Application of Randomness in a Potency-Based Best-First Heuristic Search Strategy

Emil G. Henriksen, Alan M. Khorsid, Esben Nielsen, Theodor Risager, Adam M. Stück, and Andreas S. Sørensen

Department of Computer Science, Aalborg University, Aalborg, Denmark
{eghe19,akhors19,eniel19,trisag19,astack19,asa19}@student.aau.dk

Abstract. Efficient state space exploration has a significant impact on solving reachability queries. We describe four variants of our heuristic search strategy for Petri nets. All variants assign dynamic weights called potencies to transitions, and they differentiate in the way they initialise and use potencies. The two best variants, RPFS and RPFS-LP, add randomness in the selection process, with one of them adding smarter potency initialisation. We implement all variants in the tool Tapaal, and document their performances using the models and queries from the Model Checking Contest 2022 (MCC'22). The experiment shows that our proposed heuristic search outperforms the standard search strategies in multiple metrics. Furthermore, substituting Tapaal's previous best heuristic function with RPFS leads to an additional 516 solved instances, which is more than a 1% increase.

1 Introduction

Heuristic search strategies are widely applied in areas such as pathfinding and planning [11], where the popular A* algorithm [10] is often used. Heuristic search is mostly used to search through a state space to either satisfy a goal in the case of planning or find the shortest path in pathfinding. Instead of naively trying to go through an entire state space using, e.g., Breadth First Search (BFS) or Depth First Search (DFS), it can, in some cases, be beneficial to use a heuristic search to guide an algorithm in the correct direction.

Heuristic search is also used in model checking [22,18,20]. We specifically focus on Petri nets [17], a high-level modeling formalism used to model distributed systems. Since 2011 an annual Model Checking Contest (MCC) [4] has been held where multiple tools compete to solve a variety of problems, such as reachability analysis, deadlock detection, and more. According to the MCC'22 results [3] and [23], the tools ITS-Tools [20] and LoLA [22] have implemented a random walk, and Tapaal [18] has implemented a random depth-first search to help solve problems from the reachability category [23], which shows that using randomness is not a new idea for the verification of Petri nets. These tools have placed top three in the reachability category in the MCC'21 [1] and MCC'22 [3].

We propose a modification to the existing best-first heuristic search strategy implemented in Tapaal, as shown in [12] that can more efficiently assist in answering reachability cardinality queries for Petri nets. Our heuristic search uses

randomness and an extension to Petri nets called potencies which essentially are dynamic weights on transitions. It then modifies these potencies during the state space exploration to guide the search in the desired direction. The randomness is added due to comparison results showing that it often improves the number of solved problems. Randomness helps by making the queue of transitions stochastic. This way, if the heuristic search is guided on a wrong path, it will not necessarily explore the entire path, but it may, at some point, take a more promising path. Furthermore, we set the initial value of potencies using the solution to a linear program created from the given Petri net and a reachability query.

We benchmark the standard search strategies present in Tapaal using the MCC'22 models, and compare all strategies on the number of reachability cardinality instances they solve in a given time. We then compare this benchmark with our proposed heuristic search strategy. The results show that the proposed heuristic search strategy finds 706 additional answers on reachability cardinality problems compared to the benchmark, which is an increase of 8.54% compared to Tapaal's Best First Reachability Search.

1.1 Related Work

In [12], the Integer Linear Programming (ILP) technique (state equations [16]) are used for disproving reachability by over-approximating the state space. This can be an efficient method for avoiding full state space exploration if the integer linear program does not have a solution, as this means a reachability cardinality query cannot be satisfied [12]. In LoLA [22,14], the Counter-Example Guided Abstraction Refinement (CEGAR) [8] approach, in combination with state equations, is used to help disprove reachability problems. The CEGAR approach involves starting with an initial abstraction that is an overapproximation of the original problem and then iteratively refining the abstraction until a solution is found. In [14], the state equation for a Petri net is used as an initial abstraction for a reachability problem. Based on the feasibility of solving the state equation, the reachability problem is either proved, disproved, or the abstraction is refined due to finding a counter-example.

The distance function used in the proposed heuristic search and is originally from [12]. It is implemented in Tapaal, where it is used in a deterministic best-first search algorithm. We use this function in our heuristic search strategy to dynamically adjust the potencies using distance instead of a static value.

The principles of potencies and randomness used in the proposed heuristic search strategy are discussed in [5], which uses potencies together with Monte-Carlo simulations to aid verification in reachability problems from the MCC. This shows some promising results, although the method used cannot disprove any queries using Monte-Carlo simulations, making it unusable in the MCC. Contradictory to the results from [5], given enough time, our proposed heuristic search will find an answer if it exists.

2 Preliminaries

A Petri net is a directed, weighted, bipartite graph with an initial state called the initial marking. A Petri net consists of places and transitions connected by arcs. Arcs are directed and can go from a transition to a place or from a place to a transition [16]. Places can have a non-negative number of tokens in them. These tokens can then be consumed and produced using transitions. A formal definition of Petri nets follows.

Definition 1 (Petri net). A Petri net is a 4-tuple $N = (P, T, W, I)$ where

- P is a finite set of places,
- T is a finite set of transitions, where $\mathrm{T} \cup P \neq \emptyset$ and $T \cap P = \emptyset$,
- $W : (P \times T) \cup (T \times P) \to \mathbb{N}_0$ is the weight function, and
- $I : (P \times T) \to \mathbb{N}_\infty$ is the weight function for inhibitor arcs.

Here $\mathbb{N}_\infty$ is the set of natural numbers extended with infinity, and $\mathbb{N}_0$ is the set of natural numbers extended with 0.

When a Petri net has any tokens assigned to any place(s), it is called a Marked Petri net.

Definition 2 (Marked Petri net). Let $M : P \to \mathbb{N}_0$ be a marking that assigns tokens to places, and $\mathcal{M}(N)$ the infinite set of all possible markings on N. Then $PN = (N, M_0)$ is a Marked Petri net, where $M_0 \in \mathcal{M}(N)$ is the initial marking.

We use the following notation to concisely denote the set of places/transitions that precede or succeed other places/transitions. The preset of a place/transition is defined as ${}^\bullet y \stackrel{\text{def}}{=} \{z \in P \cup T \mid W(z, y) > 0\}$ and the postset of a place or transition is defined as $y^\bullet \stackrel{\text{def}}{=} \{z \in P \cup T \mid W(y, z) > 0\}$. Let $I(t)$ be the set of places that inhibits a transition t defined as $I(t) \stackrel{\text{def}}{=} \{p \in P \mid (p, t) \in I\}$, and let $I(p)$ be the set of transitions that a place p inhibits defined as $I(p) \stackrel{\text{def}}{=} \{t \in T \mid (p, t) \in I\}$.

A transition is said to be enabled when each place connected with an incoming arc contains at least the same number of tokens as the weight of the arc. However, if the arc is an inhibitor, the place must contain fewer tokens. The formal definition of when a transition is enabled follows.

Definition 3 (Enabled). Let $\mathit{Enabled}(N, M)$ be the set of enabled transitions in M, defined as $\mathit{Enabled}(N, M) \stackrel{\text{def}}{=} \{t \in T \mid M(p) \geq W(p, t)$ for all $p \in {}^\bullet t$ and $M(p) < I(p, t)$ for all $p \in I(t)\}$.

An enabled transition can be fired. When firing an enabled transition, a new marking is produced.

Definition 4 (Fire). A transition $t \in \mathit{Enabled}(N, M)$ can fire in marking M producing a new marking M', written as $M \xrightarrow{t} M'$ where $M'(p) \stackrel{\text{def}}{=} M(p) - W(p, t) + W(t, p)$ for all $p \in P$.

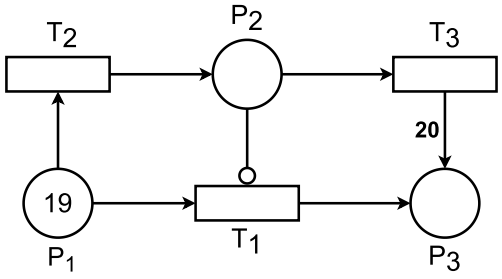


Fig. 1: Petri Net Example

Graphically, places are drawn as circles, transitions are drawn as rectangles, arcs are drawn as arrows, inhibitor arcs are drawn as arrows with a small circle instead of an arrowhead, and tokens are denoted as dots or alternatively as a number in places. If the weight of an arc is 1, it is left out of the graphical representation. An example of a Petri net can be seen in Figure 1.

2.1 Cardinality Propositions

We define the reachability problem specifically for cardinality queries. A cardinality proposition φ of a Petri net $N = (P, T, W, I)$ is defined using the following abstract syntax, which is inspired by [12].

$$\varphi ::= e \bowtie e \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi$$
$$e ::= n \mid p \mid e + e \mid e - e \mid n \cdot e$$

where $\bowtie \in \{\leq, <, =, \neq, >, \geq\}, n \in \mathbb{N}_0$ and $p \in P$.

An expression e can be evaluated in a marking M such that $eval(M, e)$ is a number. The satisfaction relation $M \models \varphi$ is defined such that $M \models e_1 \bowtie e_2$ iff $eval(M, e_1) \bowtie eval(M, e_2)$. We assume that $eval(M, p) = M(p)$ where $p \in P$, meaning that the evaluation of the place p in a marking M is the number of tokens currently in that place.

A number n evaluates to itself, and arithmetic operations are evaluated in their natural way as both sides of the operand are numbers in $\mathbb{N}_0$. The semantics of when a proposition φ holds in a given marking M is defined as follows:

$M \models e_1 \bowtie e_2$ iff $eval(M, e_1) \bowtie eval(M, e_2)$
$M \models \varphi_1 \wedge \varphi_2$ iff $M \models \varphi_1$ and $M \models \varphi_2$
$M \models \varphi_1 \vee \varphi_2$ iff $M \models \varphi_1$ or $M \models \varphi_2$
$M \models \neg\varphi$ iff $M \not\models \varphi$

If there exists a marking M which can be reached through a sequence of transitions from the initial marking M_0 of a marked Petri net (N, M_0) and $M \models \varphi$, we write $(N, M_0) \models EF\ \varphi$.

Algorithm 1: TAPAAL's Standard Reachability Search Algorithm

Input: Petri net N, initial marking M_0, proposition φ
Output: A Boolean representing if N satisfies φ

```
1  Function Generic-Reachability-Search(N, M_0, φ):
2      if M_0 ⊨ φ then
3          return true
4      Initialise(Waiting, M_0)
       // Initialise Waiting with M_0
5      Passed := {M_0}
6      while Waiting is non-empty do
7          M := SelectAndRemove(Waiting)
           // Select and remove an element M from Waiting
8          for M' such that M -t-> M' where t ∈ Enabled(N, M) do
9              if M' is not in Passed then
10                 if M' satisfies φ then
11                     return true
12                 Passed := Passed ∪ {M'}
13                 Update(Waiting, t, M, M', φ)
                   // Update Waiting with M'
14     return false
```

2.2 TAPAAL

TAPAAL [18] is a modelling, simulation, and verification tool developed at the Department of Computer Science at Aalborg University. In particular, we use TAPAAL's VERIFYPN [21] engine to answer reachability cardinality queries. TAPAAL not only supports model checking via explicit search but also performs other reduction and optimisation techniques. These techniques include query simplification [6], structural net reduction, and the use of state equations [12]. These techniques aim to reduce the state space of the exploration before running the explicit search strategies. These techniques can often prove or disprove reachability problems without running an explicit search [12].

We aim to improve TAPAAL's explicit search strategy. TAPAAL supports four standard search strategies - Breadth First Search (BFS), Depth First Search (DFS), Random Depth First Search (RDFS), and a heuristic search called Best First Reachability Search (BestFS).

2.3 Standard reachability search in TAPAAL

We define TAPAAL's standard reachability search algorithm in a high-level fashion in Algorithm 1. Each search strategy can be implemented as shown in Algorithm 1, where the difference is whether the *Waiting* set is a stack, queue, or priority queue. As seen in the `for-loop` on line 8 of Algorithm 1, if one of the successor markings of M satisfies the proposition φ, the function returns `true`.

Search Algorithm	**Initialised type of** *Waiting*
BFS	FIFO Queue
DFS	Stack
RDFS	Stack in which the elements are shuffled before being pushed[1]
BestFS	Priority queue sorted by $Min(\texttt{Distance}(M, \varphi))$

Table 1: An overview of how *Waiting* is initialised for different search algorithm in Algorithm 1 on line 4

Otherwise, M' is added to *Waiting*. The standard search strategies mentioned above only differentiate on the order in which the set *Waiting* is explored and, therefore, what data structure *Waiting* is. Table 1 shows the different types that *Waiting* can be and thus define how the element M shall be picked on line 7. On line 4, *Waiting* is initialised to a set only containing the initial marking M_0 for all the standard search strategies. On line 13, only M' is added to *Waiting* for all standard search strategies.

The priority queue for BestFS is sorted by the markings with the shortest distance to the query φ, as shown in Table 1. In Algorithm 2, the distance is computed by evaluating the cardinality queries for each marking put into the waiting queue. The `Distance` function taken from [12] returns an integer that describes how far away a given marking is from satisfying a given query. The Δ-function at the bottom of Algorithm 2 evaluates the given numbers differently depending on what operator is in the query. For example, if the query consists only of one proposition with an '='-operator, the absolute difference between the two given values determines the distance to the goal marking. The general ideas and structures of Algorithm 1 and Algorithm 2 are from [12].

3 Comparison of Standard Search Strategies

To compare different search strategies, we first present the methodology in Section 3.1. We use four types of evaluation techniques, namely, cactus-, ratio-, and throughput plots, as well as tables. Section 3.2 looks into how different random seeds affect the result of running RDFS. Lastly, Section 3.3 presents the final results of the four search strategies and evaluates how each strategy performs.

3.1 Methodology

This section outlines the data we use for comparing different strategies and how the strategies are compared. It also describes the hardware which we use for execution. The dataset used to compile all results in this and later sections can

[1] The element shuffle is implemented using a cache that stores the elements until they are ready to be shuffled, then pops the next element before the rest is pushed unto the stack.

Algorithm 2: TAPAAL's Current Distance Heuristics

Input: Marking M, proposition φ
Output: Integer representing distance from satisfying φ

```
1  Function Distance(M, φ):
2      if φ = e1 ⋈ e2 then
3          return Δ(eval(M, e1), ⋈, eval(M, e2))
4      else if φ = φ1 ∧ φ2 then
5          return Distance(M, φ1) + Distance(M, φ2)
6      else if φ = φ1 ∨ φ2 then
7          return min{Distance(M, φ1), Distance(M, φ2)}
8      else if φ = ¬(e1 ⋈ e2) then
9          return Δ(eval(M, e1), ⋈̄, eval(M, e2))
10     else if φ = ¬(φ1 ∧ φ2) then
11         return min{Distance(M, ¬φ1), Distance(M, ¬φ2)}
12     else if φ = ¬(φ1 ∨ φ2) then
13         return Distance(M, ¬φ1) + Distance(M, ¬φ2)
14     else if φ = ¬(¬φ1) then
15         return Distance(M, φ1)
```

where $\overline{\bowtie}$ is the dual arithmetical operation of $\bowtie$ (for example $\overline{>}$ is the notation for $\leq$) and where

$$\Delta(v_1, =, v_2) = \mid v_1 - v_2 \mid$$

$$\Delta(v_1, \neq, v_2) = \begin{cases} 1 & \text{if } v_1 = v_2 \\ 0 & \text{otherwise} \end{cases}$$

$$\Delta(v_1, <, v_2) = \max\{v_1 - v_2 + 1, 0\} \qquad \Delta(v_1, >, v_2) = \Delta(v_2, <, v_1)$$

$$\Delta(v_1, \leq, v_2) = \max\{v_1 - v_2, 0\} \qquad \Delta(v_1, \geq, v_2) = \Delta(v_2, \leq, v_1)$$

be found on GitHub[2], along with the program used to compile all plots and tables.

Models The models we use in this project are from the Model Checking Contest 2022 (MCC'22) [2] model set, which consists of 1321 models, each containing 16 reachability cardinality queries. Here we introduce the notion of an instance, which refers to a specific query for a specific model. This results in a total of 21136 instances. Before we apply any of the explicit search strategies, 10952 instances are solved by reductions and optimisation techniques, as mentioned in Section 2.2. These are referenced as simple instances. The explicit search strategies attempt to solve the remaining 10184 instances. These are referenced as difficult instances and are the instances used for the remainder of the paper.

Executions We define an execution as a specific strategy being executed on an instance. When comparing different strategies, multiple experiments are per-

[2] `https://github.com/theodor349/P7-Potency-Based-Search-Heuristic`

formed. An experiment is a collection of executions where each strategy is executed on all instances. Each strategy is executed in series on a given instance to make sure that each strategy is executed on the same core for the given instance. This is referred to as a job. This means that an experiment consists of 10184 jobs, each containing multiple executions. Unless otherwise specified, the experiments are run with default settings which enable all optimisations, where each execution has a timeout of five minutes and a memory limit of 16 GB.

For each execution, the following data points are collected; total time, total memory, and exit code. We also collect discovered-, explored-, expanded states, max tokens, and search time for those instances that use explicit search. The difference between total time and search time is that total time includes all phases of VerifyPN, and search time only includes the time spent in the while loop outlined in Algorithm 1 on line 7.

Hardware All experiments are run on the DEIS CPU cluster [9]. The cluster consists of AMD EPYC 7551 32-Core Processors. Each CPU is set up to have a single thread with a clock speed between 2.42 GHz and 2.56 GHz, and Linux has estimated the BogoMIPS to be between 3992.23 and 3992.66. These results are based on an experiment with 84.672 executions, where the *lscpu* command is used to get information about the current core, and this equates to a possible clock speed variation of ∼5 % from core to core. This means there may be a runtime variation upwards of ∼5 % on an instance between different experiments.

Software The version of VerifyPN used for all experiments in this section is compiled from the August 26$^{\text{th}}$, 2022 version. The git commit 4*e*20572 can be found here [19].

Evaluation A cactus plot is used to summarize the performance of different search strategies. For example, Figure 2 shows how fast different strategies solve an instance and how many are solved. The x-axis is sorted from the fastest-solved instance to the slowest-solved for each strategy. This means that the same x value for different strategies may not correspond to the same instance. The y-axis displays the time it takes for a strategy to solve the instance on a logarithmic scale. In conclusion, we use the cactus plot to compare the total time used to solve the instance for multiple strategies simultaneously, as it is possible to see how much faster a strategy is compared to another and how many more instances it solves.

The ratio plot compares one strategy (A) to another (B). An example can be seen in Figure 5. It is split into two separate plots, where the left one displays all instances where A is slower than B, and the right plot displays instances where A is faster than B. The x-axis is sorted from smallest to largest. The left plot's y-axis displays how many times faster B is compared to A, and the right plot shows how many times A is faster than B. Both y-axes are calculated by dividing the total time of the slowest strategy by the fastest and are displayed

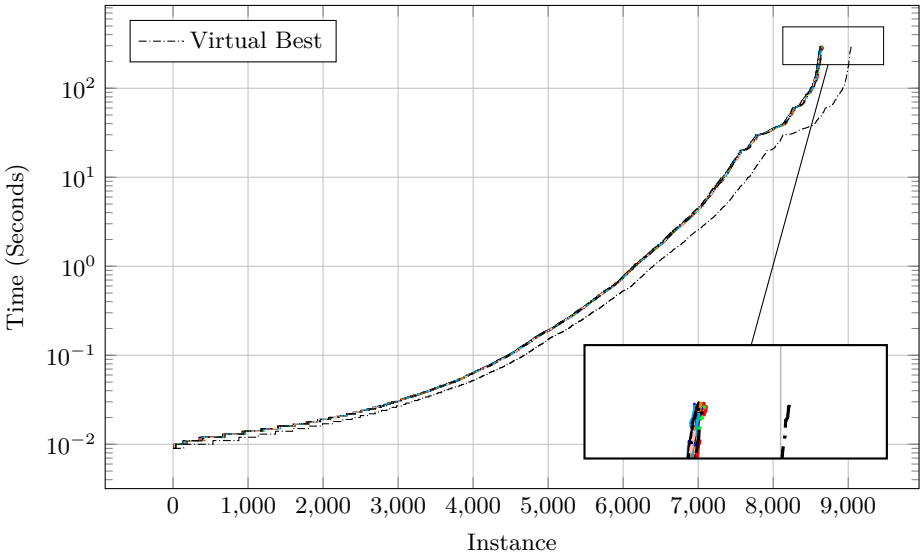


Fig. 2: Cactus plot of RDFS using 21 different seed offsets

on a logarithmic scale. For convenience, the ratio plot compares one strategy to multiple other strategies, which results in multiple legends.

In addition to the plots, we also present tables displaying the results of an experiment. It contains six columns; strategy, total, solved, timeouts, errors, fastest, and unique. An example can be seen in Table 2. The fastest column is a measure of how many instances the specific strategy is the fastest. The unique column is a measure of how many instances a strategy is the only one to solve.

Virtual Best We introduce a virtual best strategy for the cactus plot and results table. It contains all the executions which are the fastest for a given instance. If strategy A solves the instance in 2 seconds and strategy B solves it in 4 seconds, then A is chosen as the virtual best for that instance.

3.2 RDFS Comparison

To determine how much RDFS varies given different seeds, we run RDFS 21 times, all with different seeds. The experiment results can be seen in the cactus plot in Figure 2. As one can see, all of the 21 RDFS strategies vary a relatively small amount in the total number of instances solved. In fact, out of the total 21136 instances, the largest difference in the number of instances solved between any two RDFS strategies is 50. This difference is not large enough to justify running 21 differently seeded RDFS strategies each time we run an experiment.

There can be a significant variation in the time used to solve an instance if we compare each strategy on the same instance instead of sorting them all by time per instance.

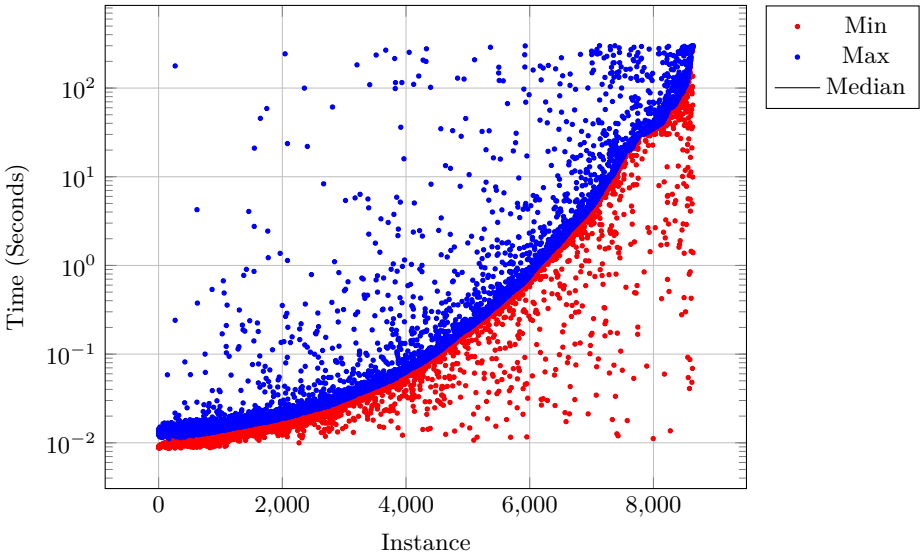


Fig. 3: Plot showing the *min* and *max* time to solve a query per seed.

In Figure 3, we plot the median RDFS strategy from the previous comparison, and each instance is sorted so that *min* and *max* for each x-axis point match the same instance, which is the median's instance. Here we see some variation in the time it takes per instance depending on the seed given to RDFS. In the beginning, there are a few outliers in the *max* category, indicating that the easier instances sometimes have a seed that makes it much slower. However, more often than not, the *min* and *max* do not vary much from the median.

Towards the more challenging instances, the *min* category has increasingly more RDFS seeds that solve the instance much faster. Although there are some outliers in the *min* and *max* categories, the average deviation from the median is only 3 seconds. This means that there is little difference between the 21 seeds regarding the time it takes to solve an instance.

In Figure 2, one can see from the virtual best that there is a deviation across all seeds in the number of instances solved. This is enough to justify using the median from these 21 seeds instead of using a single random seed, which potentially could be good or bad compared to the median. Therefore, we use the median strategy from the first comparison to represent RDFS going forward. The median strategy is the median of the strategies from Figure 2 after they have been sorted by the number of unique, fastest, and solved queries in that order. The actual median seed is 5760351.

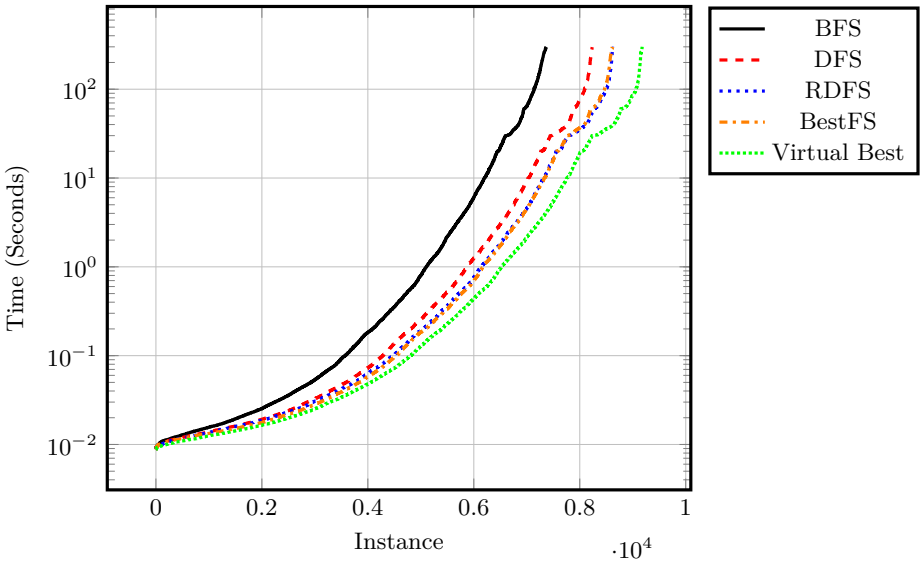


Fig. 4: A Cactus plot showing the comparison between standard search strategies. The timeout is set to 5 min.

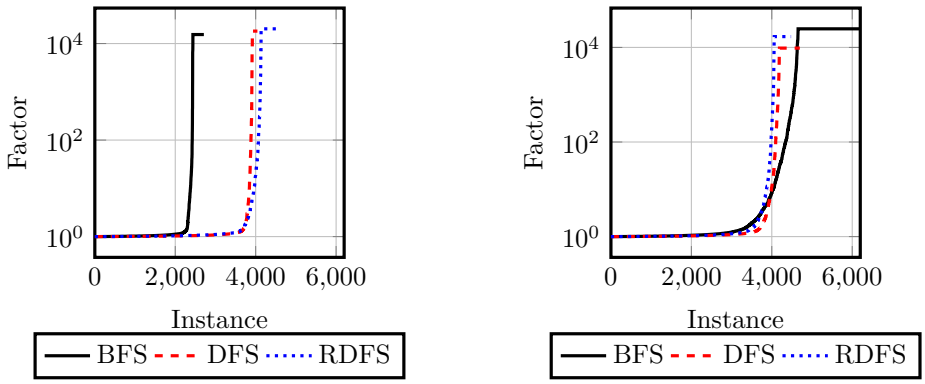
Strategy	**Total**	**Solved**	**Timeouts**	**Errors**	**Fastest**	**Unique**
BFS	10184	7367	2812	5	1215	100
DFS	10184	8235	1942	7	2066	14
RDFS	10184	8641	1520	23	3036	218
BestFS	10184	8617	1562	5	2861	177
Virtual Best	10184	9179	1000	-	-	-

Table 2: Standard Statistics

3.3 Standard Search Strategies Comparison

We conduct an experiment to determine the performance of TAPAAL's standard search heuristics. These are BFS, DFS, RDFS, and BestFS. Furthermore, as mentioned in Section 3.1, we add a virtual best.

As seen in Table 2 and Figure 4, the strategy that solves the most instances besides the theoretical virtual best is RDFS which is between BestFS and Virtual Best. Furthermore, RDFS also solves the most unique instances. From this result, we find RDFS interesting as it indicates that adding randomness can improve performance. This is because the non-random DFS is slower and solves fewer instances than RDFS.



(a) BestFS is slower ($\frac{BestFS}{strategy}$) (b) BestFS is faster ($\frac{strategy}{BestFS}$)

Fig. 5: A Ratio plot comparing BestFS to the other search strategies on an individual basis

3.4 BestFS Ratio Comparison

To compare the BestFS search strategy with the rest of the search strategies, we present the ratio plot in Figure 5. And as shown in Figure 5a, both DFS and RDFS perform well compared to BestFS. Both search strategies are faster than BestFS in slightly more than 4000 instances. Comparing this to Figure 5b, we see that BestFS is also faster than both DFS and RDFS in slightly more than 4500 instances. This means that BestFS is slightly faster at solving instances than DFS and RDFS, even though all three methods solve a similar number of instances in the same amount of time.

Looking at BFS in Figure 5a, we can see that BFS is faster than BestFS in approximately 3000 instances. If we compare this to Figure 5b, we can see that BestFS is faster in approximately 6000 instances. This means that BestFS is significantly faster at solving instances when compared to BFS.

3.5 Adding Randomness to BestFS

Section 3.3 shows that adding randomness to DFS significantly increases performance. Because of this, it is reasonable to question whether randomness can improve other strategies. In particular, BestFS is interesting as it already performs quite well without randomness. To test this, we run a variation of BestFS that does not only consider the very best element in the *Waiting* queue but rather the first n elements. The variation implements the waiting list as a priority queue ordered by distance. We randomly pick a marking from the first n elements, so markings with shorter distances are more likely to be selected. We run this with n-values of 1, 10, 100, 1000, 10000, and without a limit. Table 3 shows the results. The results show that adding this randomness has a limited

n	1	10	100	1000	10000	no limit
Solved	8622	8715	8761	8761	8175	6818

Table 3: Result of adding randomness to BestFS. n is the limit of the random choice, and **Solved** is the number of instances solved.

effect on performance and that running without a limit is especially inefficient. At best, when $n = 1000$, 1.612% more instances are solved, which is quite a small difference. This strategy is not included further in this report, as we later introduce another strategy that significantly improves performance. However, it still improves BestFS, which motivates the question if another strategy with randomness can produce even better results.

4 Potency-First-Search

This section introduces our potency-based search strategy, of which there are four variations. Potency First Search (PFS) is our basic potency strategy based on [5]. Distance Potency First Search (DPFS) is a slight variation, where potencies are updated using the delta between markings. Random Potency First Search (RPFS) adds randomness to our strategy, as this was found to help solve more queries. Random Potency First Search-Linear Programming (RPFS-LP) is an extension to RPFS, in which we use linear programming to initialize potencies.

We now describe our algorithm PFS. The idea of the algorithm is based on a paper [5] that introduces dynamic weights to transitions. These weights, called *potencies*, are used to perform Monte Carlo simulations. This often finds positive answers efficiently. Crucially, however, this strategy does not perform an exhaustive search and can never disprove a query.

Similar to the existing strategies, PFS uses TAPAAL's standard reachability search algorithm, Algorithm 1 on page 5, as a basis for the search. The search mainly differs in the addition of potencies. The potency of a transition is a number representing the likelihood of that transition bringing the Petri Net closer to a marking solving the query if it were to be fired. To explain the behaviour of PFS, we show how the *Waiting* set is initialised, updated, and how markings are selected from it. Throughout these steps, we assume a global function $Potencies(t)$, which maps a transition to its potency.

Initialise We initalise *Waiting* as a set of marking-transition pairs. The element (M, t) represents a marking M and the transition t that we fire to reach it. Initially, *Waiting* contains the initial marking M_0. Afterward, we set the potency of every transition to 100. The pseudocode for this is in Figure 6a. The $-$ symbol in line 2 represents a non-existent transition. Since this initial marking is the only element in *Waiting* in the first iteration of the for-loop, it will always be selected regardless of the potency.

```
Initialise(Waiting, M_0) =
Waiting := {(M_0, −)}
foreach transition t in T do
    Potencies(t) := 100
```

(a)

```
SelectAndRemove(Waiting) =
t := arg max_{t∈T s.t. (M',t)∈Waiting} Potencies(t)
(M, t) := arg min_{(M,t)∈Waiting} Distance(M, φ)
Waiting = Waiting \ (M, t)
return M
```

(b)

```
Update(Waiting, t, M, M', φ) =
Waiting := Waiting ∪ {(M', t)}
if Distance(M', φ) < Distance(M, φ) then
    Potencies(t) := Potencies(t) + 1
if Distance(M', φ) > Distance(M, φ) then
    Potencies(t) := Potencies(t) − 1
```

(c)

Fig. 6: The pseudocode for `Initialise` (a), `SelectAndRemove` (b) and `Update` (c) regarding the implementation of PFS

Select And Remove When selecting a new marking to explore, we want to select a marking that was reached by firing the transition with the highest potency. Hence, for every $(M, t) \in \mathit{Waiting}$, we select the t which maximises $\mathit{Potencies}(t)$. In *Waiting*, there can be multiple elements containing t. For all these elements, we select the marking M that minimises the heuristic function `Distance(`M, φ`)`. Now, M is the most promising state to expand next. The pseudocode for this is in Figure 6b.

Implementation-wise, *Waiting* is implemented as a collection of priority queues, one for each transition. As such, for an element (M, t), the marking M is only stored in the priority queue for t. Because of this, we can select markings effeciently, as we do not need to search through all markings. We can simply select the top element of the chosen potency's priority queue.

Update After exploring the new marking M', it is added to *Waiting* together with the fired transition t. Now, if the `Distance()` of the parent marking M is larger than M''s distance, the potency of t should be increased by 1. This increases the likelihood that other markings reached by firing t will be explored later on. Similarly, if the distance increases, the potency of t is decreased by 1. The pseudocode for this can be seen in Figure 6c.

PFS only adds or subtracts 1 from potencies. Therefore, the potency change does not reflect how much better the reached marking is. To address this, we introduce a variation of PFS that adds or subtracts the difference in distance between the parent marking and the reached marking. We refer to this strategy asDistance Potency First Search (DPFS).

```
SelectAndRemove(Waiting) =
best := −1
n := 0
foreach transiton t appearing in Waiting do
    n := n + Potencies(t)
    r := a random number in [0, 1]
    if r ≤ Potencies(t)/n then
        t_best := t
(M, t_best) := arg min_{(M,t_best)∈Waiting} Distance(M, φ)
Waiting = Waiting \ (M, t_best)
return M
```

Fig. 7: The pseudocode for `SelectAndRemove` regarding the implementation of RPFS

4.1 Randomness

Another interesting approach is to incorporate a random choice. We refer to this variation as Random Potency First Search (RPFS). The motivation for this approach comes from the observations in Section 3.2, where we observe that RDFS performs better than DFS, and another observation from Section 3.5 where adding randomness also slightly improved BestFS. Therefore, we introduce a random choice when selecting a new marking to expand, such that the highest potency transition is not necessarily taken. This is done using a weighted random choice where transitions with higher potencies are more likely to be chosen. This technique is similar to Algorithm 2 from [5], which also contains a proof of correctness. Figure 7 shows the implementation.

4.2 Applying Integer Linear Programming

Until now, we have not looked at how potencies are reasonably initialised. They are currently initialised with a generic value of 100. The disadvantage of setting all potencies to a common value is that the heuristic is blindly searching in the beginning until a marking reducing the distance is found. We want to approach this by giving a more reasonable initial potency for every transition. By doing so, we can guide the heuristic algorithm into taking transitions that are more likely to help find a solution. We introduce Integer Linear Programming (ILP) [6] as a technique to compute a more reasonable potency.

We recall the fundamentals of ILP, as described in [6] and [7].

Definition 5 (Integer Linear Programming). Let $X = \{x_1, x_2, ..., x_n\}$ be a set of variables and let $\overline{x} = (x_1, x_2, \ldots, x_n)^T$ be a column vector over the variables X. A *linear equation* is given by $\overline{c} \cdot \overline{x} \bowtie k$, where $\overline{c} = (c_1, c_2, \ldots, c_n)$ is a row vector of integers, we have $\bowtie \in \{=, <, \leq, >, \geq\}$ and $k \in \mathbb{Z}$. We can now define an integer linear program (LP) as a finite set of linear equations. A solution is a mapping $u : X \to \mathbb{N}^0$ where the column vector $\overline{u} = (u(x_1), u(x_2), \ldots, u(x_n))^T$

satisfies all linear equations. An optimal solution is a solution that minimises a given linear objective function.

In linear programming, we are concerned with finding an *optimal solution* where we want to either maximise or minimise some linear objective function. As for Petri nets, there is a technique for checking if a marking is unreachable. The technique is called *state equations* and disproves reachability by over-approximating the state space, hence avoiding the full state space exploration [12]. State equations are an algebraic description, and we use them to observe how markings change by firing transitions.

We recall the definition of a Petri net as N, an initial marking M_0 on N, a marking M on N we use the set of variables $X = \{x_t \mid t \in T\}$. Given these inputs, we can construct the linear program over the variables X by writing a linear equation for each place $p \in P$ in the Petri net. The equation should express the relationship between the initial marking M_0, the final marking M, and the number of times each transition is fired. This relationship is given by the following state equation:

$$M_0(p) + \sum_{t \in T} (W(t,p) - W(p,t)) \cdot x_t = M(p) \quad \text{for all } p \in P.$$

Once we have written a state equation for each place p in the Petri net, we solve the linear program using a linear programming solver. We use the *branch-and-cut* [15] method to solve the integer linear program.

If there is no solution to the linear program, then M is not reachable from M_0. As state equations do not consider if a transition is enabled, a solution is inconclusive on whether M can be reached from M_0. The purpose of using state equations in this paper is to help set more reasonable values for initial potencies.

Recall that a reachability cardinality problem is given by a Marked Petri net (N, M_0) and a proposition φ. This motivates the question, is there an M s.t. M is reachable from M_0 and $M \vDash \varphi$. In [6] there is an algorithm that given (N, M_0) and φ produces a set of linear programs (LPS) over the variables x_t where $t \in T$. In [6] it is proved that: If M is reachable from M_0 and $M \vDash \varphi$ then there is an LP s.t. $LP \in LPS$ with a feasible solution. If the condition holds, then we take the first LP that is feasible and minimise it with respect to the objective function $\sum_{t \in T} x_t$. It returns an optimal solution u which we use as a basis for setting the initial potencies of each transition to $\mathit{Potencies}(t) = 100 \cdot (u(x_t) + 1)$. Note that we have tried changing the constant 100 to values between 1 and 10000 as well as squaring the right factor. Changing the value does not provide any meaningful difference.

To retrieve the solution x_t from an LP, we use the linear programming kit GLPK (GNU Linear Programming Kit) [15]. This software allows us to specify the constraints and objectives of an LP and then solve for the optimal solution.

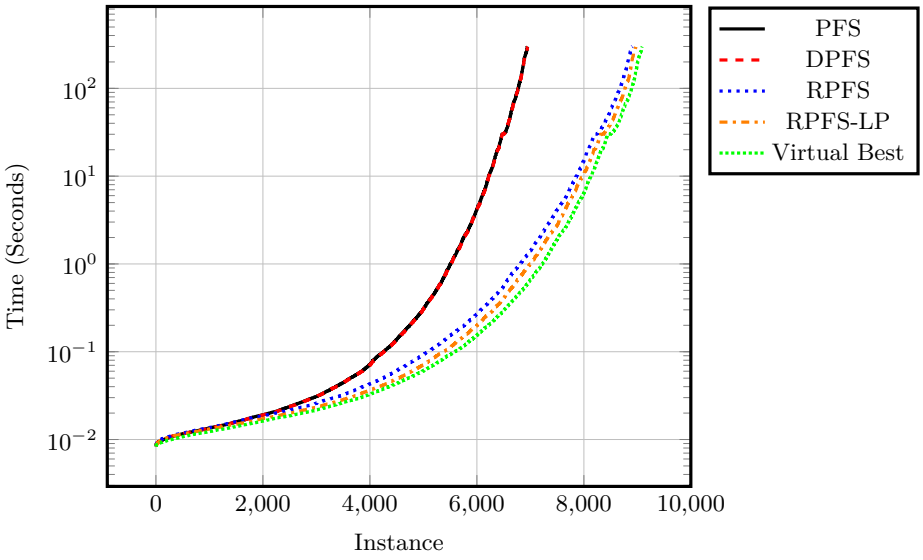


Fig. 8: Comparison of PFS, DPFS, RPFS, and RPFS-LP

Strategy	Total	Solved	Timeouts	Errors	Fastest	Unique
PFS	10029	6944	3085	0	1462	2
DPFS	10029	6951	3078	0	1722	3
RPFS	10029	8902	1126	1	2082	49
RPFS-LP	10029	8972	1057	0	3825	116
Virtual Best	10029	9091	938	-	-	-

Table 4: Statistics for the potency based strategies

5 Potency Strategies in Practice

In this section, we look at the performance of our four potency search strategies (PFS, DPFS, RPFS, and RPFS-LP) from Section 4. We then compare the best performing one to BestFS by looking at the same metrics used to compare the standard search strategies from Section 3.1. Lastly, we test the performance of our improved algorithm for the MCC'22 competition in place of BestFS.

5.1 Experiment Setup

In this section, we use the same setup described in Section 3.1 with an updated version of VerifyPN. This version is based on the same commit but with our strategies added. The source code can be found at `https://github.com/theodor349/P7-verifypn`. We were unable to implement RPFS-LP in a way s.t. it could reduce the model and query. Therefore, we have slightly changed

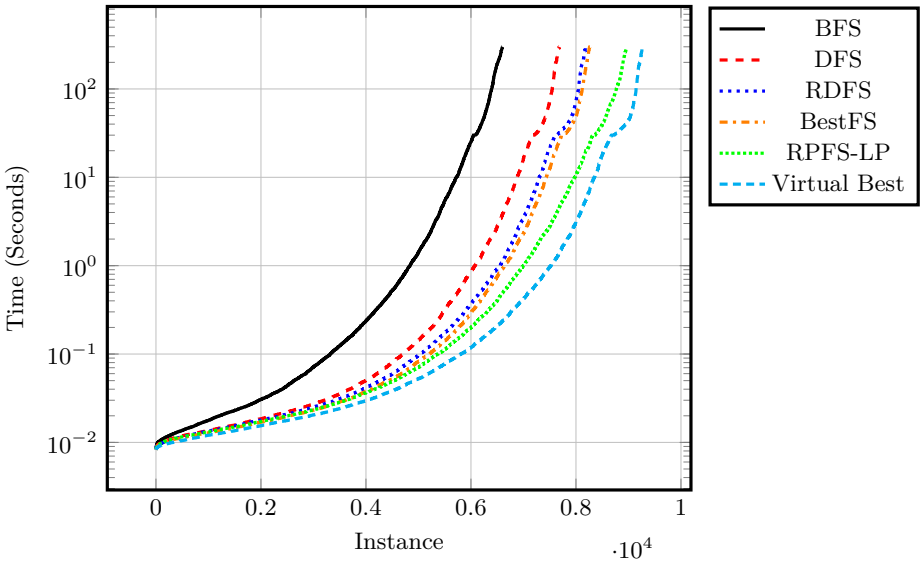


Fig. 9: A Cactus plot showing the comparison between standard search strategies and RPFS-LP. The timeout is set to 5 min.

the experiment format s.t all experiments are executed on already reduced models and queries[3]. As all strategies are executed on reduced models, we can still compare them. All results for non-reduced models can be seen in Appendix A.

5.2 Comparison of Potency Strategies

We conduct an experiment to determine how our search strategies perform compared to each other. The results are shown in Figure 8 and Table 4. It can be seen that RPFS and RPFS-LP perform roughly the same, but RPFS-LP performs slightly better. PFS and DPFS perform very similarly in terms of instances solved. From this comparison, we conclude that adding randomness to PFS will increase the performance. As RPFS-LP outperforms the other three strategies, we will choose this as our main heuristic. Moving forward in this section, we will further measure the performance of RPFS-LP.

5.3 Comparing RPFS-LP and the Standard Search Strategies

We compare RPFS-LP to the standard strategies. Figure 9 and Table 5 show the result. It shows that RPFS-LP outperforms all other strategies with regard to

[3] We reduce the models and queries using VerifyPN. However, each model is reduced using all queries. This means that the reduced model may not be the same as if it is reduced only using a single query, as was the case for the earlier experiments. Therefore the results from the reduced models might not be comparable to the non-reduced results.

Strategy	Total	Solved	Timeouts	Errors	Fastest	Unique
BFS	10029	6604	3425	0	623	15
DFS	10029	7684	2344	1	1666	14
RDFS	10029	8185	1801	43	2576	58
BestFS	10029	8266	1763	0	2479	59
Virtual Best Standard	10029	9021	1008	-	-	-
RPFS-LP	10029	8972	1057	0	1920	244
Virtual Best All	10029	9265	764	-	-	-

Table 5: RPFS-LP compared to standard

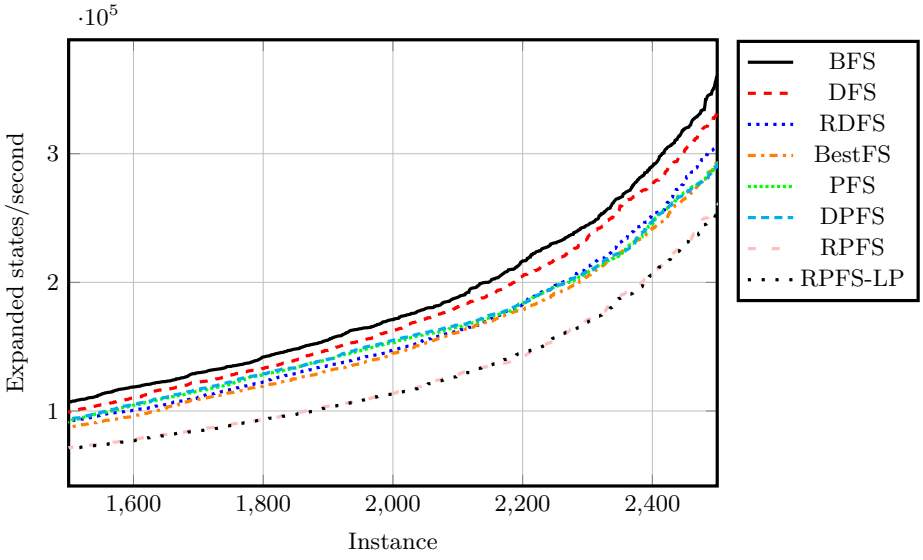
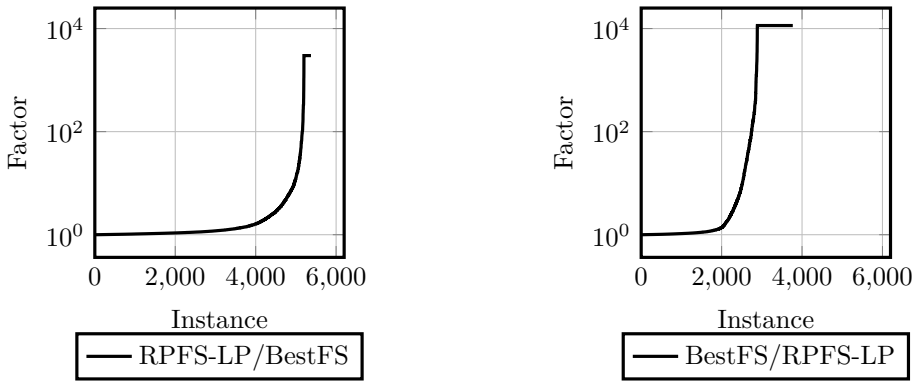


Fig. 10: Throughput plot (x-axis constrained between 1500 and 2500)

unique solved instances and is noticeably faster in more instances than BestFS. Note that the virtual best for the standard strategies have also been added to the table. It shows that RPFS-LP performs almost as well as the other strategies combined, as it almost solves the same number of instances.

We also compared each strategy's throughput, which measures how many states are expanded per second. This plot can be seen in Figure 10, and the full plot can be found in Appendix B. As expected, naive strategies such as BFS and DFS expand more states per second, whereas strategies such as BestFS and RPFS-LP expand fewer. This means that the improved heuristics of BestFS and RPFS-LP searches more efficiently through the state space, to outweigh the lower throughput.



(a) RPFS is slower ($\frac{RPFS-LP}{BestFS}$) (b) RPFS is faster ($\frac{BestFS}{RPFS-LP}$)

Fig. 11: A Ratio plot comparing RPFS-LP to BestFS

Strategy	**Total**	**Solved**	**Timeouts**	**Errors**	**Fastest**	**Unique**
BestFS	10029	8266	1763	0	5375	175
RPFS-LP	10029	8972	1057	0	3771	881

Table 6: RPFS-LP vs BestFS Statistics

Comparing RPFS-LP and BestFS Now that we have seen that RPFS-LP performs better than all standard strategies, we go into detail about how RPFS-LP compares to BestFS. Table 6 shows that BestFS is faster than RPFS-LP in significantly more instances. However, Figure 11 shows that 4236 of them are with a factor of less than 2. This means that of the 5375 instances where BestFS is faster, it is only noticeably faster in 1139 instances. The same is true for RPFS-LP, as it is noticeably faster in only 1600. Furthermore, Table 6 shows that RPFS-LP has 706 more uniquely solved instances than BestFS. From these observations, we can conclude that RPFS-LP is noticeably better than BestFS.

Clearly, RPFS-LP outperforms BestFS by a substantial margin. To highlight a drawback of BestFS, consider the Petri net in Figure 1 on page 4 and the proposition $\varphi = P_3 \geq 20$. BestFS will attempt to find a solution by continuously firing T_1, as it reduces the distance by one each time. Only when all 19 tokens have been removed from P_1, BestFS fires T_2 and finds the solution after expanding a total of 20 markings. As for RPFS-LP, because of the random choice, the number of expanded markings is not deterministic. We run RPFS-LP 100000 times to estimate the average number of expanded markings on φ. The results show that, on average, RPFS-LP expands 1.34 markings. This difference is to be expected, as RPFS-LP's random choice will prevent it from tunnel-visioning and allow it to explore other paths while still having a preference towards promising markings.

5.4 Competition Setup

Now that we have found that RPFS-LP outperforms all other search strategies, we test it in a competition setting. However, because the implementation of RPFS-LP does not work for non-reduced models, we have to use RPFS in its place. However, as we found out in Section 5.3, RPFS is only slightly slower and still outperforms the standard strategies.

For this test, we use the reachability cardinality queries as before. However, we also add the reachability fireability queries as they are converted to cardinality queries in Tapaal [12, p. 3]. This is done by using the competition script developed by Tapaal [13] at AAU for the MCC'22 competition. We first create a baseline by running the competition script without any modifications. Afterwards, we modify the script to use RPFS instead of BestFS. The results produced by RPFS are consistent with the baseline competition results. As seen in Table 7, RPFS increases the total number of solved instances by 516.

	BestFS	**RPFS**	**BestFS %**	**RPFS %**
Cardinality	20308	20471	96.08%	96.85%
Fireabilty	19541	19894	92.45%	94.12%
Total	39849	40365	94.27%	95.49%

Table 7: Solved instances from running the competition script. In total there are 42272 instances where 21136 are cardinality and 21136 are fireability.

6 Conclusion

Based on the thorough comparison of existing search strategies used in explicit state space search, we proposed a novel combination of random search and potency-based search where the likelihood of selecting a transition for the state space exploration is dynamically updated during the search itself. We implemented the heuristic search strategy in Tapaal, and the experimental results demonstrate that it outperforms all standard strategies of Tapaal.

We solved 8972 instances out of 10029 non-trivial reachability cardinality queries, 244 of which were unique. This is only 49 fewer than the total number solved by all standard strategies combined. Furthermore, we were able to improve the results for the MCC'22 contest by 516 more solved instances.

As we have shown in Section 5.2 on page 18, a very promising improvement would be to implement RPFS-LP so that it does not require models to be reduced in advance. This will allow us to use it in an MCC setting and will likely produce even better results than RPFS.

A possible improvement of RPFS-LP may be to select a more effective initialisation of our potencies, as we currently choose the first LP with a solution from LPS. However, it may be more beneficial for state space exploration to look at

all linear programs in LPS with a solution. The minimum solution across all of the linear program solutions can be used to optimise the potencies by providing better initialisation. One can also explore other options, such as aggregating all the optimal solutions in LPS. However, this will require a lot of computation, as LPS can become very large.

Another approach for solving reachability cardinality queries is to use the A* search algorithm. To do this, one will need to find a Petri net-specific heuristic that is guaranteed to over-approximate the distance from a given marking to a marking satisfying the query. A clever heuristic could lead to another improvement in state space exploration.

References

1. 2021, M.C.C.: Results. https://mcc.lip6.fr/2021/results.php (2022)
2. 2022, M.C.C.: Models. https://mcc.lip6.fr/2022/models.php (2022), copied from cluster path "/nfs/home/cs.aau.dk/pgj/MCC"
3. 2022, M.C.C.: Results. https://mcc.lip6.fr/2022/results.php (2022)
4. 2023, M.C.C.: Model Checking Contest 2023. https://mcc.lip6.fr/2023/ (2023)
5. Aagreen, E.F.L., Hansen, T.B.S., Herum, R.E.N., A.Jensen, F., Jensen, M.T.: Extending Petri Nets with Transition Weights to Improve Model-Checking using Monte Carlo Simulations (2022)
6. Bønneland, F., Dyhr, J., Jensen, P.G., Johannsen, M., Srba, J.: Simplification of CTL formulae for efficient model checking of petri nets. In: Petri Nets. Lecture Notes in Computer Science, vol. 10877, pp. 143–163, 9–14. Springer (2018)
7. Bønneland, F., Dyhr, J., Johannsen, M.: A Simplified and Stubborn Approach to CTL Model Checking of Petri Nets. Trans. Petri Nets Other Model. Concurr. **60**, 10–12 (2017), https://projekter.aau.dk/projekter/files/259777979/master.pdf
8. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) Computer Aided Verification. pp. 154–169. Springer Berlin Heidelberg, Berlin, Heidelberg (2000)
9. DEIS cluster: Distributed, embedded and intelligent systems cluster. https://github.com/DEIS-Tools/DEIS-MCC, accessed: 02/12/2022
10. Hart, P.E., Nilsson, N.J., Raphael, B.: A Formal Basis for the Heuristic Determination of Minimum Cost Paths. IEEE Trans. Syst. Sci. Cybern. **4**(2), 100–107 (1968). https://doi.org/10.1109/TSSC.1968.300136, https://doi.org/10.1109/TSSC.1968.300136
11. Hoffmann, J., Nebel, B.: The FF planning system: Fast plan generation through heuristic search. J. Artif. Intell. Res. **14**, 253–302 (2001). https://-doi.org/10.1613/jair.855, https://doi.org/10.1613/jair.855
12. Jensen, J.F., Nielsen, T., Oestergaard, L.K., Srba, J.: TAPAAL and Reachability Analysis of P/T Nets. Trans. Petri Nets Other Model. Concurr. **11**, 307–318 (2016). https://doi.org/10.1007/978-3-662-53401-4_16, https://doi.org/10.1007/978-3-662-53401-4_16
13. Jensen, P.G.: tapaal.sh. https://github.com/TAPAAL/verifypn/blob/main/Scripts/MCC22/competition-scripts/tapaal.sh (2022), accessed on 09/12/2022
14. Liebke, T., Wolf, K.: Solving finite-linear-path ctl-formulas using the CEGAR approach. Trans. Petri Nets Other Model. Concurr. **15**, 150–164 (2021).

https://doi.org/10.1007/978-3-662-63079-2_7, https://doi.org/10.1007/978-3-662-63079-2_7
15. Makhorin, A.: GNU Linear Programming Kit. http://most.ccib.rutgers.edu/glpk.pdf (2013)
16. Murata, T.: Petri nets: Properties, analysis and applications. Proc. IEEE **77**(4), 541–580 (1989). https://doi.org/10.1109/5.24143, https://doi.org/10.1109/5.24143
17. Petri, C.: Kommunikation mit Automaten. Ph.D. thesis, TU Darmstadt (1962)
18. TAPAAL: Tool for verification of timed-arc Petri nets. https://www.tapaal.net/, accessed: 21/11/2022
19. Base commit for this report's fork of VerifyPN (2022), https://github.com/TAPAAL/verifypn/commit/4e205729a1518727007f2971f9d123d8effa9650
20. Thierry-Mieg, Y.: Symbolic model-checking using its-tools. In: Baier, C., Tinelli, C. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 231–237. Springer Berlin Heidelberg, Berlin, Heidelberg (2015)
21. Verifypn. https://github.com/TAPAAL/verifypn, accessed: 12/12/2022
22. Wolf, K.: Lola - Theoretische Informatik - Universität Rostock. https://theo.informatik.uni-rostock.de/en/theo-forschung/werkzeuge/, accessed: 13/12/2022
23. Wolf, K.: Petri Net Model Checking with LoLA 2. In: Khomenko, V., Roux, O.H. (eds.) Application and Theory of Petri Nets and Concurrency - 39th International Conference, PETRI NETS 2018, Bratislava, Slovakia, June 24-29, 2018, Proceedings. Lecture Notes in Computer Science, vol. 10877, pp. 351–362. Springer (2018). https://doi.org/10.1007/978-3-319-91268-4_18, https://doi.org/10.1007/978-3-319-91268-4_18

A RPFS Experiment Results

In this appendix, we describe the experiment we perform to compare the Random Potency First Search (RPFS) strategy with the standard TAPAAL strategies. The experiment is conducted in accordance with the experimental method described in Section 3.1 on page 6.

A.1 21 RPFS

Figure 12 shows an experiment similar to the experiment described in Section 3.2, but we use RPFS instead of RDFS. Similarly to our previous experiment, there is little difference in the number of solved instances between the different seeds, but the virtual best still diverges. This means that we solve different unique instances depending on the seed we choose.

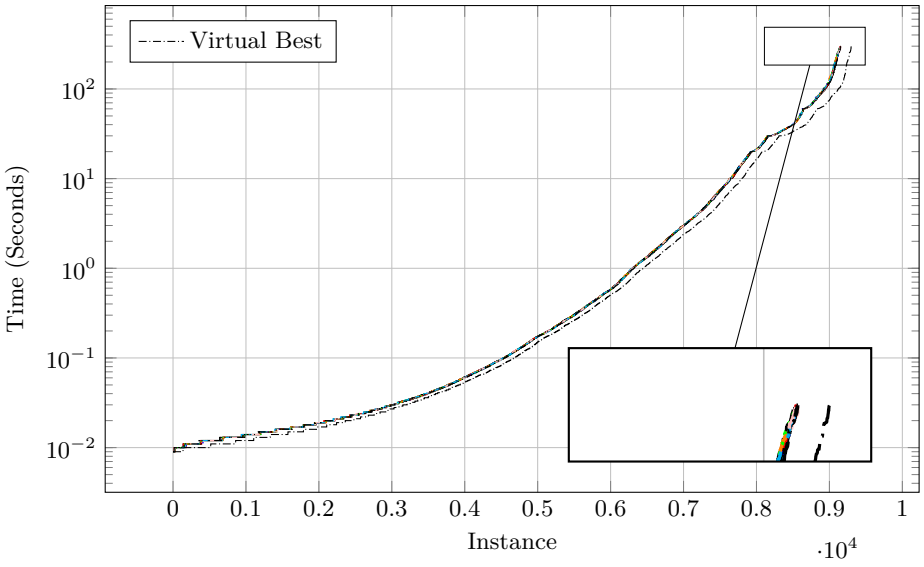
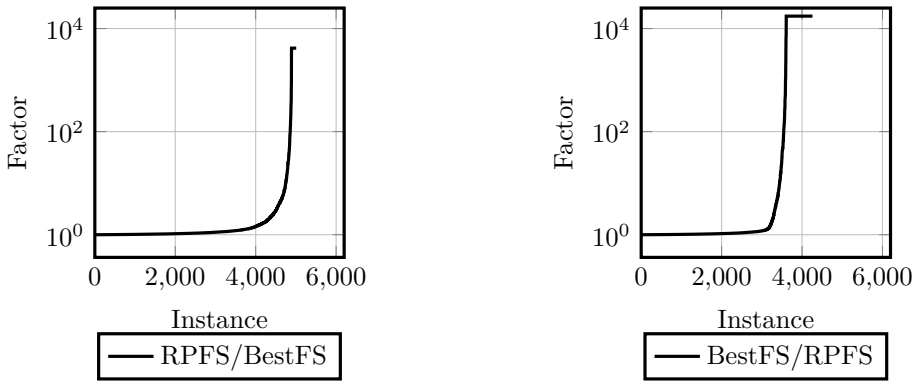


Fig. 12: Cactus plot of RPFS using 21 different seed offsets

A.2 RPFS vs BestFS Ratio Plot

In this ratio plot comparing the RPFS strategy to the BestFS strategy, we can see that BestFS is generally slightly faster at solving many instances. However, in the cases where RPFS is faster at solving an instance, it is generally faster by a significant factor. Furthermore, RPFS has significantly more unique instances solved.



(a) RPFS is slower ($\frac{RPFS}{BestFS}$) (b) RPFS is faster ($\frac{BestFS}{RPFS}$)

Fig. 13: A Ratio plot comparing RPFS to BestFS

A.3 RPFS vs BestFS table

In Table 8, we can see the specific values for the ratio plot of RPFS and BestFS. We can see that BestFS is faster in slightly more than half of the cases, whereas RPFS finds solutions to significantly more unique instances.

Strategy	**Total**	**Solved**	**Timeouts**	**Errors**	**Fastest**	**Unique**
BestFS	10217	8620	1592	5	5097	148
RPFS	10217	9134	1039	44	4190	662
Virtual Best	10217	9282	930	-	-	-

Table 8: RPFS vs BestFS Statistics

A.4 RPFS Cactus

Figure 14 shows the cactus plot of the standard TAPAAL strategies and the RPFS strategy. We can see that RPFS has gained an increase in the number of instances solved when compared to the standard TAPAAL strategies.

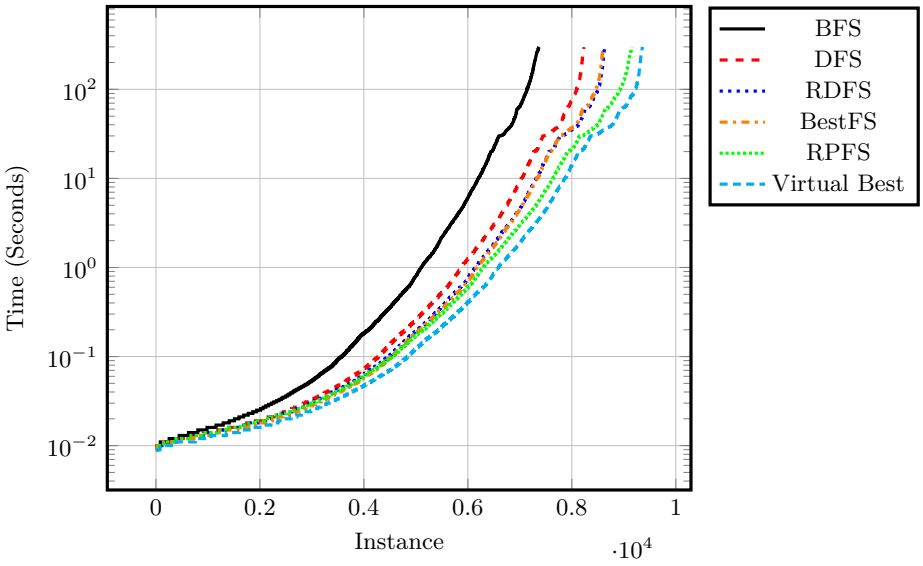


Fig. 14: A Cactus plot showing the comparison between standard search strategies and RPFS. The timeout is set to 5 min.

A.5 RPFS Table

Table 9 shows the specific values for the experiment with the standard TAPAAL strategies and RPFS. We can see that both BestFS and RDFS have more instances in which they are faster than RPFS. However, RPFS finds significantly more solutions to unique instances.

Strategy	**Total**	**Solved**	**Timeouts**	**Errors**	**Fastest**	**Unique**
BFS	10184	7367	2812	5	856	16
DFS	10184	8235	1942	7	1691	7
RDFS	10184	8641	1520	23	2452	41
BestFS	10184	8617	1562	5	2358	31
Virtual Best Standard	10184	9179	1000	-	-	-
RPFS	10184	9153	1026	5	2000	178
Virtual Best All	10184	9357	822	-	-	-

Table 9: RPFS compared to standard strategies

B Throughput Plot

Figure 15 shows the full throughput plot of the different models from the experiment described in Section 5 on page 17.

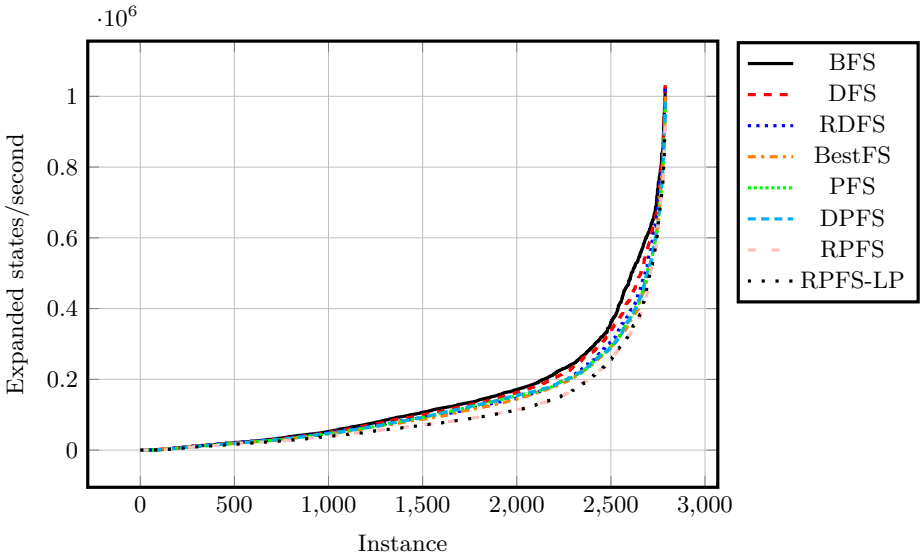


Fig. 15: Throughput plot