Guiding Probabilistic Model Checkers by Reinforcement Learning

University of Oxford
Department of Computer Sience

Qiyi Tang
Kellogg College

A thesis submitted for the degree of
Master of Science
Trinity 2013

Abstract

In this thesis, we study different solution methods of reinforcement learning problems and adapt those methods for model checking Java code containing probabilistic choices. In particular, we propose three new search strategies: softmax search (SMS), ϵ-greedy search (EGS) and reinforcement learning search (RLS). SMS and EGS are inspired by choice selection in reinforcement learning and RLS is based on the popular Q-learning method. We implement the above-mentioned search strategies within jpf-probabilistic, an extension of the model checker Java PathFinder. Furthermore, we use a progress measure to compare the performance of different search strategies on verifying a randomized version of quicksort.

Acknowledgements

I would like to express sincere gratitude to my supervisor, Prof. Franck van Breugel, for his extensive guidance and generous support. With his wisdom and experience as a trusted mentor, he encouraged me and motivated me to learn throughout the project. I feel so grateful to have had the opportunity to work with him. I'm also thankful for his expert advice and feedback on my thesis writing.

I would also like to thank my course mates for their friendship, inspiration, and support during this year at Oxford University. Without the thoughts and laughter we shared together, my year in Oxford would not have been so memorable.

Last but not least, many thanks to my family, for always believing in me, understanding me, and being there for me. Without their everlasting love, I would not have had the courage to explore the world away from home.

Contents

Chapter 1

Introduction

The notorious state space explosion problem is one of the major challenges in model checking in general and probabilistic model checking in particular (see, for example, [BK08]). Numerous techniques have been developed to tackle this problem. Experimental studies have shown that combining different techniques is most effective (see [Pel08]).

1.1 Research Question

The order in which the different alternatives of probabilistic choices are explored by the model checker, also known as its search strategy, impacts its effectiveness. Determining this order can be viewed as a planning problem. Techniques to address planning problems can therefore be used to combat the state space explosion problem. For example, the Monte Carlo method has been used in model checking (see, for example, [GS05]). In this project, we focus on another technique to tackle planning problems: reinforcement learning.

1.2 Proposed Method

One recent proposal to combat the state space explosion problem is to measure the amount of progress that has been made by the model checker towards verifying the property of interest (see [ZvB11]). Notions from reinforcement learning such as model, policy, reward and value correspond in our setting to the model checker together with the system being verified, probability distributions over the alternatives of probabilistic choices, (the average of) the progress made so far, and the expected progress. In this project, we design a new search strategy based on the notion of progress measure using techniques of reinforcement learning.

The model checker Java PathFinder[1] [VHB+03] (JPF) provides search strategies that do not take the probabilities into account such as depth-first search and breadth-first search. An extension of JPF named jpf-probabilistic[2] [ZvB10] contains several

[1] babelfish.arc.nasa.gov/trac/jpf

[2] bitbucket.org/discoveri/jpf-probabilistic

search strategies that do take the probabilities into account, such as probability-first search and random search. We implement our new search strategies within this framework. Our new search strategies are variations of the strategies of jpf-probabilistic. Furthermore, we also compare the performance of our search strategy with the above mentioned search strategies.

1.3 Related work

The aim of our research is to exploit techniques from planning, in particular reinforcement learning, in model checking. The opposite, that is, exploiting techniques from model checking for planning, has also been considered. In [GT99] Giunchiglia and Traverso describe the "planning as model checking" paradigm. In this approach, generating a plan amounts to determining whether a property is satisfied in a model.

Aragi and Cho [AC06] show how reinforcement learning can be used to test concurrent systems. In their approach, a concurrent system is modelled as a transition system. Each state of the system is labelled with the atomic propositions that hold in that state and each transition is labelled with the identifier of the process that corresponds to the transition. They restrict to fair executions and they focus on response properties. The latter can be expressed in linear temporal logic (LTL). For example, $\Box(p \implies \Diamond q)$, where p and q are atomic propositions, is a response property. Aragi and Cho use the Q-learning method (see, for example, [SB98, Section 6.5]) to determine which execution of the system to test. To maintain fairness, a positive reward is assigned to those outgoing transitions of the current state with the following property: in the execution checked so far and extended with that transition, each process takes the same number of transitions. In the search for violations of the response property, a positive reward is assigned to those outgoing transitions of the current state which satisfy one of the following two conditions: (1) in the execution checked so far we have not yet encountered p and the target state of the transition satisfies p but not q; (2) in the execution checked so far we have already encountered p and the target state of the transition does not satisfy q. Aragi and Cho have implemented their approach and applied it to a model of Dijkstra's mutual exclusion algorithm and a model of the dining philosopher's problem.

Behjati, Sirjani and Ahmadabadi [BSA09] also apply reinforcement learning to model checking of concurrent systems. Both the system and the negation of the property, expressed in LTL, are modelled as a Büchi automaton. In their approach not only response properties but also other LTL properties can be checked. Also Behjati et al. restrict to fair executions. They focus on finding counterexamples, that is, fair executions of the system that do not satisfy the property. To find those, they build the product of the two Büchi automata. A counterexample consists of a fair cycle that is reachable from an initial state and contains an accepting state in the product automaton. To find a counterexample, they assign a positive reward to those transitions at the end of an unfair accepting cycle and a negative reward to the ones at the end of a non-accepting cycle. Behjati et al. use the Monte Carlo method (see, for example, [SB98, Chapter 5]) to find accepting cycles. If such a cycle is found,

they check if it is fair. They have implemented their approach and applied it to the dining philosophers problem.

A large variety of other techniques are used to develop search strategies. For an overview, we refer the reader to [ES11].

1.4 Overview

The main contributions of this thesis are listed below:

- We designed three new search strategies softmax search (SMS), ϵ-greedy search (EGS) and reinforcement learning search (RLS);
- We implemented the above mentioned search strategies;
- We not only considered the implementation of random search (RS) that uses a linked list, as described in Zhang's thesis ([Zha10]), but also a new implementation of RS that uses a red-black tree instead;
- We ran experiments to compare the performance of the new search strategies with depth-first search (DFS) and breadth-first search (BFS), which are both part of JPF, and probability-first search (PFS) and RS, which are part of the extension `jpf-probabilistic`;
- We developed a blueprint for implementing search strategies in JPF using techniques from reinforcement learning;
- We developed scripts to parallelize the computation of progress.

The structure of thesis is shown as follows. In the next chapter, we characterize the reinforcement learning (RL) problems and the class of methods to solve RL problems. To illustrate how the methods can be applied to RL problems, we construct a simple example which is to find the most common path. We show how to formulate the RL problem for this example and how to solve it using two solution methods. We also discuss the suitable methods to develop a search strategy.

Chapter 3 introduces the explicit-state model checker Java PathFinder and its extension `jpf-probabilistic`. There relationship with search strategies and listeners are also covered in this chapter.

Chapter 4 provides the concept and algorithms of different search strategies. We start with the basic DFS and BFS. Then we provide PFS and RS proposed by Zhang and van Franck ([ZvB11]) which take the probabilities of the transitions into account. SMS and EGS are developed based on PFS and RS. We then propose RLS which is based on a temporal-difference method called Q-learning method.

We use the so-called progress measure ([ZvB11]) to evaluate the different search strategies and the algorithm is presented and explained in Chapter 5.

Chapter 6 describes the experiment settings and shows the results of the experiment. The performance of different search strategies are compared and analysed.

Finally, we summarise the thesis and conclude the thesis with some suggestions of future work.

Chapter 2

Background - Most Common Path Example

To introduce the readers to reinforcement learning, we will consider the following problem. Given a discrete time Markov chain (DTMC), find a most common path. That is, among all the infinite paths starting in the initial state of the DTMC, find a path whose probability is maximal. Before introducing reinforcement learning, let us first formalize the problem.

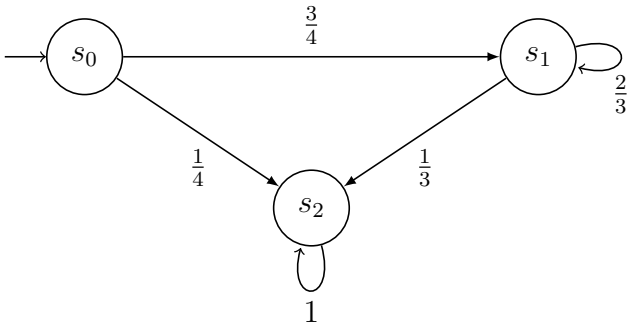
2.1 Most Commmon Path in a DTMC

Definition 2.1.1. A *discrete time Markov Chain* (DTMC) is a tuple $(S, \mathbf{P}, s_0)$ where

- S is a countable, non-empty set of *states*,
- $\mathbf{P} : S \times S \to [0, 1]$ is the *transition probability function* such that for all states $s \in S$,
$$\sum_{s' \in S} \mathbf{P}(s, s') = 1,$$
- $s_0 \in S$ is the *initial state*.

We shall use the following DTMC as our running example for the rest of this chapter.

Example 2.1.1. The DTMC depicted by



has three states and five transitions where state s_0 is the initial state. The transition probability function can be easily extracted from the above diagram. For example, $\mathbf{P}(s_0, s_1) = \frac{3}{4}$ and $\mathbf{P}(s_2, s_2) = 1$.

Definition 2.1.2. Let $M = (S, \mathbf{P}, s)$ be a DTMC. A *path of M starting in s_0* is an infinite sequence $s_0 s_1 s_2 \ldots$ such that $\mathbf{P}(s_i, s_{i+1}) > 0$ for all $i \geq 0$. We denote the set of paths of M starting in the initial state s by $Paths(M)$.

For the DTMC of Example 2.1.1, the sequence $s_0 s_1 s_2 s_2 s_2 \ldots$ is an example of a path and so is $s_0 s_1 s_1 s_1 \ldots$.

Definition 2.1.3. Let $M = (S, \mathbf{P}, s)$ be a DTMC and let $s_0 s_1 s_2 \ldots \in Paths(M)$. The *probability* of $s_0 s_1 s_2 \ldots$ is defined by

$$prob(s_0 s_1 s_2 \ldots) = \lim_{n \to \infty} \prod_{0 \leq i \leq n} \mathbf{P}(s_i, s_{i+1}).$$

In the DTMC of Example 2.1.1, the probability of paths $s_0 s_1 s_2 s_2 s_2 ...$ and $s_0 s_1 s_1 s_1 ...$ are $\frac{1}{4}$ and 0, respectively.

Now that we have defined the probability of a path, we can formalize the most common path problem as follows. Given an DTMC M, find a $\sigma \in Paths(M)$ such that $prob(\sigma) \geq prob(\sigma')$ for all $\sigma' \in Paths(M)$.

In Example 2.1.1, the paths $s_0 s_1 s_2 s_2 s_2 \ldots$ and $s_0 s_2 s_2 s_2 \ldots$ are the two most common paths, both have probability of $\frac{1}{4}$. All the other paths in the DTMC have smaller probabilities.

Note that for the most common path problem, we restrict ourselves to DTMCs which satisfy two conditions:

- the probability is less than one for all the transitions except self loops;
- and the DTMCs must contain self loops.

2.2 Reinforcement Learning

Reinforcement learning (RL) is a learning method in connection with an agent and its environment. The agent aims to achieve a goal by interacting with the environment. In the following section, we first describe the environment. Second, the role of agent is introduced and the interaction between the environment and the agent is discussed. Next, we show how the goal of a RL problem is formalized and explain the basic elements of RL in detail. Finally, we formulate our own RL problem – finding a most common path.

The environment is everything around the agent and can be modelled by a Markov decision process (MDP).

Definition 2.2.1. A *Markov decision process* (MDP) is a tuple $(S, Act, A, \mathbf{P}, s_0, \gamma, R)$ where

- S is a countable, non-empty set of *states*,
- Act is a set of *actions*,
- $A : S \to 2^{Act}$ is the *available action function*,
- $\mathbf{P} : S \times Act \times S \to [0, 1]$ is the *transition probability function* such that for all states $s \in S$ and actions $a \in Act$,

$$\sum_{s' \in S} \mathbf{P}(s, a, s') \in \{0, 1\},$$

- $s_0 \in S$ is the *initial state*.
- $\gamma \in [0, 1]$ is the *discount rate*,
- $R : S \times Act \to \mathbb{R}$ is the *reward function*. $R(s, a)$ is the reward of taking action a from state s and $a \in A(s)$.

For all state $s \in S$, $A(s)$ is the set of actions that are available in state s. A reward function $S \times Act \to \mathbb{R}$ maps a state-action pair to a numerical number. The reward function captures how good the action in the given state is.

If the task is divided into separate episodes, the environment also should include terminal states. The environment could appoint a set of states to be terminal directly or otherwise could specify a set of states to be terminal by some properties such as no outgoing transitions.

The agent is a learner and decision maker. It is the agent's role to determine which action to take in a state of the environment. The agent interacts with the environment. The interaction consists of choosing actions and observing state and reward. It learns how good is each action for each state of the environment. Figure 2.1 shows an agent interacts with its environment in a sequence of discrete time step, t = 0, 1, 2, 3, ... At each time step t, the agent observes the current state of the environment $s_t \in S$ and selects a possible action $a_t \in A(s_t)$. One time step later, as a consequence of selecting a_t the agent receives a reward r_{t+1} and observes the new environment state s_{t+1}. At each time step, the agent selects an action according to a policy which maps each state s of the environment to a probability distribution on $A(s)$. The reward is a number generated by the environment. It can be either positive, zero or negative. We will talk about policy and reward in detail later.

The goal of a RL problem is to maximize the cumulative (discounted) rewards in the long run. Upon visiting the sequence of states $s_t, s_{t+1}, ...$ (s_t does not have to be the initial state) with actions $a_t, a_{t+1}, ...$ the sum of the discounted rewards is given by

$$R(s_t, a_t) + \gamma R(s_{t+1}, a_{t+1}) + \gamma^2 R(s_{t+2}, a_{t+2}) + \cdots = \sum_{k=0}^{\infty} \gamma^k R(s_{t+k}, a_{t+k}) \qquad (2.1)$$

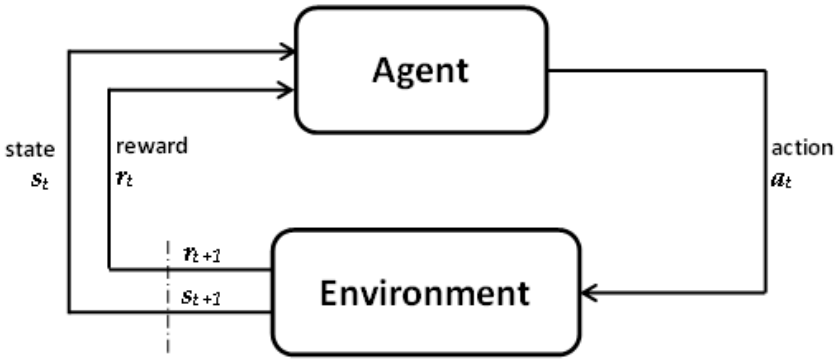


Figure 2.1: The agent-environment interaction in reinforcement learning.

Equation (2.1) is called *discounted return*, R is the reward function and γ is the discount rate. The role of γ is illustrated here. The future rewards are discounted by γ. For example, the reward k time steps further than the current state s_t is only accounted γ^k times the value it would be if it were the immediate reward. If $\gamma = 0$, only the immediate reward counts and all the future rewards are disregarded, which makes the process quite nearsighted. As γ becomes larger, the future rewards are taken more into account.

There are four basic elements of a RL system: a policy, a reward function, a value function and the model of the environment ([SB98]). The model of the environment and the reward function have already been discussed earlier.

A policy tells the agent how to choose an action given the current state of the environment. It is a function $\pi : S \times Act \to [0, 1]$ such that $\sum_{a \in A} \pi(s, a) = 1$. It maps a state s to a probability distribution on $A(s)$. When for every state s, $\pi(s, a) = 1$ where $a \in A$, the policy actually directly maps a state to a possible action. The policy is changing alongside with the interaction between the agent and the environment. The agent is learning towards an optimal policy.

A value function estimates how good each state is in the long run. It is the total future (discounted) reward that is expected in a state continuing with the given policy. The value of a state s under a policy π, denoted $V^\pi(s)$, is given by

$$V^\pi(s) = \mathbb{E}_\pi\{\sum_{k=0}^{\infty} \gamma^k R(s_{t+k}, a_{t+k}) | s_t = s\} \tag{2.2}$$

The value of taking action a in state s under a policy π, denoted $Q^\pi(s, a)$, is given by

$$Q^\pi(s, a) = \mathbb{E}_\pi\{\sum_{k=0}^{\infty} \gamma^k R(s_{t+k}, a_{t+k}) | s_t = s, a_t = a\} \tag{2.3}$$

There are two types of RL tasks, episodic tasks and continuing tasks, which can be differentiate according to if the task is broken into separate episodes. As mentioned earlier, episodic task has a set of terminal states. Each episode will end when reaching one of the terminal state and start a new episode from the initial state. A continuing task does not have terminal states so it will not terminate. Since episodic task will not run infinitely long time, the cumulative discounted reward function (2.1) should be modified. For one episode with T time steps, it is given by

$$R(s_t, a_t) + ... + \gamma^{T-1} R(s_{t+T-1}, a_{t+T-1}) = \sum_{k=0}^{T-1} \gamma^k R(s_{t+k}, a_{t+k}) \tag{2.4}$$

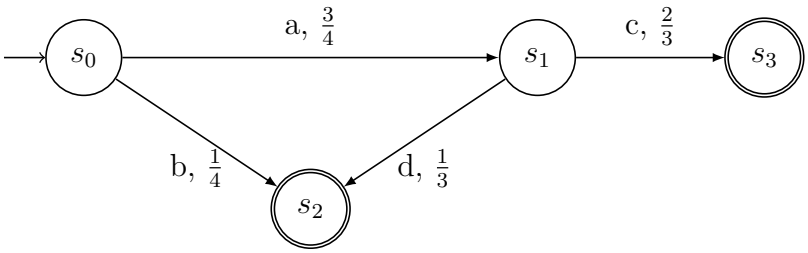
Equation (2.2) and Equation (2.3) should be modified accordingly for episodic tasks.

2.3 RL Problem Formulation

With all the elements explained, we formulate the elements of the RL problem - searching for a most common path of a DTMC. There are lots of ways to formulate the problem, the way we proposed is only one example among them.

The problem can be broken down into episodes when using Monte Carlo methods and temporal-difference learning. The terminal states are the states with self loops. Each time such a state is reached, i.e. a path is found, the current episode ends and a task of finding a path is restarted from the initial state. The set of states of the environment is the same as that of the DTMC. The transition in the DTMC are all labelled differently. The set of actions Act in the environment is then $\{a, b, c, d\}$. For each state s in the environment, available action function $A(s)$ can be obtained from Figure 2.3. For all the states s, s' and action a, the transition probability function $P'(s, a, s')$ is set to be equal $P(s, s')$, where $P(s, s')$ is the transition probability function of the DTMC. Discount rate γ is set to be 1.

Our environment now is depicted by



For the most common path example, the agent's goal is to find a path with the greatest probability. We assign a negative value (e.g. -1) to the recurring state with probability less than 1 and a positive value (e.g. 1) to the recurring state with probability of exactly 1 and zero to all the other transitions. The reward of each step is set to be -0.1.

2.4 Solution Methods

There are three elementary solution methods: dynamic programming (DP) methods, Monte Carlo methods and temporal-difference (TD) learning . We show how to apply DP methods and TD learning to the most common path problem respectively. To apply DP methods, we must have the full knowledge of the environment model, so it does not fit our search strategy. We introduce it since it helps understand TD learning methods.

2.4.1 Dynamic Programming

There are two forms of dynamic programming methods: policy iteration and value iteration ([SB98]). Policy iteration starts with a random policy. The policy is evaluated and improved iteratively ([SB98, Section 4.3]). The evaluation is based on Equation (2.2). Rewrite Equation (2.2) to be the form of Bellman equation for V^π:

$$V^\pi(s) = \mathbb{E}_\pi\{R(s,a) + \gamma V^\pi(s_{t+1}) | s_t = s\} \tag{2.5}$$

The value of each state s under policy π could be approximated by iteratively applying Equation (2.5).

$$\pi'(s) = \arg\max_a Q^\pi(s,a) \tag{2.6}$$

Assume the original policy is π and the policy after applying Equation (2.6) is π'. For any state s, $V_\pi(s) \leq Q^\pi(s, \pi'(s))$, then the policy π' is better than the policy π. The proof is as follows.

$$\begin{aligned}
V_\pi(s) &\leq Q^\pi(s, \pi'(s)) \\
&= \mathbb{E}_{\pi'}\{R(s_t,a_t) + \gamma V^\pi(s) | s_t = s, a_t = a\} \\
&\leq \mathbb{E}_{\pi'}\{R(s_t,a_t) + \gamma Q^\pi(s_{t+1}, \pi'(s_{t+1})) | s_t = s, a_t = a\} \\
&= \mathbb{E}_{\pi'}\{R(s_t,a_t) + \gamma \mathbb{E}_{\pi'}\{R(s_{t+1},a_{t+1}) + \gamma V^\pi(s_{t+2})\} | s_t = s, a_t = a\} \\
&\vdots \\
&\leq \mathbb{E}_{\pi'}\{R(s_t,a_t) + \gamma R(s_{t+1},a_{t+1}) + \gamma^2 R(s_{t+2},a_{t+2}) + ... | s_t = s, a_t = a\} \\
&= V_{\pi'}(s)
\end{aligned}$$

For all state s, $V_\pi(s) \leq V_{\pi'}(s)$. So the policy could be improved by applying Equation (2.6).

Value iteration is the truncated version of policy iteration ([SB98]). It applies Equation (2.7) to all states in the RL system iteratively until the values of all states become stable.

$$V^{\pi'}(s) = \arg\max_a Q^\pi(s,a) \tag{2.7}$$

After several iterations, if $V^\pi(s) = V^{\pi'}(s)$ holds for all state s, then the policy is the optimal policy as it satisfies the Bellman optimality equation Equation (2.8). The

derivation from $V^{\pi}(s) = V^{\pi'}(s)$ to the Bellman optimality equation is shown below.

$$V^{\pi'}(s) = \max_a \mathbb{E}\{R(s_t, a_t) + \gamma V^{\pi}(s_{t+1}) | s_t = s, a_t = a\} (\text{substitute } \pi \text{ by } \pi')$$
$$V^{\pi'}(s) = \max_a \mathbb{E}\{R(s_t, a_{s_t}) + \gamma V^{\pi'}(s_{t+1}) | s_t = s, a_t = a\} \quad (2.8)$$

We apply value iteration on the MDP shown in Figure2.3. The values of s_0 and s_1 of the first two iterations are shown in Table 2.1. The values of s_0 and s_1 are initialized to be 0. The values of states remain stable after the first iteration. The optimal policy we obtain is then $\pi(s_0) = b$ and $\pi(s_1) = d$, which is correct.

Table 2.1: Value iteration (DP method)

The 1st iteration:	$V(s_0) = 0.9$	$V(s_1) = 0.9$.
The 2nd iteration:	$V(s_0) = 0.9$	$V(s_1) = 0.9$.

2.4.2 Temporal-Difference Learning

When model checking a probabilistic program, we do not know the state space of the program at the beginning. We only know the part we have already model checked. For this reason, DP methods can not be applied. Temporal-difference (TD) learning can, however, learn from its experience without the full knowledge of the environment ([SB98, Page 133]).

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)] \quad (2.9)$$

TD learning also involves policy evaluation and policy improvement. Here we present one TD learning algorithm which is the Q-learning algorithm [SB98, Section 6.5]. In Q-learning algorithm, the function which updates $Q(s, a)$ is independent of the policy being followed. Equation (2.9) is used to update the Q function. The algorithm is shown in Algorithm 2.1.

Algorithm 2.1: Q-learning: TD learning Algorithm

```
initialize Q(s, a) for all s and a;
repeat
    choose action a using policy derived from Q (e.g. ε-greedy);
    while state s is not the end state do
        take action to state s', observe reward r;
        Q(s, a) ← Q(s, a) + α[r + γ max_a' Q(s', a') − Q(s, a)];
        s ← s';
    end
until Q(s, a) for all s and a converges;
```

Chapter 3

Java PathFinder

There are two types of model checker: symbolic ones and explicit-state ones. The former is often used to verify hardware whereas the latter is more suited to check software. An explicit-state model checker visits the states of the system under test in a systematic way. Each visited state is represented explicitly by the model checker. In contrast, a symbolic model checker represents the set of visited states compactly by a data structure such as a binary decision diagram. Since we are interested in verifying Java (byte)code, we focus on explicit-state model checking. For a more detailed comparison of symbolic and explicit-state model checking, we refer the reader to, for example, [EP02].

Java PathFinder (JPF)[1] [VHB+03] is an explicit-state model checker. It can verify Java (byte)code. JPF systematically visits the states of the system under test, that is, the bytecode of a Java application. It goes from state to state traversing transitions. These transitions correspond to sequences of bytecode instructions.

3.1 Virtual Machines

JPF is a virtual machine. To contrast JPF's virtual machine with an ordinary Java virtual machine we consider the following Java application.

```
1 import java.util.Random;
2
3 public class Example {
4   public static void main(String[] args) {
5     Random random = new Random();
6     System.out.print(random.nextInt(10));
7   }
8 }
```

The execution of the above application by an ordinary Java virtual machine results in the printing of a randomly chosen integer in the interval $[0, 9]$. The execution of the application by JPF's virtual machine results in the following output.

[1] babelfish.arc.nasa.gov/trac/jpf

```
1 JavaPathfinder v6.0 (rev 1035+) - (C) RIACS/NASA Ames Research Center
2 ========================================== system under test
3 Example.main()
4 ========================================== search started: ...
5 0123456789
6 ========================================== results
7 no errors detected
8 ...
9 ========================================== search finished: ...
```

In particular, the execution prints `0123456789`, that is, it prints all integers in the interval $[0, 9]$. While an ordinary Java virtual machine carries out one of the ten potential executions, JPF's virtual machine performs all ten potential executions in a systematic way.

Now, let us replace line 6 with

```
6     System.out.print(1 / random.nextInt(10));
```

In 80% of the cases, the execution by an ordinary Java virtual machine prints zero, in 10% it prints one, and in the remaining 10% it throws an exception. Of course, it may take more than ten executions before we encounter the exception. In case we choose an integer in the interval $[0, 99999]$ it may take many executions before encountering the exception. If we execute the application one million times, there is a 36% chance that we do not encounter the exception. In contrast, JPF's virtual machine checks all one million executions within ten seconds and reports the exception.

```
1 JavaPathfinder v6.0 (rev 1035+) - (C) RIACS/NASA Ames Research Center
2 ========================================== system under test
3 Example.main()
4 ========================================== search started: ...
5
6 ========================================== error 1
7 gov.nasa.jpf.vm.NoUncaughtExceptionsProperty
8 java.lang.ArithmeticException: division by zero
9         at Example.main(Example.java:6)
10 ...
11 ========================================== search finished: ...
```

JPF is implemented in Java. Its virtual machine is implemented by means of the class `VM`, which is part of the package `gov.nasa.jpf.vm`. Later, we will come back to this class and its methods.

3.2 Search Strategies

JPF's exploration of the state space can be seen as a graph traversal. The vertices of the graph are the states of the system under test and the edges of the graph are the transitions between the states. Just like graphs can be traversed in different

ways, JPF can visit the states in different orders as well. JPF supports several search strategies including depth-first search (DFS) and breadth-first search (BFS).

In case JPF cannot explore the whole state space (due to lack of memory or time), different search strategies may visit different parts of the state space. Consider, for example, the following Java application.

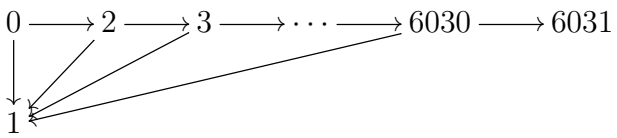
```
1 public class Example {
2   public static void main(String[] args) {
3     Random random = new Random();
4     long count = 0;
5     while (random.nextInt(2) == 0) {
6       count++;
7     }
8   }
9 }
```

By default, JPF uses DFS. In this case, JPF runs out of memory after roughly two and a half minutes. During that time, it visits 272050 states. The states and transitions explored by JPF can be depicted as follows.

$$0 \longrightarrow 1 \longrightarrow 2 \longrightarrow \cdots \longrightarrow 272048 \longrightarrow 272049$$

The states are numbered in the order they are visited.

If we configure JPF to use BFS by setting the JPF property `search.class` to `gov.nasa.jpf.search.heuristic.BFSHeuristic`, it also runs out of memory, this time after less than two minutes. It only visits 6032 states. It runs out of memory quicker and visits fewer states since the queue used to implement BFS takes additional memory. In this case, the states and transitions can be depicted as follows.



State 0 of the above diagram corresponds to state 0 of the DFS diagram. State 1 of the BFS diagram is a final state and is not visited by DFS. State i, for $i \geq 0$, of the BFS diagram corresponds to state $i+1$ of the DFS diagram. Hence, there is one state that is visited by BFS that is not visited by DFS, but there are many states visited by DFS that are not visited by BFS.

Now, let us consider which executions DFS and BFS check. On the one hand, DFS considers a prefix of a single execution, where in each iteration the randomly chosen integer is zero. On the other hand, BFS checks 6031 executions, where in the i iteration the randomly chosen integer is one, for $0 \leq i \leq 6030$. We will come back to this later when discussing progress. The algorithm of different search strategies are presented in Chapter 4 and the detailed implementation of those search strategies are given in Appendix A.

3.3 Listeners

A listener allows us to extract information from JPF during its traversal of the state space. For example, the listener `SimpleDot`, which is part of package `gov.nasa.jpf.listener`, generates a dot-file that contains a graphical representation of the state space of the system under test. Both the search and JPF's virtual machine notify listeners of particular events. How to implement a listener is shown in Appendix B.

3.4 The `jpf-probabilistic` Extension

The extension `jpf-probabilistic`[2] [ZvB10] of JPF allows us to associate probabilities with transitions. For example, in the above Java application we may want to associate probability 0.5 with each transition. For JPF to easily recognize these probabilities, we use the method `make` of the class `Choice`, which is part of the package `probabilistic`. For example, if

```
1 double[] p = { 0.4, 0.6 };
```

then the method invocation `Choice.make(p)` returns 0 with probability 0.4 and 1 with probability 0.6. More generally, if $\texttt{p}[0] + \cdots + \texttt{p}[\texttt{p.length} - 1] = 1.0$ then the method invocation `Choice.make(p)` returns i, where $0 \leq i < \texttt{p.length}$, with probability $\texttt{p}[i]$. We can refactor the above Java application as follows.

```
1 public class Example {
2   public static void main(String[] args) {
3     final double[] p = { 0.5, 0.5 };
4     long count = 0;
5     while (Choice.make(p) == 0) {
6       count++;
7     }
8   }
9 }
```

The probabilities associated with the transitions can be used two different ways. First of all, they can be exploited by search strategies. We will come back to this later. Secondly, they can be used to measure the amount of progress JPF has made with its verification effort. We will discuss this next.

[2] `bitbucket.org/discoveri/jpf-probabilistic`

Chapter 4

Algorithms of Search Strategies

In this chapter we provide the algorithms of different search strategies. We start with depth-first search (DFS) and breadth-first search (BFS) and then introduce random search (RS) and probability-first search (PFS) proposed by [Zha10, page 71-76] which take the probabilities of transitions into account. Then we propose two new search strategies, softmax search (SMS) and ϵ-greedy search (EGS), which are based on RS and PFS and inspired by some selection methods in [SB98]. The details of the implementation of these search strategies are given in the Appendix A.

4.1 Depth-First Search

Depth-first search (DFS) is part of JPF. As its name suggests, it starts at the initial state and explores along each branch as far as possible before backtracking. It can be implemented as Algorithm 4.1.

Algorithm 4.1: Depth-First Search

```
S ← empty stack;
push the initial state s onto S;
while S is not empty do
    pop(S);
    for all transitions t from S do
        if target(t) is not visited then
            push target(t) onto S ;
        end
    end
end
```

4.2 Breadth-First Search

Breadth-first search (BFS) is also part of JPF. It starts at the initial state and traverses all its unexplored transitions. In the mean time, each of the newly discovered states are stored into a queue. Then for each of the states in the queue in turn, it traverses every unexplored transition and adds the newly discovered states into the queue, and so on. An algorithm of BFS is shown in Algorithm 4.2.

Algorithm 4.2: Breadth-First Search

```
Q ← an empty queue;
enqueue the initial state;
while Q is not empty do
    dequeue state s from Q;
    for all transitions t from s do
        if target(t) is not visited then
            enqueue target(t);
        end
    end
end
```

4.3 Probability-First Search

Now we describe several search strategies which take the probabilities of transitions into consideration. These search strategies are provided in the `jpf-probabilistic` extension of `JPF`.

probability-first search (PFS) is proposed in [Zha10]. It uses the probability to select the next state to explore, where the policy is always choosing the state whose path along which it is discovered has the highest probability. The algorithm is as follows.

Algorithm 4.3: Probability-First Search

```
Q ← an empty priority queue;
add the initial state with weight 1 to Q;
while Q is not empty do
    remove a state s with largest weight p from Q;
    for all transitions t from s do
        if target(t) is not visited then
            add target(t) with weight prob(t) * p to Q;
        end
    end
end
```

The weight of the state is the probability of the path along which the state is discovered. We use a priority queue as the container to store a state and its weight. The priority queue orders its elements opposite to the natural ordering of their probabilities, and the head of the queue corresponds to the state with the largest weight.

4.4 Random Search

Random search (RS) is also proposed by Zhang ([Zha10]). Similar to PFS, RS also uses the probability to select the state to explore next, but the policy is that the chance of choosing a state is proportional to the probability of the path along which the state is discovered. Let us make that precise. Assume that $\{s_1, ..., s_n\}$ is the set of states that have been discovered but their outgoing transitions have not been explored yet. Then RS chooses state s_j with probability

$$\frac{p(s_j)}{\sum_{i=1}^{n} p(s_i)} \tag{4.1}$$

where $p(s_i)$ is the probability of the path along which s_i is discovered. The algorithm is shown in Algorithm 4.4.

Algorithm 4.4: Random Search

```
distribution ← an empty distribution;
add the initial state with weight 1 to distribution;
while distribution is not empty do
    remove a state s with weight p according to the above policy from
    distribution;
    for all transitions t from s do
        if target(t) is not visited then
            add target(t) with weight prob(t) * p to distribution;
        end
    end
end
```

The weight is the same as in PFS. In Algorithm 4.4, the container to store the state and its weight is called *distribution*. It supports the method to remove a state and the probability of removing any state is proportional to their weight. We implement *distribution* by means of a red-black tree whereas Xin implements it by a linked list in [Zha10].

4.5 Softmax Search

From now on, we show several search strategies designed and implemented by ourselves. Inspired by softmax action selection [SB98, Section 2.3], we implement softmax search (SMS) where the chance of choosing a state is based on Gibbs distribution. It chooses state s_j with probability

$$\frac{e^{p(s_j)/\tau}}{\sum_{i=1}^{n} e^{p(s_i)/\tau}} \tag{4.2}$$

where $p(s_i)$ is the probability of the path along which s_i is discovered and the constant τ is called the temperature and should be a positive real number. As the temperature

approaches zero, SMS behaves more and more like PFS. High temperatures cause the states to become nearly equiproportional.

Since we can obtain Equation (4.2) by replacing $p(s_i)$ in Equation (4.1) with $e^{p(s_i)/\tau}$, we can change the weight of a state element to $e^{p(s_i)/\tau}$. We can implement SMS on the basis of RS with the weights of the states changed as follows.

Algorithm 4.5: Softmax Search

```
distribution ← an empty distribution;
add the initial state with weight e^{1/τ} to distribution;
while distribution is not empty do
    remove a state s with weight p according to the above policy from
    distribution;
    for all transitions t from s do
        if target(t) is not visited then
            add target(t) with weight p^{prob(t)} to distribution;
        end
    end
end
```

There are only two places in the above search method that are different from RS. First, we add the initial state with weight $e^{1/\tau}$ instead of 1 to *distribution*. Secondly, the weight of the newly discovered state is calculated differently. Let $p(s_i)$ denote the probability of the path along which s_i is discovered. When we find a new state s_t starting from a restored state s_r, $p(s_t)$ is equal to $p(s_r) * p(s_r, s_t)$, where $p(s_r, s_t)$ denotes the probability of the transition from s_r to s_t. So the weight of s_t can be calculated as

$$e^{p(s_r)*p(s_r,s_t)/\tau} = (e^{p(s_r)/\tau})^{p(s_r,s_t)} \tag{4.3}$$

where $e^{p(s_r)/\tau}$ is the weight of s_r. According to Equation (4.3), we can obtain the weight of s_t by raising the weight of s_r to the power of the probability of the transition.

4.6 ϵ-Greedy Search

Inspired by another action selection method, the ϵ-greedy method mentioned in [SB98, page 28], we implement the so-called ϵ-greedy search (EGS). It combines RS and PFS in such a way that with probability $1 - \epsilon$ it behaves like PFS, that is choosing the state whose path along which it is discovered has the highest probability, and with probability ϵ it behaves like RS, that is the chance of choosing a state is proportional to the probability of its path.

Algorithm 4.6: ϵ-Greedy Search

```
distribution ← an empty distribution ;
add the initial state with weight 1 to distribution;
while distribution is not empty do
    random ← a random number in the range [0..1);
    if random < ε then
        remove a state s according to RS policy from distribution;
    else
        remove a state s according to PFS policy from distribution;
    end
    for all transitions t from s do
        if target(t) is not visited then
            add target(t) with weight prob(t) * p to distribution;
        end
    end
end
```

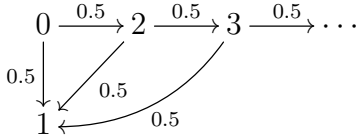
4.7 Reinforcement Learning Search

In a search strategy we need to decide

- which state to consider next, and
- which unexplored transition of that state to explore next.

For example, PFS selects a state which is not fully explored and the probability of the path along which the state is discovered is maximal. It also selects the first unexplored transition of that state.

Below, we will show how we can use reinforcement learning to select a state and transition. We use temporal-difference learning [SB98, Chapter 6] as the basis of our search strategy. In particular, we base it in Q-learning [SB98, Section 6.5]. In order to explain our approach, we will again use the `Example` application of Section 3.4. Below we present the states and transitions, where 0 is the initial state and 1 is a final state.



At the level of states and transitions there is very little we can learn since we traverse each transition at most once and we visit each state at most $n + m$ times, where n and m are the number of incoming and outgoing transitions of the state.

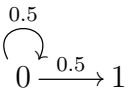
Since we restrict ourselves to sequential code, each state with multiple outgoing transitions corresponds to an invocation of the `Choice.make` method. For example, in the above example the states 0, 2, 3, ... all correspond to an invocation of the `Choice.make` method. Note that such an invocation of the `Choice.make` method

may be executed multiple times. Hence, this suggests that we may be able to learn something at the level of invocations of the `Choice.make` method.

Given an invocation `Choice.make(p)`, different executions of this invocation may use a different probability distributions `p`. We will see an example of this scenario in Section 6.2. The invocations of the `Choice.make` method together with their probability distributions `p` are the states of the MDP representing the environment. We will call these the RL states, to distinguish them from the JPF states. Note that multiple states visited by JPF may correspond to a single RL state. For each RL state, we keep track of the set of corresponding JPF states by means of a function $\mathbf{F}_S$ which maps RL states to sets of JPF states.

The alternatives of an invocation `Choice.make(p)`, that is, the set $\{0, \ldots, \texttt{p.length}-1\}$, play the role of actions of the MDP. Hence, the actions of the MDP are natural numbers. The available action function of the MDP is defined in the obvious way. It maps the RL state corresponding to the `Choice.make(p)` to the set $\{0, \ldots, \texttt{p.length} - 1\}$.

An RL state s corresponding to an invocation `Choice.make(p)` has an RL transition for each alternative i, with $0 \leq i < \texttt{p.length}$. The probability of this RL transition is $\texttt{p}[i]$ and the transition leads to the RL state corresponding to the JPF states that can be reached from $\mathbf{F}_S(s)$. Note that this may in general not be well defined, but for the `Example` application, this defines a transition probability function for the MDP representing the environment. The MDP representing the environment of the `Example` application is shown below.



where 0 is the initial state and 1 is the final state.

However, for the randomized quicksort application, this does not define a proper transition probability function. Recall that TD learning does not assume the full knowledge of the environment (Section 2.4.2), we do not need to define a transition probability function. For a state s, we only keep track of the most recent next states of it. Improving our approach so that the transitions and transition probability function are well defined is left for future research.

The initial RL state corresponds to the first invocation of `Choice.make`. Also this may in general not be well defined, but again for all the examples we considered so far, it is. Handling the general case we also leave for future work. We set the discount rate γ to one. That is, we do not discount the future.

Before we define the reward function, we briefly introduce a few notions that will be considered in detail in Chapter 5. We call a JPF state fully explored if it is either a final state or the probabilities of its explored outgoing transitions add up to one. In case JPF has explored states 0, 1 and 2 in the above example, states 0 and 1 are fully explored but state 2 is not. We call a JPF state processed if all its reachable states are fully explored. In case JPF has explored states 0, 1 and 2 in the above example, only state 1 is processed.

Recall that the reward function assigns a numerical value to each RL state. Given an RL state s, we are interested in the fraction of corresponding JPF states that are processed. That is, we define the reward function R as

$$R(s) = \frac{|\{\, s' \in \mathbf{F}_S(s) \mid s' \text{ is processed} \,\}|}{|\mathbf{F}_S(s)|}.$$

Now that we have defined all the ingredients defining the MDP representing the environment, let us look at another reinforcement learning notion. In our setting, a policy is a mapping from an RL state to a probability distribution on the actions available in that state. Hence, it tells us given an invocation `Choice.make(p)` which alternative of that invocation to choose. Therefore, it can be used to drive a search strategy.

Recall that the value function assigns to each RL state the expected accumulative reward. We estimate the value function using a simplified version of Equation (2.9) shown below.

$$V(s) \leftarrow V(s) + \alpha[r + \gamma \max_{s'} V(s') - V(s)] \tag{4.4}$$

Note that s' in Equation (4.4) is chosen from the set of next states which is most recently verified. The value of the final state is set to be the value of the source state which leads to it. α is set to be the probability of the transition which leads to the state with the largest value. For all the other state s, $V(s)$ is initialized to be 0.

Algorithm 4.7: Reinforcement Learning Search

```
Q_small ← an empty priority queue;
add initial state with value 0 to Q_small;
Q_large ← an empty priority queue;
while true do
    if Q_small is not empty then
        remove state s with largest value from Q_small;
        add all states in Q_small to Q_large;
    end
    else if Q_large is not empty then
        remove state s with largest value from Q_large ;
    end
    else
        end the search;
    end
    for all transitions t from s do
        if target(t) is not visited then
            add target(t) with value V(target(t)) to Q_small;
        end
    end
end
```

RLS uses two priority queues to store the states with their values: Q_{small} and Q_{large}. The policy π we follow is adding all next states available to Q_{small} and then choosing the state with the largest value in Q_{small}. All the other states in Q_{small} are added to Q_{large}. If for a state s, there are no next states available, we choose the state with the largest value from Q_{large}. The search strategy is shown in Algorithm 4.7. It is a greedy policy since RLS always chooses the state with the largest value.

We implement a new listener called `NewTransitionsAndTime`. It not only prints the transitions and timing information but also maps the JPF states to the RL states. In addition, it keeps track of the set of states which are processed. It updates the value of each RL state. A separate thread is created to calculate the set of processed states while model checking, which will affect the speed of search process as little as possible. When RLS visits a new JPF state, it gets the corresponding RL state from the listener and saves the JPF state along with the RL state in the priority queues.

Chapter 5

Evaluation of Search Strategies - Progress Measure

In Section 4 we present the algorithms of seven different search strategies: DFS, BFS, PFS, RS, SMS, EGS and RLS. Next, we evaluate the different search strategies. In this chapter we introduce the evaluation method proposed by [ZvB11].

5.1 Measuring Progress for Invariants

We restrict our attention to checking invariants for sequential code. Roughly speaking, an invariant captures that each state of the system under test satisfies a particular property. For example, "no state of the system under test throws an uncaught exception" is an invariant. For a formal definition of the notion of invariant, we refer the reader to, for example, [BK08, Section 3.3.1]. In particular when checking source code, as we do, invariants play a key role. JPF can check for uncaught exceptions. JPF's extension `jpf-numeric` can check for overflow of integer valued variables. Both are examples of invariants.[1]

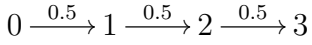
Zhang and Van Breugel [ZvB11] introduced a general notion of progress. This is a quantitative notion that captures the amount of progress the model checker has made with its verification effort. The amount of progress is captured by a real number in the interval $[0, 1]$. The larger the number, the more progress the model checker has made.

As shown by Zhang and Van Breugel in [ZvB11], the progress measure provides a bound on the probability that an error is encountered. Assume that the model checker runs out of memory when verifying property φ, does not detect any violations of φ, and has made 0.97 progress. Then the probability of encountering a violation of φ when running the code is at most $1 - 0.97 = 0.03$.

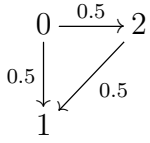
When restricting to invariants and sequential code, the progress can be computed as follows. Consider all the states and transitions that have been explored by the model checker. We call an explored state fully explored if it is either a final state or the probabilities of its explored outgoing transitions add up to one. Consider the

[1] `babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-numeric`

`Example` presented Section 3.4. Assume that the code has been explored using DFS. After three transitions, the following states and transitions have been explored.

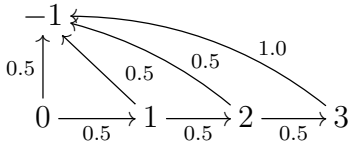


No state is fully explored. If we use BFS instead, the following states and transitions have been explored.

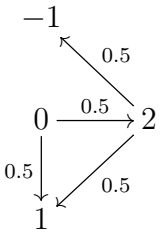


State 0 is fully explored. Since state 1 is a final state and, hence, also fully explored. State 2 is not fully explored yet.

To compute the progress for invariants for sequential code, we conceptually add a new state, which we call the sink state and denote it by -1. For each state that is not fully explored, we conceptually add a transition from that state to the sink state. The probability of that transition is the "remaining" probability, that is, it is $1-$ (the sum of the probabilities of its outgoing explored transitions). In the case of DFS, this gives rise to



and for BFS



As shown by Zhang and Van Breugel [ZvB11] the progress for invariants for sequential code is equal to $1-$ (probability of reaching the sink state -1 from the initial state 0). Hence, DFS has made progress

$$1-(0.5\times 0.5+0.5\times 0.5\times 0.5+0.5\times 0.5\times 0.5)=1-(0.25+0.125+0.0625)=0.5625$$

and BFS has made progress

$$1-0.5\times 0.5=0.75.$$

We can map the progress in terms of the number of explored transitions as follows.

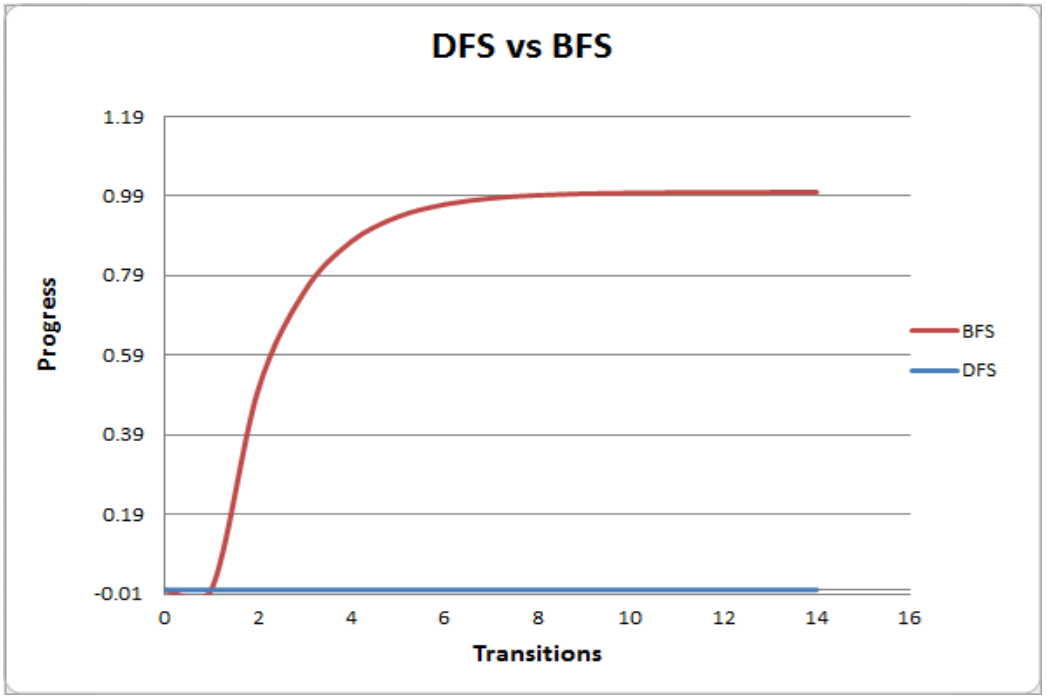


Figure 5.1: Progress vs transition

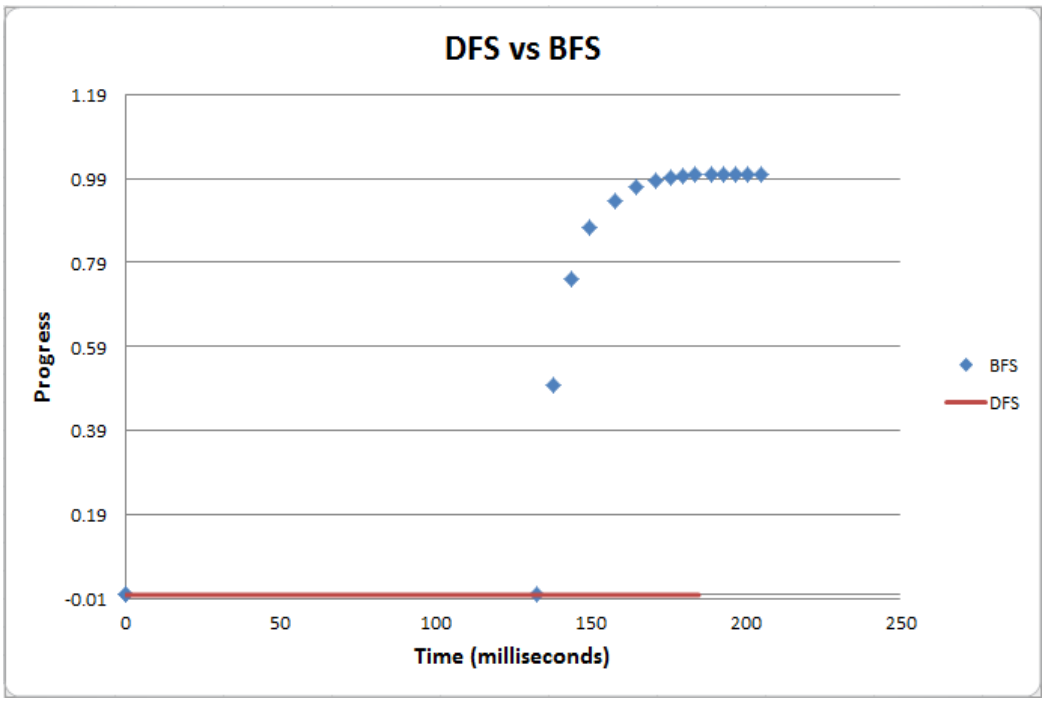


Figure 5.2: Progress vs time

However, since different search strategies may take different amounts of time to process a transition, one can also map the amount of progress versus the amount

of time it took to make that progress (Figure 5.2). BFS runs out of memory after 7 minutes and 6 seconds. DFS does not make any progress before running out of memory in 7 minutes and 53 seconds.

In Chapter 6, we will use the latter type of graphs to compare the different search strategies.

5.2 Algorithm to Compute Progress

As we have seen in the previous section, to compute the progress it suffices to compute the probability of reaching the sink state from the initial state. Next, we sketch how to compute such reachability probabilities. For more details we refer the reader to, for example, [BK08, Section 10.1.1].

First, we determine the set of states S_0 that cannot reach the sink state. This can done by a simple graph algorithm. For these states, the probability of reaching the sink state is zero. If the initial state belongs to S_0, then the progress is one. For the BFS example of the previous section, $S_0 = \{1\}$.

Next, we determine the set of states S_1 that always reach the sink state. Also this can be accomplished by a simple graph algorithm. For these states, the probability of reaching the sink state is one. Note that the sink state is included in S_1. If the initial state belongs to S_1, then the progress is zero. For the BFS example of the previous section, $S_1 = \{-1\}$.

The set of remaining states is denoted by $S_?$. For the BFS example of the previous section, $S_? = \{0, 2\}$. In case the initial state belongs to $S_?$, we compute the reachability probabilities for the remaining states as follows.

Let $\mathbf{P}$ be the probability matrix, that is, the entry $\mathbf{P}[i, j]$ is the probability of going from state i to state j. For the BFS example of the previous section,

$$\mathbf{P} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 0 \\ 0.5 & 0 & 0.5 & 0 \end{bmatrix}$$

Next, we will compute the vector $\mathbf{x}$, where the entry $\mathbf{x}[i]$ is the probability of reaching the sink state from state i. Note that $\mathbf{x}[0]$ is the probability of reaching the sink state from the initial state 0.

The vector $\mathbf{x}$ satisfies the equation

$$\mathbf{x}[i] = \sum_{j \in S} \mathbf{P}[i, j] \times \mathbf{x}[j] = \left(\sum_{j \in S_?} \mathbf{P}[i, j] \times \mathbf{x}[j] \right) + \sum_{j \in S_1} \mathbf{P}[i, j]. \tag{5.1}$$

We define the vector $\mathbf{b}$ such that the entry $\mathbf{b}[i] = \sum_{j \in S_1} \mathbf{P}[i, j]$ for each state $i \in S_?$. For the BFS example of the previous section, $\mathbf{b} = [0, 0, 0, 0.5]$. Hence, we can rewrite Equation (5.1) as

$$\mathbf{x}[i] = \left(\sum_{j \in S_?} \mathbf{P}[i, j] \times \mathbf{x}[j] \right) + \mathbf{b}[i].$$

That is,

$$\mathbf{x} = \mathbf{P} \times \mathbf{x} + \mathbf{b}. \tag{5.2}$$

Equation (5.2) can be rewritten as

$$(\mathbf{I} - \mathbf{P}) \times \mathbf{x} = \mathbf{b},$$

where $\mathbf{I}$ is the identity matrix. Now let $\mathbf{A} = \mathbf{I} - \mathbf{P}$. Then

$$\mathbf{A}[i,j] = \begin{cases} 1 - \mathbf{P}[i,j] & \text{if } i = j \\ -\mathbf{P}[i,j] & \text{otherwise} \end{cases}$$

and

$$\mathbf{A} \times \mathbf{x} = \mathbf{b}. \tag{5.3}$$

We solve Equation (5.3) by means of the Jacobi method. For a detailed discussion of the Jacobi method, we refer the reader to, for example, [Saa03, Section 4.1]. Our implementation of the progress computation can be found in the CD.

Chapter 6

Experimental Results

In this chapter, we present the experimental results of the search strategies and analyse the performance of them. To evaluate and performance of the search strategies we proposed, we model checked a randomized version of quicksort for each of the search strategies: DFS, BFS, RS, PFS, SMS and EGS. The last two ones are new. All the search strategies except DFS store states in a container which is implemented by means of a linked list or a red-black tree. RSlist and BFS use the linked list based container while RSrbt, SMS and EGS use the red-black tree based container. We compare the amount of progress different search strategies make and discuss why some search strategies work better.

6.1 Experiment Configuration

To ensure the correctness of the new search strategies (SMS and EGS), before doing an experiment for a newly proposed search strategy we first tested the implementation of them. We automatically created 178508 simple applications that use the method `Choice.make`. An example is shown in Appendix C.

We created a listener that keeps track of the number of states and transitions visited by a search. A state is only counted when it is first discovered. For each test, we ran DFS and a new search strategy and we made sure that the number of states and transitions visited were the same for both search strategies. Both SMS and EGS passed this correctness test.

For each search strategy, we model checked the randomized quicksort application ten times. We will take a closer look at the application in Section 6.2. Below is some information about the setting.

Experiment Setting	
Machine:	two AMD Athlon 64 X2 dual core processors and 4 GB of memory.
Linux version:	CentOS release 6.4
Java version:	1.7.0_25
JPF version:	7 (despite the fact that the output says 6)

In the directory where we can find the randomized quicksort application `QuickTest.class`, we create the application properties file named `QuickTest.jpf` with the following contents.

```
1 @using=jpf-probabilistic
2 target=probabilistic.QuickTest
3 classpath=/cs/fac/packages/jpf/jpf-probabilistic/build/jpf-
     probabilistic-examples.jar
4 search.class=.......
5 listener=TransitionsAndTime
```

The key `target` is set to be `probabilistic.QuickTest` as its value. `probabilistic.QuickTest` is the name of the randomized quicksort application to be checked by JPF. The key `classpath` has JPF's classpath as its value. We set `search.class` to be the search class we use for the model checking process. For example, if we use PFS as the search strategy, it uses `probabilistic.search.PFSearch` as its value. The key `listener` is set to `TransitionsAndTime`.

The listener `TransitionsAndTime` provided in Appendix D prints the transitions and timing information. It prints the current time in milliseconds at the start of the search. Then it prints the timestamps in milliseconds after every 1000 transitions. The transitions and timing information are saved to a file. In total, there are 442889 transitions. Here are the first few lines of the output for one of the experiments for DFS.

```
1 T: 1374864281262
2 JavaPathfinder v6.0 (rev 1035+) - (C) RIACS/NASA Ames Research Center
3
4
5 ========================================================= system under
     test
6 probabilistic.QuickTest.main()
7
8 ========================================================= search started:
      7/26/13 2:44 PM
9 0 0.07692307692307693 1
10 1 0.1111111111111111 2
11 2 0.16666666666666666 3
12 3 0.2 4
13 4 0.25 5
14 5 0.3333333333333333 6
15 6 0.5 7
16 7 0.3333333333333333 8
17 8 0.5 9 *
18 8 0.5 9 *
19 7 0.3333333333333333 10
```

The first line shows the current time in millisecond when the search started. Line 2

shows the version and copyright information of JPF. Line 5-6 describes the system under test which is the `main` method of the `probabilistic.QuickTest` class. Line 9-19 shows the state transitions. Line 12 shows that it transfers from state 3 to state 4 with probability 0.2. Line 17 shows a transition from state 8 to state 9 with probability 0.5. This line also indicates that state 9 is a final state by means of the *.

This file is used as input for an application that computes the progress after every 1000 transitions. Since computing the progress usually takes a lot of time, we split up the problem. Given number n and m, with $n < m$, the subproblem `progress(n,m)` computes the progress after $n*1000$ transitions, $(n+1)*1000$ transitions, ..., $m*1000$ transitions. We split the problem `progress(0,443)` into 48 subproblems. The set of subproblems are shown in Appendix E. The subproblems were run on 48 machines in parallel. As a consequence, the time to compute the progress was reduced by almost a factor 50.

A script started the progress computations on the different machines. Another script combined the output produced by the 48 machines into a single file. Here are the first few lines for one of the DFS experiments.

```
1  0       1.000000000F0   0
2  1       0.9880638651    4168
3  2       0.9841100258    6193
4  3       0.9643806810    8045
5  4       0.9388524284    9836
6  5       0.9357091982    11591
7  6       0.8458966445    13190
8  7       0.8450928894    14808
9  8       0.8444630986    16657
10 9       0.8431126487    18148
11 10      0.8410406916    19618
```

where the first column is a counter, the second one represents $1 - progress$, the third one is the time (in milliseconds). For example, after 4,000 transitions it has made $1 - 0.9388524284 \approx 0.06$ of progress and it took 9836 milliseconds to check the 4,000 transitions.

6.2 Randomized Quicksort

The randomized algorithm which we model checked is the randomized quicksort application. It is a three-step process for sorting a list of n integers *list*. Firstly, randomly choose an element as *pivot* from *list* and partition *list* into two sublists *list_smaller* and *list_larger*. All the elements of *list_smaller* are smaller than or equal to *pivot* while all the elements of *list_larger* are larger than *pivot*. Then recursively sort *list_smaller* and *list_larger* in the same way. Finally, combine the sorted lists. Algorithm 6.1 implements randomized quickshort.

Algorithm 6.1: Randomized_Quicksort(*list*)

```
Data: list (a list of integers)
Result: list (sorted in ascending order)
if list.size > 1 then
    pivot ←an element randomly chosen from list;
    for i ← 0 to list.size − 1 do
        if list.get(i) ≤ pivot then
            add list.get(i) to list_smaller;
        else
            add list.get(i) to list_larger;
        end
    end
    Randomized_Quicksort(list_smaller);
    Randomized_Quicksort(list_larger);
    clear list;
    add all elements of list_smaller to list;
    add pivot to list;
    add all elements of list_larger to list;
end
```

Recall that the amount of progress when JPF runs out of memory is of mainly interest. Since running the experiments and computing their progress is very time consuming, we simply do not have the time to consider a list such that the randomized version of quicksort runs out of memory. Given time restrictions, we only consider the list $[10, 7, 13, 1, 2, 11, 6, 8, 4, 3, 12, 9, 5]$ for which randomized quicksort terminates.

6.3 Experiment Results

Figure 6.1 shows the relationship between the progress measure and the running time after model checking using EGS with $\epsilon = 0.1$ ten times.

The horizontal axis is the running time (in milliseconds) of the model checking and the vertical axis is the amount of progress. As we can see from Figure 6.1, the results of the ten experiments using EGS differ very little. The same applies for the other search strategies except SMS.

The experiment results of SMS are shown in Figure 6.2. The chance of choosing most states stored in the container are almost equal for SMS ($\tau = 0.5$), so that the result of each experiment appears to be very random. We did experiment on SMS ($\tau = 10^{-30}$) and the results are shown in Figure 6.3. Compared to the other search strategy, the difference of the ten data sets are quite large. The performance is getting even worse, we will give the explanation later.

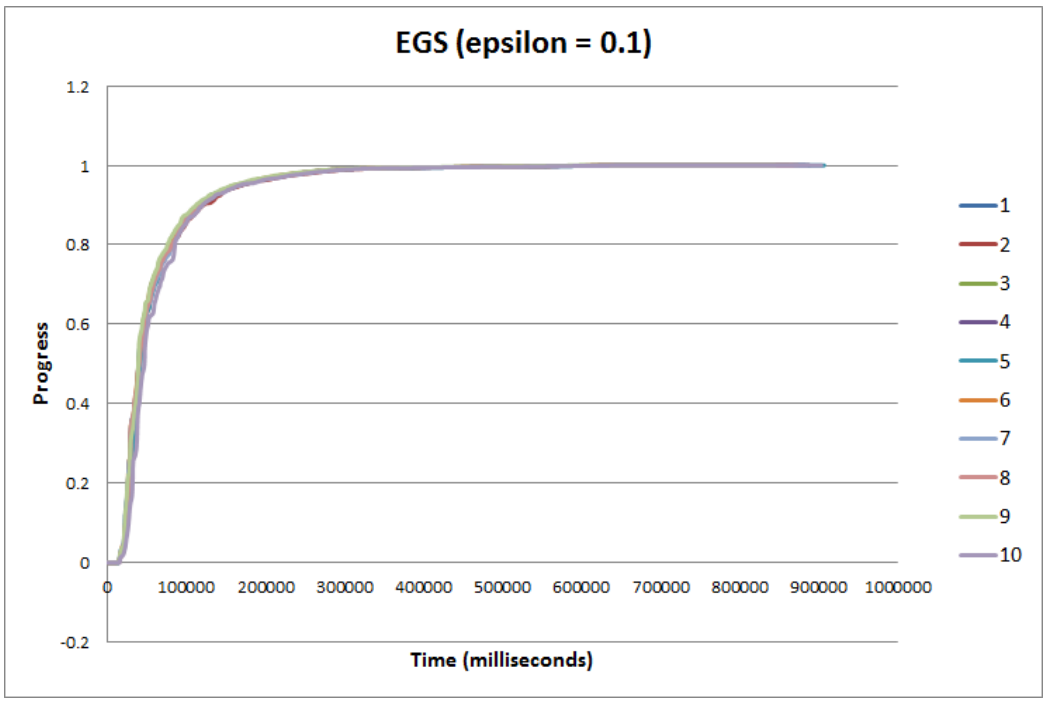


Figure 6.1: The data sets of EGS.

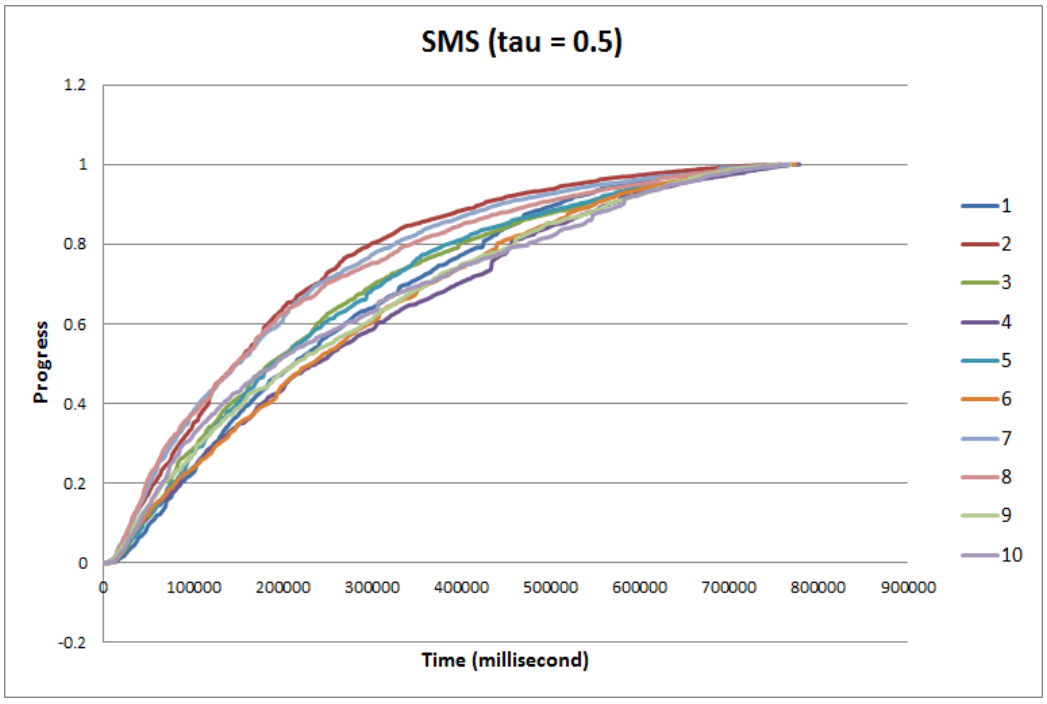


Figure 6.2: The data sets of SMS ($\tau = 0.5$).

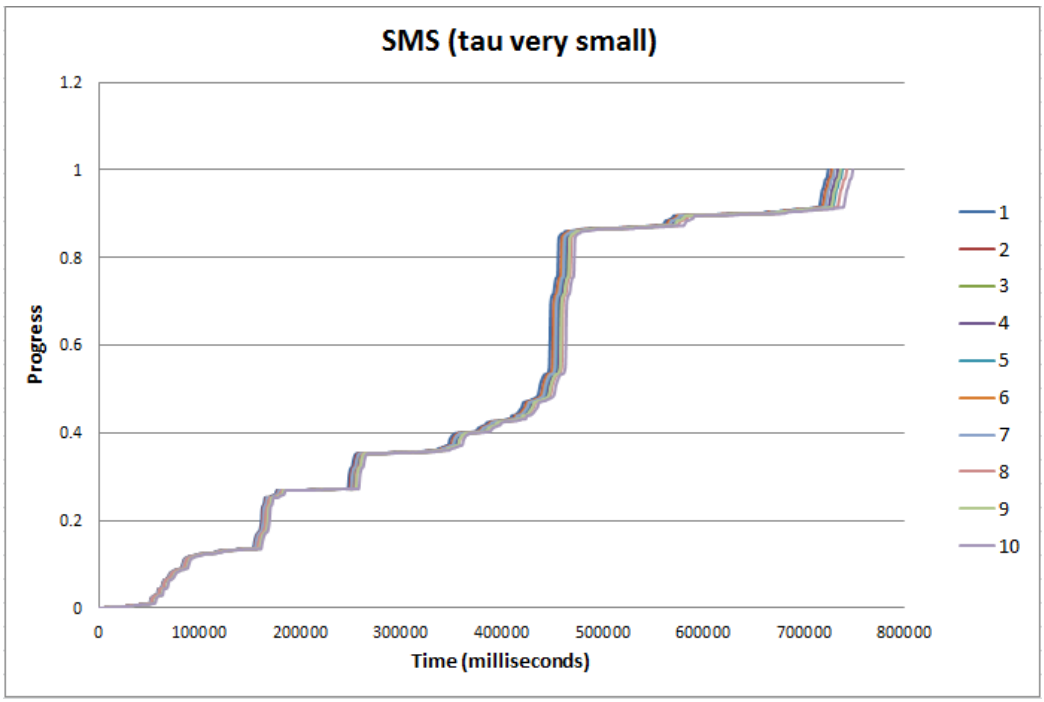


Figure 6.3: The data sets of SMS ($\tau = 10^{-30}$).

In Table 6.1 and Figure 6.4, we take the data of the first experiment for each search strategy except SMS. For SMS, we take the data of the fifth experiment. As we can see in Figure 6.2 it is almost the middle line of the experiments.

Table 6.1: Comparison of different search strategy

Search Strategies	Progress Measure 0.2	0.4	0.6	0.8	0.9	0.95	1
DFS	149812	605732	620992	713270	725882	728000	728495
BFS	113326	131972	149090	283343	313565	315148	799452
RSlist	52250	79144	130889	217445	346019	501025	2705955
RSrbt	31429	43756	63327	100297	145633	202982	1282618
PFS	24764	33368	43512	78134	114080	161882	911678
SMS ($\tau = 0.5$)	77244	150661	246914	384888	507689	583092	775617
EGS ($\epsilon = 0.1$)	26291	33939	45593	79854	116150	167627	881678
EGS ($\epsilon = 0.5$)	28492	37361	54478	89380	127849	175273	975558
RLS	8084	20301	39973	105756	255427	294390	744878

Table 6.1 shows the time (in milliseconds) it takes JPF to achieve different amount of progress for different search strategies. It is obvious from the table that DFS takes the least amount of time. However, we are not so much interested in the total amount

of time it takes for JPF to verify the application. We are much more interested in the time it takes to reach progress 0.8, 0.9, 0.95 etc.

PFS and EGS ($\epsilon = 0.1$) take the least amount of search time to reach 0.8 of progress, which remains the same for 0.9 and 0.95 of progress. EGS ($\epsilon = 0.5$) takes slightly more time since it spends more time exploring. RSrbt takes less than half the time of RSlist to achieve 0.8, 0.9 ,0.95 of progress. This is because the implementation of the red-black tree is much more efficient than the linked list. RLS performs the best until reaching the progress of 0.6. After that, it slows down its speed of progressing. It takes almost the same amount of time as RSrbt to reach the progress of 0.8. It takes less time to reach the progress of 0.9 and 0.95 than BFS. DFS performs worst. It takes about 80% of the total time to achieve the progress of 0.4.

Figure 6.4 compares all the search strategies. It shows clearly that PFS and EGS ($\epsilon = 0.1$) make the fastest progress. EGS ($\epsilon = 0.5$) and RSrbt are a little worse. BFS and RSlist are comparable to each other. DFS makes the slowest progress because DFS always tries to search as far as along each path and does not take probabilities into consideration.

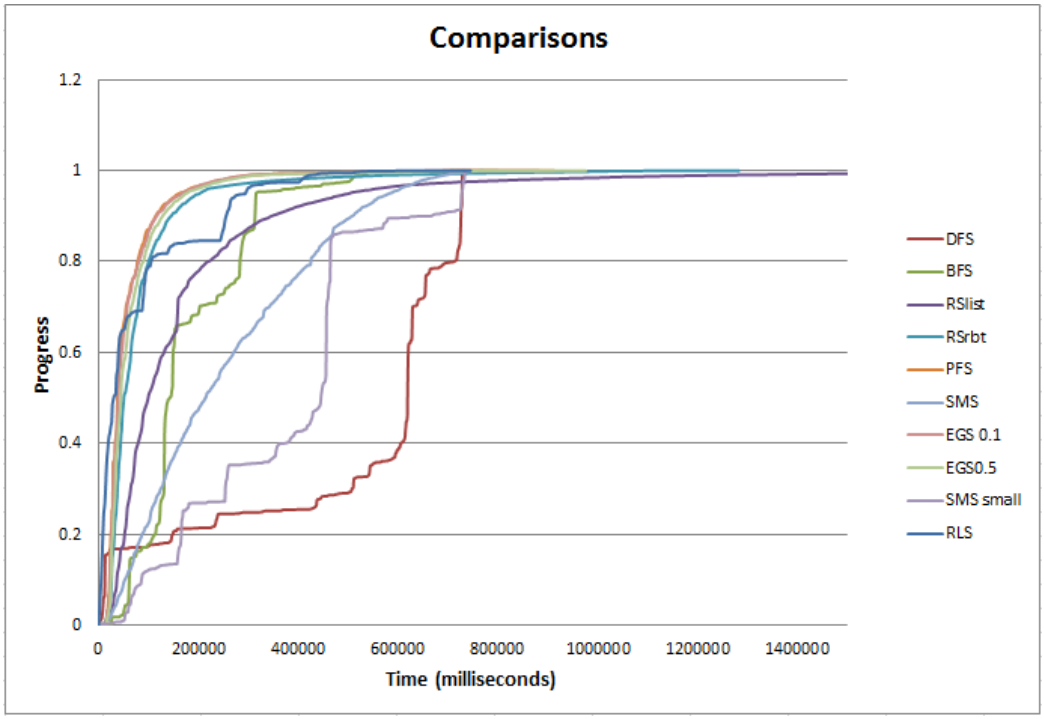
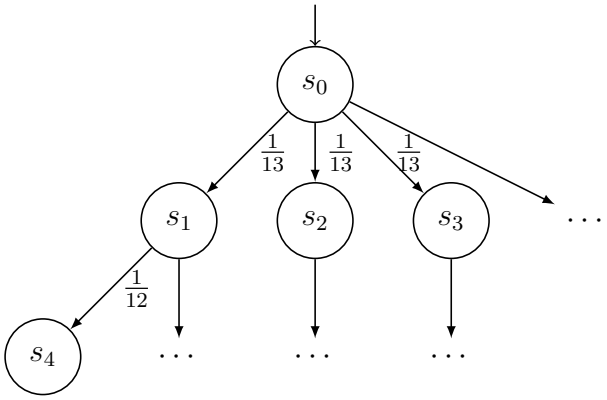


Figure 6.4: Comparison of different search strategies.

SMS ($\tau = 0.5$) makes a little faster progress than DFS since it chooses states randomly. For the random quicksort application under checking, there are 13 numbers in the list to sort. A small part of the state space of the application is shown below.



Recall that in SMS, the weight of a state is determined by $e^{p(s_j)/\tau}$. Then the weight of s_1, s_2 and s_3 is

$$e^{\frac{1}{13}/0.5} \approx 1.163$$

respectively. The weight of s_4 is

$$e^{\frac{1}{13}*\frac{1}{12}/0.5} \approx 1.013$$

We can see that the weight of any state in the state space approaches 1. So the probability of choosing any state is quite similar. A smaller value of τ would not improve the situation. Take $\tau = 0.01$ as an example. The weight of s_1, s_2 and s_3 is

$$e^{\frac{1}{13}/0.01} \approx 2191.415$$

respectively. The weight of s_4 is

$$e^{\frac{1}{13}*\frac{1}{12}/0.01} \approx 1.898$$

When the probability of the path along which the state is discovered becomes even smaller, the weight approaches 1 again. There are only several states with large weights.

With $\tau = 10^{-30}$, the overflow occurs when computing the weight of each state. All the states then are chosen completely random. In the situation that there is no proper value of τ, we conclude that SMS is not a favourable search strategy.

RLS outperforms PFS at first and stops making progress at progress around 0.7. It recovers its speed of making progress after some time and stops again when reaching progress around 0.85. It stops making progress since it keeps choosing the JPF states of which the corresponding RL states have big values but do not contribute to the progress making. It suggests that the equation which estimates the values of states is not optimal. Overall, RLS outperforms BFS and RSlist. Though our implementation of RLS does not give the best performance, there is plenty of room for improvement and we will suggest several methods in the conclusion.

Chapter 7

Conclusion

7.1 Summary

In this thesis we introduced three new search strategies softmax search (SMS), ϵ-greedy search (EGS) and reinforcement learning search (RLS) which are all based on some methods in tackling reinforcement learning problems. SMS is inspired by the softmax action selection and EGS is inspired by ϵ-greedy action selection. Dynamic programming methods, Monte Carlo methods and temporal-difference methods are studied. We showed how to solve a reinforcement learning problem using dynamic programming methods and temporal-difference methods by a small example. We then characterized the model checking problem into a reinforcement learning problem. and implemented RLS based on the popular Q-learning method which is a kind of temporal-difference methods. We implement the search strategies within the framework of an extension of JPF which is called `jpf-probabilistic` (Section 3.4). Random search (RS), which is described in Xin's thesis ([Zha10]), is reimplemented by a red-black tree. Furthermore, we developed scripts to parallelize the computation of progress.

We measured the amount of progress in terms of the amount of time it took to make that progress by different search strategy. Most of the time, model checkers like JPF would suffer state explosion problem, so we are more interested in the progress when JPF runs out of memory. We mainly focus on the time the search strategy takes to reach progress of 0.8, 0.9, 0.95 etc. It is shown in Section 6.3, EGS is comparable with PFS, while in most cases SMS chooses states randomly regardless of the probability of the transition. Our implementation of RS in red-black tree performs much better than the one in linked list. RLS shows its potential by outperforming all the other search strategies before reaching progress 0.7, though it slows down afterwards. Our implementation does provide a blueprint for applying the techniques from reinforcement learning to search strategies in JPF.

7.2 Future Work

To improve the performance of RLS, as mentioned in Section 4.7, we can formulate the transitions and the transition probability function. When estimating the value of RL state in Q-learning, instead of using the simplified version (Equation (4.4)), it is suggested to replace the value function $V(s)$ with the action-value function $Q(s, a)$, which results in the original equation Equation (2.9). As the performance of RLS suggests, we could use a different reward function or a different value function so that RLS would not keep choosing the states which have big values but are not the most favourable ones. Moreover, to balance the exploration and exploitation ([SB98, Section 2.2]), when estimating the value function, the ϵ-greedy method could replace the greedy method to generate a new episode.

Appendix A

Implementing a Search Strategy in JPF

In this appendix we first explain how to implement a search strategy within JPF. As a running example, we use depth-first search (DFS). We name our class `DFSearch`. Later in this appendix, we shall also consider BFS, PFS, RS, SMS and EGS.

A.1 The Structure of the Class

The `Search` class, which resides in the package `gov.nasa.jpf.search`, contains numerous attributes and methods that may be of use for implementing a search strategy. Hence, we extend this class.

```
1 import gov.nasa.jpf.search.Search;
2
3 public class DFSearch extends Search {
4   ...
5 }
```

The constructor of the `Search` class takes two arguments. The first argument is a `Config` object. The class `Config` is part of the package `gov.nasa.jpf`. The `Config` object contains the JPF properties. These properties can be set, for example, in the `jpf.properties` file. The second argument is a `VM` object. The `VM` object provides the search a reference to JPF's virtual machine. To properly initialize the `Search` object, we add the following constructor to our `DFSearch` class (and import the classes `Config` and `VM`).

```
1 public DFSearch(Config config, VM vm) {
2   super(config, vm);
3 }
```

If JPF is configured to use our `DFSearch`, JPF will construct a `DFSearch` object with a `Config` object capturing JPF's configuration and a `VM` object representing JPF's virtual machine.

The `Search` class is abstract. It contains the abstract method `search`. This method drives the search. It visits the states of the system under test in a systematic way by traversing transitions. In our `DFSearch` we implement the search method. We develop our implementation of the `search` method in a number of steps. We start with the implementation of

A.2 The Basic Search

To implement the basis of the search, we use the following three methods from the `Search` class that categorize the current state. The method `isNewState()` tests whether the current state is new,

that is, it has not been visited before by the search. The method `isEndState()` tests whether the current state is a final state. The method `isIgnoredState()` tests whether the current state should be ignored by the search. States can, for example, be ignored by using in the system under test the method `ignoreIf(boolean)` of JPF's class `Verify` which is part of the package `gov.nasa.jpf.vm`. Several other methods that characterize the current state can be found in the `Search` class.

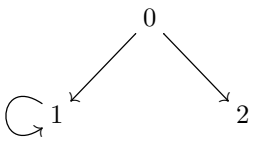
To visit the states, the `Search` class provides the following two methods. The `backtrack()` method returns the search to the source state of the transition along which the current state was discovered by the search. The method returns a boolean: whether the backtrack was successful. A backtrack fails, for example, in the initial state. In that case, the methods returns false and the search stays in the same state. The `forward()` method moves the search from the current state to another state along an unexplored transition. The method returns a boolean: whether the forward was successful. A forward fails, for example, if there are no unexplored transitions from the current state. In that case, the methods returns false and the search stays in the same state. The `forward()` method also checks whether any property is violated after the unexplored transition has been traversed (we will come back to this later).

The class `VM` contains the method `restoreState(RestorableVMState)` which restores the given state, which has been visited before. The class `RestorableVMState`, which is part of package `gov.nasa.jpf.vm`, represents a state that can be restored. Although we do not need the method `restoreState(RestorableVMState)` to implement DFS, it comes in handy when implementing BFS as we will show later.

Using the above mentioned methods, we can implement DFS as follows.

```
1  public void search() {
2    while (true) {
3      if (!isNewState() || isEndState() || isIgnoredState()) {
4        if (!backtrack()) {
5          break;
6        }
7      }
8      forward();
9    }
10 }
```

Consider a system under test that gives rise to the following states and transitions.



Starting in the initial state 0, DFS gives rise to the following sequence of invocations of `forward` and `backward`.

`forward`	move from state 0 to state 1
`forward`	move from state 1 to state 1
`backtrack`	move from state 1 to state 1
`forward`	fails; stay in state 1
`backtrack`	move from state 1 to state 0
`forward`	move from state 0 to state 2
`backtrack`	move from state 2 to state 0
`forward`	fails; stay in state 0
`backtrack`	fails; stay in state 0

Below, we extend the above `search()` method in several ways by adding different, orthogonal, aspects to `search()`.

A.3 Other Components

Other components of JPF, such as listeners, can end a search by setting the attribute `done` of the class `Search` to true. This is reflected in the `search()` method as follows.

```
1  public void search () {
2    while (!done) {
3      if (!isNewState() || isEndState() || isIgnoredState()) {
4        if (!backtrack()) {
5          break;
6        }
7      }
8      forward();
9    }
10 }
```

Other components of JPF can also request a search to backtrack by means of the method `requestBacktrack()` of the class `Search`. This method simply sets the boolean attribute `doBacktrack` to true. The method `checkAndResetBacktrackRequest()` of the class `Search` tests whether a backtrack has been requested and resets the attribute `doBacktrack` to false. Requests of backtracks can be addressed in our search method as follows.

```
1  public void search () {
2    while (true) {
3      if (!isNewState() || isEndState() || isIgnoredState() ||
4          checkAndResetBacktrackRequest()) {
5        if (!backtrack()) {
6          break;
7        }
8      }
9      forward();
10   }
11 }
```

A search can be configured in several ways. Next, we will introduce the JPF properties relevant to a search.

A.4 JPF Properties

JPF can be configured to limit the depth of the search by setting the JPF property `search.depth_limit`. The default value of `search.depth_limit` is `Integer.MAX_VALUE`. The `Search` class contains the attribute `depth` that can be used to keep track of the depth of the search. It also provides the method `getDepthLimit()` which returns the maximal allowed depth of the search.

We can limit the depth of the search as follows.

```
1  public void search () {
2    final int MAX_DEPTH = getDepthLimit();
3    depth = 0;
4    while (true) {
5      if (!isNewState() || isEndState() || isIgnoredState() ||
6          depth >= MAX_DEPTH) {
7        if (!backtrack()) {
8          break;
9        }
10       depth--;
11     }
```

```
12      if (forward()) {
13        depth++;
14      }
15    }
16  }
```

Note that if the maximal depth has been reached, the search tries to backtrack. If that fails, the search terminates.

The JPF property `search.min_free` captures the minimal amount of memory, in bytes, that needs to remain free. The default value is 2^{20}. By leaving some memory free, JPF can report that it ran out of memory and provide some useful statistics instead of simply throwing an `OutOfMemoryError`. The method `checkStateSpaceLimit()` of the class `Search` checks whether the minimal amount of memory that should be left free is still available.

We end the search if we run almost out of memory as follows.

```
1  public void search () {
2    while (true) {
3      if (!isNewState() || isEndState() || isIgnoredState()) {
4        if (!backtrack()) {
5          break;
6        }
7      }
8      if (forward()) {
9        if (!checkStateSpaceLimit()) {
10         break;
11       }
12     }
13   }
14 }
```

The JPF property `search.multiple_errors` tells us whether the search should report multiple errors (or just the first one). The default value is false (that is, by default only the first error is reported after which the search ends).

Recall that the `forward()` method also checks whether any property is violated after the unexplored transition has been traversed. Immediately after an invocation of the `forward()` method, the attribute `currentError` of the class `Search` is null if and only if no such a violation has been detected. Furthermore, if a violation has been detected then the attribute `done` is set to true if and only if JPF has been configured to report at most one error.

The method `hasPropertyTermination()` of the class `Search` checks whether a violation was encountered during the last transition. The method returns true if and only if the value of the attribute `currentError` is different from null and the value of the attribute `done` is true. We often use the method `hasPropertyTermination()` as follows.

```
1  if (forward()) {
2    // forward succeeded
3    if (currentError != null) {
4      // violation detected
5      if (hasPropertyTermination()) {
```

In the above context, `hasPropertyTermination()` returns true if and only if JPF has been configured to report at most one error. Furthermore, if `hasPropertyTermination()` returns false (which in this context denotes that JPF has been configured to report multiple errors), then it also sets the attribute `doBacktrack` requesting a backtrack to true.

```
1  public void search () {
2    while (!done) {
3      if (!isNewState() || isEndState() || isIgnoredState() ||
```

```
4         checkAndResetBacktrackRequest()) {
5       if (!backtrack()) {
6         break;
7       }
8     }
9     if (forward()) {
10      if (currentError != null) {
11        if (hasPropertyTermination()) {
12          break;
13        }
14      }
15    }
16  }
17 }
```

In the case that the `forward()` method encounters a violation, it either terminates the search (if JPF has been configured to report at most one error) or it requests a backtrack (if JPF has been configured to report multiple errors).

A.5 Notifications

A search should also notify listeners of particular events. The `Search` class provides the following methods. Note that the methods below correspond to the methods of the interface `SearchListener`, which can be found in the package `gov.nasa.jpf.search`. We have divided the methods into four groups.

The first group contains methods that notify listeners of events related to the current state of the search. The method `notifyStateAdvanced()` notifies the listeners that the current state has been reached as a result of a successful `forward()` invocation. The method `notifyStateProcessed()` notifies the listeners that the current state has been fully explored, that is, it has no unexplored outgoing transitions. The method `notifyStateBacktracked()` notifies the listeners that the current state has been reached by means of a backtrack.

The method `notifySearchStarted()` notifies the listeners that the search has started and the method `notifySearchFinished()` notifies the listeners that the search has finished.

The method `notifyPropertyViolated()` notifies the listeners that a violation has been encountered in the current state.

The method `notifySearchConstraintHit(String)` notifies the listeners that the given constraint has been violated. The string specifies which constraint has been violated. For example, the string `"memory limit reached"` can be used if we run almost out of memory.

Below, we present the simplest extension of the basic search so that we can introduce all the notifications apart from `notifyPropertyViolated()` and `notifySearchConstraintHit()`. The latter two types of notification will be added later.

```
1  public void search () {
2    notifySearchStarted();
3    while (true) {
4      if (!isNewState() || isEndState() || isIgnoredState()) {
5        if (!backtrack()) {
6          break;
7        }
8        notifyStateBacktracked();
9      }
10     if (forward()) {
11       notifyStateAdvanced();
12     } else {
```

```
13        notifyStateProcessed();
14      }
15    }
16    notifySearchFinished();
17  }
```

A.6 The Complete Search

Combining all the above and adding the appropriate invocations of the `notifyPropertyViolated()` and `notifySearchConstraintHit()` methods, we arrive at the following.

```
1   public void search () {
2     final int MAX_DEPTH = getDepthLimit();
3     depth = 0;
4     notifySearchStarted();
5     while (!done) {
6       if (!isNewState() || isEndState() || isIgnoredState() ||
7           depth >= MAX_DEPTH || checkAndResetBacktrackRequest()) {
8         if (!backtrack()) {
9           break;
10        }
11        depth--;
12        notifyStateBacktracked();
13      }
14      if (forward()) {
15        depth++;
16        notifyStateAdvanced();
17        if (currentError != null) {
18          notifyPropertyViolated();
19          if (hasPropertyTermination()) {
20            break;
21          }
22        }
23        if (depth >= MAX_DEPTH) {
24          notifySearchConstraintHit("depth limit reached");
25        }
26        if (!checkStateSpaceLimit()) {
27          notifySearchConstraintHit("memory limit reached");
28          break;
29        }
30      } else {
31        notifyStateProcessed();
32      }
33    }
34    notifySearchFinished();
35  }
```

The above search method is slightly different from the one in the `DFSearch` class of the `gov.nasa.jpf.search` package.

A.7 Breadth-First Search

As a second example, we implement BFS in a class named `BFSearch`. We start with extending the class `Search`.

```
1  import gov.nasa.jpf.Config;
2  import gov.nasa.jpf.vm.VM;
3  import gov.nasa.jpf.search.Search;
4
5  public class BFSearch extends Search
6  {
7    public BFSearch(Config config, VM vm) {
8      super(config, vm);
9    }
10 }
```

To implement the basic search, we need a few new ingredients. To implement BFS, we not only need `forward()` and `backtrack()`, but also `restoreState(RestorableVMState)`. The latter method restores the state given as an argument and can be found in the `VM` class. This class also contains the method `getRestorableState()` which returns a `RestorableVMState` object representing the current state. The `Search` class contains an attribute named `vm` of type `VM` that represents JPF's VM.

```
1  public void search() {
2    List<RestorableVMState> currentLevel = new LinkedList<RestorableVMState>();
3    currentLevel.add(vm.getRestorableState());
4    while (!currentLevel.isEmpty()) {
5      List<RestorableVMState> nextLevel = new LinkedList<RestorableVMState>();
6      Iterator<RestorableVMState> iterator = currentLevel.iterator();
7      while (iterator.hasNext()) {
8        vm.restoreState(iterator.next());
9        while (true) {
10         if (!forward()) {
11           break;
12         } else {
13           if (isNewState() && !isEndState() && !isIgnoredState()) {
14             nextLevel.add(vm.getRestorableState());
15           }
16           backtrack();
17         }
18       }
19     }
20     currentLevel = nextLevel;
21   }
22 }
```

As before, the attribute `done` is used to end the search.

```
1  public void search() {
2    List<RestorableVMState> currentLevel = new LinkedList<RestorableVMState>();
3    currentLevel.add(vm.getRestorableState());
4    while (!currentLevel.isEmpty() && !done) {
5      List<RestorableVMState> nextLevel = new LinkedList<RestorableVMState>();
6      Iterator<RestorableVMState> iterator = currentLevel.iterator();
7      while (iterator.hasNext() && !done) {
8        vm.restoreState(iterator.next());
9        while (!done) {
```

```
10        if (!forward()) {
11          break;
12        } else {
13          if (isNewState() && !isEndState() && !isIgnoredState()) {
14            nextLevel.add(vm.getRestorableState());
15          }
16          backtrack();
17        }
18      }
19    }
20    currentLevel = nextLevel;
21  }
22 }
```

In our BFS implementation we decided not to support backtrack requests by other JPF components to keep things simple. The class `Search` contains the method `supportBacktrack()` which tests whether a search supports backtrack requests. This method of the `Search` class always returns true. In our subclass `BFSearch`, we override this method as follows.

```
1 public boolean supportBacktrack() {
2   return false;
3 }
```

The depth of the search can be limited as follows.

```
1  public void search() {
2    final int MAX_DEPTH = getDepthLimit();
3    depth = 0;
4    List<RestorableVMState> currentLevel = new LinkedList<RestorableVMState>();
5    currentLevel.add(vm.getRestorableState());
6    while (!currentLevel.isEmpty() && !done && depth < MAX_DEPTH) {
7      List<RestorableVMState> nextLevel = new LinkedList<RestorableVMState>();
8      Iterator<VMState> iterator = currentLevel.iterator();
9      while (iterator.hasNext() && !done) {
10       vm.restoreState(iterator.next());
11       while (!done) {
12         if (!forward()) {
13           break;
14         } else {
15           if (isNewState() && !isEndState() && !isIgnoredState()) {
16             nextLevel.add(vm.getRestorableState());
17           }
18           backtrack();
19         }
20       }
21     }
22     currentLevel = nextLevel;
23     depth++;
24   }
25 }
```

To end the search when insufficient memory is available, we use the method `checkStateSpaceLimit()` and the attribute `done` as follows.

```
1 public void search() {
2   final int MAX_DEPTH = getDepthLimit();
3   depth = 0;
```

```
4   List<RestorableVMState> currentLevel = new LinkedList<RestorableVMState>();
5   currentLevel.add(vm.getRestorableState());
6   while (!currentLevel.isEmpty() && !done && depth < MAX_DEPTH) {
7     List<RestorableVMState> nextLevel = new LinkedList<RestorableVMState>();
8     Iterator<VMState> iterator = currentLevel.iterator();
9     while (iterator.hasNext() && !done) {
10      vm.restoreState(iterator.next());
11      while (!done) {
12        if (!forward()) {
13          break;
14        } else {
15          if (isNewState() && !isEndState() && !isIgnoredState()) {
16            nextLevel.add(vm.getRestorableState());
17          }
18          backtrack();
19        }
20       if (!checkStateSpaceLimit()) {
21          done = true;
22          break;
23        }
24      }
25    }
26    currentLevel = nextLevel;
27    depth++;
28  }
29 }
```

The JPF property `search.multiple_errors` can be dealt with in the same way as in our implementation of DFS.

In the notification code, we use two methods that we have not discussed before. The method `notifyStateStored()` notifies the listeners that the current state has been stored (so that it can be restored later). The method `notifyStateRestored()` notifies the listeners that the current state has been restored (by means of the `restoreState(RestorableVMState)` method). Below, we introduce all the notifications apart from `notifyPropertyViolated()` and `notifySearchConstraintHit(String)`.

```
1  public void search() {
2    final int MAX_DEPTH = getDepthLimit();
3    depth = 0;
4    List<RestorableVMState> currentLevel = new LinkedList<RestorableVMState>();
5    notifySearchStarted();
6    currentLevel.add(vm.getRestorableState());
7    notifyStateStored();
8    while (!currentLevel.isEmpty() && !done && depth < MAX_DEPTH) {
9      List<RestorableVMState> nextLevel = new LinkedList<RestorableVMState>();
10     Iterator<RestorableVMState> iterator = currentLevel.iterator();
11     while (iterator.hasNext() && !done) {
12       vm.restoreState(iterator.next());
13       notifyStateRestored();
14       while (!done) {
15         if (!forward()) {
16           notifyStateProcessed();
17           break;
18         } else {
```

```
19          notifyStateAdvanced();
20          if (currentError != null) {
21            if (hasPropertyTermination()) {
22              break;
23            }
24          }
25          if (isNewState() && !isEndState() && !isIgnoredState()) {
26            nextLevel.add(vm.getRestorableState());
27            notifyStateStored();
28          }
29          backtrack();
30          notifyStateBacktracked();
31        }
32        if (!checkStateSpaceLimit()) {
33          done = true;
34          break;
35        }
36      }
37    }
38    currentLevel = nextLevel;
39    depth++;
40  }
41  notifySearchFinished();
42 }
```

We conclude by adding the appropriate invocations of the `notifyPropertyViolated()` and `notifySearchConstraintHit(String)` methods.

```
1  public void search() {
2    final int MAX_DEPTH = getDepthLimit();
3    depth = 0;
4    List<RestorableVMState> currentLevel = new LinkedList<RestorableVMState>();
5    notifySearchStarted();
6    currentLevel.add(vm.getRestorableState());
7    notifyStateStored();
8    while (!currentLevel.isEmpty() && !done && depth < MAX_DEPTH) {
9      List<RestorableVMState> nextLevel = new LinkedList<RestorableVMState>();
10     Iterator<RestorableVMState> iterator = currentLevel.iterator();
11     while (iterator.hasNext() && !done) {
12       vm.restoreState(iterator.next());
13       notifyStateRestored();
14       while (!done) {
15         if (!forward()) {
16           notifyStateProcessed();
17           break;
18         } else {
19           notifyStateAdvanced();
20           if (currentError != null) {
21             notifyPropertyViolated();
22             if (hasPropertyTermination()) {
23               break;
24             }
25           }
26           if (isNewState() && !isEndState() && !isIgnoredState()) {
27             nextLevel.add(vm.getRestorableState());
```

```
28            notifyStateStored();
29          }
30          backtrack();
31          notifyStateBacktracked();
32        }
33        if (!checkStateSpaceLimit()) {
34          done = true;
35          notifySearchConstraintHit("memory limit reached");
36          break;
37        }
38      }
39    }
40    currentLevel = nextLevel;
41    depth++;
42    if (depth >= MAX_DEPTH) {
43      notifySearchConstraintHit("depth limit reached");
44    }
45  }
46  notifySearchFinished();
47 }
```

Although The above search method is quite different from the one in the class `BFSHeuristic` of the package `gov.nasa.jpf.search.heuristic`, it introduced several ingredients that we will use below.

A.8 Random Search

The policy of RS is that the chance of choosing a state is proportional to the probability of the path along which the state is discovered.

RS is implemented in a class named `RandomSearch`. We start with extending the class `Search` as before. To implement the basic search, we need a probability distribution to store weighted elements where the element is a `RestorableVMState` object and its weight is the probability of the path along which the state is discovered. The class named `Weighted` works as a blueprint for such weighted elements. It implements the `Comparable` interface and overrides the `compareTo()` method, so that the elements are ordered. Besides the `compareTo()` method and constructor, it also provides two methods to give access to its element and weight. `Weighted` is a generic class parameterized over the type of the element.

```
1  public class Weighted<T> implements Comparable<Weighted<T>> {
2    private T element;
3    private double weight;
4
5    Weighted(T element, double weight) {
6      super();
7      this.element = element;
8      this.weight = weight;
9    }
10
11   public T getElement() {
12     return this.element;
13   }
14
15   public double getWeight() {
16     return this.weight;
```

```
17   }
18
19   public int compareTo(Weighted<T> other) {
20     if (this.weight < other.weight) {
21       return -1;
22     } else if (this.weight > other.weight) {
23       return 1;
24     } else {
25       return 0;
26     }
27   }
28 }
```

The methods supported by the probability distribution are listed in the interface which is called `Distribution`. The method `isEmpty()` returns true if the distribution contains no element and false otherwise. The method `add(T,double)` adds a weighted element to the distribution by giving the element and its weight. The method `remove()` removes and returns a weighted element randomly from the distribution according to the policy mentioned above.

```
1  public interface Distribution<T> {
2    public boolean isEmpty();
3    public void add(T element, double weight);
4    public Weighted<T> remove();
5  }
```

We implement the distribution by means of a linked list and a red-black tree in the classes `List` and `RedBlackTree`, respectively. These two classes are provided in the Appendix.

Using the above mentioned methods along with `forward()`, `backward()` and `restoreState(RestorableVMState)` we can implement RS as follows.

```
1  public void search() {
2    Distribution<RestorableVMState> distribution = new RedBlackTree<
       RestorableVMState>();
3    distribution.add(vm.getRestorableState(), 1.0);
4    do {
5      Weighted<RestorableVMState> selected = distribution.remove();
6      RestorableVMState source = selected.getElement();
7      vm.restoreState(source);
8      while (true) {
9        if (!forward()) {
10         break;
11       } else {
12         if (isNewState() && !isEndState() && !isIgnoredState()) {
13           double probability = selected.getWeight();
14           ChoiceGenerator<?> cg = vm.getChoiceGenerator();
15           if (cg instanceof Probabilistic) {
16             probability *= ((Probabilistic) cg).getProbability();
17           }
18           RestorableVMState target = vm.getRestorableState();
19           distribution.add(target, probability);
20         }
21         backtrack();
22       }
23     }
24   } while (!distribution.isEmpty());
25 }
```

Before starting to explore the state space in the `search()` method, we first define the earlier mentioned distribution which stores states and their probabilities. It is named `distribution`. Then we start the search from the initial state. We store the initial state in `distribution` first. In the loop, `distribution` chooses a state `selected` to restore by calling the `distribution.remove()` method.

Starting from the restored state s_r, we traverse its unexplored outgoing transitions. If it ends up in a new state s_t that should neither be ignored nor is a final state, we add the state and the probability of the path along which the state is discovered to `distribution`. Let $p(s_r)$ denote the probability of the path along which s_r is discovered. Then $p(s_r)$ is returned by `selected.getWeight()`.

Roughly, a `ChoiceGenerator` represents the outgoing transitions of a state. The method `getChoiceGenerator()` of the class `VM` returns the current `ChoiceGenerator`. `JPF` and its extension `jpf-probabilistic` contain different types of `ChoiceGenerator`. The latter contains `ChoiceGenerator`s that keep track of the probabilities of the transitions associated with the `Choice.make` method. These `ChoiceGenerator`s implement the interface `Probabilistic`. This interface contains the method `getProbability()` which returns the probability of the latest traversed transition.

Let $p(s_r, s_t)$ denote the probability of the transition from s_r to s_t. Then $p(s_r, s_t)$ is returned by `((Probabilistic)cg).getProbability()` if the current `ChoiceGenerator cg` is an instance of `Probabilistic`. In that case, the probability of the path along which s_t is discovered is $p(s_r) * p(s_r, s_t)$.

After traversing all the outgoing transitions and adding each of the visited states to `distribution`, the loop condition is checked. If `distribution` is empty, the search completes. Otherwise it continues by going back to the start of the loop.

Similar to DFS and BFS, we extend the above search method in several ways by adding different, orthogonal aspects to `search()`.

The depth of the search is limited with the help of two methods defined in the `Search` class: `setStateDepth(int, int)` and `getStateDepth(int)`. In the search, we set the depth in terms of the id of the state by invoking `setStateDepth(vm.getStateId()+ 1, depth + 1)`. Here we add 1 to the state id since the initial state has id -1. So the depth can be retrieved by invoking the method `getStateDepth(vm.getStateId()+ 1)`. If the depth limit has been reached, the search will ignore this state and restore another one. We limit the depth of the search with attribute `done` added as follows.

```
1  public void search() {
2    final int MAX_DEPTH = getDepthLimit();
3    Distribution<RestorableVMState> distribution = new RedBlackTree<
         RestorableVMState>();
4    distribution.add(vm.getRestorableState(), 1.0);
5    setStateDepth(vm.getStateId() + 1, 0);
6    do {
7      Weighted<RestorableVMState> selected = distribution.remove();
8      RestorableVMState source = selected.getElement();
9      vm.restoreState(source);
10     int depth = getStateDepth(vm.getStateId() + 1);
11     if (depth >= MAX_DEPTH) {
12       notifySearchConstraintHit("depth limit reached: " + MAX_DEPTH);
13       continue;
14     }
15     while (!done) {
16       if (!forward()) {
17         break;
18       } else {
19         if (isNewState() && !isEndState() && !isIgnoredState()) {
20           double probability = selected.getWeight();
```

```
21          ChoiceGenerator<?> cg = vm.getChoiceGenerator();
22          if (cg instanceof Probabilistic) {
23            probability *= ((Probabilistic) cg).getProbability();
24          }
25          RestorableVMState target = vm.getRestorableState();
26          distribution.add(target, probability);
27          setStateDepth(vm.getStateId() + 1, depth + 1);
28        }
29        backtrack();
30      }
31    }
32  } while (!distribution.isEmpty() && !done);
33 }
```

Insufficient memory and the JPF property `search.multiple_errors` are dealt with the same way as before. The appropriate invocations of different notifications are added in the same way as in DFS and BFS. The complete search class is given below.

```
1  public void search() {
2    notifySearchStarted();
3    final int MAX_DEPTH = getDepthLimit();
4    Distribution<RestorableVMState> distribution = new RedBlackTree<
         RestorableVMState>(); // new List<RestorableVMState>();
5    distribution.add(vm.getRestorableState(), 1.0);
6    setStateDepth(vm.getStateId() + 1, 0);
7    notifyStateStored();
8    do {
9      Weighted<RestorableVMState> selected = distribution.remove();
10     RestorableVMState source = selected.getElement();
11     vm.restoreState(source);
12     notifyStateRestored();
13     int depth = getStateDepth(vm.getStateId() + 1);
14     if (depth >= MAX_DEPTH) {
15       notifySearchConstraintHit("depth limit reached: " + MAX_DEPTH);
16       continue;
17     }
18
19     while (!done) {
20       if (!forward()) {
21         break;
22       } else {
23         notifyStateAdvanced();
24         if (currentError != null) {
25           notifyPropertyViolated();
26           if (hasPropertyTermination()) {
27             break;
28           }
29         }
30         if (isNewState() && !isEndState() && !isIgnoredState()) {
31           double probability = selected.getWeight();
32           ChoiceGenerator<?> cg = vm.getChoiceGenerator();
33           if (cg instanceof Probabilistic) {
34             probability *= ((Probabilistic) cg).getProbability();
35           }
36           RestorableVMState target = vm.getRestorableState();
```

```
37            distribution.add(target, probability);
38            setStateDepth(vm.getStateId() + 1, depth + 1);
39            notifyStateStored();
40            if (!checkStateSpaceLimit()) {
41              notifySearchConstraintHit("memory limit reached: "
42                  + minFreeMemory);
43              done = true;
44              break;
45            }
46          }
47          backtrack();
48          notifyStateBacktracked();
49        }
50      }
51    } while (!distribution.isEmpty() && !done);
52    notifySearchFinished();
53  }
```

A.9 Probability-First Search

PFS is implemented in the class `PFSearch` which extends `Search` as before. Like for RS, we need a container to store a state and the probability of the path along which the state is discovered. We should be able to retrieve from the container the state with the largest probability. The class `PVMState` contains a `RestorableVMState` and its probability. We impose a total ordering on the state by implementing the `Comparable` interface and overriding its `compareTo` method. The ordering on `PVMState`s is the opposite of the natural ordering of their probabilities. We then use the class `PriorityQueue` as the container. The head of the queue can be obtained by invoking the `poll()` method.

```
1  private static class PVMState implements Comparable<PVMState> {
2    private RestorableVMState state;
3    private double probability;
4
5    private PVMState(RestorableVMState state, double probability) {
6      this.state = state;
7      this.probability = probability;
8    }
9
10   public int compareTo(PVMState other) {
11     if (this.probability < other.probability) {
12       return 1;
13     } else if (this.probability > other.probability) {
14       return -1;
15     } else {
16       return 0;
17     }
18   }
19 }
```

The basic search is almost the same as RS with `Distribution` in RS replaced by `PriorityQueue` and `Weighted` by `PVMState`.

```
1  public void search() {
2    PriorityQueue<PVMState> queue = new PriorityQueue<PVMState>();
```

```
3   queue.add(new PVMState(vm.getRestorableState(), 1.0));
4   while (!queue.isEmpty()) {
5     PVMState head = queue.poll();
6     vm.restoreState(head.state);
7     while (forward()) {
8       if (isNewState() && !isEndState() && !isIgnoredState()) {
9         double probability = head.probability;
10        ChoiceGenerator<?> cg = vm.getChoiceGenerator();
11        if (cg instanceof Probabilistic) {
12          probability *= ((Probabilistic) cg).getProbability();
13        }
14        queue.add(new PVMState(vm.getRestorableState(), probability));
15      }
16      backtrack();
17    }
18  }
19 }
```

The complete search method adds all the other aspects to `search()` in the same way as RS and can be found below.

```
1  public void search() {
2    notifySearchStarted();
3    final int MAX_DEPTH = getDepthLimit();
4    PriorityQueue<PVMState> queue = new PriorityQueue<PVMState>();
5    setStateDepth(vm.getStateId() + 1, 0);
6    queue.add(new PVMState(vm.getRestorableState(), 1.0));
7    notifyStateStored();
8    while (!queue.isEmpty() && !done) {
9      PVMState head = queue.poll();
10     vm.restoreState(head.state);
11     notifyStateRestored();
12     int depth = getStateDepth(vm.getStateId() + 1);
13     if (depth >= MAX_DEPTH) {
14       notifySearchConstraintHit("depth limit reached: " + MAX_DEPTH);
15     } else {
16       while (forward()) {
17         notifyStateAdvanced();
18         if (currentError != null) {
19           notifyPropertyViolated();
20           if (hasPropertyTermination()) {
21             break;
22           }
23         }
24         if (isNewState() && !isEndState() && !isIgnoredState()) {
25           double probability = head.probability;
26           ChoiceGenerator<?> cg = vm.getChoiceGenerator();
27           if (cg instanceof Probabilistic) {
28             probability *= ((Probabilistic) cg).getProbability();
29           }
30           setStateDepth(vm.getStateId() + 1, depth + 1);
31           queue.add(new PVMState(vm.getRestorableState(), probability));
32           notifyStateStored();
33         }
34         backtrack();
```

```
35          notifyStateBacktracked();
36        }
37        notifyStateProcessed();
38      }
39      if (!checkStateSpaceLimit()) {
40        notifySearchConstraintHit("memory limit reached: " + minFreeMemory);
41        done = true;
42      }
43    }
44    notifySearchFinished();
45  }
```

A.10 Softmax Search

We implement SMS in a class named `SoftmaxSearch` and we start with extending the class `Search`. The search can be configured by the user by specifying the temperature τ. The method invocation `config.getDouble("tau", 0.5)` returns the value specified for `tau` if it has been configured by the user. Otherwise, it returns the default value 0.5. A new field `tau` of type double which represents the temperature is introduced in the class.

```
1  public class SoftmaxSearch extends Search {
2    private double tau;
3    public SoftmaxSearch(Config config, VM vm) {
4      super(config, vm);
5      tau = config.getDouble("tau", 0.5);
6    }
7  }
```

We can implement the basic search on the basis of RS with the weights of the states changed as follows.

```
1  public void search() {
2    Distribution<RestorableVMState> distribution = new RedBlackTree<
         RestorableVMState>();
3    distribution.add(vm.getRestorableState(), Math.exp(1.0 / tau));
4    do {
5      Weighted<RestorableVMState> selected = distribution.remove();
6      RestorableVMState source = selected.getElement();
7      vm.restoreState(source);
8      while (true) {
9        if (!forward()) {
10         break;
11       } else {
12         if (isNewState() && !isEndState() && !isIgnoredState()) {
13           double probability = selected.getWeight();
14           ChoiceGenerator<?> cg = vm.getChoiceGenerator();
15           if (cg instanceof Probabilistic) {
16             probability = Math.pow(probability,((Probabilistic) cg).getProbability
                   ());
17           }
18           RestorableVMState target = vm.getRestorableState();
19           distribution.add(target, probability);
20         }
21         backtrack();
```

```
22      }
23    }
24  } while (!distribution.isEmpty());
25 }
```

The complete search method adds all the other aspects to `search()` in the same way as RS and PFS and can be found below.

```
1  public void search() {
2    notifySearchStarted();
3    final int MAX_DEPTH = getDepthLimit();
4    Distribution<RestorableVMState> distribution = new RedBlackTree<
         RestorableVMState>();
5    distribution.add(vm.getRestorableState(), Math.exp(1.0 / tau));
6    setStateDepth(vm.getStateId() + 1, 0);
7    notifyStateStored();
8    do {
9      Weighted<RestorableVMState> selected = distribution.remove();
10     RestorableVMState source = selected.getElement();
11     vm.restoreState(source);
12     notifyStateRestored();
13     int depth = getStateDepth(vm.getStateId() + 1);
14     if (depth >= MAX_DEPTH) {
15       notifySearchConstraintHit("depth limit reached: " + MAX_DEPTH);
16       continue;
17     }
18     while (!done) {
19       if (!forward()) {
20         break;
21       } else {
22         notifyStateAdvanced();
23         if (currentError != null) {
24           notifyPropertyViolated();
25           if (hasPropertyTermination()) {
26             break;
27           }
28         }
29         if (isNewState() && !isEndState() && !isIgnoredState()) {
30           double probability = selected.getWeight();
31           ChoiceGenerator<?> cg = vm.getChoiceGenerator();
32           if (cg instanceof Probabilistic) {
33             probability = Math.pow(probability,((Probabilistic) cg).getProbability
                   ());
34           }
35           else{
36             System.out.println("not random");
37           }
38           RestorableVMState target = vm.getRestorableState();
39           distribution.add(target, probability);
40           setStateDepth(vm.getStateId() + 1, depth + 1);
41           notifyStateStored();
42           if (!checkStateSpaceLimit()) {
43             notifySearchConstraintHit("memory limit reached: "
44                 + minFreeMemory);
45             done = true;
```

```
46            break;
47          }
48        }
49        backtrack();
50        notifyStateBacktracked();
51      }
52    }
53  } while (!distribution.isEmpty() && !done);
54  notifySearchFinished();
55 }
```

A.11 ϵ-Greedy Search

EGS is implemented in a class named `EpsilonGreedySearch` by starting with extending the class `Search` and adding a new field `epsilon` of type double. If the user does not specify a value for ϵ, then we set `epsilon` to 0.1.

```
1 public class EpsilonGreedySearch extends Search {
2   private double epsilon;
3   public EpsilonGreedySearch(Config config, VM vm) {
4     super(config, vm);
5     epsilon = config.getDouble("epsilon", 0.1);
6   }
7 }
```

We based our search method on the search method of RS since with probability ϵ it behaves like RS. The `Weighted` class, which represents an element and its weight, can be reused. We add a method `removeMax()` which returns the element with the highest weight to the `Distribution` interface. Any class that implements the interface should implement the method `removeMax()`. In the basic search, we generate a random value between 0.0 and 1.0 by invoking `Math.random()`. If the value is greater than or equal to `epsilon`, we restore the state with the highest probability by invoking `distribution.removeMax()`. Otherwise, like in RS we choose a random state to restore by invoking the `distribution.remove()`.

```
1  public void search() {
2    Distribution<RestorableVMState> distribution = new RedBlackTree<
         RestorableVMState>();
3    distribution.add(vm.getRestorableState(), 1.0);
4    do {
5      double chance = Math.random();
6      Weighted<RestorableVMState> selected;
7      if (chance >= epsilon) {
8        selected = distribution.removeMax();
9      } else {
10       selected = distribution.remove();
11     }
12     RestorableVMState source = selected.getElement();
13     vm.restoreState(source);
14     int depth = getStateDepth(vm.getStateId() + 1);
15     while (true) {
16       if (!forward()) {
17         break;
18       } else {
19
```

```
20        if (isNewState() && !isEndState() && !isIgnoredState()) {
21          double probability = selected.getWeight();
22          ChoiceGenerator<?> cg = vm.getChoiceGenerator();
23          if (cg instanceof Probabilistic) {
24            probability *= ((Probabilistic) cg).getProbability();
25          }
26          RestorableVMState target = vm.getRestorableState();
27          distribution.add(target, probability);
28        }
29        backtrack();
30      }
31    }
32  } while (!distribution.isEmpty());
33 }
```

The basic search is shown above and the complete search method which adds all the other aspects to `search()` in the same way as RS, PFS and SMS can be found below.

```
1  public void search() {
2    notifySearchStarted();
3    final int MAX_DEPTH = getDepthLimit();
4    Distribution<RestorableVMState> distribution = new RedBlackTree<
         RestorableVMState>();
5    distribution.add(vm.getRestorableState(), 1.0);
6    setStateDepth(vm.getStateId() + 1, 0);
7    notifyStateStored();
8    do {
9      double chance = Math.random();
10     Weighted<RestorableVMState> selected;
11     if (chance >= epsilon) {
12       selected = distribution.removeMax();
13     } else {
14       selected = distribution.remove();
15     }
16     RestorableVMState source = selected.getElement();
17     vm.restoreState(source);
18     notifyStateRestored();
19     int depth = getStateDepth(vm.getStateId() + 1);
20     if (depth >= MAX_DEPTH) {
21       notifySearchConstraintHit("depth limit reached: " + MAX_DEPTH);
22       continue;
23     }
24     while (!done) {
25       if (!forward()) {
26         break;
27       } else {
28         notifyStateAdvanced();
29         if (currentError != null) {
30           notifyPropertyViolated();
31           if (hasPropertyTermination()) {
32             break;
33           }
34         }
35         if (isNewState() && !isEndState() && !isIgnoredState()) {
36           double probability = selected.getWeight();
```

```
37          ChoiceGenerator<?> cg = vm.getChoiceGenerator();
38          if (cg instanceof Probabilistic) {
39            probability *= ((Probabilistic) cg).getProbability();
40          }
41          RestorableVMState target = vm.getRestorableState();
42          distribution.add(target, probability);
43          setStateDepth(vm.getStateId() + 1, depth + 1);
44          notifyStateStored();
45          if (!checkStateSpaceLimit()) {
46            notifySearchConstraintHit("memory limit reached: "
47                + minFreeMemory);
48            done = true;
49            break;
50          }
51        }
52        backtrack();
53        notifyStateBacktracked();
54      }
55    }
56  } while (!distribution.isEmpty() && !done);
57  notifySearchFinished();
58 }
```

Appendix B

Implementing a Listener in JPF

In this section we provide a recipe for implementing a listener within JPF. As a first example, we implement a listener which prints the states and transitions visited by the search in the following simple format.

```
0 -> 1
1 -> 2
0 -> 3
3 -> 4
4 -> 2
```

The above tells us that the search started in the initial state 0 and made a transition to a new state 1. From state 1 it made a transition to a new state 2. Next, it made a transition from state 0 to state 3. Etcetera. We name our class `StateSpacePrinter`.

As we already mentioned earlier, both the search and JPF's virtual machine notify listeners of particular events. The methods corresponding to those events can be found in the interfaces `SearchListener` and `VMListener`, which are part of the packages `gov.nasa.jpf.search` and `gov.nasa.jpf.vm`, respectively. A listener implements either one of these or both interfaces.

Since usually we are only interested in a few events, we can provide the methods corresponding to the remaining events with a default implementation (since all methods are void, we can just provide a method with an empty body). To avoid coding these methods, JPF has already provided the classes `SearchListenerAdapter` and `ListenerAdapter`, which are part of the packages `gov.nasa.jpf.search` and `gov.nasa.jpf`, respectively. The former provides default implementations for all the methods specified in the `SearchListener` interface, and the latter provides default implementations for all the methods specified in the `SearchListener` and `VMListener` interface.

In our `StateSpacePrinter`, we are only interested in the events signalling a change of state. Therefore, we implement the `SearchListener` interface. Since we are not interested in all event notifications by the search, we start from default implementations by extending the `SearchListenerAdapter` class. Hence, we arrive at the following method header.

```
public class StateSpacePrinter extends SearchListenerAdapter implements
    SearchListener
```

In order to print a transition, we need both the source and target state of the transition. As we will see, it is sufficient to introduce an attribute that keeps track of the source state. In JPF each state has a unique identifier, which is a nonnegative integer.

```
private int source;
```

We initialize this attributes in the constructor to an unknown state, which we represent by -1.

```
public StateSpacePrinter() {
  source = -1;
}
```

The only three methods that signal a state change are `stateAdvanced(Search)`, `stateBacktracked(Search)` and `stateRestored(Search)`. These can be implemented in the following straightforward way.

```
1  public void stateAdvanced(Search search) {
2    int target = search.getStateId();
3    if (source != -1) {
4      System.out.printf("%d -> %d%n", source, target);
5    }
6    source = target;
7  }
8
9  public void stateBacktracked(Search search) {
10   source = search.getStateId();
11 }
12
13 public void stateRestored(Search search) {
14   source = search.getStateId();
15 }
```

Note that these methods receive a reference to a `Search` object as an argument. This `Search` object contains information about the search of the state space by JPF. In our implementation of the methods we use `search.getStateId()` to get the identifier of the current state.

Appendix C

An Example of Correctness Test

```
1  import probabilistic.Choice;
2  public class Test46445 {
3    public static void main(String[] args) {
4      while (true) {
5        double[] p = { 0.75, 0.25 };
6        switch (Choice.make(p)) {
7        case 0:
8          double[] p1 = { 0.5, 0.5 };
9          switch (Choice.make(p1)) {
10         case 0:
11           double[] p11 = { 1.0 };
12           switch (Choice.make(p11)) {
13           case 0:
14             break;
15           }
16           break;
17         case 1:
18           while (true) {
19             double[] p12 = { 0.5, 0.5 };
20             switch (Choice.make(p12)) {
21             case 0:
22               break;
23             case 1:
24               break;
25             }
26           }
27         }
28         break;
29       case 1:
30         double[] p2 = { 0.75, 0.25 };
31         switch (Choice.make(p2)) {
32         case 0:
33           while (true) {
34             double[] p21 = { 1.0 };
35             switch (Choice.make(p21)) {
36             case 0:
37               break;
38             }
```

```
39            }
40          case 1:
41            double[] p22 = { 1.0 };
42            switch (Choice.make(p22)) {
43            case 0:
44              break;
45            }
46            break;
47          }
48          break;
49        }
50      }
51    }
52  }
```

Appendix D

TransitonsAndTime Listener

```
1  package probabilistic.listener;
2
3  /* Copyright (C) 2013 Xin Zhang, Qiyi Tang and Franck van Breugel
4   *
5   * This program is free software: you can redistribute it and/or modify
6   * it under the terms of the GNU General Public License as published by
7   * the Free Software Foundation, either version 3 of the License, or
8   * (at your option) any later version.
9   *
10  * This program is distributed in the hope that it will be useful,
11  * but WITHOUT ANY WARRANTY; without even the implied warranty of
12  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
13  * GNU General Public License for more details.
14  *
15  * You can find a copy of the GNU General Public License at
16  * <http://www.gnu.org/licenses/>.
17  */
18 import gov.nasa.jpf.ListenerAdapter;
19 import gov.nasa.jpf.vm.ChoiceGenerator;
20 import gov.nasa.jpf.search.Search;
21 import gov.nasa.jpf.search.SearchListener;
22 import probabilistic.vm.Probabilistic;
23 /**
24  * This listener prints the states and the transitions visited by the search in
25  * the following formats: 0 0.5 1, denoting that there is a transition from
26  * state 0 to state 1 with probability 0.5, and 0 1.0 1, denoting that there is
27  * a transition from state 0 to state 1. Transitions to an end state are
28  * followed by a '*'. After each 1000 transitions, the current time in
29  * milliseconds preceeded by "T: " is printed.
30  *
31  * @author Xin Zhang
32  * @author Qiyi Tang
33  * @author Franck van Breugel
34  */
35 public class TransitionsAndTime extends ListenerAdapter implements
36     SearchListener {
37   private int source; // id of the source of the next transition
38   private int count;
```

```
39  private static int INTERVAL = 2;
40  /*
41   * Initializes the listener.
42   */
43  public TransitionsAndTime() {
44    source = -1; // -1 represents undefined
45    count = 0;
46  }
47  /*
48   * Executed whenever the search advances to the next state.
49   */
50  public void stateAdvanced(Search search) {
51    count++;
52    int target = search.getStateId();
53    if (source != -1) {
54      ChoiceGenerator<?> cg = search.getVM().getChoiceGenerator();
55      if (cg instanceof Probabilistic) {
56        double probability = ((Probabilistic) cg).getProbability();
57        System.out.print(source + " " + probability + " " + target);
58      } else {
59        System.out.print(source + " 1.0 " + target);
60      }
61      if (search.isEndState()) {
62        System.out.print(" *");
63      }
64      System.out.println();
65    }
66    source = target;
67    if (count == INTERVAL) {
68      System.out.println("T: " + System.currentTimeMillis());
69      count = 0;
70    }
71  }
72  /*
73   * Executed whenever the search backtracks.
74   */
75  public void stateBacktracked(Search search) {
76    source = search.getStateId();
77  }
78  /*
79   * @param search object that provides information about the search.
80   */
81  public void stateRestored(Search search) {
82    source = search.getStateId();
83  }
84  /*
85   * Executed when the search starts.
86   */
87  public void searchStarted(Search search) {
88    System.out.println("T: " + System.currentTimeMillis());
89  }
90  /*
91   * Executed when the search finishes.
92   */
```

```
93    public void searchFinished(Search search) {
94      System.out.println("T: " + System.currentTimeMillis());
95    }
96  }
```

Appendix E

Subproblems of Progress

progress(0, 125), progress(125, 165), progress(165, 190), progress(190, 200), progress(200, 208), progress(208, 216), progress(216, 224), progress(224, 232), progress(232, 238), progress(238, 244), progress(244, 250), progress(250, 256), progress(256, 262), progress(262, 268), progress(268, 274), progress(274, 280), progress(280, 286), progress(286, 292), progress(292, 298), progress(298, 304), progress(304, 310), progress(310, 316), progress(316, 321), progress(321, 326), progress(326, 331), progress(331, 336), progress(336, 341), progress(341, 346), progress(346, 351), progress(351, 356), progress(356, 361), progress(361, 366), progress(366, 371), progress(371, 376), progress(376, 381), progress(381, 386), progress(386, 391), progress(391, 396), progress(396, 401), progress(401, 406), progress(406, 411), progress(411, 416), progress(416, 421), progress(421, 426), progress(426, 431), progress(431, 435), progress(435, 439), progress(439, 443),

Bibliography

[AC06] Tadashi Araragi and Seung Mo Cho. Checking liveness properties of concurrent systems by reinforcement learning. In Stefan Edelkamp and Alessio Lomuscio, editors, *Proceedings of the 4th Workshop on Model Checking and Artificial Intelligence*, volume 4428 of *Lecture Notes in Computer Science*, pages 84–94, Riva del Garda, Italy, August 2006. Springer-Verlag.

[BK08] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, Cambridge, MA, USA, 2008.

[BSA09] Razieh Behjati, Marjan Sirjani, and Majid Nili Ahmadabadi. Bounded rational search for on-the-fly model checking of ltl properties. In Farhad Arbab and Marjan Sirjani, editors, *Revised Selected Papers of the 3rd IPM International Conference on Fundamentals of Software Engineering*, volume 5961 of *Lecture Notes in Computer Science*, pages 292–307, Kish Island, Iran, April 2009. Springer-Verlag.

[EP02] Cindy Eisner and Doron Peled. Comparing symbolic and explicit model checking of a software system. In *In Proc. SPIN Workshop on Model Checking of Software, volume 2318 of LNCS*, pages 230–239. Springer, 2002.

[ES11] Stefan Edelkamp and Stefan Schrödl. *Heuristic Search: Theory and Applications*. Morgan Kaufmann, Waltham, MA, USA, 2011.

[GS05] Radu Grosu and Scott A. Smolka. Monte Carlo model checking. In Nicolas Halbwachs and Lenore D. Zuck, editors, *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, volume 3440 of *Lecture Notes in Computer Science*, pages 271–286, Edinburgh, UK, April 2005. Springer-Verlag.

[GT99] Fausto Giunchiglia and Paolo Traverso. Planning as model checking. In Susanne Biundo and Maria Fox, editors, *Proceedings of the 5th European Conference on Planning*, volume 1809 of *Lecture Notes in Artificial Intelligence*, pages 1–20, Durham, UK, September 1999. Springer-Verlag.

[Pel08] Radek Pelánek. Fighting state space explosion: Review and evaluation. In Darren D. Cofer and Alessandro Fantechi, editors, *Revised Selected Papers of the 13th International Workshop on Formal Methods for Industrial Critical Systems*, volume 5596 of *Lecture Notes in Computer Science*, pages 37–52, L'Aquila, Italy, September 2008. Springer-Verlag.

[Saa03] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Cambridge, MA, USA, 2nd edition, 2003.

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: an introduction*. The MIT Press, Cambridge, MA, USA, 1998.

[VHB+03] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, April 2003.

[Zha10] Xin Zhang. Measure progress of model checking randomized algorithms. Master's thesis, York University, Toronto, July 2010.

[ZvB10] Xin Zhang and Franck van Breugel. Model checking randomized algorithms with Java PathFinder. In *Proceedings of 7th International Conference on Quantitative Evaluation of Systems*, pages 157–158, Williamburgh, VA, USA, September 2010. IEEE.

[ZvB11] Xin Zhang and Franck van Breugel. A progress measure for explicit-state probabilistic model-checkers. In Luca Aceto, Monika Henzinger, and Jirí Sgall, editors, *Proceedings of the 38th International Colloquium on Automata, Languages and Programming*, volume 6756 of *Lecture Notes in Computer Science*, pages 283–294, Zurich, Switzerland, July 2011. Springer-Verlag.