Using Monte Carlo Tree Search With A Branch & Bound Algorithm To Find A Minimum Directed Feedback Vertex Set

JD O'Hea
Department of Advanced Computing Sciences
Maastricht University
Maastricht, The Netherlands

***Abstract*—A new algorithm which uses Monte Carlo Tree Search (MCTS) with a Branch & Bound (B&B) algorithm to solve the Minimum Directed Feedback Vertex Set (MDFVS) problem, on a directed graph, is proposed.**

The directed feedback vertex set (DFVS) problem asks to construct a set of vertices such that the deletion of that set of vertices from its graph, makes the graph acyclic. The MDFVS problem asks to find the smallest DFVS in a graph.

MCTS provides information when used at the root-node such as finding an initial set of cycles in a graph which is then used in the lower-bounding mechanism within the B&B algorithm, and an initial upper bound. An (Integer) Linear program was used for generating lower bounds. Running MCTS at the root-node improved the solving time of a B&B algorithm by 43%. Further parameters and implementations are proposed which may further improve the overall performance of the algorithm.

***Index Terms*—Monte Carlo Tree Search, Branch & Bound, Minimum Directed Feedback Vertex Set**

I. Introduction

A directed feedback vertex set (DFVS) of G, where G is a directed graph (as shown in Figure 1 (a)), is a set of vertices in G such that every directed cycle in G contains at least one vertex in the DFVS. Equivalently, the deletion of each vertex in the DFVS from G leaves a directed acyclic graph [1]. The Minimum DFVS (MDFVS) is the smallest DFVS that can be found in G. Figure 1 (b) shows 3 deleted vertices (marked with red stars), which would form a DFVS for the graph in Figure 1 (a), but this is not a MDFVS. In Figure 1 (c) we see a MDFVS of size 2.

The MDFVS problem, which asks for such a MDFVS to be found, is an NP-Hard problem [2]. This means the MDFVS problem is at least as hard as the hardest problems in NP, and if there is a polynomial-time algorithm that can solve the MDFVS problem, there would be a polynomial-time algorithm for all problems in NP as all problems in NP can be polynomial-time reduced to the MDFVS problem.

The MDFVS problem has a range of applications in areas such as chip design [3], circuit testing, database and operating systems - specifically with deadlock recovery [1]. In systems requiring deadlock recovery, the resources and processes of a system can be modelled as a directed graph, the system resource-allocation graph. A cycle in the system resource-allocation graph is a deadlock and in order to resolve this deadlock a DFVS must be found. Removing vertices from the graph is equivalent to killing running processes in the system. It is desirable to kill a minimal number of processes that have been allocated resources, hence the desire of finding a MDFVS.

For these and other reasons it was the problem of choice for this years Parameterized Algorithms and Computational Experiments (PACE) competition [4].

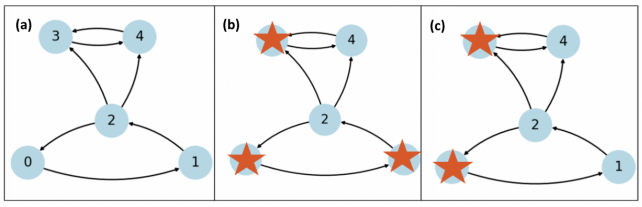

Fig. 1: (a) is a cyclic graph. (b) is a graph with a DFVS marked. (c) is a graph with a MDFVS marked.

The aim of this paper is to explore ways to marry the Branch & Bound (B&B) paradigm with Monte Carlo Tree Search (MCTS). In Section II different tools that are used to

Thanks to my supervisor Steven, I must say your help and guidance knows no bound. Thanks to my parents, for making me the strong component I am today. To all my friends and family, for keeping me on a directed path. And to Caoimhe, whom without I would be stuck in a cycle without a DFVS.

This thesis was prepared in partial fulfilment of the requirements for the Degree of Bachelor of Science in Data Science and Artificial Intelligence, Maastricht University. Supervisor(s): [Dr. Steven Kelk]

create such an algorithm are introduced in the context of the MDFVS problem. In Section III two algorithms are formally introduced and defined, incorporating the preliminary techniques. Algorithm 1 is the baseline B&B algorithm before integrating any MCTS techniques. Algorithm 2 then includes MCTS, with which the experiments for this paper were ran.

This work is inspired by the "Learning to branch" movement [5] in which machine learning techniques were adapted to learn branching strategies for a given application domain, among other work that uses machine learning techniques with B&B for solving combinatorial problems such as the MDFVS problem. This paper argues that there is a place for MCTS in the intersection of the machine learning and combinatorics fields.

This paper empirically analyses Algorithm 2 and provides a baseline for what is possible. MCTS with B&B is extremely flexible and there are many more ways the algorithm can be enhanced to yield possible further runtime improvements. These possibilities are outlined in Section III-A.

The details of experimentation using the PACE competition instances are outlined in Section IV. The runtime improvements that MCTS achieved are reported in Section V and the importance of those results are analysed in Section VI.

II. Preliminaries

A. Cycle Detection

Detecting a cycle in a strongly connected graph can be achieved in linear time ($O(V+E)$) using a depth first search (DFS) algorithm. The DFS works as follows.

For each vertex that has not been visited, begin DFS while keeping track of each vertex visited along a path using a stack. If we visit a vertex that is already in the stack, we have found a cycle. The stack consists of each vertex in the discovered cycle.

B. Branch & Bound (B&B)

B&B is an algorithm design technique used to explore candidate solutions in a tree search manner. The root of the tree contains the full graph G, while its DFVS is the empty set. Candidate vertices to branch on are found by searching for a cycle in G. The set of vertices C in a detected cycle are the set of candidates to branch on. A branching is completed by deleting a vertex v from G and adding it to the $DFVS$.

Before branching in B&B we first check the upper bound and an estimated lower bound to see if we can prune the search tree. This bounding process works as follows.

The upper bound gets updated to be the $DFVS$ of a candidate tree-node when a new $DFVS$ of a smaller size to the current upper bound is found. At any given tree-node of the search tree we can prune the search if the current depth of the tree (equivalently the number of vertices that have been added to the $DFVS$) plus the lower bound (a lower bound on the number of vertices that need to be deleted a given tree-node to get a DFVS) is greater than the size of the current upper bound as there is no potential to improve the upper bound by branching at this tree-node.

Initially the upper bound is set to be G, all of the vertices in the graph, as a $DFVS$ equal to G would leave a graph with no vertices, which is trivially an acyclic graph and the largest DFVS of G. Thus G is a DFVS, and a valid upper bound.

A lower bound can be derived from a set of known vertex-disjoint cycles in G at a given point in the search tree. A vertex of a DFVS intersects at most one such cycle, so a set of k vertex-disjoint cycles constitutes a certificate that a MDFVS has size at least k [6].

C. Monte Carlo Tree Search (MCTS)

MCTS is used to search a problem space; usually as a simulation based algorithm for multiplayer games. It can be adapted to tackle the MDFVS problem by formulating it as a single player game [7]. Similar to the B&B formulation, at the root-node of the search tree is the empty DFVS. Each action is undertaken by deleting a vertex from the graph and adding it to the DFVS. MCTS typically finds a reasonable solution to a problem by performing many simulations.

As described in [8], MCTS consists of 4 phases:

- **Selection:** When traversing a path to a terminal tree-node, the child tree-node is selected from the pool of children using a predefined selection policy. For the purpose of this research Upper Bound Confidence Applied to Trees (UCT) is used as the selection policy [8].
- **Expansion:** At a leaf-node of the search tree, if it is not terminal, select a random candidate action to get to a new state. Repeat until a terminal tree-node is reached.
- **Simulation:** At each tree-node the state of the graph needs to be simulated. A simulation in the MDFVS problem, for a given tree-node, is deleting the vertices from G that have already been selected for the DFVS earlier in the tree, then checking G for cycles. The set of vertices that are returned from the cycle detection in the simulation are the candidate actions that are for generating the child tree-nodes. If the cycle detection returns the empty set, then the DFVS is valid and the search has reached a terminal tree-node.
- **Backpropagation:** At the terminal tree-node, evaluate the state using a reward function and propagate this back along the solution path all the way to the root-node. The propagation consists of aggregating the total rewards achieved at the terminal to the tree-node's total rewards and incrementing the number of visits of each tree-node on the path.

Equation 1 is a good choice for a reward function as the reward will be in the range $[0, 1]$ which simplifies the tuning of parameter c [9] in Equation 2. Equation 1 meets another requirement for the reward function in MCTS as better terminal states must yield a higher reward - the smaller the size of the DFVS, the higher the reward.

$$Reward = 1 - \frac{|DFVS|}{|G|} \tag{1}$$

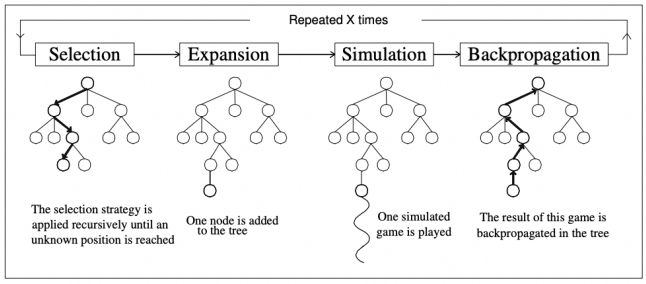

Fig. 2: Phases of MCTS, from [12].

UCT guarantees that the search converges to the optimal solution as the number of samples goes to infinity [10]. The UCT formula for selecting the next tree-node to search is shown in Equation 2. It takes the current tree-node N and an exploration weight c as parameters.

In the equation, $\max_{n \in N}$ means the maximum value the function evaluates of all the child-nodes (n) of N. $\frac{n.rewards}{n.visits}$ represents the average reward from searching at tree-node n, $n.rewards$ being the aggregated rewards achieved when searching at that tree-node and $n.visits$ is the total number of times that tree-node has been explored, as shown in Algorithm 2 in the next section. c is the exploration weight and is typically set to be $\sqrt{2}$, as this ensures that non-optimal tree-nodes get played much less than optimal ones if rewards are in the range $[0, 1]$ [11] .

MCTS can be easily adapted to many problems. We must only be able to simulate a state from an action, determine and evaluate terminal states, and generate further possible actions from the current state. For combinatorial problems such as the MDFVS problem it is straightforward to implement. Another advantage of MCTS is it can easily be built into the recursive framework of a B&B algorithm. This means that both MCTS and B&B are working on the the same search tree.

$$UCT(N, c) = \max_{n \in N} \left(\frac{n.rewards}{n.visits} + c * \sqrt{\frac{\log N.visits}{n.visits}} \right) \quad (2)$$

D. Linear Programming

Linear programming is used to solve problems with the following formulation; minimise Z such that $Z = c^T x$, $Ax \geq b$, and $x \geq 0$. For any given set of cycles C from G (not necessarily disjoint), we can write an (I)LP to find a minimum size set of vertices which intersects each cycle in C at least once. This is then a lower bound on the size of a MDFVS in G.

Each variable x_j corresponds to a vertex in G. x_j is binary. A value of 1 for vertex x_j means it is included in the MDFVS, and a value 0 meaning the opposite. c^T is a vector of 1's as each vertex is of equal weighting when trying to minimise Z, meaning to minimise the number of vertices in the DFVS. The matrix A contains a row for each known cycle. For each entry a_{ij} in A, $a_{ij} = 1$ for every vertex j that is in cycle i, else $a_{ij} = 0$.

Having x_j as binary makes the formulation an Integer Linear Program (ILP). x_j can also be relaxed to be $0 \leq x_j \leq 1$ instead of binary, which makes this formulation a standard Linear Program (LP). LPs can be solved quickly in practice using standard solvers but ILP solvers typically require much more time due to the integrality constraints (ILP is NP-hard). The trade-off here is that the ILP will give a stronger bound compared to an LP, albeit at a slower rate.

E. Strongly Connected Component (SCC)

An SCC is a maximally connected sub-graph. Each vertex in the sub-graph is reachable from every other vertex. SCC's of a graph G are vertex disjoint. Figure 3 shows a graph with it's 2 SCC's circled with red dotted lines. These two SCC's happen to also be the graph's only two disjoint cycles.

Solving the MDFVS problem for each SCC of G and taking the union of the MDFVS's of each SCC is equivalent to solving the MDFVS problem for G.

A linear time algorithm, known as Kosaraju's Algorithm, was proposed in [13] which finds all SCCs of a graph. By solving the SCCs of G instead of solving G as a single component reduces the branching factor of the search tree, thus reducing overall solving time.

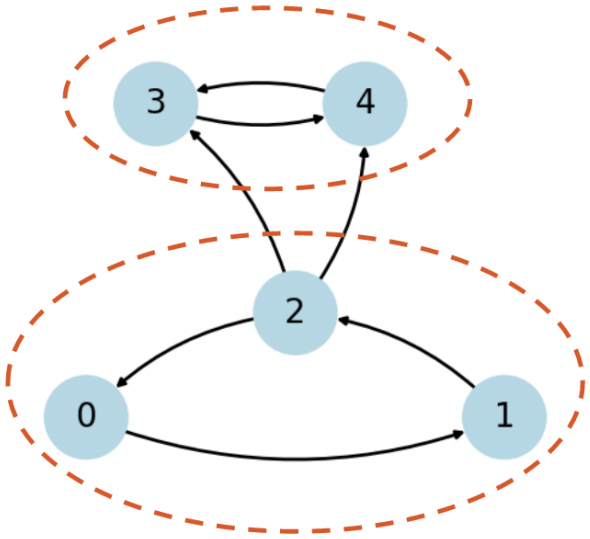

Fig. 3: A graph with it's SCC's highlighted with red dotted lines.

F. Graph Preprocessing

G can be reduced using the following reduction rules introduced in [14]:

1) Any vertex with in-degree 0 can never be in the MDFVS (IN0).
2) Any vertex with out-degree 0 can never be in the MDFVS (OUT0).

3) Any vertex v with in-degree 1, having the unique incoming edge (u, v), v can be deleted and all edges (v, x_i) can be reformatted to be (u, x_i) (IN1).
4) Any vertex v with out-degree 1, having the unique outgoing edge (v, u), v can be deleted and all edges (x_i, v) can be reformatted to be (x_i, u) (OUT1).
5) Any vertex v that has an incident edge (v, v) can be immediately added to the MDFVS as this cycle is a self-loop and the only vertex that can hit the cycle is v itself, thus should be deleted from G (LOOP).

Point 1 is true because such a vertex could never be on a cycle as a cycle needs incoming edges. An equivalent reasoning applies for Point 2 but in the opposite direction.

Point 3 is true because vertex u will be an in all the cycles v would be in. This would mean both u and v would never be in the same MDFVS. An equivalent reasoning applies for Point 4 but in the opposite direction.

Point 5 is true trivially because there is a cycle containing only vertex v thus v has to be included in the MDFVS in order to make G acyclic.

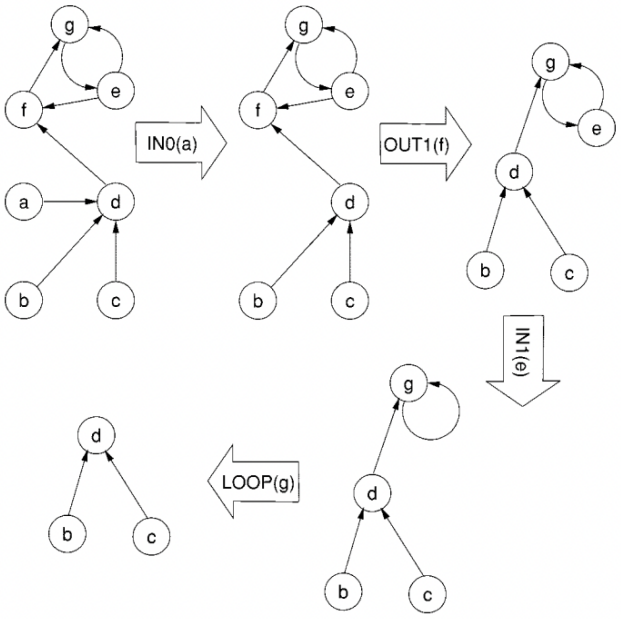

Fig. 4: Demonstration of preproccessing operations, from [14].

Figure 4 demonstrates the 5 rules in action. For points 1, 2, 3, and 4, their respective methods take vertices, deletes them from the graph and discards them as they will not be required to find an MDFVS. Point 5's method, LOOP, deletes the vertex from the graph and stores it for after the search is complete as this vertex has to be in the final MDFVS.

III. Algorithms

The $BOUND\_SOLVER$ parameter is required as input for Algorithm 1. This parameter specifies whether to use an ILP or LP for the lower-bounding method BOUND as seen in Line 7, Algorithm 1. The bounding uses (integer) linear programming as described in Section II-D.

The preprocessing in Line 1, Algorithm 1, undertakes the reductions described in Section II-F.

The $MDFVS$ variable in the context of the algorithms is used to track the current best solution found, and gets updated as better solutions are found throughout the search, eventually terminating with a correct MDFVS.

The CYCLE method called in Algorithm 1 Line 23 takes a graph as input and returns a set of vertices contained within a cycle of the input, as described in Section II-A. Each cycle found should not only be used as a set of candidate children (i.e. vertices which we should branch on) for the given tree-node, but also stored in memory to be used by the BOUND method.

In Algorithm 2, the amount of time to run MCTS must be specified with the $MCTS\_TIME$ parameter. This algorithm reuses elements from Algorithm 1. Algorithm 2 is a formulation for what was introduced in Section II-C

There is no reference to SCCs in the algorithms described in this section as splitting G into SCCs prior to initiating the algorithm is trivial. The SCCs can be solved individually and the results of each can be combined. This is equivalent to solving G entirely. It would have also been possible to continuously split G into SCC's at each tree-node after the vertices in the DFVS have been deleted. This would have reduced the branching factor of Algorithm 1. This was not implemented as part of this work due to lack of time.

A. Further Possible Parameters

Other parameters which were considered but not fully explored as part of this work include defining a parameter which controls the depth into the search tree of Algorithm 1 before the BOUND method is utilised. For example if the ratio of $\frac{|node.DFVS|}{|MDFVS|}$ (the ratio of the size of a DFVS at a given tree-node to the size of minimum known DFVS/the upper bound) is greater than x, s.t. $x \in [0, 1]$, then Algorithm 1 should begin bounding. A similar parameter can be implemented for a depth that stops using the BRANCH mechanism after a certain depth ratio. The reason a ratio is used instead of a fixed depth is so the parameter generalises for any sized graph.

When we are using the BRANCH mechanism it may not be optimal to employ it at every tree-node. Another possibility is to run it with a tuned probability p.

MCTS does not need to be independent to BRANCH as formulated in Algorithm 2. There could be a point in which it may be useful to relaunch the MCTS machinery within BRANCH. A possible parameter for deciding this is when the gap between the lower bound and upper bound is significant. This gap could be interpreted as having high potential for a good solution, or containing unexplored information which prevented the algorithm from finding a good enough bound to prune the search. The parameter m, s.t. $m \in [0, 1]$, if it is greater than $\frac{lower\_bound}{|MDFVS|}$; restart MCTS. In this case $lower\_bound$ would be the lower bound achieved at a tree-node and $|MDFVS|$ is the size of the best solution found throughout the search up to that point. This

heuristic generalises for any size graph. Implementing such a mechanism would also require tuning another parameter determining when MCTS is run within BRANCH, and for how long it should be run.

These parameters were not explored as part of this research because preliminary experiments showed that although they offer potential to improve the overall performance of an algorithm, they will not make a significant enough difference to rend the current running time feasible. It is left to future work to explore further performance improvement potential.

Prioritising tree-nodes to branch on based on a given heuristic is another possible enhancement of Algorithm 1. For example prioritising the tree-nodes with the highest average reward ($\frac{n.rewards}{n.visits}$). Another branching technique would be prioritising tree-nodes that would break a graph into a disconnected graph. The new components would then be solved separately reducing the branching factor, and thus the solving time of the overall search space.

Algorithm 1 B&B Algorithm

Require: $G : G$ is a directed, strongly connected graph
Require: $BOUND\_SOLVER \in \{ILP, LP\}$
PREPROCESS(G)
$MDFVS \leftarrow G$
$root\_node \leftarrow$ NODE($\{\}$)
BRANCH($root\_node$)

procedure BRANCH($node$)
  $lower\_bound \leftarrow$ BOUND($node, BOUND\_SOLVER$)
  if $lower\_bound > |MDFVS|$ **then**
    return
  else if $node$ **is** terminal **then**
    $MDFVS \leftarrow node.DFVS$
    return
  end if
  while $node$ **has** $candidates$ **do**
    INITIALISE_CANDIDATE($node$)
  end while
  for n **in** $node.children$ **do**
    BRANCH(n)
  end for
end procedure

procedure NODE($DFVS$)
  $this.candidates \leftarrow$ CYCLE($G \cap DFVS^c$)
  $this.DFVS \leftarrow DFVS$
end procedure

procedure INITIALISE_CANDIDATE($node$)
  $v \leftarrow$ POP($node.candidates$)
  $n \leftarrow$ NODE($node.DFVS \cup v$)
  add n **to** $node.children$
  return n
end procedure

Algorithm 2 MCTS with B&B Algorithm 1

Require: $G : G$ is a directed, strongly connected graph
Require: $BOUND\_SOLVER \in \{ILP, LP\}$
Require: $MCTS\_TIME \in [0, T)$
PREPROCESS(G)
$MDFVS \leftarrow G$
$EXPLORATION \leftarrow \sqrt{2}$
$root\_node \leftarrow$ NODE($\{\}$)

while $time\_elapsed < MCTS\_TIME$ **do**
  MCTS_EXPAND($root\_node$)
end while

BRANCH($root\_node, \{\}$)

procedure MCTS_EXPAND($node$)
  if $node$ **is** terminal **then**
    $r \leftarrow 1 - (|node.DFVS|/|G|)$
  else if $node$ **does not have** $candidates$ **then**
    $n \leftarrow$ UCT($node, EXPLORATION$)
    $r \leftarrow$ EXPAND(n)
  else if $node$ **has** $candidates$ **then**
    $n \leftarrow$INITIALISE_CANDIDATE($node$)
    $r \leftarrow$ PLAY_OUT(n)
  end if
  $node.rewards \mathrel{+}= r$
  $node.visits \mathrel{+}= 1$
  return r
end procedure

procedure PLAY_OUT($node$)
  while $node$ **is not** terminal **do**
    $node \leftarrow$INITIALISE_CANDIDATE($node$)
  end while
  $r \leftarrow 1 - (|node.DFVS|/|G|)$
  $node.rewards \mathrel{+}= r$
  $node.visits \mathrel{+}= 1$
  return r
end procedure

IV. Experiments

Experiments are modelled on the (PACE) 2022 competition setup [4]. This year's competition involved solving the MDFVS problem.

Experiments were run on the 100 instances provided by PACE for their exact solver competition [15]. Each combination of the parameter values outlined in TABLE I were explored. The graphs were split into SCCs and the SCCs were each passed to the algorithms outlined in Section III and the final result was the union of the results from the SCCs.

PACE allows 30 minutes to solve an instance exactly, so this was the time allowed for each instance here. If no solution is returned, the algorithm is marked as having failed.

The algorithms were limited to 8GB of RAM. The algorithm described in section III was developed using Python 3.8. The open source software GNU Linear Programming Kit (GLPK) is used to solve the linear programs. The experiments were run on Google Cloud virtual machines using Google's N2 series with machine specifications that match the requirements for the PACE 2022 challenge as described.

Parameter	Values
$MCTS\_TIME$ (mins)	{ 0, 5, 10, 15}
$BOUND\_SOLVER$	{ LP, ILP }

TABLE I: Table of parameter values for experimentation

V. Results

		MCTS_TIME				
		0	1	2	5	10
Ranking	1st	2	47	21	4	2
	2nd	6	19	46	5	0
	3rd	12	4	9	45	6
	4th	16	6	0	22	32
	5th	40	0	0	0	36

TABLE II: Table of counts of rankings achieved for the given $MCTS\_TIME$ parameter on the SCC's

$MCTS\_TIME$ i	Mean time	Mean time 0 - Mean time i
0	45.98	0
1	28.27	17.71
2	27.88	18.10
5	31.61	14.37
10	39.40	6.58
Best	26.11	19.87

TABLE III: Table of mean times (s) by MCTS_TIME

The algorithm was only able to solve 1 graph, graph "e_001", within the 30 minute time frame. This graph is the smallest graph in the set with 512 vertices and 613 edges, compared to the next largest graph of 1024 vertices and 6055 edges. The |MDFVS| is 2 for this graph, where as for every other graph this is significantly larger.

In order to compare whether or not MCTS improved the B&B algorithm, graphs that are solvable in a reasonable time by Algorithm 1 are required. To do this the PACE instances were split into their SCCs and ran on Algorithm 1 (equivalent to Algorithm 2 with $MCTS\_TIME$ set to 0). The SCCs that ran in under 150 seconds but more than 15 seconds were selected for further investigation. This resulted in 76 SCCs, having graph sizes in the range of 40 to 110 vertices, and MDFVS sizes in the range of 27 to 69.

Each SCC was solved 5 times for each of the following $MCTS\_TIME$ parameter values: {0, 1, 2, 5, 10}. $BOUND\_SOLVER$ was set to ILP. The result for a graph parameter pairing was taken to be the mean time of the 5 runs.

For each graph, the parameters can be ranked by the time the algorithm took to solve with a given parameter value. Table II shows the counts of how many times each parameter achieved a particular ranking. For example when the algorithm was run with with $MCTS\_TIME$ of 1, it was the fastest parameter for 47 of the 76 SCC's. $MCTS\_TIME$ 2 was fastest on 21 graphs. $MCTS\_TIME$ 0 was the slowest (came 5th) on 40 of the 76 SCC's.

The mean of the times for all graphs by parameter are shown in Table III in the "Mean time" column. The second column subtracts the "Mean time" column with the mean time of $MCTS\_TIME$ set to 0. This highlights the average time improvement achieved by having a value greater than 0 set for $MCTS\_TIME$. For the the 76 graphs the algorithm with $MCTS\_TIME$ set to 0 took a mean time of 45.98 seconds per graph, where as the same algorithm with $MCTS\_TIME$ set to 1 took a mean time of 28.27 seconds per graph. This is a time improvement of 17.71 seconds on average per graph.

VI. Discussion & Conclusion

The reason the algorithms proposed above were not able to solve the majority of the PACE instances is likely because the techniques proposed here alone are not enough to obtain a solution in a reasonable time. Algorithm 1 & 2 require more enhancements, perhaps such as those described in Section III-A, or a deeper understanding of the graph's underlying structures to implement heuristics that would improve runtime.

The results in Section V shows that MCTS improves the overall algorithm solution time in all but 2 graphs. Table III shows that simply setting $MCTS\_TIME$ to 2 seconds for the given graphs, saw a 39% improvement on the runtime. When always taking the best value of $MCTS\_TIME$ for a given graph, this improvement increases to a 43% runtime improvement.

The optimal time for the given set of graphs is likely to be close to 1 or 2 seconds as the runtime gains begin to dwindle as $MCTS\_TIME$ is increased to 5 and 10 seconds because the time spent on MCTS starts to eat into the B&B solving time required to get the exact answer.

The reason MCTS improves the standalone B&B algorithm is because it gives the algorithm an initial upper bound, pruning the search tree before ever beginning the B&B search, and MCTS provides a initial set of cycles which contributes to lower-bounding, thus strengthening the pruning mechanism within B&B.

For this research, at every tree-node the lower bound is evaluated. There may be potential for runtime improvement by tuning a parameter or implementing a heuristic for when to use the pruning machinery. This was left to future work.

Although it is not surprising that finding an initial upper bound at the root-node improved the overall runtime of a baseline B&B algorithm, it is worth highlighting that MCTS is a flexible technique that can be adapted to tackle any problem that uses a tree search mechanism to find a solution, thus not needing domain based heuristics to find an upper bound. Also, as a side effect of MCTS we gain additional information about the instance, that is domain specific, in the MDFVS case in the form of cycles.

Here we used the cycles found in MCTS throughout the B&B algorithm to find a lower bound. Note however there are other formulations of exact solvers that MCTS can also be used to strengthen. For example in the exact ILP formulation for finding a MDFVS as described in [16], edge constraints are used to enforce a topological order. This formulation uses binary decision variables, the same as described in Section II-D. This formulation could also be strengthened by adding constraints using the cycles discovered with MCTS, to strengthen the lower bound and the LP-relaxation; and warm-starting the ILP with the initial upper bound it discovers. Preliminary experiments adding cycle constraints to the formulation show promising results for certain instances of the PACE exact dataset, quickly strengthening the bounds achieved. However in some cases the ILP formulation already becomes extremely large (due to many vertices and edges in the graph) and adding the cycles seems to slow down the already over-burdened ILP machinery more than it helps it.

The results show that running MCTS for a small fraction of the expected solving time of an instance is likely to improve the overall solving time of a B&B algorithm. However the underlying algorithm needs to be sufficient enough to solve the instances in a reasonable time before the gains from MCTS can be realised. It is possible there are more performance gains to be realised, especially strategically restarting the MCTS machinery within the B&B machinery when there is significant potential information to be gained from doing so, as described for the MDFVS problem in Section III-A when the gap between lower bound estimate and the upper bound is significant. Exploring the effect of restarting MCTS within B&B has on performance is left to future work.

Furthermore, it is possible to integrate MCTS with an existing high performing B&B algorithm or tree search algorithm at little cost with the potential for performance improvements. MCTS is an extremely flexible and adaptable technique. It shows high potential in helping improve solving times of exact solvers similar to those described in this work.

References

[1] Jianer Chen et al. "A fixed-parameter algorithm for the directed feedback vertex set problem". In: *Proceedings of the fortieth annual ACM symposium on Theory of computing*. 2008, pp. 177–186.

[2] Richard M Karp. "Reducibility among combinatorial problems". In: *Complexity of computer computations*. Springer, 1972, pp. 85–103.

[3] Paola Festa, Panos M Pardalos, and Mauricio GC Resende. "Feedback set problems". In: *Handbook of combinatorial optimization*. Springer, 1999, pp. 209–258.

[4] *Pace 2022 Challenge Description*. URL: https://pacechallenge.org/2022/.

[5] Maria-Florina Balcan et al. "Learning to branch". In: *International conference on machine learning*. PMLR. 2018, pp. 344–353.

[6] Guoli Ding, Zhenzhen Xu, and Wenan Zang. "Packing cycles in graphs, II". In: *Journal of Combinatorial Theory, Series B* 87.2 (2003), pp. 244–253.

[7] Maarten PD Schadd et al. "Single-player Monte-Carlo tree search for SameGame". In: *Knowledge-Based Systems* 34 (2012), pp. 3–11.

[8] Jacek Mańdziuk. "MCTS/UCT in solving real-life problems". In: Jan. 2018, pp. 277–292. ISBN: 978-3-319-67945-7. DOI: 10.1007/978-3-319-67946-4_11.

[9] Tobias Joppen and Johannes Fürnkranz. "Ordinal Monte Carlo Tree Search". In: *Monte Carlo Search International Workshop*. Springer. 2020, pp. 39–55.

[10] Levente Kocsis and Csaba Szepesvári. "Bandit based monte-carlo planning". In: *European conference on machine learning*. Springer. 2006, pp. 282–293.

[11] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. "Finite-time analysis of the multiarmed bandit problem". In: *Machine learning* 47.2 (2002), pp. 235–256.

[12] Guillaume Maurice Jean-Bernard Chaslot Chaslot. *Monte-carlo tree search*. Vol. 24. Maastricht University, 2010.

[13] Micha Sharir. "A strong-connectivity algorithm and its applications in data flow analysis". In: *Computers & Mathematics with Applications* 7.1 (1981), pp. 67–72.

[14] Hen-Ming Lin and Jing-Yang Jou. "On computing the minimum feedback vertex set of a directed graph by contraction operations". In: *IEEE Transactions on computer-aided design of integrated circuits and systems* 19.3 (2000), pp. 295–307.

[15] *Pace 2022 Public Instances*. URL: https://github.com/PACE-challenge/pacechallenge.org/tree/master/2022.

[16] Leo Van Iersel et al. "A practical approximation algorithm for solving massive instances of hybridization number for binary and nonbinary trees". In: *BMC bioinformatics* 15.1 (2014), pp. 1–12.