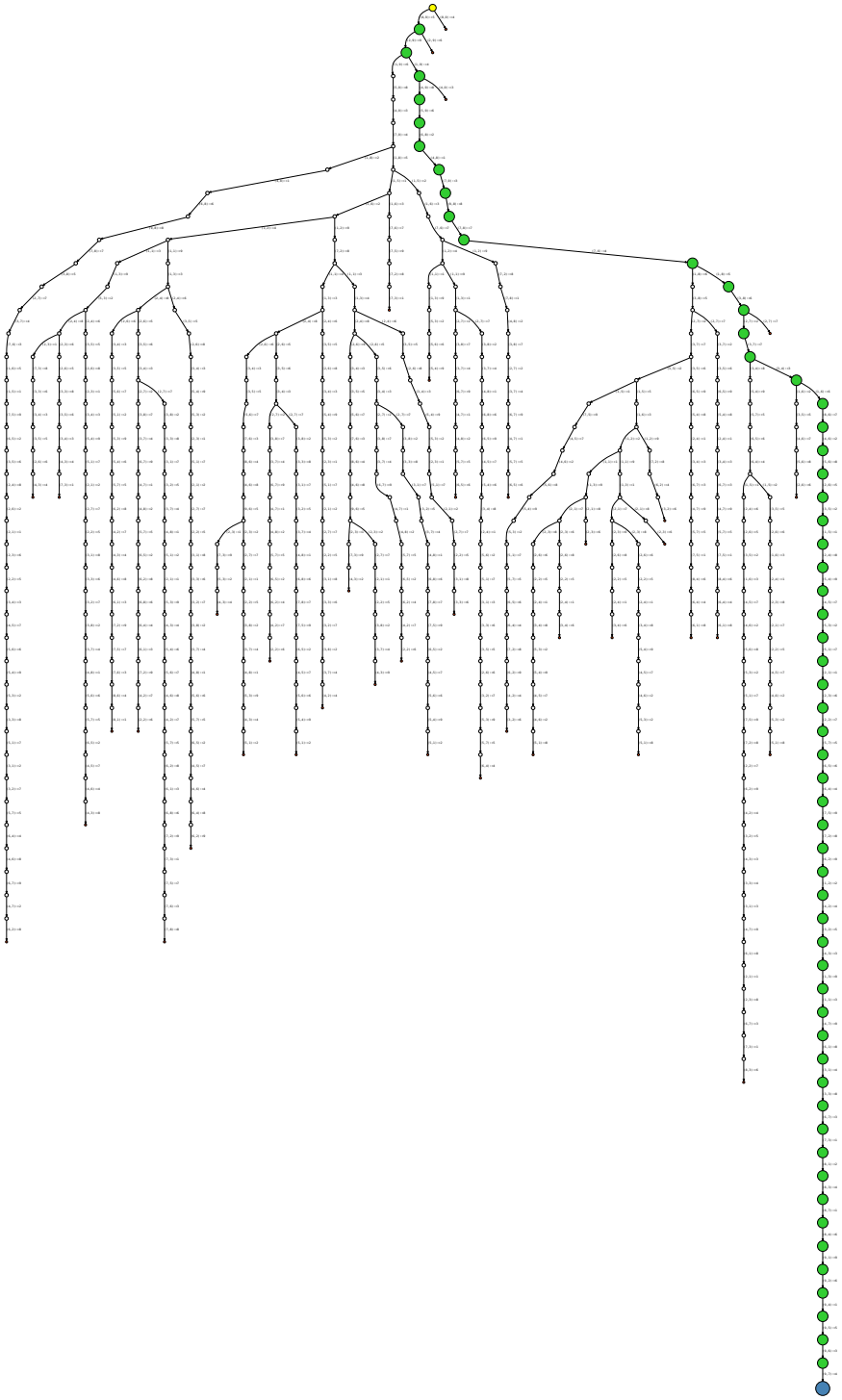


Figure 3.13: Graph searched by *SaC* using "empty cells" heuristics for the 'Qassim Hamza' sudoku.

Minimum sudoku

The next puzzle is an example of a so called minimum sudoku. Minimum sudokus are such initial arrangements which lead to a unique (single) solution and the number of givens is minimal. Up to recently it was not known what is the size of minimum sudokus for the 9×9 board. In 2012 a group of researches from Dublin University College proved that 17 is the size[14]. In other words all 9×9 sudokus with 16 or fewer givens lead to more than one solution. For strictness we also remark that obviously not all sudokus with 17 givens are necessarily minimal.

Below, we show a particular minimum sudoku and results of two executions of our *SaC* solver using two heuristics. This time the results are comparable (with a slight advantage of the 'sum of remaining possibilities' function).

```
SUDOKU TO SOLVE:
0,0,0,0,0,2,7,5,0
0,1,8,0,9,0,0,0,0
0,0,0,0,0,0,0,0,0
4,9,0,0,0,0,0,0,0
0,3,0,0,0,0,0,0,8
0,0,0,7,0,0,2,0,0
0,0,0,0,3,0,0,0,9
7,0,0,0,0,0,0,0,0
5,0,0,0,0,0,0,8,0
---
HEURISTICS: sac.examples.sudoku.HFunctionSumRemainingPossibilities.
SOLUTION:
9,4,6,1,8,2,7,5,3
3,1,8,5,9,7,4,2,6
2,7,5,6,4,3,8,9,1
4,9,2,3,1,8,5,6,7
6,3,7,2,5,4,9,1,8
8,5,1,7,6,9,2,3,4
1,2,4,8,3,5,6,7,9
7,8,3,9,2,6,1,4,5
5,6,9,4,7,1,3,8,2

CLOSED STATES: 5348.
OPEN STATES: 717.
DURATION TIME 530 ms.

HEURISTICS: sac.examples.sudoku.HFunctionEmptyCells.
SOLUTION:
9,4,6,1,8,2,7,5,3
3,1,8,5,9,7,4,2,6
2,7,5,6,4,3,8,9,1
4,9,2,3,1,8,5,6,7
6,3,7,2,5,4,9,1,8
8,5,1,7,6,9,2,3,4
1,2,4,8,3,5,6,7,9
7,8,3,9,2,6,1,4,5
5,6,9,4,7,1,3,8,2

CLOSED STATES: 5430.
OPEN STATES: 723.
DURATION TIME 561 ms.
```

[14] See: https://en.wikipedia.org/wiki/Mathematics_of_Sudoku

Suppose we want to check that the given sudoku is in fact minimal. Up to now our search procedure was stopped at the moment it reached the first goal state. From the status of the *Open* set at the stoppage moment one can see there are still potential leads to be followed. Let us force *SaC* to pursue those leads. The code excerpt below shows how this can be achieved.

```
1  GraphSearchConfigurator configurator = new GraphSearchConfigurator();
2  configurator.setWantedNumberOfSolutions(Integer.MAX_VALUE);
3
4  GraphSearchAlgorithm algorithm = new BestFirstSearch(sudoku, configurator);
5  algorithm.execute();
6
7  System.out.println("FOUND SOLUTIONS: " + algorithm.getSolutions().size() + ".");
8  System.out.println("CLOSED STATES: " + algorithm.getClosedStatesCount() + ".");
9  System.out.println("OPEN STATES: " + algorithm.getOpenSet().size() + ".");
10 System.out.println("DURATION TIME " +  algorithm.getDurationTime() + " ms.");
```

The crucial point is to explicitly instantiate a `GraphSearchConfigurator` object and to tell it we want to find all solutions[15] via `setWantedNumberOfSolutions(Integer.MAX_VALUE)`. In fact, `setWantedNumberOfSolutions(2)` would work too if we were only interested in checking the uniqueness of the solution.

The execution of our program leads now to the output shown below. It confirms the fact our particular sudoku was minimal — the number of found solutions is one and we can see that the *Open* set is now emptified (all valid leads are exhausted).

```
FOUND SOLUTIONS: 1.
CLOSED STATES: 9730.
OPEN STATES: 0.
DURATION TIME 764 ms.
```

Now, suppose we want to play a bit and we remove one given number (the left most given in the first row) from our initial board. Again, we run our solver and we tell it (by means of a configurator) to find the first three solutions. Here is the full code, slightly modified, and the program output.

```
1  Sudoku sudoku = new Sudoku(3,
2          "0,0,0,0,0,0,7,5,0," +
3          "0,1,8,0,9,0,0,0,0," +
4          "0,0,0,0,0,0,0,0,0," +
5          "4,9,0,0,0,0,0,0,0," +
6          "0,3,0,0,0,0,0,0,8," +
7          "0,0,0,7,0,0,2,0,0," +
8          "0,0,0,0,3,0,0,0,9," +
9          "7,0,0,0,0,0,0,0,0," +
10         "5,0,0,0,0,0,0,8,0");
11 System.out.println("SUDOKU TO SOLVE:\n" + sudoku);
12
13 GraphSearchConfigurator configurator = new GraphSearchConfigurator();
14 configurator.setWantedNumberOfSolutions(3);
15
16 GraphSearchAlgorithm algorithm = new BestFirstSearch(sudoku, configurator);
17 algorithm.execute();
```

[15] If RAM allows and if the set of all solutions is smaller than `Integer.MAX_VALUE`.

```
18
19 System.out.println("FOUND SOLUTIONS: " + algorithm.getSolutions().size() + ".");
20 System.out.println("CLOSED STATES: " + algorithm.getClosedStatesCount() + ".");
21 System.out.println("OPEN STATES: " + algorithm.getOpenSet().size() + ".");
22 System.out.println("DURATION TIME " +  algorithm.getDurationTime() + " ms.");
23
24
25 int i = 0;
26 for (GraphState solution : algorithm.getSolutions()) {
27     System.out.println("---");
28     System.out.println("SOLUTION " + (++i) + ":");
29     System.out.println(solution);
30 }
```

```
SUDOKU TO SOLVE:
0,0,0,0,0,0,7,5,0
0,1,8,0,9,0,0,0,0
0,0,0,0,0,0,0,0,0
4,9,0,0,0,0,0,0,0
0,3,0,0,0,0,0,0,8
0,0,0,7,0,0,2,0,0
0,0,0,0,3,0,0,0,9
7,0,0,0,0,0,0,0,0
5,0,0,0,0,0,0,8,0
FOUND SOLUTIONS: 3.
CLOSED STATES: 202.
OPEN STATES: 36.
DURATION TIME 109 ms.
---
SOLUTION 1:
9,6,4,8,2,3,7,5,1
2,1,8,5,9,7,6,3,4
3,7,5,6,4,1,8,9,2
4,9,2,3,6,8,5,1,7
1,3,7,4,5,2,9,6,8
8,5,6,7,1,9,2,4,3
6,8,1,2,3,5,4,7,9
7,4,9,1,8,6,3,2,5
5,2,3,9,7,4,1,8,6
---
SOLUTION 2:
9,6,4,8,2,3,7,5,1
2,1,8,5,9,7,6,3,4
3,7,5,6,4,1,8,9,2
4,9,2,3,6,8,5,1,7
1,3,7,4,5,2,9,6,8
8,5,6,7,1,9,2,4,3
6,8,1,2,3,5,4,7,9
7,4,3,9,8,6,1,2,5
5,2,9,1,7,4,3,8,6
---
SOLUTION 3:
9,6,4,8,2,3,7,5,1
2,1,8,5,9,7,6,4,3
3,7,5,6,4,1,8,9,2
4,9,2,3,6,8,5,1,7
1,3,7,4,5,2,9,6,8
8,5,6,7,1,9,2,3,4
6,8,1,2,3,5,4,7,9
7,4,3,9,8,6,1,2,5
```

```
5,2,9,1,7,4,3,8,6
```

We now know that our removal of one given caused that at least three solutions exist. How many solutions are there in total now? We run the code again changing `setWantedNumberOfSolutions(3)` to `setWantedNumberOfSolutions(Integer.MAX_VALUE)`. Below we show the program output. For shortness we do not display the particular solutions, just their multiplicity.

```
FOUND SOLUTIONS: 1414.
CLOSED STATES: 90594.
OPEN STATES: 0.
DURATION TIME 3619 ms.
```

The output may be a bit surprising — there are now 1 414 solutions in total and our search algorithm had to visit 90 594 states before stopping. The duration time was now noticeable — 3 619 ms.

All 4×4 sudokus

We use the ideas from the previous example to discover all distinct sudokus for the 4×4 board. It is sufficient to create an empty 4×4 sudoku and to ask *SaC* for all the solutions. The program code and its output are as follows.

```
1  Sudoku sudoku = new Sudoku(2);
2  System.out.println("SUDOKU TO SOLVE:\n" + sudoku);
3
4  GraphSearchConfigurator configurator = new GraphSearchConfigurator();
5  configurator.setWantedNumberOfSolutions(Integer.MAX_VALUE);
6
7  GraphSearchAlgorithm algorithm = new BestFirstSearch(sudoku, configurator);
8  algorithm.execute();
9
10 System.out.println("FOUND SOLUTIONS: " + algorithm.getSolutions().size() + ".");
11 System.out.println("CLOSED STATES: " + algorithm.getClosedStatesCount() + ".");
12 System.out.println("OPEN STATES: " + algorithm.getOpenSet().size() + ".");
13 System.out.println("DURATION TIME " +  algorithm.getDurationTime() + " ms.");
```

```
SUDOKU TO SOLVE:
0,0,0,0
0,0,0,0
0,0,0,0
0,0,0,0
FOUND SOLUTIONS: 288.
CLOSED STATES: 2273.
OPEN STATES: 0.
DURATION TIME 188 ms.
```

It turns out there are 288 different 4×4 sudokus. Fig. 3.14 illustrates the graph involved in that search.

One might be tempted to do the same thing for a traditional 9×9 sudoku. Yet, the trial must be abandoned, as it has been demonstrated combinatorically that the number of different sudoku puzzles is approximately $6.7 \cdot 10^{21}$. Thus, it is impossible to collect them within RAM memory.

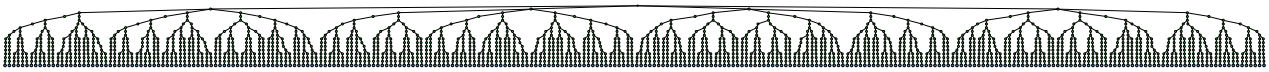


Figure 3.14: Graph searched by *SaC* in order to discover all 4×4 sudokus.

Sudoku console solver

Along with the distribution of the *SaC* library comes a console solver dedicated for the sudoku puzzle. The solver can be accessed by the included `run_sudoku.bat` file (or directly by the `sac.examples.sudoku.ConsoleSolver` class). A default execution triggered via

```
java -Xmx2048M -cp "sac-1.0.1.jar;jfreechart-1.0.14.jar;jcommon-1.0.17.jar" sac.examples.sudoku.ConsoleSolver
```

(the line from the .bat file) produces the following output to the screen with help information and a default 'Qassim Hamza' sudoku solved:

```
SUDOKU SOLVER
-----------------------------------------------------------------------------------------------------------------------------
PARAMETERS:
-s - input path to text file (one line, comma-separated) with sudoku to be solved (possible sudokus: 4x4, 9x9, 16x16, etc.)
-h - full class name of heuristic function be used (deafault: sac.examples.sudoku.HFunctionSumRemainingPossibilities)
-c - input path to .properties file with configuration settings for search process
-g - output path to .dot file in Graphviz format representing graph that was searched
-gWithContent - true/false flag stating if points in Graphviz graph should be drawn with a content or no
-----------------------------------------------------------------------------------------------------------------------------
DEFAULT SUDOKU 'QASSIM HAMZA'.
SUDOKU TO SOLVE:22
0,0,0,7,0,0,8,0,0
0,0,0,0,4,0,0,3,0
0,0,0,0,0,9,0,0,1
6,0,0,5,0,0,0,0,0
0,1,0,0,3,0,0,4,0
0,0,5,0,0,1,0,0,7
5,0,0,2,0,0,6,0,0
0,3,0,0,8,0,0,9,0
0,0,7,0,0,0,0,0,2
HEURISTICS: sac.examples.sudoku.HFunctionSumRemainingPossibilities.
SOLVING...
DURATION TIME 655 ms.
CLOSED STATES: 5267.
OPEN STATES: 452.
FOUND 1 SOLUTION(S):
---
SOLUTION 1:
3,2,9,7,1,6,8,5,4
1,7,6,8,4,5,2,3,9
4,5,8,3,2,9,7,6,1
6,4,3,5,7,2,9,1,8
7,1,2,9,3,8,5,4,6
8,9,5,4,6,1,3,2,7
5,8,1,2,9,4,6,7,3
2,3,4,6,8,7,1,9,5
9,6,7,1,5,3,4,8,2
ALL DONE.
```

As an additional example we would like to show here a 16×16 sudoku solved by the console solver. We prepared the string representing such a sudoku in a text file `d:/sudoku16.txt` and we passed the file path to the console solver via `-s` parameter. The following command:

```
java -Xmx2048M -cp "sac-1.0.1.jar;jfreechart-1.0.14.jar;jcommon-1.0.17.jar" sac.examples.sudoku.ConsoleSolver -s d:/sudoku16.txt
```

produced the ouput beneath to the console. As one can note the search procedure took over 90 s to finish.

```
SUDOKU SOLVER
--------------------------------------------------------------------------------------------------------------------------------
PARAMETERS:
-s - input path to text file (one line, comma-separated) with sudoku to be solved (possible sudokus: 4x4, 9x9, 16x16, etc.)
-h - full class name of heuristic function be used (deafault: sac.examples.sudoku.HFunctionSumRemainingPossibilities)
-c - input path to .properties file with configuration settings for search process
-g - output path to .dot file in Graphviz format representing graph that was searched
-gWithContent - true/false flag stating if points in Graphviz graph should be drawn with a content or no
--------------------------------------------------------------------------------------------------------------------------------
SUDOKU TO SOLVE:
7,0,0,0,0,5,1,0,3,11,0,0,0,0,0,0
12,8,0,0,0,15,14,0,4,0,9,0,11,0,16,2
0,15,10,2,13,0,0,0,0,7,0,5,8,0,3,0
1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
0,0,1,0,0,2,0,0,15,0,0,0,5,0,0,11
15,0,0,3,0,0,0,0,0,0,7,14,6,0,1,0
14,0,16,0,0,0,0,0,0,5,6,0,10,2,0,0
0,0,12,0,0,0,0,8,9,1,10,13,16,0,0,3
0,0,0,0,0,0,0,0,0,15,0,0,9,8,0,0
0,6,4,0,0,10,0,0,7,0,14,0,0,0,11,0
0,16,0,12,0,3,9,0,10,0,0,0,8,0,0,5,0
3,0,0,0,15,0,0,0,0,13,0,4,0,14,0,16
0,0,0,0,0,14,15,13,0,10,0,8,0,0,4,0,9
9,0,7,0,6,0,0,0,0,0,0,1,0,0,0,0
5,13,8,0,3,0,10,0,0,0,0,11,6,0,0,15,0
10,1,15,0,0,0,5,16,14,0,4,0,0,6,0,0
HEURISTICS: sac.examples.sudoku.HFunctionSumRemainingPossibilities.
SOLVING...
DURATION TIME 90870 ms.
CLOSED STATES: 333585.
OPEN STATES: 33769.
FOUND 1 SOLUTION(S):
---
SOLUTION 1:
7,14,6,9,8,5,1,12,3,11,2,16,4,10,13,15
12,8,13,5,7,15,14,3,4,6,9,10,11,1,16,2
4,15,10,2,13,16,11,6,1,7,12,5,8,9,3,14
1,3,11,16,2,9,4,10,8,14,13,15,7,5,6,12
6,10,1,13,14,2,3,9,15,4,16,12,5,7,8,11
15,5,9,3,10,13,16,11,2,8,7,14,6,12,1,4
14,4,16,8,1,12,7,15,11,5,6,3,10,2,9,13
2,11,12,7,5,4,6,8,9,1,10,13,16,15,14,3
13,7,5,1,12,6,2,14,16,15,3,11,9,8,4,10
8,6,4,15,16,10,13,5,7,12,14,9,2,3,11,1
11,16,14,12,4,3,9,7,10,2,1,8,15,13,5,6
3,9,2,10,15,11,8,1,6,13,5,4,12,14,7,16
16,12,3,6,11,14,15,13,5,10,8,7,1,4,2,9
9,2,7,14,6,8,12,4,13,16,15,1,3,11,10,5
5,13,8,4,3,1,10,2,12,9,11,6,14,16,15,7
10,1,15,11,9,7,5,16,14,3,4,2,13,6,12,8
ALL DONE.
```

Chapter 4

Searching game trees

This chapter is devoted to the part of *SaC* library responsible for searching game trees. We begin the chapter with a section reminding several well known game search algorithms. The reader familiar with these algorithms can move over to subsequent sections. The second section discusses *SaC*'s API related to game tree searching. The last one presents two game-related examples in which we have applied *SaC* for demonstration, namely: checkers and Nim.

4.1 Algorithms

Algorithms for game trees are based on the concept of *minimax*. Historically, this concept is due to von Neumann who formulated and proved the *minimax theorem* (von Neumann, 1928; von Neumann and Morgenstern, 1944). The theorem itselft has a bit different and more general setting than typically can be met in algorithms meant for games like for example chess. The theorem pertains to zero-sum two-person games, covers both cases where players make simultaneous or alternate moves, and implies the existance of an optimal, so called, mixed strategy for each player. If both players apply their optimal strategies the game will be driven to a saddle point (a.k.a. minimax point) resulting in a game value (minimax value) that none of the players can improve by changing his strategy. Putting it more simply, the concept of minimax can be explained as a decision rule which tells a player to minimize the maximum possible payoff for the other player.

SaC provides three well known algorithms dedicated for games where players make alternate moves. The algorithms are recursive and it is convenient to express each of them in a form of two twin procedures that call one another interchangeably. In subsequent three sections we present these algorithms. We try to give them in a compact form, limiting ourselves to the most important backbone. In later sections, we describe some possible refinements and tricks that the algorithms can be augmented with to boost up their performance.

4.1.1 Min-Max

The subsequent pseudocode shows the simplest algorithm for game trees — the Min-Max algorithm. It demonstrates the basic mechanics underlying game tree searching. In the code *s* denotes

the current state (for which the current recursive call is made), d denotes the depth of s, and D is the maximum depth limit.

Alg. 8 Min-Max

```
1:  procedure MMEvaluateMaxState(s, d, D)
2:      if IsTerminal(s, d, D) then return h(s)        ▷ h(s) — heuristic evaluation of position
3:      v := −∞
4:      generate descandants {t} of s
5:      for all t do
6:          w :=MMEvaluateMinState(t, d + 1/2, D)
7:          if s is the root state then memorize w as the score for s → t move
8:          v := max{v, w}
9:      return v
10: procedure MMEvaluateMinState(s, d, D)
11:     if IsTerminal(s, d, D) then return h(s)        ▷ h(s) — heuristic evaluation of position
12:     v := ∞
13:     generate descandants {t} of s
14:     for all t do
15:         w :=MMEvaluateMaxState(t, d + 1/2, D)
16:         if s is the root state then memorize w as the score for s → t move
17:         v := min{v, w}
18:     return v
```

The outermost call of the recursion should be made with respect to an initial state from which we want to start the analysis, in particular this can be the initial position of the game, and that state is treated as the root of game tree.

In lines 7 and 16 the algorithm memorizes a score for some move. In an actual implementation such scores are typically memorized in a data structure that resides in the global scope outside the recursion (for example this could be a hash map).

We distinguished the condition checking if given state is a terminal as a separate method IsTerminal. It can be regarded as an additional routine method. We do not want to define it here explicitly as it may depend on the rules of the game. Most commonly though, in IsTerminal one has to check if either: maximum depth has been reached i.e. $d = D$; or s is a win state, i.e. $h(s) = \pm\infty$; or s is a non-win but terminal state due to some special rule, for example a stalemate or a perpetual check in chess.

We should remark that in some situations it may be reasonable to pursue consequences of a state deeper although it is a terminal according to the maximum depth reached ($d = D$). For example, in chess or checkers when a series of captures by both players is involved, one should analyze the series to the very end and return its final evaluation. An algorithmic gadget for that purpose, called *Quiescence*, is described in further sections.

As regards the computational complexity, it is easy to note that it is exponential with respect to the imposed depth limit D — the seach horizon. If the given game has a constant branching factor

b (or if it is possible to estimate that factor to be b on avarage) then the computational complexity of Min-Max is of $O(b^D)$ order. In that sense the algorithm can be treated as an exhaustive search up to the imposed horizon. It does not discard any subtrees.

4.1.2 Alpha-beta pruning

There are several people regarded as independent and almost simultaneous discoverers of the *alpha-beta pruning* algorithm (a.k.a. *alpha-beta cut-offs*) around late 1950s and early 1960s. In particular, the discoverers or contributors were: Daniel J. Edwards, Allen Newell, Hebert A. Simon, John McCarthy, Arthur Samuel, Alexander Brudno; see e.g. (Edwards and Hart, 1963; Brudno, 1963; Newell and Simon, 1976). Later, Knuth and Moore (1975) refined the algorithm and gave its detailed analysis. Pearl (1982) proved its optimality.

The pseudocode of the α-β pruning algorithm is presented below.

Alg. 9 Alpha-beta pruning

```
1:  procedure AlphaBetaEvaluateMaxState(s, d, D, α, β)
2:      if IsTerminal(s, d, D) then return h(s)          ▷ h(s) — heuristic evaluation of position
3:      generate descandants {t} of s
4:      for all t do
5:          v :=AlphaBetaEvaluateMinState(t, d + 1/2, D, α, β)
6:          if s is the root state then memorize v as the score for s → t move
7:          α := max{α, v}
8:          if α ≥ β then return α                       ▷ cut off — no further child t will be checked
9:      return α
10: procedure AlphaBetaEvaluateMaxState(s, d, D, α, β)
11:     if IsTerminal(s, d, D) then return h(s)          ▷ h(s) — heuristic evaluation of position
12:     generate descandants {t} of s
13:     for all t do
14:         v :=AlphaBetaEvaluateMaxState(t, d + 1/2, D, α, β)
15:         if s is the root state then memorize v as the score for s → t move
16:         β := min{β, v}
17:         if α ≥ β then return β                       ▷ cut off — no further child t will be checked
18:     return β
```

Alpha-beta pruning belongs to the class of "branch and bound" algorithms. It works by tracking two bounds on the game value along the recursion: a lower bound α and an upper bound β. At any given point of the recursion one can understand α as the guaranteed payoff for the maximizing player and β as the guaranteed payoff for the minimizing player. The outermost call of the recursion is made with the settings $\alpha = -\infty$ and $\beta = \infty$, i.e. the most pessimistic values for the maximizing and the minimizing player respectively.

The legal situation for the bounds is that they remain satisfying the $\alpha < \beta$ inequality. Clearly, $\alpha > \beta$ is a contradiction. For example, a maximizing player cannot be guaranteed with a payoff of

5, while the minimizing player is simultaneously guaranteed a payoff of −2. Therefore, if at some stage of tree analysis one comes across a state for which the initial condition to run the subrecursion is $\alpha \geq \beta$, then, it means there is no point to do so because the subtree rooted by that state will not affect the game value. All such subtrees can be discarded. Note that the equality $\alpha = \beta$ is not a contradiction, but it also means that neither player can improve his score within a given subtree. In other words, entering subtrees with $\alpha \geq \beta$ would not be an effect of optimal behaviour of either player. On the other hand, one must understand that α-β is an exact algorithm, not an approximation. Its results are identical with results of Min-Max. It means that even though we do not analyze some subtrees we do not miss anything relevant in the game. If an actual game being played entered in its course of actions a discarded subtree that would mean one of the players made a mistake and the opponent can only gain from that — he can improve or sustain his result, not deteriorate it.

It can be shown that an optimistic complexity of the alpha-beta pruning is $O(b^{1/2D})$. It arises when the ordering of children nodes is optimal, meaning that the best moves are always analyzed in first order, which allows for early cut-offs. The pessimistic complexity is $O(b^D)$ — the same as for Min-Max. With random ordering of moves one obtains an average complexity of $O(b^{3/4D})$

It is also worth mentioning that the shown pseudocode represents the alpha-beta pruning algorithm in the so called *fail-soft* version. It means that outcomes returned from particular recursion calls do not have to fall into the starting $[\alpha, \beta]$ interval the call was initially made with. There also exist the *fail-hard* version, in which each outcome must be bounded to the initial $[\alpha, \beta]$ interval[1]. The difference between the two versions does not affect neither the final result of the outermost call nor the collection of memorized move scores. One can check that even if some subrecursion returns and outcome outside the initial $[\alpha, \beta]$ interval, then, at the resursive stage one level up, this outcome will be ignored because it does not improve the suitable α or β payoff. On the other hand the *fail-soft* version is important because it helps to construct more advanced algorithms (like Scout or MTD-f) that are based on the notion of so called *zero windows*. We discuss that concept in the next section.

4.1.3 Scout

The main idea behind the Scout algorithm is driven from the following Knuth-Moore theorem.

Theorem 1 *(Knuth and Moore, 1975, "theorem about α-β window") Let v^* denote the true, exact game value returned by the Min-Max procedure executed with respect to some state. Let v denote the outcome of a fail-soft alpha-beta pruning procedure executed for that state and with a search window defined by α and β numbers imposed arbitrarily. Then, the following three cases are possible.*

1. $\alpha < v < \beta \Rightarrow v = v^*$,
2. $v \leq \alpha \Rightarrow v^* \leq v$,
3. $\beta \leq v \Rightarrow v \leq v^*$.

[1] In the *fail-hard* version, lines 8 and 17 would return: β and α respectively.

The meaning of the theorem for the design of new search algorithm is the following. We may try to *guess* a more narrow α-β window than in the traditional pruning procedure. Note that the more narrow the window is the more cutoffs are likely to be produced within the recursion. If the outcome value v, being the consequence of our guess, falls inside the (α, β) interval then we are lucky, because v is equal to the true game value v^*, and we can experience a computational gain (if more cutoffs occur). On the other hand if the outcome falls outside the interval, then one obtains a lower or an upper bound on true game value. Now, two cases are possible: either we have to repeat the execution with a wider window (and we experience a computational loss), or we can make use of the bound (and we experience a gain again). For example, if the maximizing player is guaranteed with a payoff α and we obtained an outcome $v < \alpha$, then, despite the fact that v is not an exact game value, we do not have to repeat the computations because v upperbounds v^*, and the computations (if repeated) would not improve the α.

Historically, the first one to study this idea was Pearl (1980). He postulated that one can more cheaply test whether a payoff can be improved. He introduced two recursive procedures `test(...)` and `eval(...)`, where the first one returned a boolean value stating if an improvement was possible, while the other calculated the exact game value if needed.

The idea was then refined by Reinefeld (1983) who introduced the concept of *zero search windows* (a.k.a. *null windows* or *scout windows*). If the payoffs in a game are integer numbers then a zero window is the one satisfying the condition

$$\alpha + 1 = \beta. \tag{4.1}$$

We first give some initial definitions and remarks that help to understand the Scout algorithm. Then, we demonstrate the actual pseudocode (Alg. 10).

1. We shall say that an imposed α-β window *succeeded* if the value v returned by tha fail-soft procedure satisfies: $\alpha < v < \beta$. This implies that the true game value v^* equals v.

2. We shall say that an imposed α-β window *failed low* if the value v returned by tha fail-soft procedure satisfies: $v \leq \alpha$. This implies that v is an upper bound on the true game value, i.e.: $v^* \leq v$.

3. We shall say that an imposed α-β window *failed high* if the value v returned by tha fail-soft procedure satisfies: $\beta \leq v$. This implies that v is a lower bound on the true game value, i.e.: $v \leq v^*$.

4. A zero window (with $\alpha + 1 = \beta$) *must* fail either low or high.

5. In the algorithm, only the first descendant of every state is analyzed with a full α-β window, the second and further descendants are analyzed with zero windows, i.e. α-$(\alpha+1)$ or $(\beta-1)$-β, respectively for the states of type MAX and MIN.

6. If a zero window, imposed for a descendant of a MAX state, fails low then we do not care — the payoff of the maximizing player could not have been improved within that descendant's subtree. This usually carries a computational gain because more cutoffs are likely to occur with a zero window imposed.

7. If a zero window, imposed for a descendant of a MAX state, fails high then we have to repeat the search with respect to that descendant with a wider window: v-β, to obtain an exact value for the subtree. The repetition carries a computational loss, but note that the new window v-β is still more narrow than a full initial α-β window.

8. The above two remarks are symmetrically opposite for the MIN states.

9. The algorithm contains a condition checking if the current state is two half-moves away from the maximum depth or deeper ($D - d \leq 2 \cdot 1/2$). If so, the repetition of search is not needed despite a fail-situation because it can be shown the algorithm works correctly at those depths.

10. Experimental studies, in particular from (Reinefeld, 1983), indicate that computational gains consequencing from zero windows and thus more frequent cutoffs are typically greater than computational losses consequencing from search repetitions.

11. Reinefeld (1983) experimented with random games imposing branching factors within $[20, 60]$ (similar to chess) and depths of $\{4, 5\}$ half-moves. He experienced that his Scout algorithm visited about 20% less of tree terminals than in the case of a traditional alpha-beta pruning.

Alg. 10 Scout

```
1: procedure ScoutEvaluateMaxState(s, d, D, α, β)
2:     if IsTerminal(s, d, D) then return h(s)          ▷ h(s) — heuristic evaluation of position
3:     b := β
4:     generate descandants {t} of s
5:     for all t do
6:         v :=ScoutEvaluateMinState(t, d + 1/2, D, α, b)
7:         if t is not the first descendant and D − d > 2 · 1/2 and b ≤ v then          ▷ failing high
8:             v :=ScoutEvaluateMinState(t, d + 1/2, D, v, β)
9:         if s is the root state then memorize v as the score for s → t move
10:        α := max{α, v}
11:        if α ≥ β then return α          ▷ cut off — no further child t will be checked
12:        b := α + 1                                   ▷ forcing a zero window
13:    return α
14: procedure ScoutEvaluateMinState(s, d, D, α, β)
15:    if IsTerminal(s, d, D) then return h(s)          ▷ h(s) — heuristic evaluation of position
16:    a := α
17:    generate descandants {t} of s
18:    for all t do
19:        v :=ScoutEvaluateMaxState(t, d + 1/2, D, a, β)
20:        if t is not the first descendant and D − d > 2 · 1/2 and v ≤ a then          ▷ failing low
21:            v :=ScoutEvaluateMaxState(t, d + 1/2, D, α, v)
22:        if s is the root state then memorize v as the score for s → t move
23:        β := min{β, v}
24:        if α ≥ β then return β          ▷ cut off — no further child t will be checked
25:        a := β − 1                                   ▷ forcing a zero window
26:    return β
```

4.2 API

4.2.1 Game state abstraction

In *SaC*'s API for searching games, the main 'actor' is the `sac.graph.GameState` interface being an extension of the `sac.State` discussed earlier in Chapter 2. Below, we present the code listing for the `sac.graph.GameState` interface and then some excerpts from its default implementation — `sac.graph.GameStateImpl`. For full code of the implementation the reader is addressed to Appendix 6.3.

```
1  package sac.game;
2
3  import java.util.List;
4
5  import sac.State;
6
7  public interface GameState extends State {
8
9      public static final double H_SMALLEST_INFINITY = 0.5 * Double.MAX_VALUE;
10
11     @Override
12     public GameState getParent();
13     @Override
14     public List<GameState> getChildren();
15     @Override
16     public List<GameState> getPath();
17     @Override
18     public List<GameState> generateChildren();
19
20     public boolean isMaximizingTurnNow();
21     public void setMaximizingTurnNow(boolean maximizingTurnNow);
22     public boolean isWinTerminal();
23     public boolean isNonWinTerminal();
24     public List<String> getMovesAlongPrincipalVariation();
25     public boolean isQuiet();
26     public boolean isVisited();
27     public void setVisited(boolean visited);
28     public boolean isReadFromTranspositionTable();
29     public void setReadFromTranspositionTable(boolean readFromTranspositionTable);
30 }
```

The first thing to note is that the `GameState` interface reformulates the signatures of four basic methods related to the parent – children binding and path tracking. The difference (of cosmetic nature) is in the types returned, which now become `GameState` or `List<GameState>`, as the resulting objects should be treated as game states (rather than general states).

Apart from the above, there are several new elements in the interface (with respect to its ancestor `sac.State`):

- a constant `H_SMALLEST_INFINITY` storing the smallest value of the heuristic function (position evaluation) that already represents a win for a player — all values larger than the constant also represent wins, but achieved faster (at smaller depths),

- a pair of methods to check or set whose turn it is now — `isMaximizingTurnNow()`,

`setMaximizingTurnNow(...)`,

- a method designed to indicate whether the current state is a win terminal state — `isWinTerminal()`,
- a method designed to indicate whether the current state is a terminal but non-win state due to some special rule of the game (e.g. perpetual check or a stalemate in chess) — `isNonWinTerminal()`,
- a method returning the list of move names along the principal variation from the current state downwards the tree — `getMovesAlongPrincipalVariation()`,
- a method designed to indicate whether the current state is a so called quiet[2] state — `isQuiet()`,
- a pair of methods to check or set whether the given state was actually visited during the search procedure; it may be so that a state was generated but in fact not visited due to a cutoff — `isVisited()`, `setVisited(...)`,
- a pair of methods to check or set whether the evaluation for given state was read from the transposition table — `isReadFromTranspositionTable()`, `setReadFromTranspositionTable(...)`.

The last two pairs of methods are not recommended to be called explicitly by the user, they are meant for internal purposes of *SaC*'s API.

Below, we show some excerpts from the default game state implementation in *SaC*.

```
1  public abstract class GameStateImpl extends StateImpl implements GameState {
2
3      protected boolean maximizingTurnNow = true;
4      protected List<String> movesAlongPrincipalVariation = null;
5      protected boolean visited = false;
6      protected boolean readFromTranspositionTable = false;
7
8      ...
9
10
11     @Override
12     public final boolean isMaximizingTurnNow() {
13         return maximizingTurnNow;
14     }
15
16     @Override
17     public final void setMaximizingTurnNow(boolean maximizingTurnNow) {
18         this.maximizingTurnNow = maximizingTurnNow;
19     }
20
21     @Override
22     public boolean isQuiet() {
23         return true; // default implementation
```

[2] E.g. in chess or checkers typically a state is regarded as quiet if no immediate captures are possible. If a state is not quiet then it is advised it should be analyzed further (to have a good evaluation of its consequences) even though it may be a state at the maximum depth of deeper.

```
24      }
25
26      @Override
27      public final List<String> getMovesAlongPrincipalVariation() {
28          return movesAlongPrincipalVariation;
29      }
30
31      @Override
32      public boolean isNonWinTerminal() {
33          return false;
34      }
35
36      @Override
37      public final boolean isWinTerminal() {
38          return Math.abs(getH()) >= H_SMALLEST_INFINITY;
39      }
40  }
```

4.2.2 General (abstract) game search algorithm

In this section we discuss *SaC*'s internal mechanisms related to the general game searching procedure. This procedure is represented by an abstract class: `sac.game.GameSearchAlgorithm`. Obviously, the end user of *SaC* library does not have to be aware of core-level intricacies. Instead, in practice he can limit himself to instantiating a specific search algorithm (e.g.: `new MinMax()`, `new AlphaBetaPruning(...)` or `new Scout(...)`) and running it for his particular game state. Therefore, we recommend the reading of subsequent contents only to readers really interested in low level details of *SaC*, perhaps the readers intending to extend the library in the future with new algorithms or data structures.

The listing below demonstrates the most important parts of the `GameSearchAlgorithm` class. For clarity, some less interesting parts have been skipped. The full code is available in the Appendix 6.4. The class is designed to work as a general and common (model) procedure for searching games and provides a set of helper methods that the specific algorithms — the subclassess — can rely on.

```
1   package sac.game;
2
3   ...
4
5   public abstract class GameSearchAlgorithm extends SearchAlgorithm {
6
7       protected GameState initial = null;
8       protected GameState current = null;
9       protected Map<String, Double> movesScores = null;
10
11      protected TranspositionTable transpositionTable = null;
12      protected RefutationTable refutationTable = null;
13
14      protected GameSearchConfigurator configurator = null;
15
16      protected int closedCount = 0; // number of closed states (= number of calls of methods
            evaluateMaxState(), evaluateMinState) since last reset().
17      protected double depthReached = 0.0; // maximum depth that was reached in the search (owing
            to quiescence) since last reset().
```

```
18      protected boolean stopForced = false; // boolean flag stating if stop was forced (e.g. from
            an outer thread)
19
20      public GameSearchAlgorithm(GameState initial, GameSearchConfigurator configurator) {
21          ...
22      }
23
24      @Override
25      public final void execute() {
26          reset();
27          startTime = System.currentTimeMillis();
28          doExecute();
29          endTime = System.currentTimeMillis();
30      }
31
32      protected void doExecute() {
33          Double gameValue = null;
34          if (initial.isMaximizingTurnNow())
35              gameValue = evaluateMaxState(initial, Double.NEGATIVE_INFINITY, Double.
                    POSITIVE_INFINITY, 0.0, configurator.getDepthLimit());
36          else
37              gameValue = evaluateMinState(initial, Double.NEGATIVE_INFINITY, Double.
                    POSITIVE_INFINITY, 0.0, configurator.getDepthLimit());
38          if (configurator.isTranspositionTableOn())
39              transpositionTable.putOrUpdate(initial, gameValue, Double.NEGATIVE_INFINITY, Double.
                    POSITIVE_INFINITY);
40          current = null;
41      }
42
43      protected final Double evaluateMaxState(GameState gameState, double alpha, double beta,
            double depth, double depthLimit) {
44          // time limit check
45          if ((stopForced) || (configurator.getTimeLimit() < Long.MAX_VALUE)) {
46              long currentTime = System.currentTimeMillis();
47              if (currentTime - startTime > configurator.getTimeLimit()) {
48                  endTime = System.currentTimeMillis();
49                  return null;
50              }
51          }
52          closedCount++;
53          current = gameState;
54          gameState.setVisited(true);
55          return doEvaluateMaxState(gameState, alpha, beta, depth, depthLimit);
56      }
57
58      protected final Double evaluateMinState(GameState gameState, double alpha, double beta,
            double depth, double depthLimit) {
59          // time limit check
60          if ((stopForced) || (configurator.getTimeLimit() < Long.MAX_VALUE)) {
61              long currentTime = System.currentTimeMillis();
62              if (currentTime - startTime > configurator.getTimeLimit()) {
63                  endTime = System.currentTimeMillis();
64                  return null;
65              }
66          }
67          closedCount++;
68          current = gameState;
69          gameState.setVisited(true);
70          return doEvaluateMinState(gameState, alpha, beta, depth, depthLimit);
71      }
72
```

```
73      public abstract Double doEvaluateMaxState(GameState gameState, double alpha, double beta,
            double depth, double depthLimit);
74
75      public abstract Double doEvaluateMinState(GameState gameState, double alpha, double beta,
            double depth, double depthLimit);
76
77      protected void reset() {
78          ...
79      }
80
81      public Map<String, Double> getMovesScores() {
82          return movesScores;
83      }
84
85      public final String getFirstBestMove() {
86          ...
87      }
88
89      public final List<String> getBestMoves() {
90          ...
91      }
92
93      ... // getters, setters
94
95      protected final List<GameState> generateChildrenWrapper(GameState parent) {
96          List<GameState> children = parent.generateChildren();
97          for (GameState child : children) {
98              child.setParent(parent);
99              child.setDepth(parent.getDepth() + 0.5);
100             if (configurator.isParentsMemorizingChildren())
101                 parent.getChildren().add(child);
102             recalculateHIfLarge(child);
103         }
104         return children;
105     }
106
107     protected final static void recalculateHIfLarge(GameState state) {
108         double h = state.getH();
109         if (Math.abs(h) > GameState.MAX_H_VALUE) {
110             h = Math.signum(h) * GameState.MAX_H_VALUE * (1.0 + 1.0 / state.getDepth());
111             state.setH(h);
112         }
113     }
114
115     public final boolean isGameStateTerminal(GameState gameState, double depth, double depthLimit
            ) {
116         depthReached = Math.max(depthReached, depth);
117         if ((gameState.isWinTerminal()) || (gameState.isNonWinTerminal()))
118             return true;
119         if (depth >= depthLimit)
120             return (configurator.isQuiescenceOn()) ? gameState.isQuiet() : true;
121         return false;
122     }
123
124     public final static boolean isExactGameValue(double childValue, double alpha, double beta) {
125         return ((alpha < childValue) && (childValue < beta)) || ((childValue == Double.
                NEGATIVE_INFINITY) && (alpha == Double.NEGATIVE_INFINITY))
126                 || ((childValue == Double.POSITIVE_INFINITY) && (beta == Double.POSITIVE_INFINITY
                    ));
127     }
128
```

```
129     public final static void updateMovesAlongPrincipalVariation(GameState parent, GameState child
            ) {
130         List<String> movesAlongPrincipalVariation = parent.getMovesAlongPrincipalVariation();
131         movesAlongPrincipalVariation.clear();
132         movesAlongPrincipalVariation.add(child.getMoveName());
133         movesAlongPrincipalVariation.addAll(child.getMovesAlongPrincipalVariation());
134     }
135 
136     public final void forceStop() {
137         stopForced = true;
138     }
139 }
```

The main functional goal of a `GameSearchAlgorithm` object (regardless of its actual type) is to assign numeric scores to available moves in the current game position. The method exposed to the user for that purpose is the `execute()` method. Once it is finished, the move scores are memorized in a protected field: `Map<String, Double> movesScores`. It is a map type data structure storing (key, value) pairs, where the key (`String`) represents the name of a move and the value (`Double`) represents the heuristic evaluation for the move. Once the game tree analysis is finished the move scores are accessible to the user by one of the following methods: `getMovesScores()`, `getFirstBestMove()`, `getBestMoves()`. The first one is the most general getter returning all the scores. The second method returns only one name of a move which is the best or belongs to the group of several best moves (when equal highest scores occur). The third method returns a list of all best moves (again, when equal scores occur). It is worth to remind that in order for the scores map to be popoulated, the user must not forget to impose names on the descendant states, via the `setMoveName(...)` method, while they are being generated inside the `generateChildren()`. Both these methods come from the more general `sac.State` interface.

The next elements to note in the `GameSearchAlgorithm` class are references to transposition and refutation tables. These structures are discussed in detail in later sections, but it can be explained now, in short, that they allow for a quicker search either by avoiding multiple analysis of the same subtrees reached by different paths (transposition table), or by encouraging more cutoffs (refutation table).

As mentioned before the main operational method of the `GameSearchAlgorithm` class is the `execute()`. Underneath, the method calls a hierarchy of wrapping submethods, namely: it first calls the `doExecute()`, which in turn calls a suitable wrapper `evaluateMaxState(...)` or `evaluateMinState(...)` depending on whose turn it is now, then these wrappers call the actual evaluation methods — `doEvaluateMaxState(...)` or `doEvaluateMinState(...)`. The wrapping mechanism allows to suitably handle the following elements:

- resetting initial state and some data structures before a new analysis of the tree,
- time measurements — execution and an early stoppage under time control,
- checking if an outer stop on the procedure was forced,
- counting and flagging the visited states.

One can note that the most low-level methods `doEvaluateMaxState(...)` and `doEvaluateMinState(...)`

are left as abstract and their implementations are supposed to be filled in by the specific search algorithms — sublcasses of the `GameSearchAlgorithm` class.

We now comment on a few auxiliary methods of the `GameSearchAlgorithm` class that are used within search procedures.

The `generateChildrenWrapper(...)` is a method designed to be called from within the implementations of `doEvaluateMaxState(...)` or `doEvaluateMinState(...)` methods. That wrapper is meant to handle some operations accompanying the generation of descendants. Firstly, the wrapper invokes the `generateChildren()` call back method on the parent state. The implementation of that method is supposed to be provided by the user. Then, the wrapper iterates over the created descendants and for each of them it updates the depth and parent–child binding information. Finally, for each descendants its heuristic evaluation is updated, if it represents a win state, via the `recalculateHIfLarge(...)` method. The details of this operation are discussed in the next subsection.

The `isGameStateTerminal(...)` helper returns a boolean stating whether one of the following events occurred: the maximum depth was reached and the given state is quiet, the given state is a win terminal, the given state is a non-win terminal i.e. a terminal implied by some special rule of the game (e.g. perpetual check in chess).

The `isExactGameValue(...)` helper should be considered in the light of the Knuth-Moore theorem. The method returns the flag `true` if the tested game value falls inside the α-β window. Infinities are also suitably handled in the method. The method is useful in two places. Firstly, it is a condition when an actual algorithm is about to memorize a score for a move. If a game value is not exact, but rather a bound, then the algorithm should not memorize it as a score. Secondly, a similar condition occurs when a transposition table considers a state and its score to be stored.

The `updateMovesAlongPrincipalVariation(...)` forces an update of the principal variation, stored as a list of move names, for given parent object. It is done by using such a descendant of the parent that have just led to an improvement of the game value (or bound). This method is called in suitable places by `MinMax`, `AlphaBetaBruning`, and `Scout` classes inside their `doEvaluateMaxState(...)` and `doEvaluateMinState(...)` methods when the aforementioned improvement takes place.

4.2.3 Recalculation of heuristics for win states

The `recalculateHIfLarge(...)` method maps suitably the infinities (`Double.POSITIVE_INFINITY` or `Double.NEGATIVE_INFINITY`) to large values, taking into account the depth at which the winning position was achieved. This allows to 'differntiate infinities' in case several winning lines are possible. The algorithm assigns the highest score to the move which leads to a win in the fewest moves.

Let h denote the heuristic evaluation assigned originally to a given state by the user and let d be the depth of that state ($d \in \{0.0, 0.5, 1.0, 1.5, \ldots\}$). Also, let h_∞ denote the `H_SMALLEST_INFINITY` constant (equal to `8.988465674311579E307`) from the `GameState` class, i.e. the smallest value representing a win. Then, provided that $h \geq h_\infty$ the recalculation of the evaluation is carried out

according to the following formula:

$$h := \mathrm{sgn}(h) \cdot h_\infty \cdot (1 + 1/d). \tag{4.2}$$

The table 4.1 shows some recalculation results in the `Double` type (ignoring the sign) as the depth parameter grows. As one can see the successive values approach `H_SMALLEST_INFINITY` as $d \to \infty$. It is worth to comment that the recalculation value for $d = 0.0$ was not reported in the

d	recalculated unsigned h
0.5	`Double.POSITIVE_INFINITY`
1.0	`1.7976931348623157E308`
1.5	`1.498077612385263E308`
2.0	`1.3482698511467367E308`
2.5	`1.2583851944036209E308`
3.0	`1.1984620899082103E308`
⋮	⋮

Table 4.1: Initial values of recalculated evaluations for win states depending on their depths.

table because such case does not occur in the typical usage of *SaC* — the analysis of a game tree is typically ordered for some state while the game is still ongoing, not finished. Therefore, the earliest moment a win can be reached is for $d = 0.5$. Yet, it can be checked programmistically that for $d = 0.0$ the formula (4.2) responds with `Double.POSITIVE_INFINITY`, so with the same result as for $d = 0.5$.

The recalculation mechanism is a facilitation for the user (the programmer of some game). He does not have to worry about 'differentiating infinities' himself. His heuristic evaluation function, attached via the `setHFunction(...)` functionality, is required only to return `Double.POSITIVE_INFINITY` or `Double.NEGATIVE_INFINITY` in case of a win and regardless of the depth. Those values shall later be recalculated automatically on the fly by *SaC*.

4.2.4 Specific game search algorithms

In this section presented are code excerpts for the three game search algorithms included in *SaC*. Let us start with the `MinMax` class.

```
1  package sac.game;
2
3  import java.util.List;
4
5  public class MinMax extends GameSearchAlgorithm {
6
7      public MinMax(GameState initial, GameSearchConfigurator configurator) {
8          super(initial, configurator);
9      }
10
11     public MinMax(GameState initial) {
12         super(initial, null);
```

```
13      }
14
15      public MinMax() {
16          super(null, null);
17      }
18
19      @Override
20      public Double doEvaluateMaxState(GameState gameState, double alpha, double beta, double depth
            , double depthLimit) {
21          if (isGameStateTerminal(gameState, depth, depthLimit)) {
22              if (configurator.isTranspositionTableOn())
23                  transpositionTable.putOrUpdate(gameState, gameState.getH(), alpha, beta);
24              return gameState.getH();
25          }
26          List<GameState> children = generateChildrenWrapper(gameState);
27          double value = Double.NEGATIVE_INFINITY;
28          for (GameState child : children) {
29              Double childValue = null;
30              if (configurator.isTranspositionTableOn())
31                  childValue = transpositionTable.get(child, alpha, beta);
32              if (childValue == null) {
33                  childValue = (child.isMaximizingTurnNow()) ? evaluateMaxState(child, alpha, beta,
                        depth + 0.5, depthLimit) : evaluateMinState(child, alpha,
34                          beta, depth + 0.5, depthLimit);
35                  if (childValue == null)
36                      return null;
37                  if (configurator.isTranspositionTableOn())
38                      transpositionTable.putOrUpdate(child, childValue, alpha, beta);
39              }
40              if (childValue > value) {
41                  value = childValue;
42                  updateMovesAlongPrincipalVariation(gameState, child);
43              }
44              if (depth == 0.0)
45                  movesScores.put(child.getMoveName(), childValue);
46          }
47
48          return value;
49      }
50
51      @Override
52      public Double doEvaluateMinState(GameState gameState, double alpha, double beta, double depth
            , double depthLimit) {
53          if (isGameStateTerminal(gameState, depth, depthLimit)) {
54              if (configurator.isTranspositionTableOn())
55                  transpositionTable.putOrUpdate(gameState, gameState.getH(), alpha, beta);
56              return gameState.getH();
57          }
58          List<GameState> children = generateChildrenWrapper(gameState);
59          double value = Double.POSITIVE_INFINITY;
60          for (GameState child : children) {
61              Double childValue = null;
62              if (configurator.isTranspositionTableOn())
63                  childValue = transpositionTable.get(child, alpha, beta);
64              if (childValue == null) {
65                  childValue = (child.isMaximizingTurnNow()) ? evaluateMaxState(child, alpha, beta,
                        depth + 0.5, depthLimit) : evaluateMinState(child, alpha,
66                          beta, depth + 0.5, depthLimit);
67                  if (childValue == null)
68                      return null;
69                  if (configurator.isTranspositionTableOn())
```

```
70                      transpositionTable.putOrUpdate(child, childValue, alpha, beta);
71              }
72              if (childValue < value) {
73                  value = childValue;
74                  updateMovesAlongPrincipalVariation(gameState, child);
75              }
76              if (depth == 0.0)
77                  movesScores.put(child.getMoveName(), childValue);
78          }
79
80          return value;
81      }
82  }
```

Apart from constructors there are only two methods — the implementations of `doEvaluateMaxState(...)` and `doEvaluateMinState(...)`, that were undefined (abstract) in the general `GameSearchAlgorithm` class. As one can see these implementations correspond well to the algorithmic pseudocode 8 of Min-Max presented before. A notable deviation can be seen in the way the subrecursions are being called, namely, the `doEvaluateMaxState(...)` does not necessarily have to call the 'opposite' wrapper `evaluateMinState(...)` but may again call a wrapper to itself — `evaluateMaxState(...)` (and vice-versa). It all depends on whose turn it is to play and this condition is checked on every child state — `child.isMaximizingTurnNow()`. What is the point of this generalization? When can players deviate from moves being made interchangeably all the time? A good example, to answer that, are card games. Think of any card game where a player is allowed either to take a trick e.g. by playing a high card or to duck it (refuse it) by playing a small card, whatever he thinks is better for him. If he decides to take the trick then it is (commonly) his turn again to play on. This means that a sequence of two max operations (or two min operations) occurs. The algorithmic pseudocode 8 does not include such a scenario just for the sake of notational conciseness. Therefore, it is worth to remember that it is programmer's responsibility to suitably change the turn flags when the children states are being generated — inside the `generateChildren(...)` implementation.

A certain extra in the presented code is also the usage of the transposition table. The usage is conditional, i.e. it takes place only when the suitable configuration option is on, and accounts for two cases: (1) a game value for a child already exists in the transposition table, so it can be read from there without executing the search recursion, (2) a game value for a child was just calculated and should be stored or updated in the transposition table. More technical details about the transposition table are described in the subsequent section 4.2.5. It can also be noticed that `alpha` and `beta` variables, passed as arguments, are never used. These variables are foreseen for the more advanced algorithms, which we now move to.

Let us have a look at the `AlphaBetaPruning` class.

```
1   package sac.game;
2
3   import java.util.List;
4
5   public class AlphaBetaPruning extends GameSearchAlgorithm {
6
7       public AlphaBetaPruning(GameState initial, GameSearchConfigurator configurator) {
8           super(initial, configurator);
```

```
9       }
10
11      public AlphaBetaPruning(GameState initial) {
12          super(initial, null);
13      }
14
15      public AlphaBetaPruning() {
16          super(null, null);
17      }
18
19      @Override
20      public Double doEvaluateMaxState(GameState gameState, double alpha, double beta, double depth
            , double depthLimit) {
21          if (isGameStateTerminal(gameState, depth, depthLimit)) {
22              if (configurator.isTranspositionTableOn())
23                  transpositionTable.putOrUpdate(gameState, gameState.getH(), alpha, beta);
24              return gameState.getH();
25          }
26          List<GameState> children = generateChildrenWrapper(gameState);
27          if (configurator.isRefutationTableOn())
28              refutationTable.reorder(gameState, children);
29          for (GameState child : children) {
30              Double childValue = null;
31              if (configurator.isTranspositionTableOn())
32                  childValue = transpositionTable.get(child, alpha, beta);
33              if (childValue == null) {
34                  childValue = (child.isMaximizingTurnNow()) ? evaluateMaxState(child, alpha, beta,
                        depth + 0.5, depthLimit) : evaluateMinState(child, alpha,
35                          beta, depth + 0.5, depthLimit);
36                  if (childValue == null)
37                      return null;
38                  if (configurator.isTranspositionTableOn())
39                      transpositionTable.putOrUpdate(child, childValue, alpha, beta);
40              }
41              if ((depth == 0.0) && (isExactGameValue(childValue, alpha, beta)))
42                  movesScores.put(child.getMoveName(), childValue);
43              if (childValue > alpha) {
44                  alpha = childValue;
45                  updateMovesAlongPrincipalVariation(gameState, child);
46                  if (configurator.isRefutationTableOn())
47                      refutationTable.put(gameState, child);
48              }
49              if (alpha >= beta)
50                  return alpha;
51          }
52
53          return alpha;
54      }
55
56      @Override
57      public Double doEvaluateMinState(GameState gameState, double alpha, double beta, double depth
            , double depthLimit) {
58          if (isGameStateTerminal(gameState, depth, depthLimit)) {
59              if (configurator.isTranspositionTableOn())
60                  transpositionTable.putOrUpdate(gameState, gameState.getH(), alpha, beta);
61              return gameState.getH();
62          }
63          List<GameState> children = generateChildrenWrapper(gameState);
64          if (configurator.isRefutationTableOn())
65              refutationTable.reorder(gameState, children);
66          for (GameState child : children) {
```

```
67              Double childValue = null;
68              if (configurator.isTranspositionTableOn())
69                  childValue = transpositionTable.get(child, alpha, beta);
70              if (childValue == null) {
71                  childValue = (child.isMaximizingTurnNow()) ? evaluateMaxState(child, alpha, beta,
                        depth + 0.5, depthLimit) : evaluateMinState(child, alpha,
72                          beta, depth + 0.5, depthLimit);
73                  if (childValue == null)
74                      return null;
75                  if (configurator.isTranspositionTableOn())
76                      transpositionTable.putOrUpdate(child, childValue, alpha, beta);
77              }
78              if ((depth == 0.0) && (isExactGameValue(childValue, alpha, beta)))
79                  movesScores.put(child.getMoveName(), childValue);
80              if (childValue < beta) {
81                  beta = childValue;
82                  updateMovesAlongPrincipalVariation(gameState, child);
83                  if (configurator.isRefutationTableOn())
84                      refutationTable.put(gameState, child);
85              }
86              if (alpha >= beta)
87                  return beta;
88          }
89
90          return beta;
91      }
92  }
```

Again, the code contains the same elements — three constructors and the implementations of `doEvaluateMaxState(...)` and `doEvaluateMinState(...)` methods. This time, the `alpha` and `beta` variables are in use. They are updated if an evaluation returned by a child state leads to an improvement of the pay off for given player. In particular, the last 'if' statement — `if (alpha >= beta) {...}` — leads to a cut off of a subtree, in compliance with the algorithm 9. One more new element is a possible usage of the so called refutation table. More details about it are given in section 4.2.5. Hereby, it suffices to say that the refutation table may potentially reorder the children states — `refutationTable.reorder(...)` — in such a way that the most promising child is considered as the first one in the loop, and by that it may lead to an early cut off. On the other hand, each time an actual cut off is met (`alpha >= beta`) the information about the parent–child pair causing that cut off is stored in the refutation table for a plausible usage in the future. It is also worth to explain that an application of the refutation table was not possible in the previous `MinMax` class because the MinMax algorithm always analyzes all the children states, it cannot induce a cut off.

Let us now move to the `Scout` class — representing the most advanced algorithm for game trees within *SaC*.

```
1   package sac.game;
2
3   import java.util.List;
4
5   public class Scout extends GameSearchAlgorithm {
6
7       public Scout(GameState initial, GameSearchConfigurator configurator) {
8           super(initial, configurator);
```

```
9       }
10
11      public Scout(GameState initial) {
12          super(initial, null);
13      }
14
15      public Scout() {
16          super(null, null);
17      }
18
19      @Override
20      public Double doEvaluateMaxState(GameState gameState, double alpha, double beta, double depth
            , double depthLimit) {
21          if (isGameStateTerminal(gameState, depth, depthLimit)) {
22              if (configurator.isTranspositionTableOn())
23                  transpositionTable.putOrUpdate(gameState, gameState.getH(), alpha, beta);
24              return gameState.getH();
25          }
26          List<GameState> children = generateChildrenWrapper(gameState);
27          if (configurator.isRefutationTableOn())
28              refutationTable.reorder(gameState, children);
29          double b = beta;
30          for (int i = 0; i < children.size(); i++) {
31              GameState child = children.get(i);
32              Double childValue = null;
33              boolean researchNeeded = false;
34              double bound = alpha;
35              if (configurator.isTranspositionTableOn())
36                  childValue = transpositionTable.get(child, alpha, b);
37              if (childValue == null) {
38                  if (child.isMaximizingTurnNow()) {
39                      // scout search with zero window
40                      childValue = evaluateMaxState(child, alpha, b, depth + 0.5, depthLimit);
41                      if (childValue == null)
42                          return null; // time limit reached
43                      // checking if window fails high, if so, research with broader window
44                      if ((i > 0) && (b <= childValue) && (childValue < beta) && ((configurator.
                            isQuiescenceOn()) || (depthLimit - depth > 0.5))) {
45                          researchNeeded = true;
46                          bound = childValue;
47                          childValue = evaluateMaxState(child, bound, beta, depth + 0.5, depthLimit
                                );
48                          if (childValue == null)
49                              return null; // time limit reached
50                      }
51                  } else {
52                      // scout search with zero window
53                      childValue = evaluateMinState(child, alpha, b, depth + 0.5, depthLimit);
54                      if (childValue == null)
55                          return null; // time limit reached
56                      // checking if window fails high, if so, research with broader window
57                      if ((i > 0) && (b <= childValue) && (childValue < beta) && ((configurator.
                            isQuiescenceOn()) || (depthLimit - depth > 0.5))) {
58                          researchNeeded = true;
59                          bound = childValue;
60                          childValue = evaluateMinState(child, bound, beta, depth + 0.5, depthLimit
                                );
61                          if (childValue == null)
62                              return null; // time limit reached
63                      }
64                  }
```

```
65              }
66              if (configurator.isTranspositionTableOn()) {
67                  if (!researchNeeded)
68                      transpositionTable.putOrUpdate(child, childValue, alpha, b);
69                  else
70                      transpositionTable.putOrUpdate(child, childValue, bound, beta);
71              }
72              if ((depth == 0.0) && (isExactGameValue(childValue, alpha, beta)))
73                  movesScores.put(child.getMoveName(), childValue);
74              if (childValue > alpha) {
75                  alpha = childValue;
76                  updateMovesAlongPrincipalVariation(gameState, child);
77                  if (configurator.isRefutationTableOn())
78                      refutationTable.put(gameState, child);
79              }
80              if (alpha >= beta)
81                  return alpha;
82              if (Math.abs(alpha) < GameState.H_SMALLEST_INFINITY)
83                  b = alpha + 1.0;
84          }
85
86          return alpha;
87      }
88
89      @Override
90      public Double doEvaluateMinState(GameState gameState, double alpha, double beta, double depth
            , double depthLimit) {
91          if (isGameStateTerminal(gameState, depth, depthLimit)) {
92              if (configurator.isTranspositionTableOn())
93                  transpositionTable.putOrUpdate(gameState, gameState.getH(), alpha, beta);
94              return gameState.getH();
95          }
96          List<GameState> children = generateChildrenWrapper(gameState);
97          if (configurator.isRefutationTableOn())
98              refutationTable.reorder(gameState, children);
99          double a = alpha;
100         for (int i = 0; i < children.size(); i++) {
101             GameState child = children.get(i);
102             Double childValue = null;
103             boolean researchNeeded = false;
104             double bound = beta;
105             if (configurator.isTranspositionTableOn())
106                 childValue = transpositionTable.get(child, a, beta);
107             if (childValue == null) {
108                 if (child.isMaximizingTurnNow()) {
109                     // scout search with zero window
110                     childValue = evaluateMaxState(child, a, beta, depth + 0.5, depthLimit);
111                     if (childValue == null)
112                         return null; // time limit reached
113                     // checking if window fails low, if so, research with broader window
114                     if ((i > 0) && (childValue <= a) && (alpha < childValue) && ((configurator.
                            isQuiescenceOn()) || (depthLimit - depth > 0.5))) {
115                         researchNeeded = true;
116                         bound = childValue;
117                         childValue = evaluateMaxState(child, alpha, bound, depth + 0.5,
                                depthLimit);
118                         if (childValue == null)
119                             return null; // time limit reached
120                     }
121                 } else {
122                     // scout search with zero window
```

```
123                     childValue = evaluateMinState(child, a, beta, depth + 0.5, depthLimit);
124                     if (childValue == null)
125                         return null; // time limit reached
126                     // checking if window fails low, if so, research with broader window
127                     if ((i > 0) && (childValue <= a) && (alpha < childValue) && ((configurator.
                            isQuiescenceOn()) || (depthLimit - depth > 0.5))) {
128                         researchNeeded = true;
129                         bound = childValue;
130                         childValue = evaluateMinState(child, alpha, bound, depth + 0.5,
                                depthLimit);
131                         if (childValue == null)
132                             return null; // time limit reached
133                     }
134                 }
135             }
136             if (configurator.isTranspositionTableOn()) {
137                 if (!researchNeeded)
138                     transpositionTable.putOrUpdate(child, childValue, a, beta);
139                 else
140                     transpositionTable.putOrUpdate(child, childValue, alpha, bound);
141             }
142             if ((depth == 0.0) && (isExactGameValue(childValue, alpha, beta)))
143                 movesScores.put(child.getMoveName(), childValue);
144             if (childValue < beta) {
145                 beta = childValue;
146                 updateMovesAlongPrincipalVariation(gameState, child);
147                 if (configurator.isRefutationTableOn())
148                     refutationTable.put(gameState, child);
149             }
150             if (alpha >= beta)
151                 return beta;
152             if (Math.abs(beta) < GameState.H_SMALLEST_INFINITY)
153                 a = beta - 1.0;
154         }
155 
156         return beta;
157     }
158 }
```

The code corresponds well to the algorithm 10, yet, it is quite space consuming. Some overhead is related to the checks if a repeated search has to be done due to the zero window failing high or low. Also, a slightly different information is being put to the transposition table depending on wether the evaluation came from the first search or the second search.

4.2.5 Transposition table

In the context of games, the name *transposition* historically comes from chess and denotes the possibility of reaching the same position (the same state) by different sequences of moves. If the in-depth recurrence for such a position has already been performed before, then one can save time by taking a ready-made result provided that it is memorized somewhere — in a transposition table. For example, note that in chess the state reached after the initial sequence e2-e4, e7-e5, Ng1-f3, Ng8-f6 has 4! = 24 transpositions, accounting for all possible moves permutations.

For fast performance, typically a transposition table is implemented as a hash map, so that one can look up its entries in constant time — $O(1)$. Yet, it is possible to implement transposition

tables as binary search trees, less memory-consuming but leading to the logarithmic look-up time — $O(\log_2 n)$, n being the number of stored entries.

The keys to a transposition table can be state identifiers or their hash codes. For example in chess, a key should take into account the positions of remaining pieces plus some additional information about: castling possibilities, en passant captures, repetitions of moves. Sometimes, apart from the main transposition table, one can use additional transposition tables as libraries of openings or endgames.

In *SaC*, the overall implementation of the transpostion table mechanisms is constituted by the following five elements:

1. the `TranspositionTableKey` class — it uses the (identifier, depth) pair as the key to represent a state stored in the transpostion table,
2. the `TranspositionTableEntry` class — it represents an entry in the table associated to some key; the entry stores three numbers related to the state evaluation: its lower bound, its exact value, its upper bound (any of them can also be a `null` value),
3. the `TranspositionTable` interface — it defines a general set of methods that all transposition table containers should provide,
4. the `TranspostitionTableImpl` class being the default (but still abstract) implementation of the above interface,
5. the `TranspositionTableAsHashMap` or `TranspositionTableAsTreeMap` class — the ready to use extensions of the default implementation, with the actual transposition table stored as a particular data structure: a hash map or a tree map (red-black tree), respectively.

Provided that the transposition table is turned on, there are two main situations when search algorithms interact with it. The first situation is when an algorithm iterates over the generated descendants, and it checks for each of them if there is a ready evaluation stored in the transposition table — this is done via the `get(...)` method. If so, the algorithm uses such an evaluation and does not trigger the searching recurrence downward the tree (from that descendant on). The second situation is when an algorithm reaches a terminal state or finishes processing a descendant state, then, it is the right time to put the calculated evaluation into the transposition table — this is done via the `putOrUpdate(...)` method. Both mentioned situations can be observed in the listings of `AlphaBetaPruning` and `Scout` classes presented back in section 4.1. Below, we show a code excerpt from the `TranspositionTableImpl` class that demonstrates the mentioned `get(...)` and `putOrUpdate(...)` methods and we then comment on it.

```
1  package sac.game;
2
3  import java.util.AbstractMap;
4
5  public abstract class TranspositionTableImpl implements TranspositionTable {
6
7      protected AbstractMap<TranspositionTableKey, TranspositionTableEntry> map;
8      protected int usesCount = 0;
```

```
9
10      ...
11
12      @Override
13      public Double get(GameState gameState, double alpha, double beta) {
14          Double valueOrBoundOrNull = doGet(gameState, alpha, beta);
15          if (valueOrBoundOrNull != null) gameState.setReadFromTranspositionTable(true);
16          return valueOrBoundOrNull;
17      }
18
19      protected Double doGet(GameState gameState, double alpha, double beta) {
20          TranspositionTableEntry entry = map.get(new TranspositionTableKey(gameState));
21          if (entry == null)
22              return null;
23          if (entry.getExactGameValue() != null) {
24              usesCount++;
25              return entry.getExactGameValue();
26          } else {
27              if ((entry.getUpperBoundOnGameValue() != null) && (entry.getUpperBoundOnGameValue() <
                    = alpha)) {
28                  usesCount++;
29                  return entry.getUpperBoundOnGameValue();
30              } else if ((entry.getLowerBoundOnGameValue() != null) && (beta <= entry.
                    getLowerBoundOnGameValue())) {
31                  usesCount++;
32                  return entry.getLowerBoundOnGameValue();
33              }
34          }
35
36          return null;
37      }
38
39      @Override
40      public TranspositionTableEntry get(GameState gameState) {
41          return doGet(gameState);
42      }
43
44      protected TranspositionTableEntry doGet(GameState gameState) {
45          return map.get(new TranspositionTableKey(gameState));
46      }
47
48      @Override
49      public void putOrUpdate(GameState gameState, Double value, double alpha, double beta) {
50          TranspositionTableKey key = new TranspositionTableKey(gameState);
51          TranspositionTableEntry entry = map.get(key);
52          if (entry == null) {
53              // put
54              if (GameSearchAlgorithm.isExactGameValue(value, alpha, beta))
55                  entry = new TranspositionTableEntry(null, new Double(value), null);
56              else {
57                  if (value <= alpha) // alpha-beta window fails low - value is an upper bound
58                      entry = new TranspositionTableEntry(null, null, new Double(value));
59                  else
60                      // alpha-beta window fails high - value is a lower bound
61                      entry = new TranspositionTableEntry(new Double(value), null, null);
62              }
63              map.put(new TranspositionTableKey(gameState), entry);
64          } else {
65              // update
66              if (GameSearchAlgorithm.isExactGameValue(value, alpha, beta)) {
67                  entry.setExactGameValue(value);
```

```
68                  entry.setLowerBoundOnGameValue(null);
69                  entry.setUpperBoundOnGameValue(null);
70              } else {
71                  if ((value <= alpha) && ((entry.getUpperBoundOnGameValue() == null) || (value <
                        entry.getUpperBoundOnGameValue()))) {
72                      entry.setUpperBoundOnGameValue(value); // tighter upper bound
73                      if ((entry.getLowerBoundOnGameValue() != null) && (entry.
                            getUpperBoundOnGameValue() != null)
74                              && (entry.getLowerBoundOnGameValue().doubleValue() == entry.
                                    getUpperBoundOnGameValue().doubleValue())) {
75                          entry.setExactGameValue(new Double(entry.getLowerBoundOnGameValue().
                                doubleValue()));
76                          entry.setLowerBoundOnGameValue(null);
77                          entry.setUpperBoundOnGameValue(null);
78                      }
79                  } else if ((beta <= value) && ((entry.getLowerBoundOnGameValue() == null) || (
                        entry.getLowerBoundOnGameValue() < value))) {
80                      entry.setLowerBoundOnGameValue(value); // tighter lower bound
81                      if ((entry.getLowerBoundOnGameValue() != null) && (entry.
                            getUpperBoundOnGameValue() != null)
82                              && (entry.getLowerBoundOnGameValue().doubleValue() == entry.
                                    getUpperBoundOnGameValue().doubleValue())) {
83                          entry.setExactGameValue(new Double(entry.getLowerBoundOnGameValue().
                                doubleValue()));
84                          entry.setLowerBoundOnGameValue(null);
85                          entry.setUpperBoundOnGameValue(null);
86                      }
87                  }
88              }
89          }
90      }
91
92      ...
93  }
```

The presented excerpts take into account the Knuth-Moore theorem, cited in section 4.1.3, and operate accordingly.

Consider first the `get(GameState gameState, double alpha, double beta)` method. It starts by lookin up a transposition table entry for given state and if such an entry exists it checks whether the entry contains an exact game value or a bound. The exact value is returned unconditionally. A bound is returned only if it is useful. More precisely, an upper bound can be taken advantage of if the current α is greater or equal to the bound, because it means that the maximizing player cannot improve his payoff by following the given state. Conversely, a lower bound can be taken advantage of if the current β is lower or equal to the bound, because it means that the minimizing player cannot improve his payoff by following the given state. If no entry or a useful bound exists in the table, the `null` value is returned. We should remark that apart from the mentioned getter to which the current α-β window is passed on, there also exist another getter — `get(GameState gameState)` — in the `TranspositionTable` interface, which is supposed to return the whole entry (of `TranspositionTableEntry` type) that is the triplet of values: a lower bound, an exact value, an upper bound. Yet, this is only an auxiliary getter, and all game searching algorithms in *SaC* use the former variant, always passing on the current α and β values.

Consider now the `putOrUpdate(...)` method. In the case of 'put' (when no entry exists so far), the method creates an entry and suitably places the given evaluation as an exact value or a bound

depending on its relation with respect to the α-β window. The 'update' case is more expanded. When the new value can be immediately recognized as an exact value, because it falls strictly in between α and β, then it is simply stored that way (lines 67–70). Otherwise, the code tries to make an update of the suitable bound, but only in the case the new bound is tighter than the previous one (lines 71-89). In a certain peculiar case, it is possible that after a bound update is made, both upper and lower bounds meet (become equal) — then, the entry object is rearranged to represent the exact value from now on.

Please note that the default `TranspositionTableImpl` implementation is an abstract class as it does not specify the data structure underlying the transposition table:

```
1 ...
2 protected AbstractMap<TranspositionTableKey, TranspositionTableEntry> map;
3 ...
```

The aforementioned extensions existing in *SaC* — `TranspositionTableAsHashMap` and `TranspositionTableAsTreeMap` — are very light classes, consisting solely of constructors that assign a suitable data structure to the `map` object, as shown below.

```
1 package sac.game;
2
3 import java.util.HashMap;
4
5 public class TranspositionTableAsHashMap extends TranspositionTableImpl {
6     public TranspositionTableAsHashMap() {
7         this.map = new HashMap<TranspositionTableKey, TranspositionTableEntry>(512 * 1024, (float
              ) 0.75);
8     }
9 }
```

```
1 package sac.game;
2
3 import java.util.TreeMap;
4
5 public class TranspositionTableAsTreeMap extends TranspositionTableImpl {
6     public TranspositionTableAsTreeMap() {
7         this.map = new TreeMap<TranspositionTableKey, TranspositionTableEntry>();
8     }
9 }
```

4.2.6 Refutation table

Refutation table is a valuable addition in the setting of *progressive search* (a.k.a. *iterative deepening*) i.e. when multiple searches are executed successively for new positions arising along an ongoing game. It is a typical scenario under a main game loop. Then, it is possible that some information from the previous searches can be used with a benefit in a new search. More precisely, the purpose of a refutation table is to memorize the information about best continuations at shallow depths of the tree. By best continuations meant are such moves — (parent, child) pairs — that led to the best

payoff for given parent or even caused a cut-off in the previous search iteration (*refutation moves* or *cutoff moves*). This information can be taken advantage of in the future, i.e. it can be used to reorder children and try best continuations as first in the next iteration hoping to obtain the cut-offs sooner. The scenario described above is often referred to as the *principal variation search* (PSV), although the notion of 'principal variation' itself has a slightly different meaning than the notions of 'refutation' or 'refutation table'[3]. Yet, the two are conceptually very close together (Marsland, 1983).

How large are refutation tables? Obviously, it depends on the game branching factor and on how deep do we want to track the refutation information. For a moment consider for simplicity a game with alternate moves and a constant branching factor b. After both players make their moves, the root for the next analysis shifts somewhere two plies down the tree to one of b^2 possible states. Each of those states has some best descendant underneath (possibly a cut-off causer). Therefore, there are at most b^2 entries be kept in the refutation table for that particular level. If the programmer wants to track things (refutations) deeper, then the following quantities of new entries occur (at most) for successive levels: b^2, b^3, b^4, ... By default, *SaC* stores the refutation information for 4 plies. It is also worth to add that for typical games there are much more entries in a transposition table than in a refutation table.

In *SaC*, a refutation table keeps two collections inside: a 'save collection' meant for saving entries usable by the next search iteration, and a 'read collection' meant for reading entries from the previous iteration. Once the current iteration is completed, a reference to the 'read collection' is replaced by the 'save collection', whereas the reference to the 'save collection' is emptified (becomes a `null`). These operations are performed in the `reset()` method (provided by *SaC*'s API) and a particular search algorithm is supposed to call this method in-between the iterations.

The overall *SaC*'s implementation of refutation tables is represented by the following three elements:

1. the `RefutationTable` interface — it defines a general set of methods that all refutation table containers should provide,

2. the `RefutationTableImpl` class being the default (but still abstract) implementation of the above interface,

3. the `RefutationTableAsHashMap` or `RefutationTableAsTreeMap` class — the ready to use extensions of the default implementation, with the actual refutation table stored as a particular data structure: a hash map or a tree map (red-black tree), respectively.

Similarly as it was in the case for transposition tables, the realizations of a refutation table as a hashmap or a red-black tree leads to constant or logarithmic look-up times respectively.

Provided that the refutation table is turned on, there are two main situations when search algorithms interact with it. The first is when an algorithm tries to reorder the descendants — this is done via the `reorder(...)` method being called immediately after the descendants were generated. If no suitable entry is found the refutation table (or there is just one descendant) then no reordering takes place. The second situation is when an algorithm comes across a payoff

[3] The 'principal variation' represents a complete sequence of best moves starting from the given root up to a given maximum depth that leads to the minimax value of the game.

improvement. Then, the search algorithm should register or update an entry in the refutation table — this is done vie the `put(...)` method. Both mentioned situations can be observed in the listings of `AlphaBetaPruning` and `Scout` classes presented back in section 4.1. Below, shown are the most important excerpts from the default `RefutationTableImpl` class together with short full listings of `RefutationTableAsHashMap` or `RefutationTableAsTreeMap` classes.

```
1  package sac.game;
2
3  import java.util.AbstractMap;
4  import java.util.List;
5
6  import sac.Identifier;
7
8  public abstract class RefutationTableImpl implements RefutationTable {
9
10     protected final static double DEFAULT_DEPTH_LIMIT = 2.0;
11     protected double depthLimit;
12     protected int usesCount = 0;
13
14     protected AbstractMap<Identifier, Identifier> tableToSave = null;
15     protected AbstractMap<Identifier, Identifier> tableToRead = null;
16
17     ...
18
19
20     @Override
21     public void put(GameState parent, GameState child) {
22         double depth = parent.getDepth() - 0.5; // in progressive search, we assume that next
23                                                 // iteration (possibly reading from refutation
24                                                 // table) will start from level +0.5, so moves at
25                                                 // level 0.0 are not memorized in refutation
                                                        table
26         if ((depth >= 0) && (depth <= depthLimit))
27             tableToSave.put(parent.getIdentifier(), child.getIdentifier());
28     }
29
30     @Override
31     public void reorder(GameState parent, List<GameState> children) {
32         if ((children == null) || (children.size() <= 1))
33             return; // no reorder done
34         double depth = parent.getDepth();
35         if (depth > depthLimit)
36             return; // no reorder done
37         Identifier bestChildIdentifier = tableToRead.get(parent.getIdentifier());
38         if (bestChildIdentifier == null)
39             return; // no reorder done
40         GameState bestChild = null;
41         int i = 0;
42         for (GameState child : children) {
43             if (child.getIdentifier().equals(bestChildIdentifier)) {
44                 bestChild = child;
45                 break;
46             }
47             i++;
48         }
49         if (i == 0)
50             return; // no reorder done
51         if (bestChild != null) {
```

```
52              children.remove(bestChild);
53              children.add(0, bestChild); // putting best child in front
54              usesCount++;
55          }
56      }
57
58      @Override
59      public int size() {
60          return tableToSave.size() + tableToRead.size();
61      }
62
63      ...
64  }
```

```
1   package sac.game;
2
3   import java.util.HashMap;
4
5   import sac.Identifier;
6
7   public class RefutationTableAsHashMap extends RefutationTableImpl {
8
9       public RefutationTableAsHashMap() {
10          super();
11          tableToSave = new HashMap<Identifier, Identifier>();
12          tableToRead = new HashMap<Identifier, Identifier>();
13      }
14
15      public RefutationTableAsHashMap(double depthLimit) {
16          super(depthLimit);
17          tableToSave = new HashMap<Identifier, Identifier>();
18          tableToRead = new HashMap<Identifier, Identifier>();
19      }
20
21      @Override
22      public void reset() {
23          tableToRead = tableToSave;
24          tableToSave = new HashMap<Identifier, Identifier>();
25      }
26  }
```

```
1   package sac.game;
2
3   import java.util.TreeMap;
4
5   import sac.Identifier;
6
7   public class RefutationTableAsTreeMap extends RefutationTableImpl {
8
9       public RefutationTableAsTreeMap() {
10          super();
11          tableToSave = new TreeMap<Identifier, Identifier>();
12          tableToRead = new TreeMap<Identifier, Identifier>();
13      }
14
15      public RefutationTableAsTreeMap(double depthLimit) {
16          super(depthLimit);
```

```
17          tableToSave = new TreeMap<Identifier, Identifier>();
18          tableToRead = new TreeMap<Identifier, Identifier>();
19      }
20
21      @Override
22      public void reset() {
23          tableToRead = tableToSave;
24          tableToSave = new TreeMap<Identifier, Identifier>();
25      }
26  }
```

4.2.7 Configuration options for searching games

On several occasions we have mentioned the usage of a `GameSearchConfigurator` object. Below, we present a brief code listing of this class, with all configuration options and their default values. We purposely leave the javadocs present in the listing to make the meaning of options more clear. Every configuration option can be accessed by a suitable getter or setter (that are skipped in the listing).

```
1  public class GameSearchConfigurator {
2
3      /**
4       * Identifier type for states. By default: STRING.
5       */
6      private IdentifierType identifierType = IdentifierType.HASH_CODE;
7
8      /**
9       * Depth limit expressed in full moves (not plies). By default: 3.5 (i.e. 7 plies).
10      */
11     private double depthLimit = 3.5;
12
13     /**
14      * Is transposition table on. By default: true.
15      */
16     private boolean transpositionTableOn = true;
17
18     /**
19      * Class name for transposition table. By default: sac.game.TranspositionTableAsHashMap.
20      */
21     private String transpositionTableClassName = "sac.game.TranspositionTableAsHashMap";
22
23     /**
24      * Is quiescence on. By default: true.
25      */
26     private boolean quiescenceOn = true;
27
28     /**
29      * Is refutation table on. By default: true.
30      */
31     private boolean refutationTableOn = true;
32
33     /**
34      * Class name for refutation table. By default: sac.game.RefutationTableAsHashMap.
35      */
36     private String refutationTableClassName = "sac.game.RefutationTableAsHashMap";
37
38     /**
39      * Depth limit for refutation table. By default: 2.0 (RefutationTableImpl.DEFAULT_DEPTH_LIMIT
            ).
40      */
41     private double refutationTableDepthLimit = RefutationTableImpl.DEFAULT_DEPTH_LIMIT;
42
43     /**
44      * Do parents memorize references to their children. Set to false for lower memory usage (
            WARNING: in that case
45      * drawing game tree via GraphViz is impossible). By default: false.
46      */
47     private boolean parentsMemorizingChildren = false;
48
```

```
49      /**
50       * Time limit in milliseconds. By default: 'infinity' in long type (Long.MAX_VALUE).
51       */
52      private long timeLimit = Long.MAX_VALUE;
53
54      public GameSearchConfigurator() {
55      }
56
57      public GameSearchConfigurator(String propertiesFilePath) throws Exception {
58          ...
59      }
60
61      ... // getters, setters
62  }
```

A configurator can be instatiated either by a default constructor with no arguments — `GameSearchConfigurator()`, or from a properties text file — `GameSearchConfigurator(String propertiesFilePath)`. The names of options in the properties file are uppercased versions of field names, and with underscores separating successive words. Below, we show a possible exemplary contents of a configuration `.properties` file with some non-default values.

```
1  #Example of game configurator settings
2
3  identifierType=STRING
4  depthLimit=3.5
5  transpositionTableOn=true
6  transpositionTableClassName=sac.game.TranspositionTableAsHashMap
7  quiescenceOn=true
8  refutationTableOn=true
9  refutationTableClassName=sac.game.RefutationTableAsHashMap
10 refutationTableDepthLimit=2.0
11 parentsMemorizingChildren=false
12 timeLimit=Long.MAX_VALUE
```

4.3 Examples

4.3.1 Checkers

The game of checkers as such does not require an introduction. As regards the computer programs playing checkers, one of the historically important ones was a program written by Samuel in early 1950s, see (Samuel, 1983). The program was capable of self training, equipped with a weighted evaluation function. The weights (parameters) of that function were exposed to updates and improvements as two instances of the program were playing many games against each other. The program evolved this way was able to beat quite advanced players, but not the best ones of that time.

The exemplary implementation of checkers under *SaC* is not meant to introduce a strong program, but simply to demonstrate the API guidelines needed to be fulfilled by a programmer to implement similar mind games using *SaC*. We start by showing the code excerpts from the `sac.examples.checkers.Checkers` class representing a checkers state (for full source code the reader is addressed to the library), we then show several experiments and illustrations related to checkers prepared using *SaC*.

It is worth to remark that the *SaC* distribution is equipped with a simple graphical interface designed for experimenting with checkers and with different search algorithms — the program is represented by the `sac.examples.checkers.CheckersGame` class and a screenshot of it is shown in Fig. 4.1. It allows for a 'human vs computer' play or 'computer vs computer', using different algorithms and configurations. Also, it logs many informations to the screen. The program can be triggered for example by the following line:

```
java -Xmx2048M -cp "sac-1.0.1.jar;jfreechart-1.0.14.jar;jcommon-1.0.17.jar;swt.jar" sac.examples.checkers.CheckersGame
```

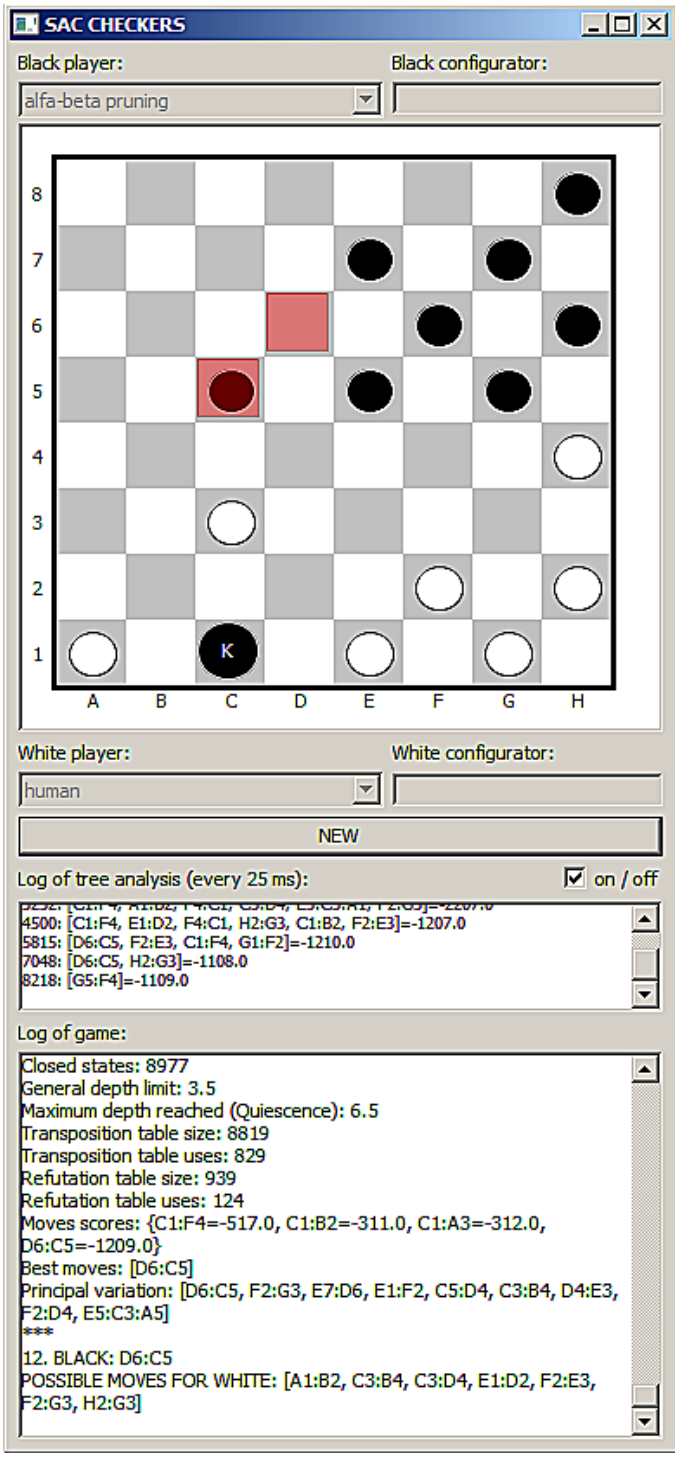


Figure 4.1: A screenshot from the examplary checkers program distributed with *SaC* — `sac.examples.checkers.CheckersGame`.

Implementation of checkers state

Our *SaC* representation of a checkers state — the `sac.examples.checkers.Checkers` class — stores four lists of pieces locations accounting for: white pawns, white kings, black pawns, and black kings. The locations themselves are encapsulated in the `BoardLocation` class, representing a pair of two-dimensional coordinates.[4] Besides, the state is equipped with necessary *SaC* routines for: (1) identification of states — by either the `toString()` or the `hashCode()` method, (2) generation of descandants — `generateChildren()`, and (3) heuristic evaluation of the position — appended using the `setHFunction(new StateFunction() {...})` static method. The code excerpts presented below skip large fragments containing auxiliary constants and helper methods (e.g. for copying states, exporting states as images or files, checking the presence of a piece at particular location, etc.). Also a large part, rather technical, related to the generation of moves and captures was skipped. We only marked the signatures of the suitable methods involved in this aspect, in particular: `populatePossiblePawnKills(...)`, `populatePossibleKingKills(...)`, `populatePossiblePawnMoves(...)`, `populatePossibleKingMoves(...)`. We can mention that methods for captures (kills) work recursively by calling themselves after each partial capture is discovered, in order to look for further capture possibilities (within the same move) from a given intermediate location. When a legal move or capture is discovered, its string representation (e.g. "A1:C3") is appended to a certain global list of moves passed as a parameter.

We now ask the reader to have a brief look at the code, we then give a few additional comments.

```
1  package sac.examples.checkers;
2
3  ...
4
5  public class Checkers extends GameStateImpl {
6
7      public static int n;
8
9      ...
10
11     private static final double PAWN_HEURISTIC_FACTOR = 100.0;
12     private static final double KING_HEURISTIC_FACTOR = 10.0 * PAWN_HEURISTIC_FACTOR;
13
14     private List<BoardLocation> whitePawns = null;
15     private List<BoardLocation> whiteKings = null;
16     private List<BoardLocation> blackPawns = null;
17     private List<BoardLocation> blackKings = null;
18
19     private List<String> possibleMoves = null;
20     private boolean hasSomeKillMoves = false;
21
22     public Checkers(Checkers parent) {
23         super();
24
25         if (parent == null) {
26             whitePawns = new ArrayList<BoardLocation>();
27             whiteKings = new LinkedList<BoardLocation>();
```

[4] We do not present that class hereby for brevity — it is a straighforward implementation, keeping a pair of integers to indicate a two dimensional location and providing the necessary routines to convert forth and back from chess-like notation (e.g.: A1, B4, etc.) to integer coordinates.

```
28              blackPawns = new LinkedList<BoardLocation>();
29              blackKings = new LinkedList<BoardLocation>();
30          } else {
31              whitePawns = copyLocationList(parent.whitePawns);
32              whiteKings = copyLocationList(parent.whiteKings);
33              blackPawns = copyLocationList(parent.blackPawns);
34              blackKings = copyLocationList(parent.blackKings);
35          }
36
37          setMaximizingTurnNow(parent.isMaximizingTurnNow());
38      }
39
40      ...
41
42      @Override
43      public String toString() {
44          return toShortString();
45      }
46
47      public String toShortString() {
48          StringBuilder builder = new StringBuilder("");
49          builder.append(locationsToString(whitePawns));
50          builder.append(DESCRIPTION_SEPARATOR);
51          builder.append(locationsToString(whiteKings));
52          builder.append(DESCRIPTION_SEPARATOR);
53          builder.append(locationsToString(blackPawns));
54          builder.append(DESCRIPTION_SEPARATOR);
55          builder.append(locationsToString(blackKings));
56          builder.append(DESCRIPTION_SEPARATOR);
57          builder.append(Boolean.valueOf(isWhiteTurnNow()).toString());
58          return builder.toString();
59      }
60
61      @Override
62      public int hashCode() {
63          List<Integer> locationsAsXY = new ArrayList<Integer>();
64
65          for (BoardLocation location : whitePawns) {
66              locationsAsXY.add(location.getX());
67              locationsAsXY.add(location.getY());
68          }
69          locationsAsXY.add(0); // serving as separator
70
71          for (BoardLocation location : whiteKings) {
72              locationsAsXY.add(location.getX());
73              locationsAsXY.add(location.getY());
74          }
75          locationsAsXY.add(0); // serving as separator
76
77          for (BoardLocation location : blackPawns) {
78              locationsAsXY.add(location.getX());
79              locationsAsXY.add(location.getY());
80          }
81          locationsAsXY.add(0); // serving as separator
82          for (BoardLocation location : blackKings) {
83              locationsAsXY.add(location.getX());
84              locationsAsXY.add(location.getY());
85          }
86          locationsAsXY.add(0); // serving as separator
87          locationsAsXY.add(isWhiteTurnNow() ? 1 : -1); // serving as 'whos turn' identifier
88
```

```
89          return locationsAsXY.hashCode();
90      }
91
92      ...
93
94      public BoardLocation makeMove(String moveString, boolean changeTurn) {
95          moveString = moveString.toUpperCase();
96          possibleMoves = null;
97          StringTokenizer tokenizer = new StringTokenizer(moveString, MOVE_SEPARATOR);
98
99          List<BoardLocation> pawns = null;
100         List<BoardLocation> kings = null;
101         List<BoardLocation> opponentPawns = null;
102         List<BoardLocation> opponentKings = null;
103
104         if (isWhiteTurnNow()) {
105             pawns = whitePawns;
106             kings = whiteKings;
107             opponentPawns = blackPawns;
108             opponentKings = blackKings;
109         } else {
110             pawns = blackPawns;
111             kings = blackKings;
112             opponentPawns = whitePawns;
113             opponentKings = whiteKings;
114         }
115
116         BoardLocation location = BoardLocation.stringToLocation(tokenizer.nextToken());
117         BoardLocation newLocation = null;
118
119         List<BoardLocation> piecesInPlay = null;
120         boolean isPawnInPlay = false;
121         if (pawns.contains(location)) {
122             piecesInPlay = pawns;
123             isPawnInPlay = true;
124         } else
125             piecesInPlay = kings;
126
127         while (tokenizer.hasMoreTokens()) {
128             newLocation = BoardLocation.stringToLocation(tokenizer.nextToken());
129             int dx = (newLocation.getX() - location.getX() > 0) ? 1 : -1;
130             int dy = (newLocation.getY() - location.getY() > 0) ? 1 : -1;
131             piecesInPlay.remove(location);
132             while (!location.equals(newLocation)) {
133                 location.setX(location.getX() + dx);
134                 location.setY(location.getY() + dy);
135                 if (opponentPawns.contains(location))
136                     opponentPawns.remove(location);
137                 else if (opponentKings.contains(location))
138                     opponentKings.remove(location);
139             }
140             piecesInPlay.add(newLocation);
141             Collections.sort(piecesInPlay);
142         }
143
144         // promotion check
145         if (isPawnInPlay) {
146             if ((isMaximizingTurnNow()) && (newLocation.getY() == n)) {
147                 whitePawns.remove(newLocation);
148                 whiteKings.add(newLocation);
149             } else if ((!isMaximizingTurnNow()) && (newLocation.getY() == 1)) {
```

```
150                 blackPawns.remove(newLocation);
151                 blackKings.add(newLocation);
152             }
153         }
154 
155         // ply
156         if (changeTurn) {
157             setMaximizingTurnNow(!isMaximizingTurnNow());
158         }
159 
160         refresh();
161 
162         return newLocation;
163     }
164 
165     ...
166 
167     public List<String> getPossibleMoves() {
168         if (possibleMoves != null)
169             return possibleMoves;
170 
171         possibleMoves = new ArrayList<String>();
172 
173         // king kill-moves
174         Iterator<BoardLocation> kingsIterator = (isWhiteTurnNow()) ? whiteKings.iterator() :
                blackKings.iterator();
175         while (kingsIterator.hasNext()) {
176             BoardLocation king = kingsIterator.next();
177             populatePossibleKingKills(this, "", isWhiteTurnNow(), king, possibleMoves,
                    NOTHING_FORBIDDEN);
178 
179             // comment the line beneath if free choice of kills is given (if maximal must be
                    chosen,
180             // leave the line uncommented)
181             eliminateNonMaximalKills(possibleMoves);
182         }
183 
184         // uncomment the line beneath if kings have the 'killing priority' over pawns
185         // if (!result.isEmpty()) return result;
186 
187         // pawn kill-moves
188         Iterator<BoardLocation> pawnsIterator = (isWhiteTurnNow()) ? whitePawns.iterator() :
                blackPawns.iterator();
189         while (pawnsIterator.hasNext()) {
190             BoardLocation pawn = pawnsIterator.next();
191             populatePossiblePawnKills(this, "", isWhiteTurnNow(), pawn, possibleMoves);
192 
193             // comment the line beneath if free choice of kills is given (if maximal must be
                    chosen,
194             // leave the line uncommented)
195             eliminateNonMaximalKills(possibleMoves);
196         }
197         if (!possibleMoves.isEmpty()) {
198             hasSomeKillMoves = true;
199 
200             // comment the line beneath if kills are not mandatory
201             return possibleMoves;
202         }
203 
204         // king regular moves
205         kingsIterator = (isWhiteTurnNow()) ? whiteKings.iterator() : blackKings.iterator();
```

```
206         while (kingsIterator.hasNext()) {
207             BoardLocation king = kingsIterator.next();
208             populatePossibleKingMoves(this, king, possibleMoves);
209         }
210 
211         // pawn regular moves
212         pawnsIterator = (isWhiteTurnNow()) ? whitePawns.iterator() : blackPawns.iterator();
213         while (pawnsIterator.hasNext()) {
214             BoardLocation pawn = pawnsIterator.next();
215             populatePossiblePawnMoves(this, isWhiteTurnNow(), pawn, possibleMoves);
216         }
217 
218         return possibleMoves;
219     }
220 
221     private static void populatePossiblePawnKills(Checkers checkers, String prefix, boolean
            isWhitesTurnNow, BoardLocation pawn, List<String> globalList) {
222         ...
223     }
224 
225     private static void checkKingKillsAlongDirection(Checkers checkers, BoardLocation king, int
            dx, int dy, List<BoardLocation> opponentPawns,
226             List<BoardLocation> opponentKings, List<BoardLocation> ownPawns, List<BoardLocation>
                ownKings, List<String> moves) {
227         ...
228     }
229 
230     private static void populatePossibleKingKills(Checkers checkers, String prefix, boolean
            isWhitesTurnNow, BoardLocation king, List<String> globalList,
231             int forbiddenDirection) {
232         ...
233     }
234 
235     private static void populatePossibleKingMoves(Checkers checkers, BoardLocation king, List<
            String> globalList) {
236         ...
237     }
238 
239     private static void populatePossiblePawnMoves(Checkers checkers, boolean isWhitesTurnNow,
            BoardLocation pawn, List<String> globalList) {
240         ...
241     }
242 
243     private void eliminateNonMaximalKills(List<String> killMoves) {
244         ...
245     }
246 
247 
248     @Override
249     public List<GameState> generateChildren() {
250         List<String> moves = getPossibleMoves();
251         List<GameState> children = new LinkedList<GameState>();
252         for (String move : moves) {
253             Checkers child = new Checkers(this);
254             child.makeMove(move, true);
255             child.setMoveName(move);
256             children.add(child);
257 
258         }
259         return children;
260     }
```

```
261
262     @Override
263     public boolean isQuiet() {
264         getPossibleMoves(); // in order to calculate hasSomeKillingMoves flag
265         return !hasSomeKillMoves;
266     }
267
268     static {
269         setHFunction(new StateFunction() {
270
271             @Override
272             public double calculate(State state) {
273                 Checkers checkers = (Checkers) state;
274                 double value = 0.0;
275                 if ((checkers.whitePawns.size() + checkers.whiteKings.size() == 0) || ((checkers.
                        isWhiteTurnNow()) && (checkers.getPossibleMoves().isEmpty()))) {
276                     value = Double.NEGATIVE_INFINITY;
277                     return value;
278                 } else if ((checkers.blackPawns.size() + checkers.blackKings.size() == 0)
279                         || ((!checkers.isWhiteTurnNow()) && (checkers.getPossibleMoves().isEmpty
                                ()))) {
280                     value = Double.POSITIVE_INFINITY;
281                     return value;
282                 }
283
284                 // material
285                 value = PAWN_HEURISTIC_FACTOR * (checkers.whitePawns.size() - checkers.blackPawns
                        .size()) + KING_HEURISTIC_FACTOR
286                         * (checkers.whiteKings.size() - checkers.blackKings.size());
287
288                 // pawn advancement
289                 for (BoardLocation location : checkers.whitePawns)
290                     value += location.getY();
291                 for (BoardLocation location : checkers.blackPawns)
292                     value -= (n + 1 - location.getY());
293
294                 return value;
295             }
296         });
297     }
298
299     ...
300 }
```

The first important element to note is that our `Checkers` class extends the default game state implementation — `sac.game.GameStateImpl`, a prerequisite of *SaC*.

As regards the states identification functionality, both `toString(...)` and `hashCode(...)` possibilities work by building up suitable concatenations (accordingly to the type returned) of pieces locations, consecutively for: white pawns, white kings, black pawns, black kings, using some recognizable separators in-between. Finally, the concatanation is appended with an information about whose turn it is now to play.

As regards the move generation functionality, noted should be the `getPossibleMoves()` method, doing the main work and calling several submethods underneath. It first collects the capture moves for kings and pawns and then, if no capture moves exist, it collects the regular displacement moves. If some capture moves do exist, the player is obliged to perform one of them (instead of a non-capture move) according to the rules of checkers. Also, it is worth to

add that when many capture moves are possible, the player must choose one of such moves that take the largest possible number of opponent's pieces. This rule is fulfilled in the code by the `eliminateNonMaximalKills(...)` method, which removes from the collected list the moves with too few captures. Finally the `getPossibleMoves()` method returns the list of strings, representing the moves in a chess-like notation. The method directly responsible for the generation of descandants, provided by *SaC*'s interace, is the `generateChildren(...)` method. In our example, the `generateChildren(...)` implementation calls first the aforementioned `getPossibleMoves()` method and then iterates over the moves, applying each of them to a copy of the parent state via a `makeMove(String)` call. Please note also that each descendant gets labeled with the name of the move it was caused by, using the `child.setMoveName(...)` call — this is also a must within *SaC* that was already described in the "Hello world" section 1.3 for games.

As regards the heuristic evaluation of states, we implemented a very simple function for the purpose of example. As it is required by *SaC*, the heuristics is attached statically to the class via the `setHFunction(...)` method. In this case, we pass to it a `StateFunction` object defined anonymously. Looking at its `calculate(...)` method, two parts can be seen. The first part pertains to win positions. It returns infinity (with a suitable sign) in case some player is left with no pieces or no moves (a blockage). The second part, calculates the actual evaluation for the non-win positions. The evaluation takes into account only two components: (1) the material, where kings are weighted as being worth ten times a pawn[5], (2) the pawn advancement, where each pawn adds 1, 2, ..., or $n-1$ points depending on the board row (rank) it is located at, counting rows towards the promotion. To be more precise, the materialistic evaluation treats each pawn as being worth 100 points (in chess such points are often referred to as *centipawns*), and the king is worth 1000 points. Relatively to it, the rewards for pawn advancements are fairly small. Each next row is being counted for 1 more point (a single centipawn). Clearly, the function described above does not translate onto a strong program playing checkers. For the sake of simplification all positional aspects[6] were neglected on purpose.

One more element worth noticing is the implementation of the `isQuiet(...)` method. The method is provided by the `sac.GameState` interface. In our example, a state of checkers is regarded as quiet if there are no immediate captures possible from it. We remind that this functionality is related to the algorithmic gadget known as *Quiescence*, which leads to locally deeper search horizons when a series of captures occurs; see also page 110 in section 4.2.1.

Some trees from initial checkers position

Hereby, we present some illustrations representing *SaC*'s checkers searches with trees generated from the initial checkers position (i.e. with white pieces to make the first move). We remark that by default *SaC* does not memorize parent-child links in order to save memory. Therefore, in order

[5]This relative value is chosen as a pure guess in our exemplary checkers. Also, owing to that choice, a user experimenting with the `sac.examples.checkers.CheckersGame` program can easily observe differences in reported evaluations due to the fact that: pawn advancements change the evaluation by 1 point, pawn captures by 100 points, and promotions to kings (or king captures) change it by 1000 points.

[6]E.g.: pawns structure / patterns, 'side vs center' placement, defended pieces, occupation of main diagonal by kings, etc.

to produce *Graphviz* illustrations one has to explicitly reconfigure the algorithm to be executed. The listing below is an example how this reconfiguration can be done.

```
1 GameSearchConfigurator configurator = new GameSearchConfigurator();
2 configurator.setParentsMemorizingChildren(true);
3
4 GameSearchAlgorithm algorithm = new AlphaBetaPruning(someCheckersState, configurator);
5 algorithm.execute();
6 GameSearchGraphvizer.go(algorithm, "d:/output.dot", true, true);
```

We remind the coloring scheme in the figures: yellow denotes the initial state, light gray denotes visited regular states, dark gray indicates non-win terminal states, blue indicates win terminal states (a win for either player), dark red indicates states for which the game value or a bound on that value was read as a ready result (because these states had occurred before) from the so called transposition table3 , light red indicates the so called cutoff states which could not affect the game value, green indicates states residing along the principal variation. In every box, displayed is an information about: depth of a state, its heuristic evaluation h, its game value v (also known as the minimax value) assigned by the search procedure, and its representation (if turned on). We also encourage the reader to zoom-in the figures.

Fig. 4.2 shows a tree generated using alpha-beta pruning algorithm with depth set to 1.0 (two half-moves). Fig. 4.4 is meant to depict the difference in trees generated using the Min-Max algorithm, without and with the *Quiescence* option. This time the depth is set to 1.5 (so for simplification the states are displayed with no contents inside). Fig. 4.4 compares the trees produced by alpha-beta pruning and Scout, both using 1.5 as the general search horizon, whereas Fig. 4.5 shows an analogical comparison with deeper general horizon — 2.0.

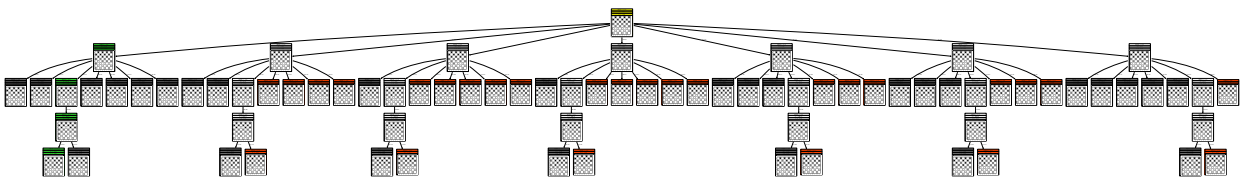


Figure 4.2: Checkers search tree generated by *SaC* from the initial position (root), using *alpha-beta pruning* algorithm and search depth set to 1.0 (two half-moves). The *Queiscence* option is turned on and makes the algorithm look beyond the 1.0 depth in several places.

(a) maximum depth 1.5, *Min-Max*, *quiescence* off

(a) maximum depth 1.5, *Min-Max*, *quiescence* on

Figure 4.3: Checkers search trees generated by *SaC* from the initial position (root), using *Min-Max* algorithm and with *Queiscence* turned off (a) and on (b), respectively. General search depth set to 1.5 (three half-moves), but *Quiescence* feature makes the algorithm visit deeper trees sometimes (when pawn captures take place).

(a) maximum depth 1.5 (unless state not quiet), *alpha-beta pruning*

(b) maximum depth 1.5 (unless state not quiet), *Scout*

Figure 4.4: Checkers search trees generated by *SaC* from the initial position (root), using *alpha-beta pruning* (a) and *Scout* (b) algorithms. General search depth set to 1.5 (three half-moves), but *Quiescence* feature makes the algorithm visit deeper trees sometimes (when pawn captures take place).

(a) maximum depth 2.0 (unless state not quiet), *alpha-beta pruning*

(b) maximum depth 2.0 (unless state not quiet), *Scout*

Figure 4.5: Checkers search trees for initial position as root, generated by *SaC* with *alpha-beta pruning* (a) and *Scout* (b) algorithms. General search depth set to 2.0 (four half-moves), but *quiescence* feature makes the algorithm visit deeper trees sometimes (when pawn captures take place).

Endgame examples

In this subsection we show some examples of *SaC* results obtained when searching endgames of checkers.

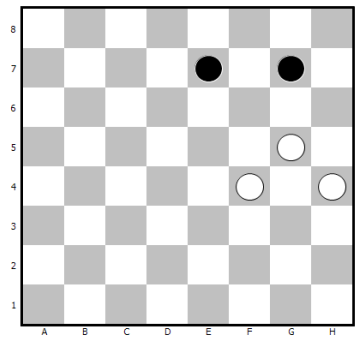
Fig. 4.6 pertains to a position with 3 white pawns against 2 black pawns, and white pieces to play now. The gap between opposing pieces is just one row, therefore the winning move for white — `G5:H6` — is discovered quickly. *SaC* visited 89 states in about 50 ms. The global search horizon was narrowed to 2.5 and the Quiescence option was on, which made the algorithm reach the 3.5 depth in several places. The displayed payoff assiociated with the `G5:H6` move is `1.1556598724114885E308`, representing a win in 7 half-moves.

Fig. 4.7 also pertains to an endgame with 3 pawns against 2, but with two rows of a gap in-between. This 'little' difference causes that the algorithm is unable to discover (with certainty) a winning move within the default search horizon of seven half-moves (3.5). Therefore, we forced the algorithm to see deeper, setting the horizon to 5.0. After visiting 2 095 and about half a second of time the right move `D2:E3` was discovered with its payoff `1.0622732160550047E308`. It represents a win in 12 half-moves, as the algorithm reached positions that deep owing to the Quiescence. The reader should also note the difference between the number of states drawn in the figure — 3 506 — and the number of states actually visited by the algorithm — 2 095. The difference is caused by cutoff states and states for which the evaluation was read from the transposition table.

Fig. 4.8 depicts a "3 kings vs 1 king" ending. It shows how white can trap opponent's king in 9 half-moves at most. Since the number of visited states — 65 438 — is now considerably large, we prepared a special visualization where the right sequence of moves (the principal variation) is exposed whereas subtrees generated by other moves (side moves) are only marked by their roots.

Endgame example 1

(a) endgame position



(b) *SaC* result

```
Searching with sac.game.AlphaBetaPruning started...
Searching with sac.game.AlphaBetaPruning done in 46 ms.
Closed states:  89
General depth limit:  2.5
Maximum depth reached (Quiescence):  3.5
Transposition table size:  89
Transposition table uses:  2
Refutation table size:  41
Refutation table uses:  0
Moves scores:  {G5:H6=1.1556598724114885E308, F4:E5=999.0}
Best moves:  [G5:H6]
Principal variation:  [G5:H6, G7:F6, F4:G5, F6:E5, G5:F6, E5:G7,
    H6:F8:D6]
```

(c) search tree for depth 2.5 using *alpha-beta pruning* (100 states drawn)

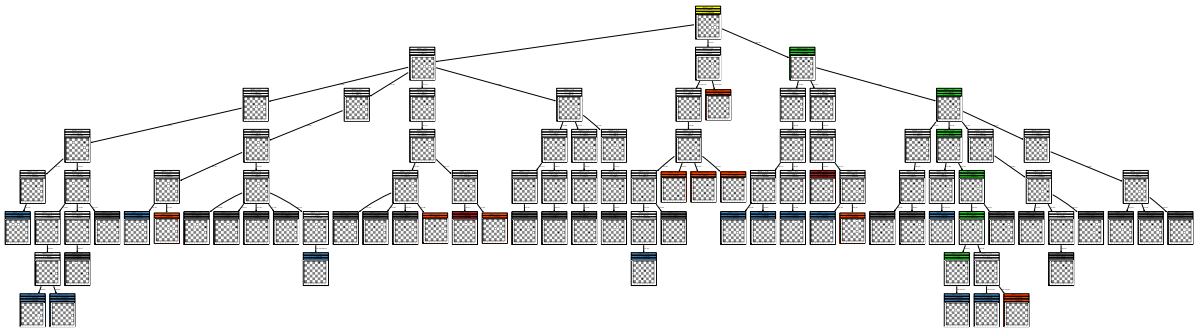
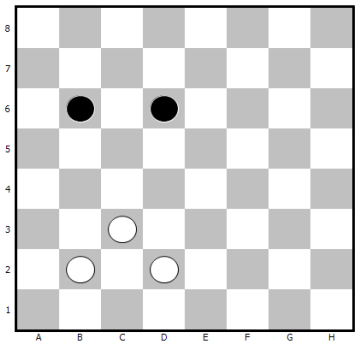


Figure 4.6: *SaC* results for an endgame checkers position. Win for white found within search horizon of depth 3.5.

Endgame example 2

(a) endgame position



(b) *SaC* result

```
Searching with sac.game.Scout...
Searching done.  Time:  577 ms.
Closed states:  2095
General depth limit:  5.0
Maximum depth reached (Quiescence):  6.0
Transposition table size:  1632
Transposition table uses:  498
Refutation table size:  127
Refutation table uses:  0
Scores:  {D2:E3=1.0622732160550047E308, C3:B4=204.0, B2:A3=101.0}
Best move:  D2:E3
Principal variation:  [D2:E3, B6:A5, B2:A3, D6:C5, C3:D4, A5:B4,
    A5:B4, D4:B6, B4:C3, B6:A7, C3:B2, A3:C1]
```

(c) search tree for depth 5.0 using *Scout* (3 506 states drawn, star-like layout)

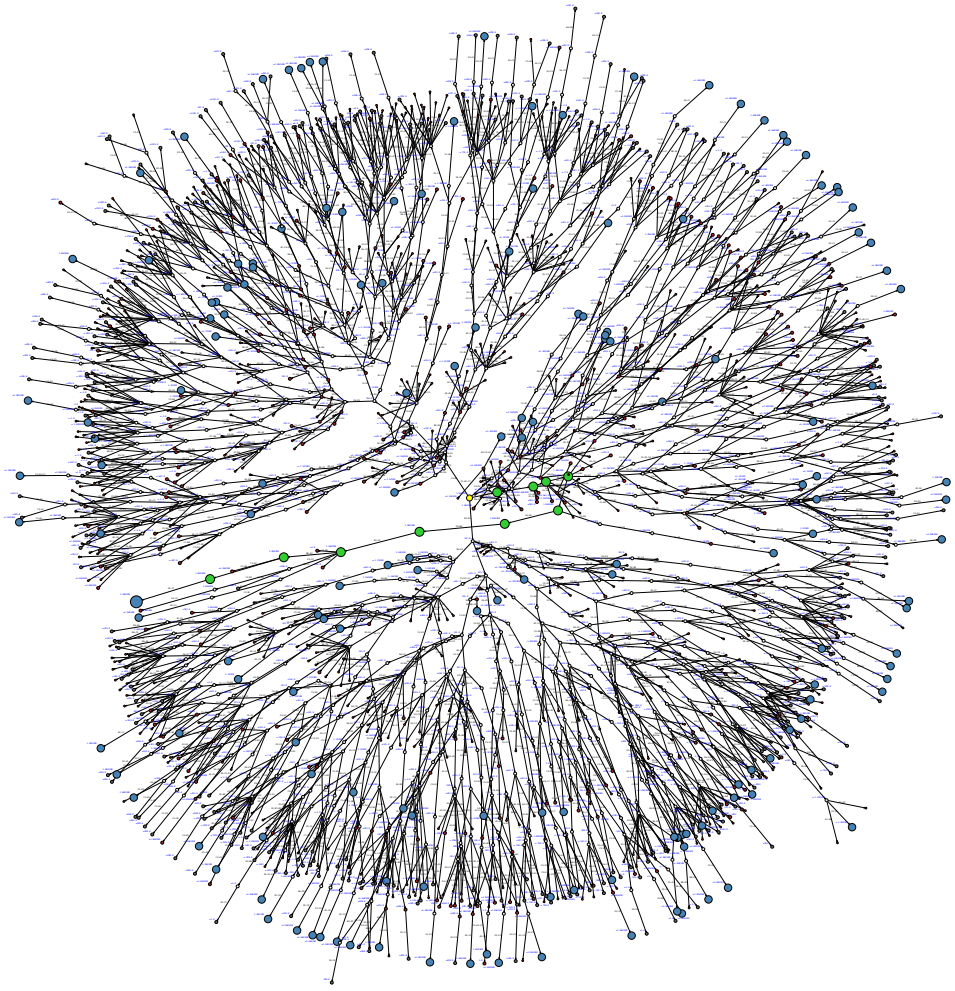
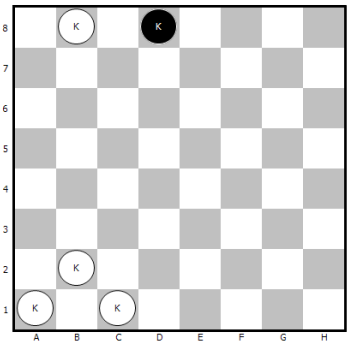


Figure 4.7: *SaC* results for an endgame checkers position. Win for white found within search horizon of depth 6.0.

Endgame example 3 — "3 kings vs 1 king"

(a) endgame position



(b) *SaC* result

```
Searching with sac.game.Scout started...
Searching with sac.game.Scout done in 4164 ms.
Closed states:  65438
General depth limit:  3.5
Maximum depth reached (Quiescence):  4.5
Transposition table size:  64106
Transposition table uses:  82549
Refutation table size:  5734
Refutation table uses:  0
Moves scores:  {B2:D4=1.0985902490825263E308, B2:A3=3000.0}
Best moves:  [B2:D4]
Principal variation:  [B2:D4, D8:A5, B8:D6, A5:E1, D6:G3, E1:H4,
    C1:G5, H4:F6:C3, A1:D4]
```

(c) fragment of search tree with principal variation using for depth 3.5 using *Scout*

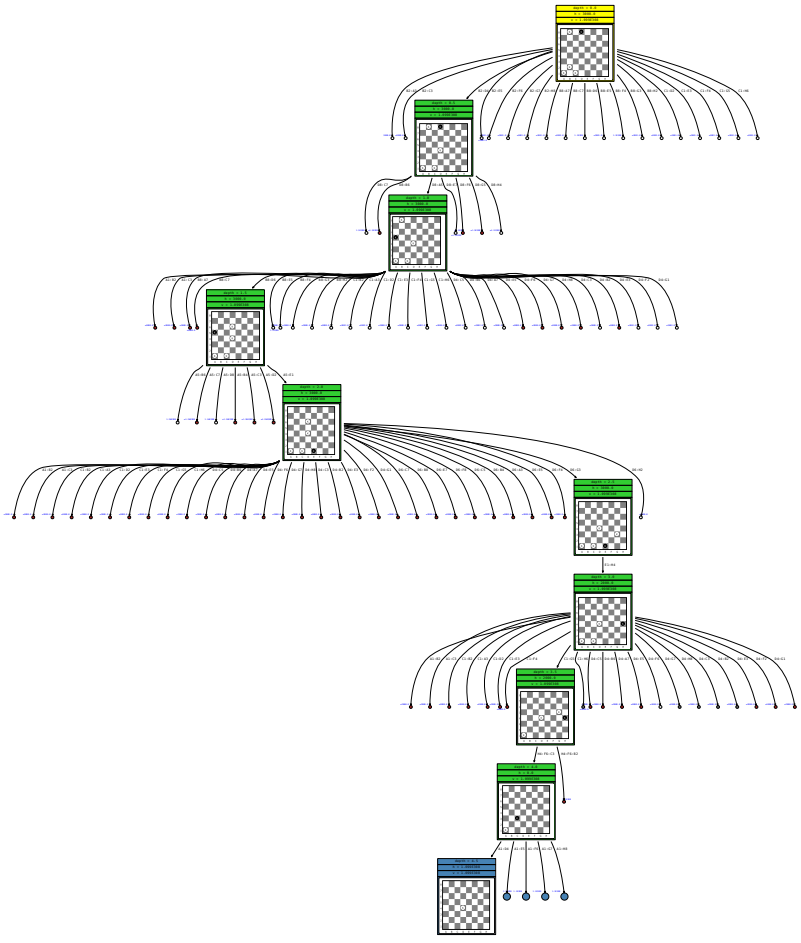


Figure 4.8: *SaC* results for an endgame checkers position. Win for white found within search horizon of depth 4.5.

4.3.2 Nim

Nim is a two person mind game, where players interchangeably remove objects from a given number of piles. A player must take at least one object and can take as many objects as he wants provided that they come from the same pile. Depending on the convention being played, the player to take the last remaining object is either the winner (normal variant) or the loser (misére variant). A common initial setup is that given are three piles, consisting of 3, 4 and 5 objects; but, other setups are possible.

Nim has been solved mathematically (for any number of piles and objects). There exist a simple calculation scheme telling which player will win at a given position with an optimal line of play. Moreover, it is possible to tell the right move. The key notion of the scheme is a so called *nim-sum* of pile sizes, coded in the binary system. The sum is taken using the exclusive-or operation (or modulo 2 sum). For example for piles of sizes 3, 4, 5, their binary representations and the nim-sum are as follows:

$$3 = (0, 1, 1)_2,$$
$$4 = (1, 0, 0)_2,$$
$$5 = (1, 0, 1)_2,$$

$$\text{nim-sum: } = (0, 1, 0)_2.$$

It turns out that the optimal strategy boils down to making moves which keep the nim-sum of the remaining piles equal to zero. Interestingly, this strategy is in general valid for both conventions of the play (normal or misère), and only the very endgames require slight changes depending on the convention. In the normal variant, when the game is reduced to two piles only, the player should make such moves to keep both piles equal. In the misère variant, when there remain piles with no more than two objects, the correct move is to leave an odd number of piles of size one. More details on Nim, its variants, history and strategy can be found at: `https://en.wikipedia.org/wiki/Nim`.

We now move to the implementation issues and *SaC*'s API. As in former examples, we first show code excerpts from the state representation class — `sac.examples.nim.NimState`, then we comment on it with the focus on vital elements: generation of descendants, identification, heuristics.

```
1  package sac.examples.nim;
2
3  ...
4
5  public class NimState extends GameStateImpl {
6
7      private List<Integer> piles;
8
9      public NimState() {
10         piles = new LinkedList<Integer>();
11         piles.add(3);
12         piles.add(4);
13         piles.add(5);
14         setMaximizingTurnNow(true);
```

```
15      }
16
17      public NimState(List<Integer> initial, boolean isWhiteTurnNow) {
18          piles = new ArrayList<Integer>(initial);
19          Collections.copy(piles, initial);
20          setMaximizingTurnNow(isWhiteTurnNow);
21      }
22
23      public NimState(NimState parent) {
24          super();
25          piles = new ArrayList<Integer>();
26          for (int item : parent.getPiles()) {
27              piles.add(item);
28          }
29          setMaximizingTurnNow(parent.isMaximizingTurnNow());
30      }
31
32      public List<Integer> getPiles() {
33          return piles;
34      }
35
36      public List<List<Integer>> getPossibleMoves() {
37          List<List<Integer>> list = new ArrayList<List<Integer>>();
38          int qi;
39          for (int i = 0; i < piles.size(); i++) {
40              qi = piles.get(i);
41              for (int j = 1; j <= qi; j++) {
42                  ArrayList<Integer> move = new ArrayList<Integer>(
43                          Collections.nCopies(piles.size() - 1, 0));
44                  move.add(i, j);
45                  list.add(move);
46              }
47          }
48          return list;
49      }
50
51      public void makeMove(List<Integer> move) {
52          assert (checkMove(move));
53          int newValue;
54          for (int i = 0; i < piles.size(); i++) {
55              newValue = piles.get(i) - move.get(i);
56              piles.remove(i);
57              piles.add(i, newValue);
58          }
59          setMaximizingTurnNow(!maximizingTurnNow);
60          refresh();
61      }
62
63      public void makeMove(String stringMove) {
64          String[] si = stringMove.substring(1, stringMove.length() - 1).split(",");
65          List<Integer> move = new LinkedList<Integer>();
66          for (String s : si) {
67              move.add(Integer.parseInt(s.trim()));
68          }
69          makeMove(move);
70      }
71
72      public boolean checkMove(List<Integer> move) {
73          if (piles.size() != move.size()) {
74              return false;
75          }
```

```
76          for (int i = 0; i < piles.size(); i++) {
77              if (!((move.get(i) <= piles.get(i)) && (move.get(i) > 0))) {
78                  return false;
79              }
80          }
81          return true;
82      }
83
84      @Override
85      public List<GameState> generateChildren() {
86          List<GameState> children = new ArrayList<GameState>();
87          for (List<Integer> move : getPossibleMoves()) {
88              NimState child = new NimState(this);
89              child.makeMove(move);
90              child.setMoveName(move.toString());
91              children.add(child);
92          }
93          return children;
94      }
95
96      @Override
97      public String toString() {
98          return piles.toString() + "|" + maximizingTurnNow;
99      }
100
101     @Override
102     public int hashCode() {
103         String string = toString();
104         return string.hashCode();
105     }
106
107
108     public boolean isTerminal() {
109         for (Integer item : piles) {
110             if (item > 0)
111                 return false;
112         }
113         return true;
114     }
115
116     static {
117         setHFunction(new StateFunction() {
118             @Override
119             public double calculate(State state) {
120                 NimState nimState = (NimState) state;
121                 if (nimState.isTerminal())
122                     return Double.POSITIVE_INFINITY * (nimState.isMaximizingTurnNow() ? -1 : 1);
123                 return 0.0;
124             }
125         });
126     }
127
128     ...
129 }
```

The piles of objects are represented by a list of integers — `List<Integer> piles` — so that `piles.get(i)` returns the number of objects remaining on the i-th pile.

There are two methods present in the class related to making a move in a Nim game — `makeMove(List<Integer> move)` and `makeMove(String stringMove)`. They differ only in the

type of the argument and work the same way underneath. As described in the introduction on Nim, typically a player is allowed to remove objects only from one pile. In the implementation though we decided to go for more generality, and a move is a sequence of length equal to the number of piles, where numbers in the sequence represent how many objects should be subtracted from successive piles. Hence, potentially one could write for example: `nimState.makeMove("2,0,1")`, which indicates that two objects should be removed from the first pile, no objects from the second pile, and one object from the third. Despite the more general code at this place, there is a method named `getPossibleMoves(...)`, which in fact generates only the moves where objects are subtracted from a single pile. Therefore, the generated sequences, representing legal moves, consist of a single non-zero value somewhere and zeros elsewhere.

As regards the identification of states, both `toString()` and `hashCode()` methods have been implemented. Yet, in fact the `hashCode()` method first calls the `toString()` and then converts its result to an integer, using the Java built-in mechanism to calculate a hash code for a string[7].

The position evaluation function is, as always, attached via the `setHFunction(...)` static method. In our Nim implementation we distinguish only three possible evaluations: $\pm\infty$ (for win positions) or 0 (for non-win positions). As it was mentioned before in the introduction, Nim is not a kind of game (contrarily e.g. to chess or checkers) where a player is capable of building slowly and incrementally an advantage for himself. A move is either correct or not at any stage of a game. That is why the code shown represents only such a primitive heuristic function.[8] In the presented form of the code, the heuristics reflects the normal game variant, i.e. the player to take the last object is the winner.

[7] see `http://docs.oracle.com/javase/7/docs/api/java/lang/String.html#hashCode%28%29`

[8] It is possible to comment that the remarks above (about a move being either correct or not) are in fact also correct for chess. Chess is not a solved game. And although its game tree is astronomically huge, it is a *finite* tree. Rules like: triple repetition of a position, pereptual check, etc. cause that the game cannot be played for ever. Since the chess tree is finite, it is easy to prove that there exist an objective answer about the outcome of the game under an optimal play. We, people, do not know that outcome (chess is still unsolved), but it certainly does exist.

Chapter 5

Tools

This chapter is devoted to additional tools provided with *SaC*. They are not indespensible in a typical usage, but provide some auxiliary functionalities that might make the work with *SaC* more convenient. First, discussed is *SaC*'s interface to *Graphviz* — a language for automatic graph drawing. Secondly, we describe how to carry out batch experiments using *SaC* and how to collect and plot statistics from such experiments. The plots are generated with the help of *jFreeChart* library. Finally, we present the possibility of monitoring the progress of graph searching (for heavy problems).

5.1 Graphviz

Graphviz (short for Graph Visualization Software) is a free software and an underlying language for automatic graph drawing. The reader can jump ahead to look at figures 5.1 and 5.2 to have an outlook on the Graphviz language. Yet, we should stress that a *SaC* user is not required to be familiar with that language and still can generate visualizations for his search experiments. As one can see, for the graph from Fig. 5.1, there is no information in the code about the formatting style of the nodes and edges. The code describes just the structure of the graph. The graph from Fig. 5.2 (or more precisely the search tree) has been generated automatically by *SaC* for a simple sliding puzzle problem. Apart from the structure, the code now describes also the formatting of its elements, partially via the HTML language.

In *SaC*, we enable a possibility to generate output text files, compliant with the language. The files contain represenations of graphs or trees searched by a particular algorithm. Having such a file and the Graphviz software installed, the user can produce visualizations using one of Graphviz engines like `dot`, `neato`, etc. For more details about the Graphviz and its language we address the reader to the Graphviz official website: `http://www.graphviz.org`. We shall now discuss some elements of *SaC*'s API related to Graphviz.

Any search algorithm from *SaC* that has stopped successfully can potentially be 'flushed out' to a .dot file compliant with the Graphviz language. There is one necessary condition — the user has to explicitly reconfigure *SaC*, before an algorithm is executed, and make parent states memorize their children states. By default, only the opposite way link is memorized — from a

(a) (b)

```
digraph g {
A;
B;
C;
A -> B;
A -> C;
B -> A;
}
```

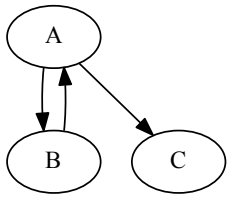


Figure 5.1: A simple directed graph coded in the Graphviz language (a) and rendered out (b).

child to its parent. The mentioned reconfiguration facilitates the generation of the Graphviz code, but obviously is more memory consuming.

The reconfiguration is done either by the `sac.graph.GraphSearchConfigurator` class or the `sac.game.GraphSearchConfigurator` class, depending on the type of search. Let us remind here an example from the checkers related section.

```
1 GameSearchConfigurator configurator = new GameSearchConfigurator();
2 configurator.setParentsMemorizingChildren(true);
3
4 GameSearchAlgorithm algorithm = new AlphaBetaPruning(someCheckersState, configurator);
5 algorithm.execute();
6 GameSearchGraphvizer.go(algorithm, "d:/output.dot", true, true);
```

In the example, the algorithm to be executed is, at its instantiation, equipped with a an initial state and additionally with a configurator. The configurator's option to memorize parent-children links is set to `true`. Once the algorithm is finished, an output .dot file compliant with Graphviz is produced by an invocation of the `sac.graphviz.GameSearchGraphvizer.go(...)` method. The arguments to that method are as follows: the algorithm object, a path to the target output file, a boolean flag stating whether the nodes should be rendered out with their contents (i.e. state representations) or just as empty circles, and finally another boolean flag stating wheter names of moves (or manipulations) should be printed near the edges.

On the Windows operating system, the following exemplary command-line invocation would produce an output .pdf file from the given .dot file (provided that Graphviz is installed in the system and that the `dot` command is visible).

```
D:\>dot -Tpdf output.dot -o d:\output.pdf
```

```
digraph g {
ranksep=0.25;
node [shape=none,height=0.1];

-2097210167
 [label=<<TABLE BORDER='0' CELLBORDER='1' CELLPADDING='2' CELLSPACING='0' BGCOLOR='yellow'><TR><TD><FONT FACE='monospace' POINT-SIZE='8'>
        depth = 0.0</FONT></TD></TR><TR><TD><FONT FACE='monospace' POINT-SIZE='8'>g = 0.0</FONT></TD></TR><TR><TD><FONT FACE='monospace'
        POINT-SIZE='8'>h = 1.0</FONT></TD></TR><TR><TD><FONT FACE='monospace' POINT-SIZE='8'>f = 1.0</FONT></TD></TR><TR><TD><TABLE BORDER='0'
        CELLBORDER='1' CELLPADDING='2' CELLSPACING='0' BGCOLOR='white'><TR><TD><FONT FACE='monospace' POINT-SIZE='8'><TABLE BORDER='0'
        CELLBORDER='1' CELLSPACING='0'><TR><TD>3</TD><TD>1</TD><TD>2</TD></TR><TR><TD></TD><TD>4</TD><TD>5</TD></TR><TR><TD>6</TD><TD>7</TD>
        <TD>8</TD></TR></TABLE></FONT></TD></TR></TABLE></TD></TR></TABLE>>,fillcolor=yellow,height=0.2];

-1986387647
 [label=<<TABLE BORDER='0' CELLBORDER='1' CELLPADDING='2' CELLSPACING='0' BGCOLOR='orangered'><TR><TD><FONT FACE='monospace' POINT-SIZE='8'>
        depth = 1.0</FONT></TD></TR><TR><TD><FONT FACE='monospace' POINT-SIZE='8'>g = 1.0</FONT></TD></TR><TR><TD><FONT FACE='monospace'
        POINT-SIZE='8'>h = 2.0</FONT></TD></TR><TR><TD><FONT FACE='monospace' POINT-SIZE='8'>f = 3.0</FONT></TD></TR><TR><TD><TABLE BORDER='0'
        CELLBORDER='1' CELLPADDING='2' CELLSPACING='0' BGCOLOR='white'><TR><TD><FONT FACE='monospace' POINT-SIZE='8'><TABLE BORDER='0'
        CELLBORDER='1' CELLSPACING='0'><TR><TD>3</TD><TD>1</TD><TD>2</TD></TR><TR><TD>4</TD><TD></TD><TD>5</TD></TR><TR><TD>6</TD><TD>7</TD>
        <TD>8</TD></TR></TABLE></FONT></TD></TR></TABLE></TD></TR></TABLE>>,fillcolor=orangered,height=0.06];

-1925441027
 [label=<<TABLE BORDER='0' CELLBORDER='1' CELLPADDING='2' CELLSPACING='0' BGCOLOR='orangered'><TR><TD><FONT FACE='monospace' POINT-SIZE='8'>
        depth = 1.0</FONT></TD></TR><TR><TD><FONT FACE='monospace' POINT-SIZE='8'>g = 1.0</FONT></TD></TR><TR><TD><FONT FACE='monospace'
        POINT-SIZE='8'>h = 2.0</FONT></TD></TR><TR><TD><FONT FACE='monospace' POINT-SIZE='8'>f = 3.0</FONT></TD></TR><TR><TD><TABLE BORDER='0'
        CELLBORDER='1' CELLPADDING='2' CELLSPACING='0' BGCOLOR='white'><TR><TD><FONT FACE='monospace' POINT-SIZE='8'><TABLE BORDER='0'
        CELLBORDER='1' CELLSPACING='0'><TR><TD>3</TD><TD>1</TD><TD>2</TD></TR><TR><TD>6</TD><TD>4</TD><TD>5</TD></TR><TR><TD></TD><TD>7</TD>
        <TD>8</TD></TR></TABLE></FONT></TD></TR></TABLE></TD></TR></TABLE>>,fillcolor=orangered,height=0.06];

-883926621
 [label=<<TABLE BORDER='0' CELLBORDER='1' CELLPADDING='2' CELLSPACING='0' BGCOLOR='steelblue'><TR><TD><FONT FACE='monospace' POINT-SIZE='8'>
        depth = 1.0</FONT></TD></TR><TR><TD><FONT FACE='monospace' POINT-SIZE='8'>g = 1.0</FONT></TD></TR><TR><TD><FONT FACE='monospace'
        POINT-SIZE='8'>h = 0.0</FONT></TD></TR><TR><TD><FONT FACE='monospace' POINT-SIZE='8'>f = 1.0</FONT></TD></TR><TR><TD><TABLE BORDER='0'
        CELLBORDER='1' CELLPADDING='2' CELLSPACING='0' BGCOLOR='white'><TR><TD><FONT FACE='monospace' POINT-SIZE='8'><TABLE BORDER='0'
        CELLBORDER='1' CELLSPACING='0'><TR><TD></TD><TD>1</TD><TD>2</TD></TR><TR><TD>3</TD><TD>4</TD><TD>5</TD></TR><TR><TD>6</TD><TD>7</TD>
        <TD>8</TD></TR></TABLE></FONT></TD></TR></TABLE></TD></TR></TABLE>>,fillcolor=steelblue,height=0.4];

-2097210167 -> -1986387647
[label=<<FONT FACE='monospace' POINT-SIZE='6'> R</FONT>>,arrowsize=0.4];

-2097210167 -> -1925441027
[label=<<FONT FACE='monospace' POINT-SIZE='6'> D</FONT>>,arrowsize=0.4];

-2097210167 -> -883926621
[label=<<FONT FACE='monospace' POINT-SIZE='6'> U</FONT>>,arrowsize=0.4];
}
```

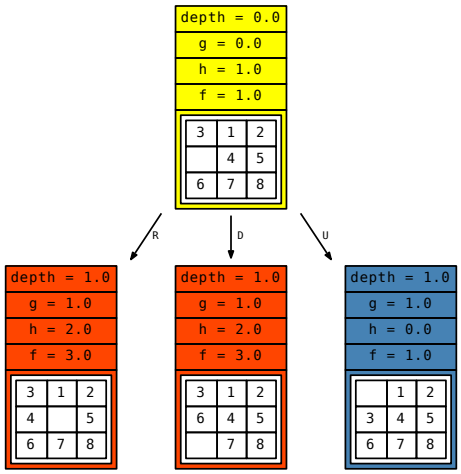


Figure 5.2: Graphviz code generated automatically by *SaC* for a simple sliding puzzle search (above) and its visualization rendered out (below).

By default, the Graphviz engine renders its outputs according to the so called *dot layouter*. For search trees, this results in natural top-down visualizations where successive rows correspond to successive depth levels and the initial state is at the top. For star-like visualizations, with the initial state in the middle, the *neato layouter* should be used. The suitable command-line invocation for that purpose could look like this:

```
D:\>dot -Tpdf output.dot -Kneato -o d:\output.pdf
```

For more information about available options, switches and command-line Graphviz invocations the reader is addressed to the documentation, in particular to: `http://www.graphviz.org/doc/info/command.html`.

As regards the displayed contents of nodes, a *SaC* user can affect those by providing his state class with an implementation of the `toGraphvizLabel()` method. This method is visible and can be overriden because the base interface in *SaC* — `sac.State` — extends a `sac.graphviz.Graphvizable` interface with that particular method. By default, the `toGraphvizLabel()` method returns the identifier of a state object. Therefore, when elegant visualizations are intended, a more meaningful state representation should be implemented. Please note, that Graphviz allows to use HTML language for that purpose. Thus, a programmer can construct a suitable HTML string within his implementation of the `toGraphvizLabel()` method. As an example, below we show a code excerpt for that purpose taken from the sliding puzzle class.

```
1  @Override
2  public String toGraphvizLabel() {
3      StringBuilder builder = new StringBuilder();
4      builder.append("<FONT␣FACE='monospace'␣POINT-SIZE='8'><TABLE␣BORDER='0'␣CELLBORDER='1'␣
           CELLSPACING='0'>");
5
6      int k = 0;
7      for (int i = 0; i < n; i++) {
8          builder.append("<TR>");
9          for (int j = 0; j < n; j++) {
10             builder.append("<TD>" + ((board[k] == 0) ? "" : board[k]) + "</TD>");
11             k++;
12         }
13         builder.append("</TR>");
14     }
15     builder.append("</TABLE></FONT>");
16
17     return builder.toString();
18 }
```

5.2 Statistics and charts for batch experiments

SaC allows to conveniently carry out *batch* experiments. What do we mean by that? Imagine a compact piece of code, containing a number of loops, that iterate for example over: mulitple random instances of a certain problem, multiple search algorithms, multiple heuristic functions, and different data structures involved; and in each iteration an execution of the search procedure for given settings is performed. Once such a collection of experiments is registered, one would

like to easily extract some statistics out of it — for example: an avarage solution path length per an algorithm, a total of visited states per a heuristic, an avarage duration time per particular combinations of an algorithm and a data structure, a variance in the size of the transposition table, and so forth. The object-oriented design of *SaC* allows to do so, see the example below.

```
1  public static void main(String[] args) throws Exception {
2      System.out.println("Starting ...");
3      long t1 = System.currentTimeMillis();
4
5      Stats stats = new Stats();
6
7      // loop over random sliding puzzle problems
8      for (int i = 0; i < 100; i++) {
9          SlidingPuzzle puzzle = new SlidingPuzzle((byte) 3);
10         puzzle.shuffle((Math.random() > 0.5) ? 1000 : 1001); // shuffling
11
12         // initial solution by A* so that optimal path length is known for further statistics
13         AStar astar = new AStar(new SlidingPuzzle(puzzle));
14         astar.execute();
15         int optimalPathLength = astar.getSolutions().get(0).getPath().size();
16
17         // loop over algorithms
18         GraphSearchAlgorithm[] algorithms = { new BestFirstSearch(), new AStar() };
19         for (GraphSearchAlgorithm algorithm : algorithms) {
20
21             // loop over heuristics
22             StateFunction[] heurs = { new HFunctionManhattan(), new HFunctionLinearConflicts() };
23             for (StateFunction h : heurs) {
24
25                 // loop over different open set implementations
26                 Class[] openSetClasses = { OpenSetAsPriorityQueue.class,
                       OpenSetAsPriorityQueueFastContains.class,
27                         OpenSetAsPriorityQueueFastContainsFastReplace.class };
28                 for (Class openSetClass : openSetClasses) {
29
30                     algorithm.setInitial(new SlidingPuzzle(puzzle));
31                     SlidingPuzzle.setHFunction(h);
32                     GraphSearchConfigurator configurator = new GraphSearchConfigurator();
33                     configurator.setOpenSetClassName(openSetClass.getName());
34                     algorithm.setConfigurator(configurator);
35
36                     // search
37                     algorithm.execute();
38
39                     // register current single run in stats object
40                     stats.addEntries(algorithm, i, algorithm.getClass(), h.getClass(),
                           openSetClass, optimalPathLength);
41                 }
42             }
43         }
44     }
45
46     long t2 = System.currentTimeMillis();
47     System.out.println("Experiment total time [s]: " + (0.001 * (t2 - t1)));
48
49     // charts based on collected statistics to be produced here ...
50 }
```

The example represents experimantations on random sliding puzzle problems, in the $3 \times 3 - 1$ variant. It involves four nested loops in which varied are: problems, algorithms, heuristics, and data structures implementing the *Open* set. An important element to notice in the code is the `sac.stats.Stats` object. The role of this object is to register singular experiments and then to produce the wanted statistics out of it.

As far as registration of experiments is concerned, the usage is simple and convenient. One instatiates a `sac.stats.Stats` object at the very start and then in the middle of all loops one registers results of a particular execution by a single line of code, calling the `addEntries(...)` method. The full signature of that method is as follows.

```
public void addEntries(SearchAlgorithm algorithm, Object... multiIndex) {
        ...
}
```

As one can see, the method requires a reference to the search algorithm (that finished its execution) as the first argument. The second argument, named `multiIndex`, is passed by the ellipsis programming mechanism — so in fact it can by an arbitrary number of arguments that together describe the settings for a given experiment. It can be explained that on the low level *SaC* uses that `multiIndex` as a key to a large hash map that stores results of experiments inside the `sac.stats.Stats` object. If a reader is interested in such low level details we address him to the source code of the library.

The exemplary lines of code shown below are the sequel to the previous batch experiment. They demonstrate how one can extract some wanted statistics (i.e. their numeric values) once all loops are finished. We encourage the reader to try to guess what the lines are meant to calculate.

```
1 stats.mean(StatsCategory.GRAPH_SEARCH_PATH_LENGTH.toString(), null, AStar.class, null, null, null
      );
2 stats.mean(StatsCategory.GRAPH_SEARCH_PATH_LENGTH.toString(), null, BestFirstSearch.class, null,
      null, null);
3 stats.mean(StatsCategory.GRAPH_SEARCH_CLOSED_STATES.toString(), null, null, HFunctionManhattan.
      class, null, null);
4 stats.variance(StatsCategory.GRAPH_SEARCH_CLOSED_STATES.toString(), null, null,
      HFunctionManhattan.class, null, null);
5 stats.mean(StatsCategory.GRAPH_SEARCH_CLOSED_STATES.toString(), null, null,
      HFunctionLinearConflicts.class, null, null);
6 stats.variance(StatsCategory.GRAPH_SEARCH_CLOSED_STATES.toString(), null, null,
      HFunctionLinearConflicts.class, null, null);
7 stats.max(StatsCategory.GRAPH_SEARCH_DURATION_TIME.toString(), null, AStar.class, null,
      OpenSetAsPriorityQueue.class);
8 stats.max(StatsCategory.GRAPH_SEARCH_DURATION_TIME.toString(), null, AStar.class, null,
      OpenSetAsPriorityQueueFastContainsFastReplace.class);
```

Well, the lines from above, if augmented with suitable `System.out.println(...)` instructions and descriptions, could result in the following output to the console:

```
Mean solution path length for A*: 23.2
Mean solution path length for Best-first-search: 57.45
Mean number of closed states for Manhattan heuristics: 616.125
Variance of number of closed states for Manhattan heuristics: 533885.259375
Mean number of closed states for Manhattan + LC heuristics: 335.325
Variance of number of closed states for Manhattan + LC heuristics: 196093.519375
Max duration time for A* and standard open set [ms]: 120.0
Max duration time for A* and FCFR open set [ms]: 80.0
```

As one can figure out the extraction of statitics from the `stats` object is based somehow on a suitable usage of the `null` symbol. The wanted aggregation operation (mean, variance, max, min) should be called on the `stats` object and two elements should be specified: (1) with respect to which quantity the aggregation should be taken, (2) what is the multiindex pattern to aggregate over — this part involves the usage of `null`s. The measured quantities at disposal are defined by the `sac.stats.StatsCategory` enumeration:

```
public enum StatsCategory {
    GRAPH_SEARCH_DURATION_TIME,
    GRAPH_SEARCH_CLOSED_STATES,
    GRAPH_SEARCH_OPEN_STATES,
    GRAPH_SEARCH_SOLUTIONS,
    GRAPH_SEARCH_PATH_LENGTH,
    GRAPH_SEARCH_PATH_G,
    GAME_SEARCH_DURATION_TIME,
    GAME_SEARCH_CLOSED_STATES,
    GAME_SEARCH_TRANSPOSITION_TABLE_SIZE,
    GAME_SEARCH_TRANSPOSITION_TABLE_USES,
    GAME_SEARCH_REFUTATION_TABLE_SIZE,
    GAME_SEARCH_REFUTATION_TABLE_USES, GAME_SEARCH_DEPTH_REACHED;
}
```

As regards the multiindex pattern, it is, again, specified by the ellipsis mechanism. So, the programmer is supposed to give a sequence of objects according to the same scheme (order) that was used when experiments were being registered. The mentioned trick at this stage is that any position within the multiindex can be either fixed or replaced by a `null` symbol, which is intended to mean *'any'*. In other words a particular arrangement of `null`s defines the scope (range) for the aggregation. For example, the two last printouts to the screen in the former example involve a fixed algorithm and a fixed type of Open set, whereas all other parameters are free to vary.

Additionally, *SaC* offers a possibility to automatically generate plots out of statitics. This functionality is underneath based on the *jFreeChart* library. Let us start by an example of a bar chart generated using the statistics. We believe the code below is self-explanatory and the resulting jpeg file with the chart is presented in the Fig. 5.3.

```
StatsBarChart statsBarChart1 = new StatsBarChart(stats, "sliding puzzle - duration", "algorithm
    and heuristics", "time [ms]");
statsBarChart1.setValue("LC", "A*", StatsOperationType.MEAN, StatsCategory.
    GRAPH_SEARCH_DURATION_TIME.toString(), null, AStar.class,
        HFunctionLinearConflicts.class, null, null);
statsBarChart1.setValue("M", "A*", StatsOperationType.MEAN, StatsCategory.
    GRAPH_SEARCH_DURATION_TIME.toString(), null, AStar.class,
        HFunctionManhattan.class, null, null);
statsBarChart1.setValue("LC", "BFS", StatsOperationType.MEAN, StatsCategory.
    GRAPH_SEARCH_DURATION_TIME.toString(), null, BestFirstSearch.class,
        HFunctionLinearConflicts.class, null, null);
statsBarChart1.setValue("M", "BFS", StatsOperationType.MEAN, StatsCategory.
    GRAPH_SEARCH_DURATION_TIME.toString(), null, BestFirstSearch.class,
        HFunctionManhattan.class, null, null);
statsBarChart1.saveAsJPEG("./sliding_puzzle_duration.jpg");
```

The code below produces three more exemplary plots: one bar chart and two x-y plots, shown

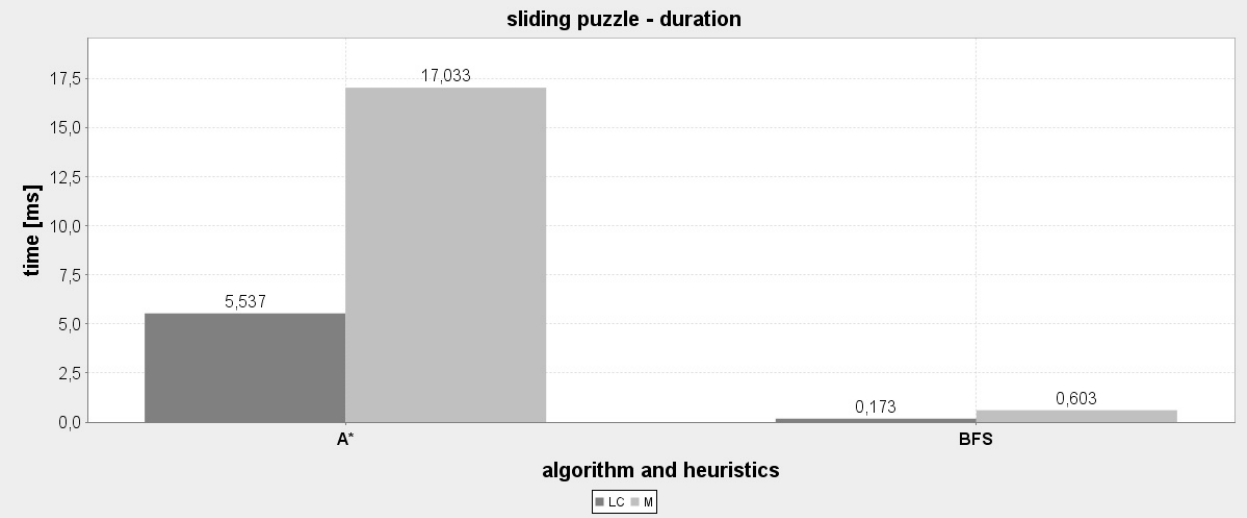


Figure 5.3: Bar chart generated by *SaC* from a batch experiment over random sliding puzzles: duration times for two algorithms (groups) and two heuristics (subgroups).

in figures 5.4, 5.5 and 5.6, respectively.

```
1  // sliding puzzle - duration over open sets (bar chart)
2  StatsBarChart statsBarChart2 = new StatsBarChart(stats, "sliding␣puzzle␣-␣duration", "open␣set␣
       and␣algorithm", "time␣[ms]");
3  statsBarChart2.setValue("A*", "PQ", StatsOperationType.MEAN, StatsCategory.
       GRAPH_SEARCH_DURATION_TIME.toString(), null, AStar.class, null,
4          OpenSetAsPriorityQueue.class, null);
5  statsBarChart2.setValue("BFS", "PQ", StatsOperationType.MEAN, StatsCategory.
       GRAPH_SEARCH_DURATION_TIME.toString(), null, BestFirstSearch.class, null,
6          OpenSetAsPriorityQueue.class, null);
7  statsBarChart2.setValue("A*", "PQFC", StatsOperationType.MEAN, StatsCategory.
       GRAPH_SEARCH_DURATION_TIME.toString(), null, AStar.class, null,
8          OpenSetAsPriorityQueueFastContains.class, null);
9  statsBarChart2.setValue("BFS", "PQFC", StatsOperationType.MEAN, StatsCategory.
       GRAPH_SEARCH_DURATION_TIME.toString(), null, BestFirstSearch.class, null,
10         OpenSetAsPriorityQueueFastContains.class, null);
11 statsBarChart2.setValue("A*", "PQFCFR", StatsOperationType.MEAN, StatsCategory.
       GRAPH_SEARCH_DURATION_TIME.toString(), null, AStar.class, null,
12         OpenSetAsPriorityQueueFastContainsFastReplace.class, null);
13 statsBarChart2.setValue("BFS", "PQFCFR", StatsOperationType.MEAN, StatsCategory.
       GRAPH_SEARCH_DURATION_TIME.toString(), null, BestFirstSearch.class,
14         null, OpenSetAsPriorityQueueFastContainsFastReplace.class, null);
15 statsBarChart2.saveAsJPEG("./sliding_puzzle_duration_2.jpg");
16
17 // sliding puzzle - closed states distribution over optimal path length (xy chart)
18 StatsXYChart statsXYChart1 = new StatsXYChart(stats, "sliding␣puzzle␣-␣closed␣states␣as␣path␣
       length␣grows", "optimal␣path␣length", "closed␣states");
19 statsXYChart1.addSeries("A*_LC", StatsOperationType.MEAN, StatsCategory.
       GRAPH_SEARCH_CLOSED_STATES.toString(), 4, null, AStar.class,
20         HFunctionLinearConflicts.class, null, null);
21 statsXYChart1.addSeries("A*_M", StatsOperationType.MEAN, StatsCategory.GRAPH_SEARCH_CLOSED_STATES
       .toString(), 4, null, AStar.class,
```

```
22          HFunctionManhattan.class, null, null);
23  statsXYChart1.addSeries("BFS_LC", StatsOperationType.MEAN, StatsCategory.
        GRAPH_SEARCH_CLOSED_STATES.toString(), 4, null, BestFirstSearch.class,
24          HFunctionLinearConflicts.class, null, null);
25  statsXYChart1.addSeries("BFS_M", StatsOperationType.MEAN, StatsCategory.
        GRAPH_SEARCH_CLOSED_STATES.toString(), 4, null, BestFirstSearch.class,
26          HFunctionManhattan.class, null, null);
27  statsXYChart1.saveAsJPEG("./sliding_puzzle_closed_states.jpg");
28
29  // sliding puzzle - found path length over optimal path length (xy chart)
30  StatsXYChart statsXYChart2 = new StatsXYChart(stats, "sliding puzzle - found path length over
        optimal path length", "optimal path length",
31          "found path length");
32  statsXYChart2.addSeries("A*_any", StatsOperationType.MEAN, StatsCategory.GRAPH_SEARCH_PATH_LENGTH
        .toString(), 4, null, AStar.class, null, null, null);
33  statsXYChart2.addSeries("BFS_LC", StatsOperationType.MEAN, StatsCategory.GRAPH_SEARCH_PATH_LENGTH
        .toString(), 4, null, BestFirstSearch.class,
34          HFunctionLinearConflicts.class, null, null);
35  statsXYChart2.addSeries("BFS_M", StatsOperationType.MEAN, StatsCategory.GRAPH_SEARCH_PATH_LENGTH.
        toString(), 4, null, BestFirstSearch.class,
36          HFunctionManhattan.class, null, null);
37  statsXYChart2.saveAsJPEG("./sliding_puzzle_path_lengths.jpg");
```

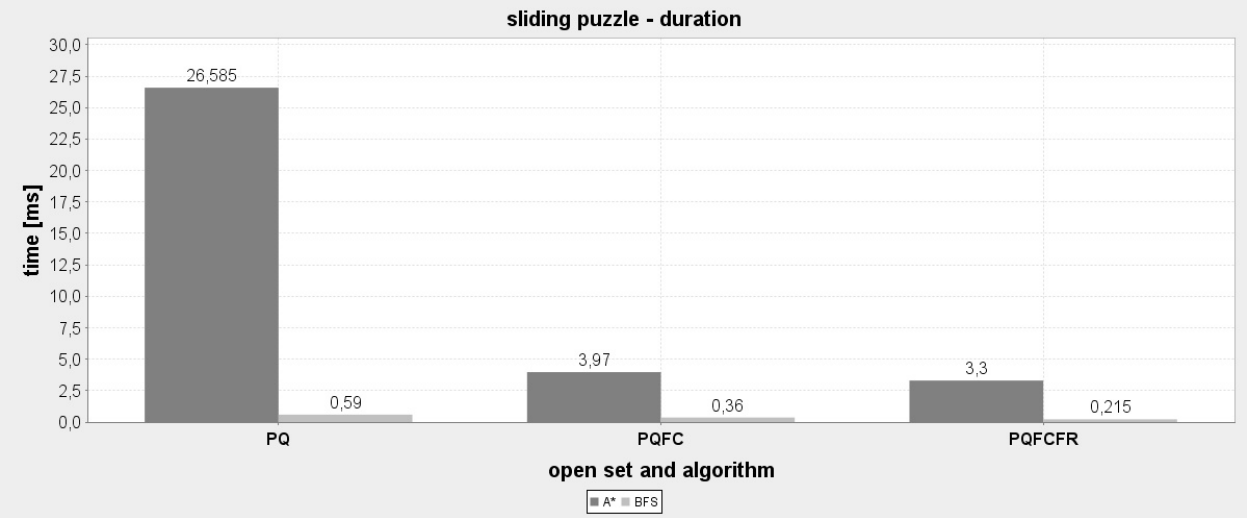


Figure 5.4: Bar chart generated by *SaC* from a batch experiment over random sliding puzzles: duration times for three data structures (groups) and two algorithms (subgroups).

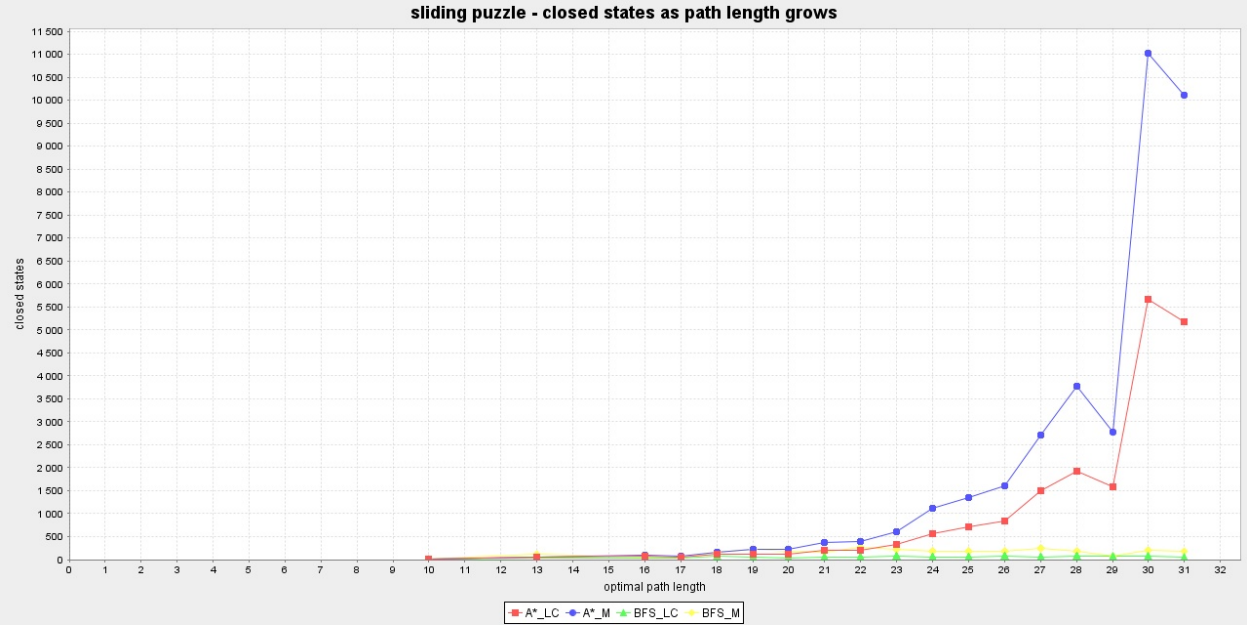


Figure 5.5: *x-y* plot generated by *SaC* from a batch experiment over random sliding puzzles: number of closed (visited) states as a function of solution path length, for different algorithms and heuristics.

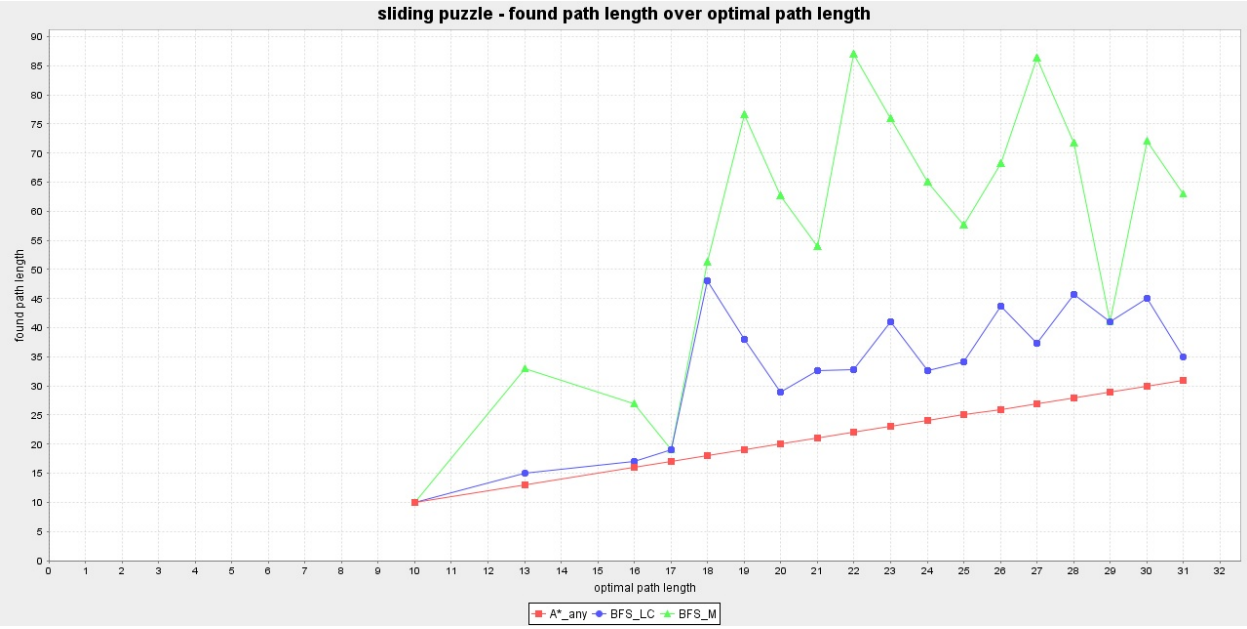


Figure 5.6: *x-y* plot generated by *SaC* from a batch experiment over random sliding puzzles: discovered path lengths vs actual optimal path lengh, for different algorithms and heuristics.

5.3 Graph search monitors

For some sufficiently hard graph search problems, the search procedure may take minutes or even hours. Often, it is impossible for the user to give even a rough estimate on the time duration needed by an algorithm, prior to its execution. In such situations the user is typically interested in monitoring the procedure, i.e. observing its progress and quantities involved. *SaC*'s API allows to do so.

Within the `sac.graph.GraphSearchConfiguration` object, there are three options related to monitors: `monitorOn`, `monitorClassName`, `monitorRefreshTime`. They can be modified either directly by suitable setters or via a properties file. The first option — `monitorOn` — serves as a switch, and is by default set to `false`. It means no monitoring is wanted. Setting this option to `true`, releases the remaining options. The second option — `monitorClassName` — is expected to be set with a name of a particular search monitor class. There are two ready-made monitor classes within *SaC*: `sac.graph.ConsoleGraphSearchMonitor` and `sac.graph.GraphicalGraphSearchMonitor`. Note however, that new custom classes can be developped (e.g. logging the progress to a file or a database, etc.) as long as they extend the base abstract class called: `sac.graph.GraphSearchMonitor`. Finally, the third option — `monitorRefreshTime` — specifies the time period in milliseconds after which the monitor is supposed to refresh its output for the user.

The fragment below shows an example of a console monitor configured for a particular 4×4 sliding puzzle problem. The refresh period is set to 1000 ms, so the monitor prints out a summary of the current progress to the console (every second) by invoking the `printCurrentSummary()` method.

```
1  GraphSearchConfigurator configurator = new GraphSearchConfigurator();
2  configurator.setMonitorOn(true);
3  configurator.setMonitorClassName(ConsoleGraphSearchMonitor.class.getName());
4  configurator.setMonitorRefreshTime(1000);
5
6  SlidingPuzzle puzzle = new SlidingPuzzle(new byte[] {0, 3, 10, 2, 1, 8, 9, 6, 12, 11, 4, 7, 14,
       5, 15, 13});
7  System.out.println(puzzle);
8
9  GraphSearchAlgorithm algorithm = new AStar(puzzle, configurator);
10 algorithm.execute();
11 System.out.println("Done.");
```

```
--------------------
|  0 |  3 | 10 |  2 |
--------------------
|  1 |  8 |  9 |  6 |
--------------------
| 12 | 11 |  4 |  7 |
--------------------
| 14 |  5 | 15 | 13 |
--------------------
[SaC] *** Initializing sac.graph.ConsoleGraphSearchMonitor with params: algorithm = sac.graph.AStar, refreshTime = 1000 ms.
[SaC] *** Starting sac.graph.ConsoleGraphSearchMonitor.printCurrentSummary()...
[SaC] Time: 0.0 s.
[SaC] Solutions so far: 0.
[SaC] Closed states: 1.
[SaC] Open states: 0.
[SaC] Best state's h: 32.0.
[SaC] Best state's f: 32.0.
[SaC] Best state's depth: 0.0.
[SaC] Current state's h: 32.0.
```

```
[SaC] Current state's f: 32.0.
[SaC] Current state's depth: 0.0.
[SaC] Free memory: 42615384
[SaC] Used memory: 128516096
[SaC] Max memory: 1907032064
[SaC] *** Done with sac.graph.ConsoleGraphSearchMonitor.printCurrentSummary().
[SaC] *** Starting sac.graph.ConsoleGraphSearchMonitor.printCurrentSummary()...
[SaC] Time: 1.012 s.
[SaC] Solutions so far: 0.
[SaC] Closed states: 75108.
[SaC] Open states: 70277.
[SaC] Best state's h: 4.0.
[SaC] Best state's f: 40.0.
[SaC] Best state's depth: 36.0.
[SaC] Current state's h: 21.0.
[SaC] Current state's f: 42.0.
[SaC] Current state's depth: 21.0.
[SaC] Free memory: 100827528
[SaC] Used memory: 184942592
[SaC] Max memory: 1907032064
[SaC] *** Done with sac.graph.ConsoleGraphSearchMonitor.printCurrentSummary().
[SaC] *** Starting sac.graph.ConsoleGraphSearchMonitor.printCurrentSummary()...
[SaC] Time: 2.174 s.
[SaC] Solutions so far: 0.
[SaC] Closed states: 121525.
[SaC] Open states: 111129.
[SaC] Best state's h: 3.0.
[SaC] Best state's f: 42.0.
[SaC] Best state's depth: 39.0.
[SaC] Current state's h: 27.0.
[SaC] Current state's f: 44.0.
[SaC] Current state's depth: 17.0.
[SaC] Free memory: 207060480
[SaC] Used memory: 313262080
[SaC] Max memory: 1907032064
[SaC] *** Done with sac.graph.ConsoleGraphSearchMonitor.printCurrentSummary().
[SaC] *** Starting sac.graph.ConsoleGraphSearchMonitor.printCurrentSummary()...
[SaC] Time: 3.245 s.
[SaC] Solutions so far: 0.
[SaC] Closed states: 233229.
[SaC] Open states: 215991.
[SaC] Best state's h: 3.0.
[SaC] Best state's f: 42.0.
[SaC] Best state's depth: 39.0.
[SaC] Current state's h: 22.0.
[SaC] Current state's f: 44.0.
[SaC] Current state's depth: 22.0.
[SaC] Free memory: 193846936
[SaC] Used memory: 362020864
[SaC] Max memory: 1907032064
[SaC] *** Done with sac.graph.ConsoleGraphSearchMonitor.printCurrentSummary().
[SaC] *** Starting sac.graph.ConsoleGraphSearchMonitor.printCurrentSummary()...
[SaC] Time: 4.246 s.
[SaC] Solutions so far: 1.
[SaC] Closed states: 282505.
[SaC] Open states: 258138.
[SaC] Best state's h: 0.0.
[SaC] Best state's f: 44.0.
[SaC] Best state's depth: 44.0.
[SaC] Current state's h: 0.0.
[SaC] Current state's f: 44.0.
[SaC] Current state's depth: 44.0.
[SaC] Free memory: 137477520
[SaC] Used memory: 362020864
[SaC] Max memory: 1907032064
[SaC] *** Done with sac.graph.ConsoleGraphSearchMonitor.printCurrentSummary().
Done.
```

Fig. 5.7 depicts a visualizaiton being generated by the `sac.graph.GraphicalGraphSearchMonitor` for the same sliding puzzle problem. This time the refresh period is set to 100 ms, so the reporting points are laidy more densly than before. An interesting gap can be seen in between the time moments of 1.5 s and 2.3 s. The gap can be explained by exhausted memory needed for the hash map underlying the *Closed* set and the JVM taking time to allocate new enlarged fragment of memory. By doing so, the JVM sets the monitor thread aside.

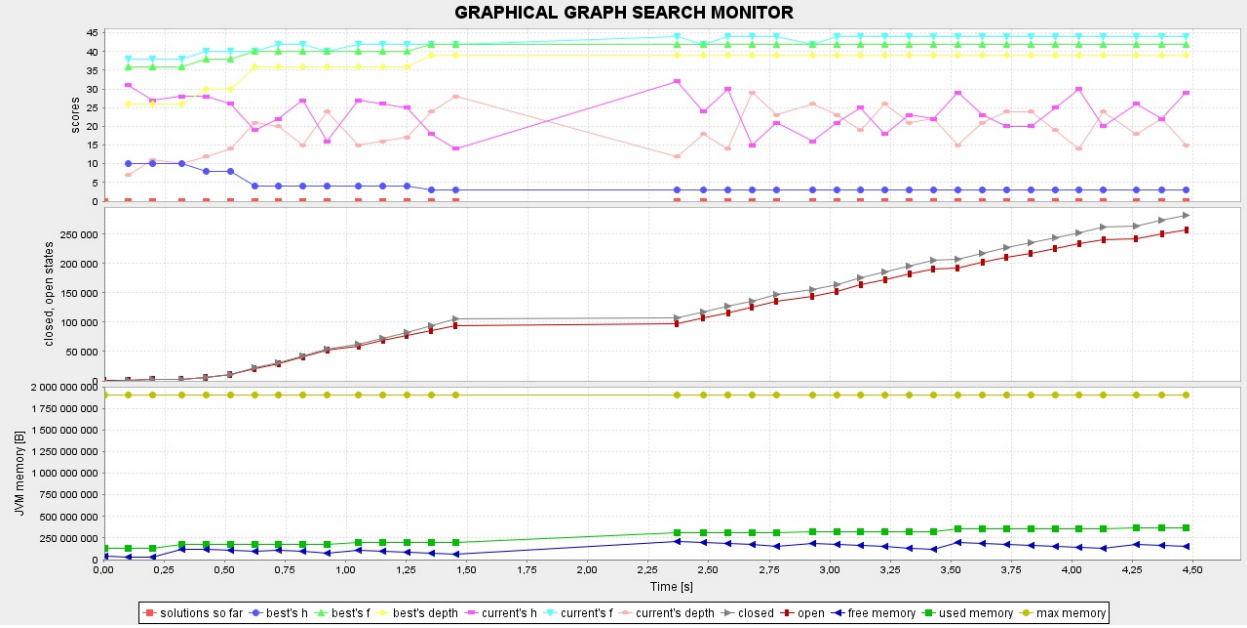


Figure 5.7: A screenshot from a running graph search monitor.

Bibliography

Bayer, R. (1972), 'Symmetric binary B-Trees: Data structure and maintenance algorithms', *Acta Informatica* **1**(4), 290–306.

Brudno, A. (1963), 'Bounds and valuations for shortening the search of estimates', *Problems of Cybernetics (Problemy Kibernetiki)* **10**, 225–241.

Dijkstra, E. (1959), 'A note on two problems in connexion with graphs', *Numerische Mathematik* **1**(1), 269–271.

Edwards, D. and Hart, T. (1963), The Alpha-Beta Heuristic, Technical Report 30, Massachusetts Institute of Technology.

Guibas, L. and Sedgewick, R. (1978), A Dichromatic Framework for Balanced Trees, *in* '19th Annual Symposium on Foundations of Computer Science', pp. 8–21.

Gutin, G., Yeo, A. and Zverovich, A. (2002), 'Traveling salesman should not be greedy: domination analysis of greedy-type heuristics for the TSP', *Discrete Applied Mathematics* **117**(1–3), 81–86.

Hansson, O., Mayer, A. and Yung, M. (1985), Generating Admissible Heuristics by Criticizing Solutions to Relaxed Models, Technical Report CUCS-219-85, Department of Computer Science, Columbia University, New York, N.Y., 10027, USA.

Hart, P., Nilsson, N. and Raphael, B. (1968), 'A Formal Basis for the Heuristic Determination of Minimum Cost Paths', *IEEE Transactions on Systems Science and Cybernetics* **4**(2), 100–107.

Hart, P., Nilsson, N. and Raphael, B. (1972), 'Correction to "A Formal Basis for the Heuristic Determination of Minimum Cost Paths"', *SIGART Bull.* **37**, 28–29.

Johnson, D. and McGeoch, L. (1997), The Traveling Salesman Problem: A Case Study in Local Optimization, *in* E. Aarts and J. Lenstra, eds, 'Local Search in Combinatorial Optimization', Wiley and Sons, London, pp. 215–310.

Kaplan, H., Shafrir, N., Lewenstein, M. and Sviridenko, M. (2003), Approximation Algorithms for Asymmetric TSP by Decomposing Directed Regular Multigraphs, *in* 'Journal of the ACM', pp. 56–65.

Knuth, D. and Moore, R. (1975), 'An analysis of alpha-beta pruning', *Artificial Intelligence* **6**(4), 293–326.

Korf, R. (1985), 'Depth-first Iterative-Deepening: An Optimal Admissible Tree Search', *Artificial Intelligence* **27**, 97–109.

Kruskal, J. (1956), 'On the shortest spanning subtree of a graph and the traveling salesman problem', *Proceedings of the American Mathematical Society* **7**, 48–50.

Marsland, T. (1983), Relative efficiency of alpha-beta implementations, *in* 'Proceedings of the eighth international joint conference on artificial intelligence', Vol. 2, Morgan Kaufman Publishers Inc., San Francisco, CA, USA, pp. 763–766.

Newell, A. and Simon, A. (1976), 'Computer Science and Empirical Inquiry: Symbols and Search', *Commununications of the ACM* **19**(3), 113–126.

Pearl, J. (1980), A Simple Game-Searching Algorithm with Proven Optimal Properties, *in* 'First Annual National Conference on Artificial Intelligence', Stanford, USA.

Pearl, J. (1982), 'The Solution for the Branching Factor of the Alpha-beta Pruning Algorithm and Its Optimality', *Commununications of the ACM* **25**(8), 559–564.

Pearl, J. (1984), *Heuristics: Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Rego, C., Gamboa, D., Glover, F. and Osterman, C. (2011), 'Traveling salesman problem heuristics: Leading methods, implementations and latest advances', *European Journal of Operational Research* **211**(3), 427–441.

Reinefeld, A. (1983), 'An Improvement to the Scout Tree-Search Algorithm', *International Computer Chess Association Journal* **6**(4), 4–14.

Samuel, A. (1983), 'Some Studies in Machine Learning Using the Game of Checkers', *IBM Journal of Research and Development* **3**(3), 210–229.

Shortz, W. (2005), *Sudoku Easy*, St. Martin's Griffin.

von Neumann, J. (1928), 'Zur Theorie der Gesellschaftsspiele', *Mathematische Annalen* **100**(1), 295–320.

von Neumann, J. and Morgenstern, O. (1944), *Theory of Games and Economic Behavior*, Princeton University Press.

Wikipedia (2014), 'Sudoku', `http://en.wikipedia.org/wiki/Sudoku`. Accessed: 28.12.2014.

Chapter 6

Appendices

6.1 Implementation of state abstraction

```java
package sac;

import java.util.Collections;
import java.util.LinkedList;
import java.util.List;

/**
 * Abstract partial implementation of State interface.
 */
public abstract class StateImpl implements State {

    /**
     * Identifier for this state.
     */
    protected Identifier identifier = null;

    /**
     * Reference to this state's parent.
     */
    protected State parent = null;

    /**
     * List of references to this state's children.
     */
    protected List<? extends State> children = null;

    /**
     * Depth of this state (number of parent states above it).
     */
    protected double depth = 0;

    /**
     * The heuristics - estimated distance to the solution state. Remains null until the first call of getH().
     */
    protected Double h = null;

    /**
     * Name of the move that led to generating this state.
     */
    protected String moveName;

    /**
     * Default h function (returns 0).
     */
    protected static StateFunction hFunction;

```

```
47 |     /**
48 |      * Constructor for this abstract class. Sets reference to parent to null, and initializes children as an empty list
49 |      * (linked list).
50 |      */
51 |     public StateImpl() {
52 |         // construction of identifier is postponed until first call of getIdentifier()
53 |         this.parent = null;
54 |         this.children = new LinkedList<State>();
55 |     }
56 |
57 |     /*
58 |      * (non-Javadoc)
59 |      *
60 |      * @see sac.State#getIdentifier()
61 |      */
62 |     @Override
63 |     public final Identifier getIdentifier() {
64 |         if (identifier == null) { // first call for identifier
65 |             // at this point toString() and hashCode() methods for classes extending StateImpl are
66 |             // ready to be used,
67 |             // identifier will call the suitable method once (on construction) and memorize the
68 |             // result
69 |             identifier = new Identifier(this);
70 |         }
71 |         return identifier;
72 |     }
73 |
74 |     /*
75 |      * (non-Javadoc)
76 |      *
77 |      * @see sac.State#refreshIdentifier()
78 |      */
79 |     @Override
80 |     public final void refreshIdentifier() {
81 |         identifier = new Identifier(this);
82 |     }
83 |
84 |     /*
85 |      * (non-Javadoc)
86 |      *
87 |      * @see sac.State#getParent()
88 |      */
89 |     @Override
90 |     public State getParent() {
91 |         return parent;
92 |     }
93 |
94 |     /*
95 |      * (non-Javadoc)
96 |      *
97 |      * @see sac.State#setParent(sac.State)
98 |      */
99 |     @Override
100|     public final void setParent(State parent) {
101|         this.parent = parent;
102|     }
103|
104|     /*
105|      * (non-Javadoc)
106|      *
107|      * @see sac.State#setDepth(double)
108|      */
109|     @Override
110|     public final void setDepth(double depth) {
111|         this.depth = depth;
112|     }
113|
114|     /*
115|      * (non-Javadoc)
116|      *
117|      * @see sac.State#getChildren()
118|      */
119|     @Override
120|     public List<? extends State> getChildren() {
```

```
121 |         return children;
122 |     }
123 |
124 |     /*
125 |      * (non-Javadoc)
126 |      *
127 |      * @see sac.State#getDepth()
128 |      */
129 |     @Override
130 |     public final double getDepth() {
131 |         return depth;
132 |     }
133 |
134 |     /*
135 |      * (non-Javadoc)
136 |      *
137 |      * @see sac.State#getPath()
138 |      */
139 |     @Override
140 |     public List<? extends State> getPath() {
141 |         List<State> path = new LinkedList<State>();
142 |         State temp = this;
143 |         path.add(temp);
144 |         while (temp.getParent() != null) {
145 |             temp = temp.getParent();
146 |             path.add(temp);
147 |         }
148 |         Collections.reverse(path);
149 |         return path;
150 |     }
151 |
152 |     /*
153 |      * (non-Javadoc)
154 |      *
155 |      * @see sac.State#getMovesAlongPath()
156 |      */
157 |     @Override
158 |     public List<String> getMovesAlongPath() {
159 |         List<String> moves = new LinkedList<String>();
160 |         List<? extends State> path = getPath();
161 |         for (State state : path) {
162 |             if (state.getParent() == null) continue;
163 |             moves.add(state.getMoveName());
164 |         }
165 |         return moves;
166 |     }
167 |
168 |     /*
169 |      * (non-Javadoc)
170 |      *
171 |      * @see java.lang.Object#equals(java.lang.Object)
172 |      */
173 |     @Override
174 |     public boolean equals(Object otherState) {
175 |         State otherState2 = (State) otherState;
176 |         return getIdentifier().equals(otherState2.getIdentifier());
177 |     }
178 |
179 |     /*
180 |      * (non-Javadoc)
181 |      *
182 |      * @see java.lang.Comparable#compareTo(java.lang.Object)
183 |      */
184 |     @Override
185 |     public int compareTo(State otherState) {
186 |         return getIdentifier().compareTo(otherState.getIdentifier());
187 |     }
188 |
189 |     /**
190 |      * Sets new h function.
191 |      *
192 |      * @param hFunction to best
193 |      */
194 |     public static final void setHFunction(StateFunction hFunction) {
```

```
195         StateImpl.hFunction = hFunction;
196     }
197 
198     /*
199      * (non-Javadoc)
200      *
201      * @see sac.State#getH()
202      */
203     @Override
204     public final double getH() {
205         if (h == null)
206             h = Double.valueOf(hFunction.calculate(this));
207         return h;
208     }
209 
210     /*
211      * (non-Javadoc)
212      *
213      * @see sac.State#setH(Double)
214      */
215     @Override
216     public final void setH(Double h) {
217         this.h = h;
218     }
219 
220     /*
221      * (non-Javadoc)
222      *
223      * @see sac.State#refreshH()
224      */
225     @Override
226     public final void refreshH() {
227         h = null;
228         hFunction.calculate(this);
229     }
230 
231     /*
232      * (non-Javadoc)
233      *
234      * @see sac.State#refresh()
235      */
236     @Override
237     public void refresh() {
238         refreshIdentifier();
239         refreshH();
240     }
241 
242     /*
243      * (non-Javadoc)
244      *
245      * @see sac.State#getMoveName()
246      */
247     @Override
248     public final String getMoveName() {
249         return (moveName != null) ? moveName : getIdentifier().toString();
250     }
251 
252     /*
253      * (non-Javadoc)
254      *
255      * @see sac.State#setMoveName(java.lang.String)
256      */
257     @Override
258     public final void setMoveName(String moveName) {
259         this.moveName = moveName;
260     }
261 
262     /*
263      * (non-Javadoc)
264      *
265      * @see sac.graphviz.Graphvizable#toGraphvizLabel()
266      */
267     @Override
268     public String toGraphvizLabel() {
```

```
269         return getIdentifier().toString(); // default content for visualization - identifier
270     }
271 
272     static {
273         hFunction = new StateFunction();
274     }
275 }
```

6.2 Implementation of graph state abstraction

```
1  package sac.graph;
2  
3  import java.util.List;
4  
5  import sac.State;
6  import sac.StateFunction;
7  import sac.StateImpl;
8  
9  /**
10  * Abstract partial implementation of GraphState interface. User's graph state classes should extend this class.
11  */
12 public abstract class GraphStateImpl extends StateImpl implements GraphState {
13 
14     /**
15      * The exact distance from the initial state.
16      */
17     protected Double g = 0.0;
18 
19     /**
20      * The sum of g and h. Remains null until the first call of getF().
21      */
22     protected Double f = null;
23 
24     /**
25      * Default g function (returns parent's g + 1).
26      */
27     protected static StateFunction gFunction;
28 
29     /**
30      * Default implementation of the true cost function (g function). Returns parent's g (if exists) plus one. Suitable
31      * for problems where the number of moves is to be minimized.
32      */
33     public static class GFunction extends StateFunction {
34         /*
35          * (non-Javadoc)
36          *
37          * @see sac.StateFunction#calculate(sac.State)
38          */
39         @Override
40         public double calculate(State state) {
41             return (state.getParent() == null) ? 0.0 : ((GraphState) state.getParent()).getG() + 1.0;
42         }
43     }
44 
45     /**
46      * Sets new g function.
47      *
48      * @param gFunction to be set
49      */
50     public final static void setGFunction(StateFunction gFunction) {
51         GraphStateImpl.gFunction = gFunction;
52     }
53 
54     /**
55      * Creates a new instance of graph state.
56      */
57     public GraphStateImpl() {
58         super();
```

```
59      }
60
61      /*
62       * (non-Javadoc)
63       *
64       * @see sac.GraphState#getParent()
65       */
66      @Override
67      public final GraphState getParent() {
68          return (parent == null) ? null : (GraphState) parent;
69      }
70
71      /*
72       * (non-Javadoc)
73       *
74       * @see sac.GraphState#getChildren()
75       */
76      @Override
77      @SuppressWarnings("unchecked")
78      public final List<GraphState> getChildren() {
79          return (List<GraphState>) children;
80      }
81
82      /*
83       * (non-Javadoc)
84       *
85       * @see sac.GraphState#getPath()
86       */
87      @Override
88      @SuppressWarnings("unchecked")
89      public final List<GraphState> getPath() {
90          return (List<GraphState>) super.getPath();
91      }
92
93      /*
94       * (non-Javadoc)
95       *
96       * @see sac.GraphState#getG()
97       */
98      @Override
99      public final double getG() {
100         if (g == null)
101             g = Double.valueOf(gFunction.calculate(this));
102         return g;
103     }
104
105     /*
106      * (non-Javadoc)
107      *
108      * @see sac.GraphState#getF()
109      */
110     @Override
111     public final double getF() {
112         if (f == null)
113             f = Double.valueOf(getG() + getH());
114         return f;
115     }
116
117     /*
118      * (non-Javadoc)
119      *
120      * @see sac.graph.GraphState#refreshCosts()
121      */
122     @Override
123     public final void refreshCosts() {
124         g = Double.valueOf(gFunction.calculate(this));
125         h = Double.valueOf(hFunction.calculate(this));
126         f = Double.valueOf(getG() + getH());
127     }
128
129     static {
130         gFunction = new GFunction();
131     }
132 }
```

6.3 Implementation of game state abstraction

```
1  package sac.game;
2
3  import java.util.LinkedList;
4  import java.util.List;
5
6  import sac.StateImpl;
7
8  /**
9   * Abstract partial implementation of GameState interface. User game state classes should extend this class.
10  */
11 public abstract class GameStateImpl extends StateImpl implements GameState {
12
13     /**
14      * Boolean value indicating whether it is the maximizing player turn to play now.
15      */
16     protected boolean maximizingTurnNow = true;
17
18     /**
19      * List of moves along principal variation.
20      */
21     protected List<String> movesAlongPrincipalVariation = null;
22
23     /**
24      * Boolean value indicating whether this state instance is flagged as visited during search. It is possible that
25      * copies of the same state in different places of search tree will have different 'visited' flags. Some of copies
26      * might be generated but cut off, or read from the transposition table. This flag is memorized only for informative
27      * purposes (in particular for Graphviz functionality).
28      */
29     protected boolean visited = false;
30
31     /**
32      * Boolean value indicating whether this state instance is flagged as read (evaluated) from transposition table. It
33      * is possible that copies of the same state in different places of search tree will have different
34      * 'readFromTranspositionTable' flags. This flag is memorized only for informative purposes (in particular for
35      * Graphviz functionality).
36      */
37     protected boolean readFromTranspositionTable = false;
38
39     /**
40      * Constructor for this abstract class. Initializes list for moves along principal variation as an empty list
41      * (linked list).
42      */
43     public GameStateImpl() {
44         movesAlongPrincipalVariation = new LinkedList<String>();
45     }
46
47     /*
48      * (non-Javadoc)
49      *
50      * @see sac.game.GameState#getParent()
51      */
52     @Override
53     public GameState getParent() {
54         return (parent == null) ? null : (GameState) parent;
55     }
56
57     /*
58      * (non-Javadoc)
59      *
60      * @see sac.game.GameState#getChildren()
61      */
62     @Override
63     @SuppressWarnings("unchecked")
64     public final List<GameState> getChildren() {
65         return (List<GameState>) children;
66     }
67
68     /*
69      * (non-Javadoc)
```

```
70  |      *
71  |      * @see sac.game.GameState#getPath()
72  |      */
73  |     @Override
74  |     @SuppressWarnings("unchecked")
75  |     public final List<GameState> getPath() {
76  |         return (List<GameState>) super.getPath();
77  |     }
78  |
79  |     /*
80  |      * (non-Javadoc)
81  |      *
82  |      * @see sac.game.GameState#isMaximizingTurnNow()
83  |      */
84  |     @Override
85  |     public final boolean isMaximizingTurnNow() {
86  |         return maximizingTurnNow;
87  |     }
88  |
89  |     /*
90  |      * (non-Javadoc)
91  |      *
92  |      * @see sac.game.GameState#setMaximizingTurnNow(boolean)
93  |      */
94  |     @Override
95  |     public final void setMaximizingTurnNow(boolean maximizingTurnNow) {
96  |         this.maximizingTurnNow = maximizingTurnNow;
97  |     }
98  |
99  |     /*
100 |      * (non-Javadoc)
101 |      *
102 |      * @see sac.game.GameState#isQuiet()
103 |      */
104 |     @Override
105 |     public boolean isQuiet() {
106 |         return true; // default implementation
107 |     }
108 |
109 |     /*
110 |      * (non-Javadoc)
111 |      *
112 |      * @see sac.game.GameState#isVisited()
113 |      */
114 |     @Override
115 |     public final boolean isVisited() {
116 |         return visited;
117 |     }
118 |
119 |
120 |     /*
121 |      * (non-Javadoc)
122 |      *
123 |      * @see sac.game.GameState#setVisited()
124 |      */
125 |     @Override
126 |     public final void setVisited(boolean visited) {
127 |         this.visited = visited;
128 |     }
129 |
130 |     /*
131 |      * (non-Javadoc)
132 |      *
133 |      * @see sac.game.GameState#isReadFromTranspositionTable()
134 |      */
135 |     @Override
136 |     public boolean isReadFromTranspositionTable() {
137 |         return readFromTranspositionTable;
138 |     }
139 |
140 |     /*
141 |      * (non-Javadoc)
142 |      *
143 |      * @see sac.game.GameState#setReadFromTranspositionTable()
```

```
     */
    @Override
    public void setReadFromTranspositionTable(boolean readFromTranspositionTable) {
        this.readFromTranspositionTable = readFromTranspositionTable;
    }

    /*
     * (non-Javadoc)
     *
     * @see sac.game.GameState#getMovesAlongPrincipalVariation()
     */
    @Override
    public final List<String> getMovesAlongPrincipalVariation() {
        return movesAlongPrincipalVariation;
    }

    @Override
    public boolean isNonWinTerminal() {
        return false;
    }
}
```

6.4 Full code of general (abstract) game search algorithm

```
package sac.game;

import java.lang.reflect.Constructor;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import sac.Identifier;
import sac.SearchAlgorithm;

/**
 * Abstract game search algorithm. Meant to be extended by actual algorithms e.g.: MIN-MAX, alpha-beta cut-offs, Scout.
 */
public abstract class GameSearchAlgorithm extends SearchAlgorithm {

    /**
     * Reference to initial state.
     */
    protected GameState initial = null;

    /**
     * Reference to currently examined state.
     */
    protected GameState current = null;

    /**
     * Map of discovered scores for moves.
     */
    protected Map<String, Double> movesScores = null;

    /**
     * Transposition table.
     */
    protected TranspositionTable transpositionTable = null;

    /**
     * Refutation table.
     */
    protected RefutationTable refutationTable = null;

    /**
     * Graph search configurator object.
     */
```

```
45      protected GameSearchConfigurator configurator = null;
46
47      /**
48       * Number of closed states (= number of calls of methods evaluateMaxState(), evaluateMinState), since last reset().
49       */
50      protected int closedCount = 0;
51
52      /**
53       * Maximum depth that was reached in the search (owing to quiescence) since last reset().
54       */
55      protected double depthReached = 0.0;
56
57      /**
58       * Boolean flag stating if stop was forced (e.g. from an outer thread).
59       */
60      protected boolean stopForced = false;
61
62      /**
63       * Creates new instance of game search algorithm.
64       *
65       * @param initial reference to initial state
66       * @param configurator reference to configurator object
67       */
68      public GameSearchAlgorithm(GameState initial, GameSearchConfigurator configurator) {
69          this.configurator = (configurator != null) ? configurator : new GameSearchConfigurator();
70
71          this.initial = initial;
72          this.movesScores = new HashMap<String, Double>();
73
74          reset();
75      }
76
77      /*
78       * (non-Javadoc)
79       *
80       * @see sac.SearchAlgorithm#execute()
81       */
82      @Override
83      public final void execute() {
84          reset();
85          startTime = System.currentTimeMillis();
86          doExecute();
87          endTime = System.currentTimeMillis();
88      }
89
90      /**
91       * Actual execution of search algorithm invoked from inside of execute() method.
92       */
93      protected void doExecute() {
94          Double gameValue = null;
95          if (initial.isMaximizingTurnNow())
96              gameValue = evaluateMaxState(initial, Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY, 0.0, configurator.
                        getDepthLimit());
97          else
98              gameValue = evaluateMinState(initial, Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY, 0.0, configurator.
                        getDepthLimit());
99          if (configurator.isTranspositionTableOn())
100             transpositionTable.putOrUpdate(initial, gameValue, Double.NEGATIVE_INFINITY, Double.POSITIVE_INFINITY);
101         current = null;
102     }
103
104     /**
105      * Wrapping method around doEvaluateMaxState(...).
106      *
107      * @param gameState given game state
108      * @param alpha lower bound on game value known for given game state
109      * @param beta upper bound on game value known for given game state
110      * @param depth current depth
111      * @param depthLimit depth limit
112      * @return calculated game value (or null if time limit is reached)
113      */
114     protected final Double evaluateMaxState(GameState gameState, double alpha, double beta, double depth, double depthLimit) {
115         // time limit check
116         if ((stopForced) || (configurator.getTimeLimit() < Long.MAX_VALUE)) {
```

```
117             long currentTime = System.currentTimeMillis();
118             if (currentTime - startTime > configurator.getTimeLimit()) {
119                 endTime = System.currentTimeMillis();
120                 return null;
121             }
122         }
123         closedCount++;
124         current = gameState;
125         gameState.setVisited(true);
126         return doEvaluateMaxState(gameState, alpha, beta, depth, depthLimit);
127     }
128 
129     /**
130      * Wrapping method around doEvaluateMinState(...).
131      *
132      * @param gameState given game state
133      * @param alpha lower bound on game value known for given game state
134      * @param beta upper bound on game value known for given game state
135      * @param depth current depth
136      * @param depthLimit depth limit
137      * @return calculated game value (or null if time limit is reached)
138      */
139     protected final Double evaluateMinState(GameState gameState, double alpha, double beta, double depth, double depthLimit) {
140         // time limit check
141         if ((stopForced) || (configurator.getTimeLimit() < Long.MAX_VALUE)) {
142             long currentTime = System.currentTimeMillis();
143             if (currentTime - startTime > configurator.getTimeLimit()) {
144                 endTime = System.currentTimeMillis();
145                 return null;
146             }
147         }
148         closedCount++;
149         current = gameState;
150         gameState.setVisited(true);
151         return doEvaluateMinState(gameState, alpha, beta, depth, depthLimit);
152     }
153 
154     /**
155      * Evaluates given game state associated with the maximizing player.
156      *
157      * @param gameState given game state
158      * @param alpha lower bound on game value known for given game state
159      * @param beta upper bound on game value known for given game state
160      * @param depth current depth
161      * @param depthLimit depth limit
162      * @return calculated game value (or null if time limit is reached)
163      */
164     public abstract Double doEvaluateMaxState(GameState gameState, double alpha, double beta, double depth, double depthLimit);
165 
166     /**
167      * Evaluates given game state associated with the minimizing player.
168      *
169      * @param gameState given game state
170      * @param alpha lower bound on game value known for given game state
171      * @param beta upper bound on game value known for given game state
172      * @param depth current depth
173      * @param depthLimit depth limit
174      * @return calculated game value (or null if time limit is reached)
175      */
176     public abstract Double doEvaluateMinState(GameState gameState, double alpha, double beta, double depth, double depthLimit);
177 
178     /**
179      * Resets this algorithm. I.e. resets initial state (cuts off its children, if they exist), clears moves scores,
180      * clears transposition table. Refutation table remains not cleared due to its purpose - the progressive search.
181      */
182     @SuppressWarnings("unchecked")
183     protected void reset() {
184         stopForced = false;
185 
186         // identifiers
187         Identifier.setType(this.configurator.getIdentifierType()); // in case, it changed since last
188                                                                    // execute() call
189         // root of the tree resetting
190         if (initial != null) {
```

```
            initial.refresh();
            initial.setParent(null);
            initial.setMoveName("");
            initial.setDepth(0.0);
            initial.getChildren().clear();
            recalculateHIfLarge(initial);
        }

        // clearing moves scores
        movesScores.clear();

        // transpostion table
        try {
            Constructor<TranspositionTable> constructor = (Constructor<TranspositionTable>) Class.forName(configurator.
                  getTranspositionTableClassName())
                     .getConstructor();
            transpositionTable = (TranspositionTable) constructor.newInstance();
        } catch (Exception e) {
            transpositionTable = new TranspositionTableAsHashMap();
            e.printStackTrace();
        }

        // refutation table
        if (refutationTable == null) {
            try {
                Constructor<RefutationTable> constructor = (Constructor<RefutationTable>) Class.forName(configurator.
                      getRefutationTableClassName())
                         .getConstructor(Double.TYPE);
                refutationTable = (RefutationTable) constructor.newInstance(configurator.getRefutationTableDepthLimit());
            } catch (Exception e) {
                refutationTable = new RefutationTableAsHashMap();
                e.printStackTrace();
            }
        } else
            refutationTable.reset();

        closedCount = 0;
        depthReached = 0.0;
    }

    /**
     * Returns the map with scores of moves.
     *
     * @return map with scores of moves
     */
    public Map<String, Double> getMovesScores() {
        return movesScores;
    }

    /**
     * Returns the best move, taking into account the player to move first (maximizing or minimizing) at initial state.
     * If more than one best move (with equal value) is available returns the first one that occured in the scores map.
     * It is possible that this method returns a null if time limit has been reached and no move score has been
     * evaluated so far.
     *
     * @return best move
     */
    public final String getFirstBestMove() {
        double factor = ((initial != null) && (initial.isMaximizingTurnNow())) ? 1 : -1;
        String bestMove = null;
        double bestMoveValue = Double.NEGATIVE_INFINITY * factor;

        for (String move : movesScores.keySet()) {
            double value = movesScores.get(move);
            if (value * factor > bestMoveValue * factor) {
                bestMove = move;
                bestMoveValue = value;
            }
        }

        if ((bestMove == null) && (!movesScores.isEmpty())) // all moves are plus or minus
                                                            // infinities, taking the first
            bestMove = movesScores.keySet().iterator().next();
        return bestMove;
```

```
263     }
264 
265     /**
266      * Returns list of all best moves (with equal highest score). It is possible that this method returns an empty list
267      * if time limit has been reached and no move score has been evaluated so far.
268      *
269      * @return list of all best moves (with equal highest score)
270      */
271     public final List<String> getBestMoves() {
272         double factor = ((initial != null) && (initial.isMaximizingTurnNow())) ? 1 : -1;
273         double bestMoveValue = Double.NEGATIVE_INFINITY * factor;
274 
275         for (String move : movesScores.keySet()) {
276             double value = movesScores.get(move);
277             if (value * factor >= bestMoveValue * factor)
278                 bestMoveValue = value;
279         }
280 
281         List<String> bestMoves = new ArrayList<String>();
282         for (String move : movesScores.keySet()) {
283             double value = movesScores.get(move);
284             if (value * factor == bestMoveValue * factor)
285                 bestMoves.add(move);
286         }
287 
288         return bestMoves;
289     }
290 
291     /**
292      * Returns reference to initial game state.
293      *
294      * @return reference to initial game state
295      */
296     public final GameState getInitial() {
297         return initial;
298     }
299 
300     /**
301      * Sets reference to initial game state.
302      *
303      * @param initial reference to initial game state
304      */
305     public final void setInitial(GameState initial) {
306         this.initial = initial;
307     }
308 
309 
310     /**
311      * Returns reference to currently examined state.
312      *
313      * @return reference to currently examined state
314      */
315     public GameState getCurrent() {
316         return current;
317     }
318 
319     /**
320      * Returns reference to transposition table.
321      *
322      * @return reference to transposition table
323      */
324     public final TranspositionTable getTranspositionTable() {
325         return transpositionTable;
326     }
327 
328     /**
329      * Returns reference to refutation table.
330      *
331      * @return reference to refutation table
332      */
333     public final RefutationTable getRefutationTable() {
334         return refutationTable;
335     }
336 
```

```
337     /**
338      * Gets reference to configurator object.
339      *
340      * @return reference to configurator object
341      */
342     public final GameSearchConfigurator getConfigurator() {
343         return configurator;
344     }
345 
346     /**
347      * Sets reference to configurator object.
348      *
349      * @param configurator reference to configurator object
350      */
351     public final void setConfigurator(GameSearchConfigurator configurator) {
352         this.configurator = configurator;
353     }
354 
355     @Override
356     public final int getClosedStatesCount() {
357         return closedCount;
358     }
359 
360     /**
361      * Returns the maximum depth that was reached in the search (owing to quiescence) since last reset().
362      *
363      * @return maximum depth that was reached in the search
364      */
365     public final double getDepthReached() {
366         return depthReached;
367     }
368 
369     /**
370      * Calls parent.generateChildren() method and increments depths of children by 0.5 with respect to their parent.
371      *
372      * @param parent reference to game state object for which children should be generated
373      */
374     protected final List<GameState> generateChildrenWrapper(GameState parent) {
375         List<GameState> children = parent.generateChildren();
376         for (GameState child : children) {
377             child.setParent(parent);
378             child.setDepth(parent.getDepth() + 0.5);
379             if (configurator.isParentsMemorizingChildren())
380                 parent.getChildren().add(child);
381             recalculateHIfLarge(child);
382         }
383         return children;
384     }
385 
386     /**
387      * Recalculates heuristic value for given state if its absolute value is greater than H_SMALLEST_INFINITY constant. The
388      * recalculation is done according to the formula: h = Math.signum(h) * H_SMALLEST_INFINITY * (1.0 + 1.0 /
389      * state.getDepth()).
390      *
391      * @param state reference to game state for which the recalculation is executed
392      */
393     protected final static void recalculateHIfLarge(GameState state) {
394         double h = state.getH();
395         if (Math.abs(h) > GameState.H_SMALLEST_INFINITY) {
396             h = Math.signum(h) * GameState.H_SMALLEST_INFINITY * (1.0 + 1.0 / state.getDepth());
397             state.setH(h);
398         }
399     }
400 
401     /**
402      * Returns a boolean flag showing if given state is a terminal (leaf).
403      *
404      * @param gameState reference to game state
405      * @param depth current depth
406      * @param depthLimit depth limit
407      * @return boolean flag showing if given state is a terminal (leaf)
408      */
409     public final boolean isGameStateTerminal(GameState gameState, double depth, double depthLimit) {
410         depthReached = Math.max(depthReached, depth);
```

```
        if ((gameState.isWinTerminal()) || (gameState.isNonWinTerminal()))
            return true;
        if (depth >= depthLimit)
            return (configurator.isQuiescenceOn()) ? gameState.isQuiet() : true;
        return false;
    }

    /**
     * Returns a boolean flag stating if given value of a child is an exact game value for given alpha-beta window.
     *
     * @param childValue value
     * @param alpha alpha value
     * @param beta beta value
     * @return boolean flag stating if given value of a child is an exact game value for given alpha-beta window
     */
    public final static boolean isExactGameValue(double childValue, double alpha, double beta) {
        return ((alpha < childValue) && (childValue < beta)) || ((childValue == Double.NEGATIVE_INFINITY) && (alpha == Double.
            NEGATIVE_INFINITY))
               || ((childValue == Double.POSITIVE_INFINITY) && (beta == Double.POSITIVE_INFINITY));
    }

    /**
     * Updates list of moves along principal variation for given parent and child (that led to an improvement). Resulting
     * list of moves along principal variation consists of: parent move name (if not empty) and list of moves along
     * principal variation for child.
     *
     * @param parent reference to parent
     * @param child reference to child
     */
    public final static void updateMovesAlongPrincipalVariation(GameState parent, GameState child) {
        List<String> movesAlongPrincipalVariation = parent.getMovesAlongPrincipalVariation();
        movesAlongPrincipalVariation.clear();
        movesAlongPrincipalVariation.add(child.getMoveName());
        movesAlongPrincipalVariation.addAll(child.getMovesAlongPrincipalVariation());
    }

    /**
     * Forces current execute() recursion to stop.
     */
    public final void forceStop() {
        stopForced = true;
    }
}
```

6.5 MST implementation using Kruskal's algorithm for the TSP solver

```
package sac.examples.tsp;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;
import java.util.SortedSet;
import java.util.TreeSet;

/**
 * Minimum Spanning Tree solver via Kruskal's algorithm, used in TSP as heuristics.
 */
public class MinimumSpanningTree {

    /**
     * List of connections.
     */
    private List<Connection> connections;

    /**
     * Cost of the minimum spanning tree.
     */
    private double cost;
```

```
23 |
24 |     /**
25 |      * Creates a new instance of minimum spanning tree as a copy.
26 |      *
27 |      * @param toCopy minimum spanning tree to be copied
28 |      */
29 |     public MinimumSpanningTree(MinimumSpanningTree toCopy) {
30 |         connections = new ArrayList<Connection>();
31 |         connections.addAll(toCopy.connections);
32 |         cost = toCopy.cost;
33 |     }
34 |
35 |     /**
36 |      * Creates a minimum spanning tree from a collection of places. The Kruskal's algorithm is performed inside this
37 |      * constructor.
38 |      *
39 |      * @param places collection of places (as a SortedSet).
40 |      */
41 |     public MinimumSpanningTree(SortedSet<Place> places) {
42 |         connections = new ArrayList<Connection>();
43 |         cost = 0.0;
44 |
45 |         SortedSet<Connection> orderedConnections = new TreeSet<Connection>(new ConnectionCostComparator());
46 |
47 |         // building sorted set of all valid connections
48 |         for (Place place : places) {
49 |             for (Connection connection : place.getConnections()) {
50 |                 if (!places.contains(connection.getPlace1()))
51 |                     continue;
52 |                 if (!places.contains(connection.getPlace2()))
53 |                     continue;
54 |                 orderedConnections.add(connection);
55 |             }
56 |         }
57 |
58 |         SortedSet<Place> placesInsideMSP = new TreeSet<Place>();
59 |
60 |         if (orderedConnections.isEmpty())
61 |             return;
62 |
63 |         // adding first connection
64 |         Connection firstConnection = orderedConnections.first();
65 |         connections.add(firstConnection);
66 |         cost += firstConnection.getCost();
67 |         placesInsideMSP.add(firstConnection.getPlace1());
68 |         placesInsideMSP.add(firstConnection.getPlace2());
69 |         orderedConnections.remove(firstConnection);
70 |
71 |         while (placesInsideMSP.size() < places.size()) {
72 |             Iterator<Connection> iterator = orderedConnections.iterator();
73 |             while (iterator.hasNext()) {
74 |                 Connection connection = iterator.next();
75 |                 boolean place1InsideMSP = placesInsideMSP.contains(connection.getPlace1());
76 |                 boolean place2InsideMSP = placesInsideMSP.contains(connection.getPlace2());
77 |                 if (((place1InsideMSP) && (!place2InsideMSP)) || ((!place1InsideMSP) && (place2InsideMSP))) {
78 |                     connections.add(connection);
79 |                     cost += connection.getCost();
80 |                     placesInsideMSP.add(connection.getPlace1());
81 |                     placesInsideMSP.add(connection.getPlace2());
82 |                     iterator.remove();
83 |                     break;
84 |                 }
85 |                 if ((place1InsideMSP) && (place2InsideMSP))
86 |                     iterator.remove();
87 |             }
88 |         }
89 |     }
90 |
91 |     /**
92 |      * Returns the connections.
93 |      *
94 |      * @return connections
95 |      */
96 |     public List<Connection> getConnections() {
```

```
97          return connections;
98      }
99
100     /**
101      * Returns the cost of this minimum spanning tree.
102      *
103      * @return cost of this minimum spanning tree
104      */
105     public double getCost() {
106         return cost;
107     }
108
109     /**
110      * Explicitly sets the cost of this minimum spanning tree. This method may be useful in case some simple
111      * modifications from outside (e.g. removal of some connection) are made on this tree, and its cost is the
112      * recalculated in a control manner.
113      *
114      * @param cost new cost to be set
115      */
116     public void setCost(double cost) {
117         this.cost = cost;
118     }
119
120     /**
121      * Checks if a given place is associated with only one connection in this minimum spanning tree. If so the
122      * connection is returned, otherwise a null is returned.
123      *
124      * @param place reference to given place
125      * @return single connection or null if more than one connection exists
126      */
127     public Connection isPlaceWithSingleConnection(Place place) {
128         int occurrences = 0;
129         Connection singleConnection = null;
130         for (Connection connection : connections) {
131             if (connection.getPlace1().equals(place) || connection.getPlace2().equals(place)) {
132                 occurrences++;
133                 singleConnection = connection;
134                 if (occurrences > 1)
135                     return null;
136             }
137         }
138         return singleConnection;
139     }
140 }
```

Index