Potency-Based Heuristic Search with Randomness for Explicit Model Checking

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

Department of Computer Science, Aalborg University, Aalborg, Denmark

Abstract. Efficient state-space exploration has a significant impact on reachability analysis in explicit model checking and existing tools use several variants of search heuristics and random walks in order to overcome the state-space explosion problem. We contribute with a novel approach based on a random search strategy, where actions are assigned dynamically (on-the-fly) updated potencies, changing according to the variations of a heuristic distance to the goal configuration as encountered during the state-space search. We implement our general idea into a Petri net model checker TAPAAL and document its efficiency on a large benchmark of Petri net models from the annual Model Checking Contest. The experiments show that our heuristic search outperforms the standard search approaches in multiple metrics and that it constitutes a worthy addition to the portfolio of classical search strategies.

1 Introduction

Heuristic search strategies are widely applied in areas such as pathfinding and planning [7], where the popular A* algorithm [6] is often used. Similarly in model checking, instead of naively exploring the state-space using, e.g., Breadth First Search (BFS) or Depth First Search (DFS), it can be beneficial to use a heuristic search to navigate in the state-space. Heuristic search has also been succesfully applied in Petri net verification tools [12,5,11]. Since 2011, the annual Model Checking Contest (MCC) [10] has been held; here Petri net tools compete to solve a variety of problems, such as reachability analysis and deadlock detection. The tools ITS-Tools [11], SMPT [2] and LoLA [12] have implemented a random walk state-space exploration, and Tapaal [5] is using a random depth-first search. These tools have placed top three in the reachability category in MCC'21 [9] and MCC'22 [10], showing that randomness improves tool efficiency [12].

We propose a novel search heuristic, called *Random Potency-First Search* (RPFS), which combines a heuristic search based on distance function together with randomness in order to achieve a competitive-edge compared to the existing strategies. Our objective is to increase the likelihood of finding (not necessarily the shortest) trace to goal configurations. The unique property of RPFS is that, while searching the state-space, it learns which transitions are more likely to contribute to achieving a given reachability goal and it dynamically modifies

transition potencies (a number that expresses how likely a transition is to be selected during the search) according to the learned information. Such dynamic potency updates proved recently beneficial for guiding random Monte Carlo walks [1] but have not yet been explored for state-space search strategies.

We implement our RPFS search in the `verifypn` engine [8] of the tool TAPAAL [5], the best scoring tool in the reachability category at the most recent edition of MCC'22 [10]. We then benchmark RPFS against the existing search strategies present in TAPAAL on a large set of MCC'22 models. The results indicate a convincing performance of RPFS: it solves additional 512 queries compared to the second best search strategy, which is a significant 5% increase in the number of answered queries. We then rerun the TAPAAL competition script (using a portfolio of several different search strategies) used at MCC'22, by replacing the existing heuristic search with RPFS and obtain over 1% additional answers. As TAPAAL, the winner of MCC'22, solves close to 95% of all queries in the reachability benchmark, the additional 1% increase is very significant.

Preliminaries. Let $\mathbb{N}_0$ be the set of natural numbers including zero. A *Petri net* is a triple $N = (P, T, W)$ where P is a finite set of *places*, T is a finite set of *transitions* s.t. $T \cap P = \emptyset$, and $W : (P \times T) \cup (T \times P) \to \mathbb{N}_0$ is a *weighted flow function*. A *marking* on N is a function $M : P \to \mathbb{N}_0$ assigning a number of *tokens* to places. A transition $t \in T$ is *enabled* in M if $M(p) \geq W(p,t)$ for all $p \in P$. An enabled transition t in M can fire and produce a marking M', written $M \xrightarrow{t} M'$, where $M'(p) \stackrel{\text{def}}{=} M(p) - W(p,t) + W(t,p)$ for all $p \in P$.

Graphically, places are drawn as circles, transitions are drawn as rectangles, and arcs are drawn as arrows. Unless an arc is annotated by a number, its default weight is 1; we do not draw normal arcs with weight 0. Tokens are denoted as a number inside a place. A Petri net example is given in Figure 1.

We are interested in the reachability of the standard cardinality queries φ where $\varphi ::= e \bowtie e \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \neg\varphi$ for $\bowtie \in \{\leq, <, =, \neq, >, \geq\}$ and where expressions e are given by $e ::= n \mid p \mid e + e \mid e - e \mid n \cdot e$ such that $n \in \mathbb{N}_0$ and $p \in P$.

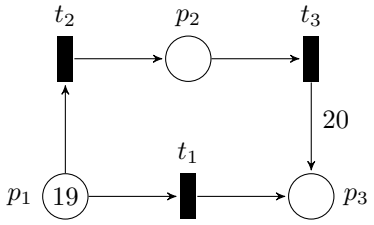

Fig. 1: Petri net example

An expression e in a marking M naturally evaluates to a number $eval(M, e)$ assuming that $eval(M, p) = M(p)$. The satisfaction relation $M \models \varphi$ is then defined in a straightforward way such that $M \models e_1 \bowtie e_2$ iff $eval(M, e_1) \bowtie eval(M, e_2)$.

For example in the net from Figure 1, we can reach a marking satisfying the query $p_3 \geq 20$ by firing the transition t_2 followed by t_3.

2 Explicit State-Space Search

In order to solve the reachability problem, explicit model checkers perform a state-space search as depicted in Algorithm 1. The algorithm is generic in the

Algorithm 1: Generic Reachability Search Algorithm

```
   Input: Petri net N, initial marking M_0, proposition φ
   Output: true if there is a reachable marking M s.t. M ⊨ φ, false otherwise
 1 Function Generic-Reachability-Search(N, M_0, φ):
 2     if M_0 ⊨ φ then return true ;
 3     Initialise(Waiting, M_0)              // Initialise Waiting with M_0
 4     Passed := {M_0}
 5     while Waiting is non-empty do
 6         M := SelectAndRemove(Waiting)
           // Select and remove an element M from Waiting
 7         for M' such that M -t-> M' where t is enabled in M do
 8             if M' is not in Passed then
 9                 if M' ⊨ φ then return true ;
10                 Passed := Passed ∪ {M'}
11                 Update(Waiting, t, M, M', φ)  // Update Waiting with M'
12     return false
```

way how it initialises the waiting set with M_0, updates the waiting set with the successor marking M', and selects and removes an element from the waiting set. In BFS strategy, the waiting set is implemented as a queue (FIFO) and in DFS it is implemented as a stack (LIFO). In RDFS, which stands for random DFS, the implementation is a stack where all successor markings M' of M are randomly shuffled before they are pushed to the waiting stack. A heuristic search strategy, called BestFS, implements the waiting set as a priority queue where markings that minimise the distance function $\texttt{Dist}(M, \varphi)$ are selected first. The distance function returns a number, estimating how far a given marking M is from satisfying the property φ. In the tool TAPAAL, the distance function is computed by Algorithm 2 taken from [8].

The four standard strategies are implemented for example in the Petri net model checker TAPAAL [5] and its engine `verifypn` [8]. The `verifypn` engine performs a number of preprocessing techniques like query simplification [4] using state equations [8] as well as structural reductions [3]. These techniques can solve a reachability query without executing the explicit state-space search. We evaluate the standard search strategies on Petri net models from MCC'22 [10]; each of the 1321 models are verified against 16 reachability cardinality queries, giving us the total of 21 136 problem instances. As 10 952 instances are solved without employing the explicit state-space search, we consider only the remaining 10 184 instances in our experiments on which we run all search strategies, enforcing 5 minute timeout and a memory limit of 16 GB. The experiments are run on a CPU cluster using AMD EPYC 7551 32-core processors with clock speed at 2.5 GHz. A reproducibility package can be obtained at GitHub[1].

[1] `https://github.com/theodor349/RPFS-reproducibility`

Algorithm 2: Distance Heuristics in TAPAAL, taken from [8]

Input: Marking M, cardinality query φ
Output: Nonnegative integer representing the distance of M from satisfying φ

Function `Dist`(M, φ)**:**
- **if** $\varphi = e_1 \bowtie e_2$ **then return** $\Delta(eval(M, e_1), \bowtie, eval(M, e_2))$;
- **if** $\varphi = \varphi_1 \wedge \varphi_2$ **then return** `Dist`(M, φ_1) + `Dist`(M, φ_2) ;
- **if** $\varphi = \varphi_1 \vee \varphi_2$ **then return** $min\{$`Dist`(M, φ_1), `Dist`$(M, \varphi_2)\}$;
- **if** $\varphi = \neg(e_1 \bowtie e_2)$ **then return** $\Delta(eval(M, e_1), \overline{\bowtie}, eval(M, e_2))$;
- **if** $\varphi = \neg(\varphi_1 \wedge \varphi_2)$ **then return** $min\{$`Dist`$(M, \neg\varphi_1)$, `Dist`$(M, \neg\varphi_2)\}$;
- **if** $\varphi = \neg(\varphi_1 \vee \varphi_2)$ **then return** `Dist`$(M, \neg\varphi_1)$ + `Dist`$(M, \neg\varphi_2)$;
- **if** $\varphi = \neg(\neg\varphi_1)$ **then return** `Dist`(M, φ_1) ;

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

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

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

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

Strategy	**Total**	**Solved**	**Solved %**	**Fastest**	**Unique**	**Unique All**
BFS	10184	7367	72.3 %	856	100	16
DFS	10184	8235	80.9 %	1691	14	7
RDFS	10184	8641	84.8 %	2452	218	41
BestFS	10184	8617	84.6 %	2358	177	31
Virtual Best	10184	9179	90.1 %	-	-	-
RPFS	10184	9153	89.9 %	2000	-	178
Virtual Best All	10184	9357	91.9 %	-	-	-

Table 1: Comparison of search strategies

The upper part of Table 1 (ignore for now the rows with RPFS strategy and Virtual Best All, as well as the column Unique All) shows the comparison of the basic search strategies as implemented in TAPAAL's verification engine, the best tool at MCC'22 in the reachability category. The numbers of solved queries for each strategy indicate that the random DFS (RDFS) as well as the standard heuristic search (BestFS) are the most succesfull strategies, both solving almost 85% of all queries. These are also the two strategies that solve largest number of queries fastest. The number of unique answers (number of queries that a given strategy solved but none of the other three provided any answer) also indicates that the random and heuristic searches are the most beneficial ones. It is worth to note that BFS also obtains a significant number (100) of unique answers. This is due to the positive queries that have a relatively short witness trace, where

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

(a) Initialise

```
Update(Waiting, t, M, M', φ) =
Waiting := Waiting ∪ {(M', t)}
Potencies(t) := max{1, Potencies(t)+
                Dist(M, φ) − Dist(M', φ)}
```

(b) Update

```
SelectAndRemove(Waiting) =
n := 0
foreach transiton t 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} Dist(M, φ)
Waiting := Waiting \ (M, t_best)
return M
```

(c) Select and remove

Fig. 2: The pseudocode for RPFS

BFS manages to find them but the other strategies miss these answers as they explore the state-space in a more depth-like manner.

These results indicate that one can consider to combine the heuristic search strategy with randomness, which we indeed tried by modifying the BestFS strategy so that we randomly select a marking from the priority queue among the first n markings that minimise the distance to the reachability query. The optimal value of n is around 100 where the performance peeks by solving 8761 instances (corresponding to 86.0% of all queries). In the next section, we shall present our idea of combining heuristic search with randomness by dynamically changing the probabilities of which transition to fire next during the state-space exploration. This novel method achieves even more significant performance improvement as it can solve almost 90% of all queries on its own.

3 Random Potency-First Search (RPFS)

We shall now describe our RPFS strategy where we assign potencies (positive numbers) to transitions. During the random search, transitions with higher potencies are more likely to be selected. We modify the transition potencies dynamically during the state-space search as we learn more information about which transitions decrease the distance to satisfying a given reachability query. The RPFS algorithm is described in Figure 2 where it instantiates the three primitives from Algorithm 1.

We initialise *Waiting* as a set of marking-transition pairs (M, t) representing a marking M that was reached by firing t (initially no transition is fired to reach M_0, so we use '-' here). The potencies of all transitions are set to 100. To select a marking from *Waiting*, we first pick some transition t from the waiting set with the probability $\mathit{Potencies}(t) / \sum_{t' \text{ appearing in } \mathit{Waiting}} \mathit{Potencies}(t')$ and return a marking that can be reached by firing t and minimises the distance function. For

an efficient implementation of the selection of t, we use an algorithm described in [1]. Finally, the update function changes the potency of the transition t by the difference between the current distance and the new distance after firing t. Should the distance increase, the potency of t is lowered accordingly but we keep the potency of all transitions strictly positive. 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 efficiently, as we do not need to search through all markings in the waiting set.

Let us consider the net from Figure 1 where we ask about the reachability of the query $p_3 \geq 20$. The standard BestFS heuristics attempts to find a solution by repeatedly firing t_1, as this reduces the distance to satisfying the reachability query. First when all 19 tokens are in the place p_3, the search backtracks and explores the firing of transition t_2 that brings us to the goal (after firing also t_3). In total, 20 markings are explored before t_2 is fired. Because of the random choice in RPFS, the number of explored markings is not deterministic; we run RPFS 100 000 times on this example and noticed that it expanded on average 3.03 markings before it reached the goal. Hence the random choice allows us to explore with a certain probability also alternative search options, while still having a preference towards the more promising markings.

4 Experimental Evaluation of RPFS

We have rerun the RPFS strategy search in the identical experimental setup as presented in Section 2 and the results are summarised in Table 1. Our RPFS strategy solves 89.9% of all queries, which is almost the same as all four remaining strategies solve together (the virtual best of these strategies is 90.1%). In the number of unique answers over all five search strategies, RPFS got the largest number of 178 unique answers; in particular the number of unique answers for BFS dropped to only 16, which is a clear indication that RPFS is more efficient in finding short witness traces that the RDFS and BestFS have difficulties dealing with. The benchmark contains a large number of easy-to-solve instances, where RDFS and BestFS are faster due to a smaller overhead compared to running RPFS. Still, on 2000 instances RPFS provides the fastest answer. We also experimented with different ways of modifying the transition potencies (e.g. by altering them by a constant value); these changes did not make any significant difference on the performance of RPFS. In Figure 3 we can see the four standard search strategies, RPFS and virtual best (over all five strategies) for our experimental evaluation. Independently for each search strategy, all instances (on x-axis) are ordered by their running times (on y-axis) and plotted in the comparison graphs. Note that y-axis uses the logarithmic scale. We can see that RPFS is clearly the best performing and it is closer to the virtual best (over all strategies) than to the BestFS and RDFS that show very similar performance.

In the final experiment, we run the competition script of TAPAAL on all models in the reachability category. The script uses a portfolio management,

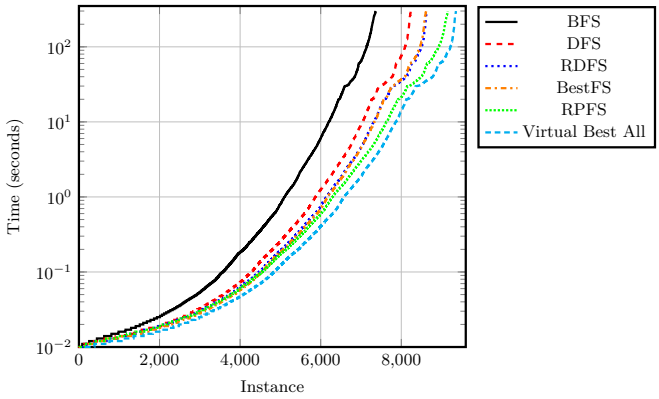

Fig. 3: A performance plot comparing standard search strategies and RPFS

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

Table 2: Solved instances from the total of 42272 (21136 for each subcategory)

where several search strategies and other optimisations are run in parallel on up to 4 cores for one hour on all 16 queries. In the script, we replace the use of BestFS with our new RPFS strategy and the number of answered queries can be seen in Table 2. We can observe a nontrivial increase in the answers, both in the cardinality queries and even more in the fireability ones (that are internally transformed into cardinality queries). In total, the RPFS heuristic provided an improvement of 1.22% over the performance of last year's winner of MCC'22.

5 Conclusion

We described RPFS, a novel idea of random, heuristic-based search strategy where transition potencies are dynamically updated during the state-space search. We instantiated this search strategy to the Petri net framework, however, we believe that the idea can be applicable to other formalisms as well. The RPFS strategy provides a significant performance boost to the existing strategies and it is now implemented in the state-of-the-art tool TAPAAL. In future work, we shall study how to further improve the performance of RPFS e.g. by guiding the search according to the solutions to state-equations obtained by linear programming, or by turning it into a variant of A* search.

Acknowledgements. We thank the anonymous reviewers for their comments and suggestions and Peter G. Jensen for his help with the experimential setup.

References

1. 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). https://doi.org/10.5281/zenodo.7690715, DAT8 Project report at Aalborg University, Denmark.
2. Amat, N., Berthomieu, B., Dal Zilio, S.: On the combination of polyhedral abstraction and smt-based model checking for petri nets. In: Buchs, D., Carmona, J. (eds.) Application and Theory of Petri Nets and Concurrency. LNCS, vol. 12734, pp. 164–185. Springer (2021)
3. Bønneland, F., Dyhr, J., Jensen, P., Johannsen, M., Srba, J.: Stubborn versus structural reductions for Petri nets. Journal of Logical and Algebraic Methods in Programming **102**(1), 46–63 (2019)
4. 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)
5. David, A., Jacobsen, L., Jacobsen, M., Jørgensen, K., Møller, M., Srba, J.: TAPAAL 2.0: Integrated development environment for timed-arc Petri nets. In: Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'12). LNCS, vol. 7214, pp. 492–497. Springer-Verlag (2012)
6. 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)
7. Hoffmann, J., Nebel, B.: The FF planning system: Fast plan generation through heuristic search. J. Artif. Intell. Res. **14**, 253–302 (2001)
8. 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)
9. Kordon, F., Bouvier, P., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Amat., N., Amparore, E., Berthomieu, B., Biswal, S., Donatelli, D., Galla, F., , Dal Zilio, S., Jensen, P., He, C., Le Botlan, D., Li, S., , Srba, J., Thierry-Mieg, Y., Walner, A., Wolf, K.: Complete Results for the 2021 Edition of the Model Checking Contest. http://mcc.lip6.fr/2021/results.php (2021)
10. Kordon, F., Bouvier, P., Garavel, H., Hillah, L.M., Hulin-Hubard, F., Amat., N., Amparore, E., Berthomieu, B., Biswal, S., Donatelli, D., Galla, F., , Dal Zilio, S., Jensen, P., He, C., Le Botlan, D., Li, S., , Srba, J., Thierry-Mieg, Y., Walner, A., Wolf, K.: Complete Results for the 2022 Edition of the Model Checking Contest. http://mcc.lip6.fr/2022/results.php (2022)
11. 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)
12. 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. LNCS, vol. 10877, pp. 351–362. Springer (2018)

A Appendix (Explicit State-Space Search)

Methodology for benchmarking against random search strategies.
Running random search strategies can result in different running times, depending on the chosen random seed. In order to eliminate such fluctuations, we run RDFS many times with 21 different random seeds. For each seed, we independently order all instances (x-axis) according to the running times (on y-axis in logarithmic scale) and present the curves in Figure 4. We can observe that the 21 random strategies vary a relatively small amount in the total number of instances solved. In fact, out of the total 10 184 instances, the largest difference in the number of instances solved between any two RDFS strategies is 50. In our experimental evaluation, we used the seed of the random run that solved the median number of answers in the benchmark. We can also see that repeating random runs with different seeds can improve the answer rates as shown by the virtual best curve in the figure (here an instance is solved if at least one of the random searches found the answer).

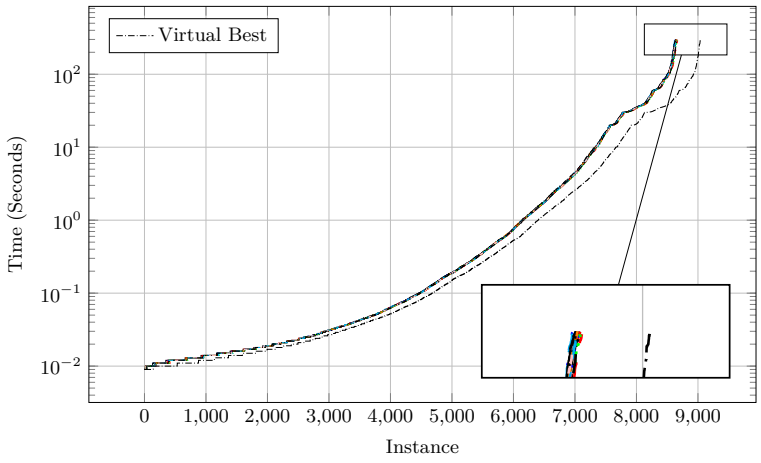

Fig. 4: Comparison of RDFS runs using 21 different seed offsets