Comparing Search Algorithms on the Retrosynthesis Problem

Anonymous submission

Abstract

In this article we try different algorithms, namely Nested Monte Carlo Search and Greedy Best First Search, on AstraZeneca's open source retrosynthetic tool : AiZynthFinder. We compare these algorithms to AiZynthFinder's base Monte Carlo Tree Search on a benchmark selected from the PubChem database and by Bayer's chemists.

We show that both Nested Monte Carlo Search and Greedy Best First Search outperform AstraZeneca's Monte Carlo Tree Search, with a slight advantage for Nested Monte Carlo Search while experimenting on a playout heuristic. We also show how the search algorithms are bounded by the quality of the policy network, in order to improve our results the next step is to improve the policy network.

Introduction

Retrosynthesis is a domain of organic chemistry which consist in finding a synthetic route (a sequence of reactions) for a given molecule in order to synthesize it from a given set of available precursor molecules (Lin, Tu, and Coley 2022). It is an important part of organic chemistry molecule synthesis, and can be used to produce newfound drugs. What we aim for in this paper is to evaluate the strength and weaknesses of two search algorithms by comparing them to AiZynthFinder's Monte Carlo Tree Search (MCTS) on a small benchmark consisting of curated and complex molecules, covering many reactions encountered by chemists.

The second section presents the AiZynthFinder retrosynthesis tool, the third section describes the search algorithms we compare, the fourth section details the benchmark used to compare the search algorithms, the fifth section gives experimental results.

AiZynthFinder

AiZynthFinder (Genheden et al. 2020) is a retrosynthesis tool made by AstraZeneca's research and development. It has the advantage of being open source, understandable and well described.

AiZynthFinder uses a neural network trained on USPTO 50K, a set containing about 50 000 reactions from organic chemistry patents (Lowe 2012). That neural network role is to select the best reactions among these 50 000 given a molecule we want to synthesize, it also gives a value to each move (the prior). Due to how the program works, it's hard to do without that neural network and the priors because it would require finding another method to evaluate the reactions available for a molecule. In this article we use AstraZeneca's open source pre-trained network. Training a network to predict more accurately the reactions for molecule retrosynthesis is another domain called "one step retrosynthesis", see (Lin, Tu, and Coley 2022), it is not what we aim to explore here.

AiZynthFinder takes the SMILES: a string representation of a molecule, as an input which makes the first state. A state is a set of molecules, from each state the neural network proposes some reactions producing a molecule from the state from precursors. If a reaction is played the molecule is removed from the state, and the precursors are added (a reaction can also be a modification of the molecule's shape only, not removing any atom, we call these "structural moves"). Retrosynthesis often uses and/or trees, here the "and" are combined into a single state as it makes the search more simple.

The goal of retrosynthesis is to obtain a state only made from molecules (precursors) that are in stock, here we use the ZINC (Irwin and Shoichet 2005) molecule database, a curated collection of commercially available chemical compounds prepared especially for virtual screening. Any state is evaluated by AiZynthFinder's base evaluation function which is $0.95 * (fraction\_of\_molecules\_making\_the\_state\_in\_stock) + 0.05 * (depth\_of\_the\_reaction\_tree)$. We don't modify it directly in this study but exploring different score functions could be interesting in future works.

Algorithms

Our algorithms use common primitives:

- M_{state} represents the moves available from state $state$.
- play($state$, m) executes the move m on the state $state$ and returns a children state of $state$.
- score($state$) evaluates a state with a float value.
- $(state, m)$ when used in a function represents the new state reached from $state$ ny playing the move m, thus $visits(state, m)$ means $visits(play(state, m))$.

AizynthFinder's MCTS

AiZynthFinder uses a MCTS algorithm with priors very similar to PUCT. PUCT stands for "Prior Upper Confidence bounds applied to Trees", it is a generalisation of the UCT algorithm (Kocsis and Szepesvári 2006) using priors for each state of the problem (the prior is the policy at the output of the neural network here), see (Rosin 2011a) for the original version of PUCT. PUCT has been used in AlphaGo (Silver et al. 2016) and Alpha Zero (Silver et al. 2018). Just like PUCT, this MCTS algorithm explores the tree using playouts: selecting the best move to play according to its evaluation, until it reaches a state not explored yet or reaches a terminal state. That state is added to the transposition table (containing the number of visits for that state, for each children and the evaluations), then the score of that newly explored state is retro-propagated to update the parent states.

AiZynthFinder's MCTS in Algorithm 1 differs from standard PUCT (Rosin 2011a) in how the *bandit* value is determined. In all our experiments the c hyper-parameter is AstraZeneca's base one of 1,4.

Algorithm 1: The MCTS algorithm

```
1:  **function** MCTS($state$)
2:      **if** terminal($state$) **then return** score($state$)
3:      **end if**
4:      **if** $state$ is not in the transposition table **then**
5:          add $state$ to the transposition table
6:          **return** score($state$)
7:      **else**
8:          $best\text{-}score \leftarrow -\infty$
9:          $mean \leftarrow prior(state, m)$
10:         **if** $visits(state) > 0$ **then**
11:             $mean \leftarrow \frac{sumEvals(state)+prior(state,m)}{visits(state)+1}$
12:         **end if**
13:         **for each** $m$ in $M_{state}$ **do**
14:             $\mu \leftarrow mean$
15:             **if** $visits(s, m) > 0$ **then**
16:                 $\mu \leftarrow \frac{sumEval(state,m)+prior(state,m)}{visits(state,m)+1}$
17:             **end if**
18:             $bandit \leftarrow \mu + c \times \sqrt{\frac{2\times(visits(state)+1)}{(visits(state,m)+1)}}$
19:             **if** $bandit > best - score$ **then**
20:                 $best\text{-}score \leftarrow bandit$
21:                 $best\text{-}move \leftarrow m$
22:             **end if**
23:         **end for**
24:         $res \leftarrow$ MCTS($play(state, best\text{-}move)$
25:         update the transposition table
26:         **return** $res$
27:     **end if**
28: **end function**
```

Nested Monte Carlo Search

Monte Carlo Tree Search (MCTS) has been successfully applied to many games and problems (Browne et al. 2012).

Nested Monte Carlo Search (NMCS) (Cazenave 2009) is an algorithm that works well for puzzles and optimization problems. It biases its playouts using lower level playouts. At level zero NMCS adopts a uniform random playout policy. Online learning of playout strategies combined with NMCS has given good results on optimization problems (Rimmel, Teytaud, and Cazenave 2011). Other applications of NMCS include Single Player General Game Playing (Méhat and Cazenave 2010), Cooperative Pathfinding (Bouzy 2013), Software testing (Poulding and Feldt 2014), heuristic Model-Checking (Poulding and Feldt 2015), the Pancake problem (Bouzy 2016), Games (Cazenave et al. 2016) and the RNA inverse folding problem (Portela 2018).

Online learning of a playout policy in the context of nested searches has been further developed for puzzles and optimization with Nested Rollout Policy Adaptation (NRPA) (Rosin 2011b). NRPA has found new world records in Morpion Solitaire and crosswords puzzles. NRPA has been applied to multiple problems: the Traveling Salesman with Time Windows (TSPTW) problem (Cazenave and Teytaud 2012; Edelkamp et al. 2013), 3D Packing with Object Orientation (Edelkamp, Gath, and Rohde 2014), the physical traveling salesman problem (Edelkamp and Greulich 2014), the Multiple Sequence Alignment problem (Edelkamp and Tang 2015) or Logistics (Edelkamp et al. 2016). The principle of NRPA is to adapt the playout policy so as to learn the best sequence of moves found so far at each level.

The use of Gibbs sampling in Monte Carlo Tree Search dates back to the general game player Cadia Player and its MAST playout policy (Finnsson and Björnsson 2008).

NMCS (Cazenave 2009) recursively calls lower level NMCS on children states of the current state in order to decide which move to play next, the lowest level of NMCS being a random playout, selecting uniformly the move to execute among the possible moves. A heuristic can be added to the playout choices.

Here we used an heuristic to penalize the structural moves (when nothing is added or removed from the molecule, it only changes shape) because these moves often occupied most of the limited depth of search, even looping to a previous state sometimes. We use the score of the children (between 0 and 1), to which we add 1 if the largest molecule weight in the state is smaller than its parent largest molecule weight, that value is then the chance to select that move, over the sum of every other move's values. The modified score function for the heuristic and the heuristic are descibed in algorithm 3.

While we did not use softmax to harden the heuristic, nor tuned the parameters, that simple heuristic allowed us to diminish the structural moves problem, giving them less than half the chance of being selected in playouts than before, and led to better results.

Greedy Best First Search

GBFS stands for Greedy Best-First Search. It is a simple algorithm which consists in opening (and removing) the best node from a list of nodes sorted by their scores, evaluating all its children and inserting them in the sorted list of nodes, and then repeat the operation by opening the new best

Algorithm 2: The NMCS algorithm.

```
1: function NMCS($c\text{-}st$, $lv$)
2:     if $lv = 0$ then                                   ▷ Playout if lv = 0
3:         $ply \leftarrow 0$
4:         $seq \leftarrow \{\}$
5:         while not terminal($c\text{-}st$) do
6:             we use this line for random playouts
7:             $move \leftarrow randomChoice(M_{c\text{-}st})$        ▷ by default we use random playouts
8:             $move \leftarrow$ HEURICHOICE$(M_{c\text{-}st})$        ▷ when possible, using an heuristic can lead to better results
9:             $c\text{-}st \leftarrow play(c\text{-}st, move)$
10:            $seq[ply] \leftarrow move$
11:            $ply+ = 1$
12:        end while
13:        return (score ($c\text{-}st$), $seq$)
14:    else
15:        $best\text{-}score \leftarrow -\infty$
16:        $best\text{-}sequence \leftarrow []$
17:        $ply \leftarrow 0$
18:        while $c\text{-}st$ is not terminal do
19:            for each $move$ in $M_{c\text{-}st}$ do
20:                $n\text{-}st \leftarrow play(c\text{-}st, move)$
21:                $(score, seq) \leftarrow$NMCS$(n\text{-}st, level - 1)$
22:                if $score \geq best\text{-}score$ then
23:                    $best\text{-}score \leftarrow score$
24:                    $best\text{-}sequence[ply..] \leftarrow move + seq$
25:                end if
26:            end for
27:            $next\text{-}move \leftarrow best\text{-}sequence[ply]$
28:            $ply \leftarrow ply + 1$
29:            $c\text{-}state \leftarrow play(ct\text{-}st, next\text{-}move)$
30:        end while
31:        return $(best\text{-}score, best\text{-}sequence)$
32:    end if
33: end function
```

node (Doran and Michie 1966). The evaluation function can use playouts to make the algorithm closer to a Monte Carlo Search algorithm. Like MCTS, that algorithm can lock itself in a local minimum, but is faster (and less accurate) as it skips the playout and the associated calculations between each node discovery. Both lack the forced progress in depth of NMCS. We describe GBFS in Algorithm 4.

The function insert($open\text{-}states$, $score$, $new\text{-}state$) inserts $new\text{-}state$ in the sorted list $open\text{-}states$ given the $score$ value.

We modified the evaluation function similarly to the NMCS playout one : if the children's biggest molecule is smaller than the parent's then add 1 to the score. That modification allows to avoid structural moves, multiplying states with high scores, but also prevents structural moves that are sometimes necessary to the resolution of a molecule, finding an alternative solution would greatly help this algorithm.

Algorithm 3: The modified score function and heuristic choice.

```
1: function VAL($s1$, $s2$)
2:     val ($s1$, $s2$) :
3:     if biggest molecule from $s1$ is smaller than biggest molecule from $s2$ then
4:         return $score(s1) + 1$
5:     end if
6:     return $score(s1)$
7: end function
8: function HEURICHOICE($moves$)
9:     $weights \leftarrow [val(play(st, m), st)$ for $m$ in $M_{st}]$
10:    $su \leftarrow 0$
11:    $rd \leftarrow randomRange(0, sum(weights))$
12:    $ind \leftarrow 0$
13:    for $w$ in $weights$ do
14:        $su \leftarrow su + w$
15:        if $su \geq rd$ then return $ind$
16:        end if
17:        $ind \leftarrow ind + 1$
18:    end for
19:    return $ind$
20: end function
```

Benchmark

To compare our algorithms to AstraZeneca's MCTS, we use a small curated subset containing 40 SMILES representation of drugs from the PübChem database selected by (Ertl and Schuffenhauer 2009) (C) and 20 SMILES representation of molecules selected by Bayer's chemists (A). The 40 molecules selected from PubChem by the authors of the benchmark were obtained randomly in such a way to cover small to large molecules, the original goal of this benchmark was to test the prediction of difficulty of retrosynthesis of these molecules, thus these 40 molecules feature some of the hardest to synthetize according to some chemists (Ertl and Schuffenhauer 2009).

Generally, one step retrosynthesis uses USPTO-50K as a benchmark. However here we are not trying to benchmark the reaction propositions of our neural network, but how our algorithme solve the retrosynthesis problem. Thus our benchmark provides a few advantages : proposing harder molecules to synthetize, reducing the benchmark size allowing us to focus more on each molecule and giving a fixed benchmark to compare with others. That is why we think this benchmark is the most appropriate.

You can find the SMILES representation of this benchmark in table 3 in the appendix.

Results

These experiences were made on a 3.50GHz intel core i5-6600K on windows 10 with 32 Gb of RAM. We used the same common parameters for every algorithm:

- Max step for substrates (how many reactions we can make from a substrate to the target molecule) : 15
- Policy cutoff cumulative : 0.995

Algorithm 4: The GBFS algorithm.

```
1: function GBFS($ini\text{-}state$, $max\text{-}iter$)
2:     $open\text{-}states \leftarrow [ini\text{-}state]$
3:     $state \leftarrow ini\text{-}state$
4:     $iter \leftarrow 0$
5:     $best\text{-}state \leftarrow ini\text{-}state$
6:     $best\text{-}score \leftarrow$ score($ini\text{-}state$)
7:     while not optimal($state$) and $open\text{-}states \neq []$ and $iter < max\text{-}iter$ do
8:         $iter \leftarrow iter + 1$
9:         $state \leftarrow$ pop( $open\text{-}states$, 0)
10:        for each $move$ in $M_{state}$ do
11:            $new\text{-}state \leftarrow$ play($state$, $move$)
12:            $score \leftarrow$ score($new\text{-}state$)
13:            insert($open\text{-}states$, $score$, $new\text{-}state$)
14:            if $score \geq best\text{-}score$ then
15:                $best\text{-}state \leftarrow state$
16:                $best\text{-}score \leftarrow score$
17:            end if
18:        end for
19:    end while
20:    return $best\text{-}state$
21: end function
```

- Policy cutoff number (maximum number of possible moves returned on a molecule) : 50
- Filter cutoff : 0.05

MCTS

To compare our algorithms, we ran AiZynthFinder MCTS at least 2 times on each of the 60 molecules of the benchmark with the base settings, C= 1.4. Running it a few times only is not problematic because the MCTS results were observed to be very stable (mostly deterministic) on the few molecules we ran it multiple times on, also our goal is not to measure the exact solving times as they heavily depend on the implementation, the language and the hardware, but see how many molecules of the benchmark a given algorithm can find a synthetic route for. The times specified in the results are not the main part of the results and are here only to give an idea of the differences of performance between the algorithms. The main part of the result is wether a molecule was solved by an algorithm or not.

First we ran the MCTS (Table 1) with a timeout of 2 minutes an maximum number of iteration of 100 (”MCTS 2mn 100it”), the C molecules (from PubChem) were mostly either solved instantly ($<$ 200ms) or not solved in 2 minutes, on the contrary the molecules selected by Bayer’s chemists took generally more time to be solved if they were. In addition, a bigger proportion of molecules from A were solved than from C, which can be explained by their sizes and the presence of distinctive atoms (like fluor). We then launched the MCTS with a time limit of 20 minutes, solving few more molecules, and finally, we launched a MCTS of 2h on some of the remaining ones : C2, C35, C37, C38. Only C37 got solved in 5236.093 seconds.

The values next to the molecule id on Table 1 is the time

Table 1: AiZynthFinder’s MCTS results

C1 (0.167), C3 (0.130), C5 (0.080), C8 (0.125), C13 (0.061), C14 (0.060), C21 (0.061), C23 (0.149), C25 (0.064), C26 (0.071), C31 (0.063), C34 (0.063), C40 (0.066), A11 (0.120)	C20(22.088), C36(41.294), A1 (5.225), A2 (12.948), A4 (0.821), A7 (84.585), A9 (0.743), A14 (0.792), A16 (12.490), A17 (2.564), A18 (15.447), A19 (0.343)	C19 (310.966), C22 (425.647), A5 (1086.547), A15 (587.830)	C2, C4, C6, C7, C9, C10, C11, C12, C15, C16, C17, C18, C24, C27, C28, C29, C30, C32, C33, C35, C37*, C38, C39, A0, A3, A6, A8, A10, A12, A13

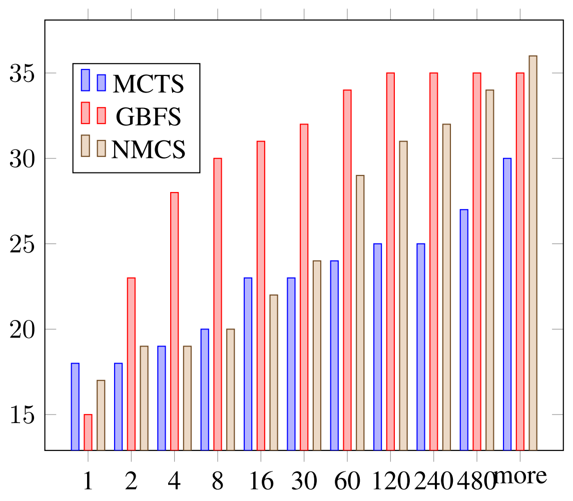

Figure 1: Distribution of the numbers of molecules solved with times in seconds

in seconds it took to find the solution.

As you can see on Table 1, increasing the MCTS search time doesn't help much, the molecules are either solved instantaneously (< 200ms) or very quickly. The instantaneous solving is due to the neural network (one-step retrosynthesis) which immediately proposes the right solution in 1 or 2 reactions for small molecules, meaning that the quality of the search algorithm doesn't matter on these molecules, and is solved equally as fast with the MCTS, the NMCS, the GBFS, and even a NMCS without playout. These molecules are not useful to our research so we remove them from the set for further experiences.

Other Algorithms

Every molecule solved by the MCTS was also solved by NMCS and GBFS, even if they may be slower than MCTS for easy states (the GBFS has to instancy 50 childs per opened state even if it uses only one or two). Thus, we are going to focus on molecules not solved by AizynthFinder base MCTS and those that took at least a minute to solve.

For the NMCS, we first perform a level 1 NMCS using only the 5 best moves from each state, instead of the 50 best given by the Policy cutoff number, if that NMCS (usually shorter than 1 minute) fails we perform a much longer level 1 NMCS using all the 50 moves. If even that fails, we launch the level 2 NMCS. The level 2 NMCS is very slow, but is able to solve molecules unsolved by both MCTS and GBFS.

The GBFS was only launched once on each molecule because the algorithm is deterministic, it was launched with a time limit of 20 minutes, molecules not solved by then are considered unsolved. Given enough time, the GBFS explores the entire tree.

First, we can notice on figure1 that NMCS and GBFS outperform astrazeneca's MCTS in the long run, but as the y-axis is starting at 15, they sligthly underperform on trivial molecules. In fact for the most trivial molecules (solved by 1 or 2 steps in less than 1s), they take about 0.6s compared to about 0.1s for MCTS, this is because they don't open the most promising node according to the neural network first. It is observed that GBFS offers a better start than both NMCS and GBFS, but reaches its potential before both of them, NMCS find retrosynthesis routes slower but can find more of them. MCTS (or any algorithm opening the reaction greedily according to the neural network) is better for very short experiments ($< 1s$), GBFS is better for medium length experiments ($< 60s$) and NMCS appears to be better for longer experiments and more complex molecules.

Here we will focus our attention to the molecules that took more than 2 minutes to solve or were not solved, as all the other molecules were solved by any algorithm:

C19, C20, C22, C37, A0, A3, A5, A6, A8, A12, A13 and A15

Table 2: Comparison of algorithms

C19	310.966	3.765	81.267
C20	119.230	X	46.166
C22	425.647	50.800	4.255
C37	5236.093	2.836	8.919
A0	X	4.569	84.582
A3	X	2.075	176.445
A5	1086.547	2.145	60.000
A6	X	29.604	1075.222
A8	X	95.365*	X
A12	X	47.78	431.095
A13	X	X	518.727
A15	587.830	3.587	37.178

The result on A8 were obtained by opening the 200 best nodes given by the neural network and not the 50 best, in addition to only searching up to a depth of 5 instead of 15. NMCS and MCTS were unable to solve the molecule with the same parameters. The reactions required to solve A8 were not present in the 50 best proposals from the neural network, but in the 200 best. On the other hand, a top 5 level 1 NMCS was enough to solve 30 of the 60 molecules, meaning the NN was very accurate in these cases. It put the emphases on how much results depend on the accuracy of the NN, again the one we used was trained by AiZynthFinder's team on the public USPTO 50K reaction dataset (Lowe 2012), which does not feature many reactions present in licensed datasets such as Reaxys or Pistachio (Thakkar et al. 2020).

Our GBFS was unable to solve C20, despite being solved by the MCTS and the NMCS, we think it was because our search heuristic favors the non structural moves when a structural move is required here to cut the carbon cycle. Overall, molecules with long carbon cycles posed problems to be solved to all the algorithms and C20 was the smallest and most simple of them from the benchmark.

Like with MCTS, running the other algorithms longer did not yield much improvement. It is because the neural network doesn't always proposes the best reactions. Some reaction patterns are not present in the USPTO-50K, the free dataset on which our neural network was trained on by AstraZeneca. Obtaining a better network, possibly trained on more complete dataset (Thakkar et al. 2020) could improve our results greatly.

To put our results in perspective, out of the 60 molecules of our benchmark we managed to solve 35 molecules with GBFS, and 36 with NMCS, while (Franz et al. 2022) managed to solve 38 molecules with a MCTS and 41 with a DFPN (AizynthFinder's DFPN does not yield such results). We hope we will be able to try with a more complete dataset and the according NN in the future. Bigger molecule would still be a challenge given all the moves (reactions) they offer, but we think it could help with the few unsolved small molecules: C4, C6, C7, C10, C12, C35, C38 and A10.

Conclusion

While MCTS solves 31 molecules out of 60 from this benchmark, GBFS solves 35 in reasonable time and NMCS solves 36. We showed that GBFS and NMCS could provide a satisfying improvement in performances, especially since GBFS and NMCS are much simpler and don't use the neural network as a search policy beyond the reaction proposition, unlike MCTS. We believe that a more accurate neural network trained on a bigger dataset, and a more complete template set would improve the performances.

Future Works

Retrosynthesis is a vast topic, and much remains to be done, we only scratched the surface of AiZynthFinder here. It would be interesting to experiment on more algorithms, including the canonical UCT and applying the prior to the algorithms we used here, use other score functions or train and use another neural network. These research would require a lot of time, and we can only encourage other computer scientists to try their algorithms and score functions on AiZynthFinder.

References

Bouzy, B. 2013. Monte-Carlo Fork Search for Cooperative Path-Finding. In *Computer Games Workshop at IJCAI*, 1–15.

Bouzy, B. 2016. Burnt Pancake Problem: New Lower Bounds on the Diameter and New Experimental Optimality Ratios. In *SOCS*, 119–120.

Browne, C.; Powley, E.; Whitehouse, D.; Lucas, S.; Cowling, P.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1): 1–43.

Cazenave, T. 2009. Nested Monte-Carlo Search. In *IJCAI*, 456–461.

Cazenave, T.; Saffidine, A.; Schofield, M. J.; and Thielscher, M. 2016. Nested Monte Carlo Search for Two-Player Games. In *AAAI*, 687–693.

Cazenave, T.; and Teytaud, F. 2012. Application of the Nested Rollout Policy Adaptation Algorithm to the Traveling Salesman Problem with Time Windows. In *Learning and Intelligent Optimization - 6th International Conference, LION 6*, 42–54.

Doran, J. E.; and Michie, D. 1966. Experiments with the graph traverser program. *Proceedings of the Royal Society of London. Series A. Mathematical and Physical Sciences*, 294(1437): 235–259.

Edelkamp, S.; Gath, M.; Cazenave, T.; and Teytaud, F. 2013. Algorithm and knowledge engineering for the TSPTW problem. In *Computational Intelligence in Scheduling (SCIS), 2013 IEEE Symposium on*, 44–51. IEEE.

Edelkamp, S.; Gath, M.; Greulich, C.; Humann, M.; Herzog, O.; and Lawo, M. 2016. Monte-Carlo Tree Search for Logistics. In *Commercial Transport*, 427–440. Springer International Publishing.

Edelkamp, S.; Gath, M.; and Rohde, M. 2014. Monte-Carlo Tree Search for 3D Packing with Object Orientation. In *KI 2014: Advances in Artificial Intelligence*, 285–296. Springer International Publishing.

Edelkamp, S.; and Greulich, C. 2014. Solving physical traveling salesman problems with policy adaptation. In *Computational Intelligence and Games (CIG), 2014 IEEE Conference on*, 1–8. IEEE.

Edelkamp, S.; and Tang, Z. 2015. Monte-Carlo Tree Search for the Multiple Sequence Alignment Problem. In *Proceedings of the Eighth Annual Symposium on Combinatorial Search, SOCS 2015*, 9–17. AAAI Press.

Ertl, P.; and Schuffenhauer, A. 2009. Estimation of synthetic accessibility score of drug-like molecules based on molecular complexity and fragment contributions. *Journal of Cheminformatics*, 1(1): 8.

Finnsson, H.; and Björnsson, Y. 2008. Simulation-Based Approach to General Game Playing. In *Aaai*, volume 8, 259–264.

Franz, C.; Mogk, G.; Mrziglod, T.; and Schewior, K. 2022. Completeness and Diversity in Depth-First Proof-Number Search with Applications to Retrosynthesis. In *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence*, 4747–4753. Vienna, Austria: International Joint Conferences on Artificial Intelligence Organization. ISBN 978-1-956792-00-3.

Genheden, S.; Thakkar, A.; Chadimová, V.; Reymond, J.-L.; Engkvist, O.; and Bjerrum, E. 2020. AiZynthFinder: a fast, robust and flexible open-source software for retrosynthetic planning. *Journal of Cheminformatics*, 12(1): 70.

Irwin, J. J.; and Shoichet, B. K. 2005. ZINC – A Free Database of Commercially Available Compounds for Virtual Screening. *Journal of chemical information and modeling*, 45(1): 177–182.

Kocsis, L.; and Szepesvári, C. 2006. *Bandit Based Monte-Carlo Planning*, volume 4212 of *Lecture Notes in Computer Science*, 282–293. Berlin, Heidelberg: Springer Berlin Heidelberg. ISBN 978-3-540-45375-8.

Lin, M. H.; Tu, Z.; and Coley, C. W. 2022. Improving the performance of models for one-step retrosynthesis through re-ranking. *Journal of Cheminformatics*, 14(1): 15.

Lowe, D. M. 2012. *Extraction of chemical structures and reactions from the literature*. Thesis, University of Cambridge. Accepted: 2013-07-23T08:23:10Z.

Méhat, J.; and Cazenave, T. 2010. Combining UCT and Nested Monte Carlo Search for Single-Player General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4): 271–277.

Portela, F. 2018. An unexpectedly effective Monte Carlo technique for the RNA inverse folding problem. *BioRxiv*, 345587.

Poulding, S. M.; and Feldt, R. 2014. Generating structured test data with specific properties using nested Monte-Carlo search. In *GECCO*, 1279–1286.

Poulding, S. M.; and Feldt, R. 2015. Heuristic Model Checking using a Monte-Carlo Tree Search Algorithm. In *GECCO*, 1359–1366.

Rimmel, A.; Teytaud, F.; and Cazenave, T. 2011. Optimization of the Nested Monte-Carlo Algorithm on the Traveling Salesman Problem with Time Windows. In *EvoApplications*, volume 6625 of *LNCS*, 501–510. Springer.

Rosin, C. D. 2011a. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3): 203–230.

Rosin, C. D. 2011b. Nested Rollout Policy Adaptation for Monte Carlo Tree Search. In *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, 649–654.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Driessche, G. v. d.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529: 484–489.

Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362(6419): 1140–1144.

Thakkar, A.; Kogej, T.; Reymond, J.-L.; Engkvist, O.; and Bjerrum, E. J. 2020. Datasets and their influence on the development of computer assisted synthesis planning tools in the pharmaceutical domain. *Chemical Science*, 11(1): 154–168.

Table 3: Benchmark

Mol	SMILES
C1	COc4ccc3nc(NC(=O)CSc2nnc(c1ccccc1C)[nH]2)sc3c4
C2	OC8Cc7c(O)c(C2C(O)C(c1ccc(O)c(O)c1)Oc3cc(O)ccc23)c(O)c(C5C(O)C(c4ccc(O)c(O)c4)Oc6cc(O)ccc56)c7OC8c9ccc(O)c(O)c9
C3	NC(=O)Nc1nsnc1C(=O)Nc2ccccc2
C4	C=CCn5c(=O)[nH]c(=O)c(=C4CC(c2ccc1OCOc1c2)N(c3ccccc3)N4)c5=O
C5	Oc1c(Cl)cc(Cl)cc1CNc2cccc3cn[nH]c23
C6	CC(C)C(C)C=CC(C)C1CCC3C1(C)CCC4C2(C)CCC(O)CC25CCC34OO5
C7	CC45CC(O)C1C(CC=C2CC3(CCC12)OCCO3)C4CCC56OCCO6
C8	CCc2ccc(c1ccccc1)cc2
C9	CC5C4C(CC3C2CC=C1CC(OC(C)=O)C(O)C(O)C1(C)C2CC(O)C34C)OC56CCC(=C)CO6
C10	CSc2ncnc3cn(C1OC(CO)C(O)C1O)nc23
C11	CCc1c(C)c2cc5nc(nc4[nH]c(cc3nc(cc1[nH]2)C(=O)C3(C)CC)c(CCC(=O)OC)c4C)C(C)(O)C5(O)CCC(=O)OC
C12	CN(COC(C)=O)c1nc(N(C)COC(C)=O)nc(N(C)COC(C)=O)n1
C13	CSc2ccc(OCC(=O)Nc1ccc(C(C)C)cc1)cc2
C14	Cc2ccc(C(=O)Nc1ccccc1)cc2
C15	CC5CC(C)C(O)(CC4CC3OC2(CCC1(OC(C=CCCC(O)=O)CC=C1)O2)C(C)CC3O4)OC5C(Br)=C
C16	COc8ccc(C27C(CC1C5C(CC=C1C2c3cc(OC)ccc3O)C(=O)N(c4cccc(C(O)=O)c4)C5=O)C(=O)N(Nc6ccc(Cl)cc6Cl)C7=O)cc8
C17	CC=CC(O)CC=CCC(C)C(O)CC(=O)NCC(O)C(C)C(=O)NCCCC2OC1(CCCC(CCC(CC=C(C)C(C)O)O1)CCC2C
C18	CCC(C)=CC(=O)OC1C(C)CC3OC1(O)C(O)C2(C)CCC(O2)C(C)(C)C=CC(C)C3=O
C19	CCC(CO)NC(=O)c2cccc(S(=O)(=O)N1CCCCCC1)c2
C20	CCCCCC1OC(=O)CCCCCCCCC=CC1=O
C21	COc1ccc(Cl)cc1
C22	CC(C)(C)C(Br)C(=O)NC(C)(C)C1CCC(C)(NC(=O)C(Br)C(C)(C)C)CC1
C23	COc2cc(CNc1ccccc1)ccc2OCC(=O)Nc3ccc(Cl)cc3
C24	COC4C=C(C)CC(C=CC=CC#CC1CC1Cl)OC(=O)CC3(O)CC(OC2OC(C)C(O)C(C)(O)C2OC)C(C)C(O3)C4C
C25	CCc2ccc(OC(=O)c1ccccc1Cl)cc2
C26	COc1ccccc1c2ccccc2
C27	CCCC(NC(=O)C1CC2CN1C(=O)C(C(C)(C)C)NC(=O)Cc3cccc(OCCCO2)c3)C(=O)C(=O)NCC(=O)NC(C(O)=O)c4ccc(NS(N)(=O)=O)cc4
C28	COC4C(O)C(C)OC(OCC3C=CC=CC(=O)C(C)CC(C)C(OC2OC(C)CC1(OC(=O)OC1C)C2O)C(C)C=CC(=O)OC3C)C4OC
C29	CC(C)(C)c4ccc(C(=O)Nc3nc2C(CC(=O)NCC#C)C1(C)CCC(O)C(C)(CO)C1Cc2s3)cc4
C30	CCC7(C4OC(C3OC2(COC(c1ccc(OC)cc1)O2)C(C)CC3C)CC4C)CCC(C6(C)CCC5(CC(OCC=C)C(C)C(C(C)C(OC)C(C)C(O)=O)O5)O6)O7
C31	O=C(OCc1ccccc1)c2ccccc2

Continued on next page

Table 3: Benchmark (Continued)

C32	CC(C)CC(NC(=O)C(CC(=O)NC2OC(CO)C (OC1OC(CO)C(O)C(O)C1NC(C)=O)C(O)C2 NC(C)=O)NC(=O)c3ccccc3)C(=O)NC(C(C)O) C(N)=O
C33	CCCC5OC(=O)C(C)C(=O)C(C)C(OC1OC(C) CC(N(C)C)C1O)C(C)(OCC=Cc3cnc2ccc(OC) cc2c3)CC(C)C4=NCCN6C(C4C)C5(C)OC6=O
C34	COC(=O)c1ccccc1NC (=O)CC(c2ccccc2)c3ccccc3
C35	Cc4onc5c1ncc(Cl)cc1n(C3CCCC (CNC(=O)OCc2ccccc2)C3)c(=O)c45
C36	CC(C)OCCCNC(=O)c3cc2c (=O)n(C)c1ccccc1c2n3C
C37	COC(=O)N4CCCC(N3CCC(n1c(=O)n(S(C) (=O)=O)c2ccccc12)CC3)C4
C38	Cc5c(C=NN3C(=O)C2C1CC(C=C1)C2C3=O) c4ccccc4n5Cc6ccc(N(=O)=O)cc6
C39	CCC5OC(=O)C(C)C(=O)C(C)C(OC1OC(C)CC (N(C)C)C1O)C(C)(OCC#Cc4cc(c3ccc2ccccc2 n3)no4)CC(C)C(=O)C(C)C6NC(=O)OC56C
C40	CC(=O)Nc1ccccc1NC(=O)COc2ccccc2
A0	COC(=O)c1cccc2c(C(=O) OC(C)C)c(nn12)c3cccc(Cl)c3
A1	CCOC(=O)C1=C(C)N(C)C(=O)NC1c2ccncc2
A2	Cn1c(nc2ccccc12)c3ncc (cc3N4CCCC4=O)c5ccccc5
A3	CC1(COc2ccccc2)CCN1C(=O)c3ccccc3
A4	Cc1cc(nn1CC(=O)N2CCC(CC2)c3nc(cs3) C4=NOC(C4)c5ccccc5OCC#C)C(F)(F)F
A5	CCC1(CC1)c2ccc(C)cc2CN(C3CC3) C(=O)c4c(F)n(C)nc4C(F)F
A6	CC1CCC(CN1C(=O)c2ccnc(NS(=O)(=O) c3cccc(Cl)c3)n2)c4cncc(c4)c5cnn(c5)C(=O)C
A7	Cc1cccc(C)c1c2csc(n2)C(=O) NCCC3=CC(=CC(=O)N3)Oc4ccccc4Cl
A8	COc1ccc(cc1)N2CC(CC2C(=O)NCc3 ccc4CCCCc4n3)NCC(F)(F)F
A9	FC(F)(F)Oc1cc(Cl)cc(c1)n2cnc3 ccc(cc23)S(=O)(=O)NC4COC4
A10	COc1cc2c3CC(NC(=O)C)C(Oc3ccc2 cc1C#N)c4ccc(F)c(F)c4
A11	FC(F)(F)c1ccc(Nc2ncc(C(=O)NCC3CCC (F)(F)CC3)c(n2)C(F)(F)F)c(Cl)c1
A12	Fc1cnc(Nc2ccc(cc2)C(=O)N3CCN(CC3) C4COC4)nc1c5cnc(n5C6CCCCC6)C(F)(F)F
A13	Cc1ccc(C2=NC(O)C(=O) Nc3cc(C)c(Cl)cc23)c(Cl)c1

Continued on next page

Table 3: Benchmark (Continued)

A14	CC1(C)CN(C(C(=O)NC2CCCCC2) c3cccc(c3)C(F)(F)F)C(=O)C1
A15	CC(=O)Nc1ccc(cc1)S(=O)(=O) c2ccc(cc2)C3CCN(C3)c4ccccc4
A16	Cc1cnc(C(=O)O)c(OC(F)F)c1
A17	FC(F)(F)c1nnc2CNCCn12
A18	CNCCC(Oc1cccc2ncncc12)c3cncs3
A19	FC(F)(F)c1nnc2CN(CCn12)c3cccc(I)c3