Solving PBQP-Based Register Allocation using Deep Reinforcement Learning

Minsu Kim
Department of Electrical and Computer Engineering
Seoul National University
South Korea
minsu.kim@snu.ac.kr

Jeong-Keun Park
Department of Electrical and Computer Engineering
Seoul National University
South Korea
jk.park@snu.ac.kr

Soo-Mook Moon
Department of Electrical and Computer Engineering
Seoul National University
South Korea
smoon@snu.ac.kr

2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) | 978-1-6654-0584-3/22/$31.00 ©2022 IEEE | DOI: 10.1109/CGO53902.2022.9741272

***Abstract*—Irregularly structured registers are hard to abstract and allocate. *Partitioned Boolean quadratic programming* (PBQP) is a useful abstraction to represent complex register constraints, even those in highly irregular processors of *automated test equipment* (ATE) of DRAM memory chips. The PBQP problem is NP-hard, requiring a heuristic solution. If no spill is allowed as in ATE, however, we have to enumerate more to find a solution rather than to approximate, since a spill means a total compilation failure. We propose solving the PBQP problem with *deep reinforcement learning* (Deep-RL), more specifically, a model-based approach using Monte Carlo tree search and deep neural network as used in *Alphazero*, a proven Deep-RL technology. Through elaborate training with random PBQP graphs, our Deep-RL solver could cut the search space sharply, making an enumeration-based solution more affordable. Furthermore, by employing backtracking with a proper coloring order, Deep-RL can find a solution with modestly-trained neural networks with even less search space. Our experiments show that Deep-RL can successfully find a solution for 10 product-level ATE programs while searching much fewer (e.g., 1/3,500) states than the previous PBQP enumeration solver. Also, when applied to C programs in llvm-test-suite for regular CPUs, it achieves a competitive performance to the existing PBQP register allocator in LLVM.**

I. Introduction

Register allocation has been studied extensively for decades due to its importance for quality code generation. The most popular abstraction is the *interference graph*, where vertices correspond to live ranges and edges represent the interference relationship between vertices. Then, the register allocation problem is mapped to the graph coloring problem where colors correspond to physical registers in the processor. A graph is called m-colorable if we can assign every vertex to one of m colors such that no two interfering vertices are assigned the same color. If the graph is not m-colorable, some vertices should be spilled to memory, so a register allocator aims to avoid any spills, but in a case when a spill is unavoidable, its goal is to minimize the total spill cost. This is an NP-hard combinatorial optimization problem with an intractable search space [1], so many heuristic algorithms have been proposed [2]–[4]. For compilers that are sensitive to compilation time such as just-in-time compilation, a greedy approach called the linear scan algorithm [5], [6] is often preferred. Indeed, GRA (greedy register allocator), the default register allocator of the LLVM, is based on the extended liner scan algorithm with aggressive live range splitting [7]–[9].

Meanwhile, it has been hard to abstract and allocate *irregularly structured registers*, from simple ones in x86 CPUs to those complex ones in *automated test equipment* (ATE), a special embedded system to test DRAM memory chips. For example, ATE requires highly irregular *register pairing* such that we can add two registers A and B but cannot add registers A and C. Also, an ATE often has multiple, interleaved processor units such that an ATE with eight processors, for example, provides 8-way interleaving, so a bundle of eight instructions constitutes a *major cycle*. Within a major cycle, a register cannot be written more than once and cannot be read ahead of a write. Finally, there is no data memory in ATE, making register spills impossible. This requires a more elaborate abstraction and allocation technique than the conventional ones such as graph coloring because an allocation failure means a total compilation failure here. On the other hand, there is a practical compilation problem in ATE, which requires high-quality irregular register allocation (see Section II).

There are many irregular register allocation techniques [10]–[14], but the most elaborate one that can model highly irregular register constraints such as those in ATE is *partitioned Boolean quadratic programming* (PBQP) [15], [16]. It is defined over a PBQP graph, where a vertex contains an $m \times 1$ cost vector and an edge contains an $m \times m$ cost matrix. An element of the cost vector or matrix is either a real number or infinity. Then, the goal is to find a coloring (selection) for each vertex, which *minimizes* the sum of cost elements determined by the selections.A graph coloring problem is a special case of a PBQP problem, where every cost vector is a zero vector and every cost matrix is a diagonal matrix where all diagonal elements are infinities. That is, PBQP provides a more fine-grained abstraction than graph coloring, suitable for handling complex register constraints. PBQP often outperforms other approaches in the domain of irregular register allocation [9], which is the reason that LLVM adopted a PBQP-based register allocator as an option [8].

As to the PBQP solvers, the original solver handles low-degree, easy vertices with some kind of full enumeration, but handles hard vertices with approximation [15]. This works

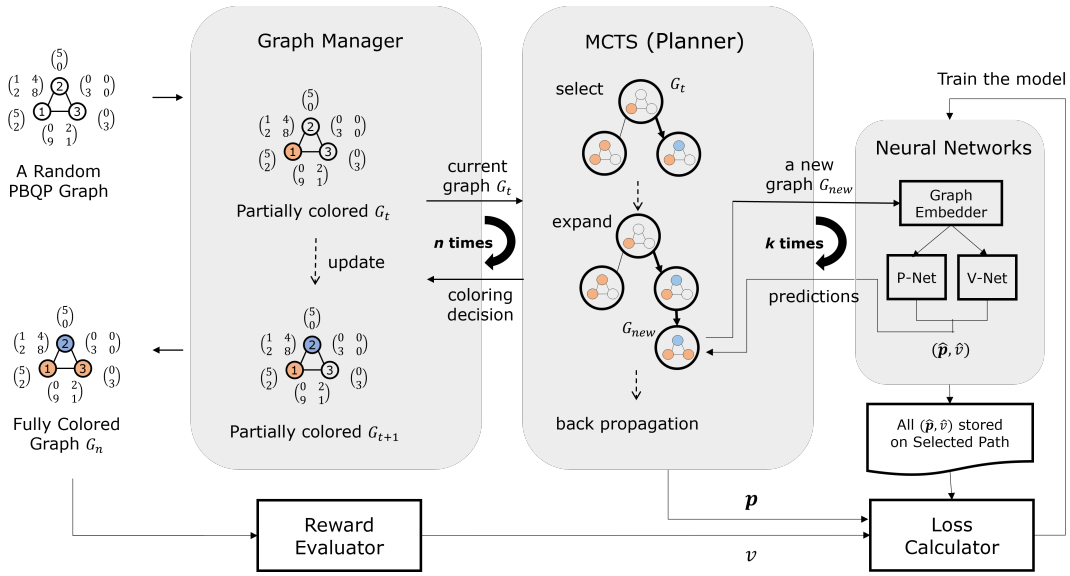


Fig. 1. A self-play of PBQP game.

relatively well for regular architectures. For highly irregular architectures such as ATE that allow no spills, however, there rarely exist easy vertices, so it mostly approximates, thus likely to fail even if a solution exists. A different solver that performs many enumerations has been proposed, which first allocates hard vertices that have fewer choices of allocatable registers (i.e., less *liberty*) with full enumeration, then allocates easy ones with approximation [17]. This liberty-based enumeration was effective, leading to successful ATE register allocation with a reasonable amount of time. However, the search time, in theory, can be exponential, since we need to explore a huge search space if there are many hard vertices as in ATE, so we need an intelligent way of narrowing down the search space. The current wisdom for reducing the search space is machine learning (ML), and we propose solving PBQP using ML, more specifically, *deep reinforcement learning* (Deep-RL).

Deep-RL has recently enjoyed remarkable success in solving intractably complex problems [18]–[22]. In general, there are two interacting elements in RL: *agent* and *environment*. An agent makes an *action* based on its *policy* for the current *state* that the environment has, then the environment transits to a new state and gives rise to a *reward*. The environment is often assumed to have a *Markov decision process*, a discrete-time stochastic process where decisions are made based only on the current state, regardless of the past. Then, the goal of RL is finding the best policy, a probabilistic distribution of available actions for a given state, which can maximize the cumulative rewards. There are many approaches to RL, but a deep-RL which is model-based with a planning method employing *Monte Carlo tree search* (MCTS), equipped with deep neural network (DNN), has shown superhuman performance in some intractable domains, such as the *AlphaZero* in Go games [20]; here, a state is a 19×19 board positioned with stones, an action is placing a stone on the board, and a reward is win/lose/tie. Before making an action each time, Deep-RL plans ahead with MCTS, which enumerates a search tree with the current state as a root. With complete knowledge of the environment model and with the help of DNN, MCTS can narrow down the search tree efficiently, producing a precise probabilistic distribution for the root state, which helps the agent to make the right action. The DNN should be trained in advance, so Deep-RL performs *self-plays* for random games, during which it collects the training data composed of DNN-predicted and the final reward values, as well as DNN-predicted and the MCTS-decided probability distributions, for each root state. The DNN is trained with a gradient descent optimization (see Section II).

This paper attempts to solve the PBQP problem using Deep-RL by formulating it as a single-player, turn-based game. At each turn, the player takes an action by selecting a color for a vertex in the PBQP graph. This makes the environment change the coloring state of the PBQP graph. Since the goal of PBQP is to minimize the cost sum decided by the selections, the reward is given at the end of the game based on how small the player managed the cost sum.

Figure 1 overviews a single self-play for a random PBQP graph with n vertices for training. To decide the next coloring, we perform MCTS with the current state (initially an uncolored PBQP graph) as a root node, which will expand the search tree by adding k nodes one by one, using the inferences from the DNN. After MCTS, we decide the color based on the probabilistic distribution for the root node. We repeat this for each of n vertices, during which we collect the training data from the MCTS process and the final reward, and they will be used later to train the DNN as depicted at the bottom of the figure. When we perform a *real-play* of register allocation for a given program, we do the same process using a final, trained DNN, so the collection and training parts are omitted.

There are four issues for solving the PBQP game, which we handle as follows. The first one is how to decide the reward. Being as a single-player game, we compare the cost sum of the current player with that of the previous best player, such that if it is smaller/bigger/same, the reward is a win/lose/tie. This way, the PBQP problem can be interpreted as a competitive game, where the proven framework of AlphaZero can seamlessly apply. The second issue is how to represent the state and its transitions. Obviously, the initial state is an uncolored PBQP graph, and as a result of a coloring action, it transits to a new graph. Instead of using a partially colored graph, we remove the colored vertex while propagating the selected costs to its neighbors, creating an equivalent, uncolored graph. This can allow the DNN to learn more easily with fewer learnable parameters. Thirdly, we need precise and efficient embedding of a PBQP graph since it is not a typical input form for feed-forward neural networks. To express a graph to a fixed-size matrix that reflects its global structure as well as local interactions, we employ a message-passing algorithm called *graph convolutional network* [23]–[26]. Since our PBQP graph has cost vectors and cost matrices, we modify the message passing algorithm to embed the costs additionally. Finally and most importantly, we should color a given graph as efficiently as possible, even with smaller MCTS trees or less-trained DNNs. We employ *backtracking*, often missing in regular Deep-RL systems that make only *one-way* actions, on an as-needed basis. For ATE programs whose no-spill requirement leads to either zero or infinity cost sum, backtracking can complement Deep-RL to find a valid solution effectively.

As to evaluation, we trained the DNN with 20,000 random PBQP graphs. We first allocate registers for ATE programs. The experiments show that our Deep-RL PBQP solver could find a valid solution for all 10 product-level ATE programs, as the previous liberty-based enumeration solver (as a note, the original PBQP solver failed to find a solution for 9 programs). In terms of the search space size, a brute-force search would require m^n states to enumerate in the worst case, which could be reduced to a few tens of million states by liberty-based enumeration, yet to a few thousand states by Deep-RL with backtracking. Actually, backtracking can lower the required k values of MCTS than that of the original Deep-RL solver, reducing both the training time and the inference time substantially. We also implement Deep-RL on LLVM and experiment with 24 C programs for regular CPUs, where the cost values are provided by the PBQP module of LLVM. We achieved a result comparable to that of LLVM's PBQP.

This paper makes the following contributions.

- We apply deep reinforcement learning to register allocation by viewing the PBPQ problem as a single-player, turn-based game. To the best of our knowledge, this is the first work to solve register allocation using deep reinforcement learning, particularly for irregular registers.
- We solve a few issues such as formulation of colored states, PBQP graph embeddings, competition-based rewards, and backtracking-based coloring actions.
- Our proposed PBQP solver can cut the search space sharply, especially when combined with backtracking, making the enumeration-based solution more affordable, even with modestly-trained neural networks.
- We evaluate the proposed solution for irregular register allocation of real ATE programs as well as for C programs for regular CPUs on a modern optimizing compiler.

The rest of the paper is as follows. Section II briefly provides the background. Section III formulates deep reinforcement learning for PBQP. Section IV details the learning algorithms. Section V evaluates the proposed solutions. Section VI describes related work. A summary is in Section VII.

II. Background

In this section, we briefly review PBQP-based register allocation, ATE programs, and deep reinforcement learning.

A. PBQP-Based Register Allocation

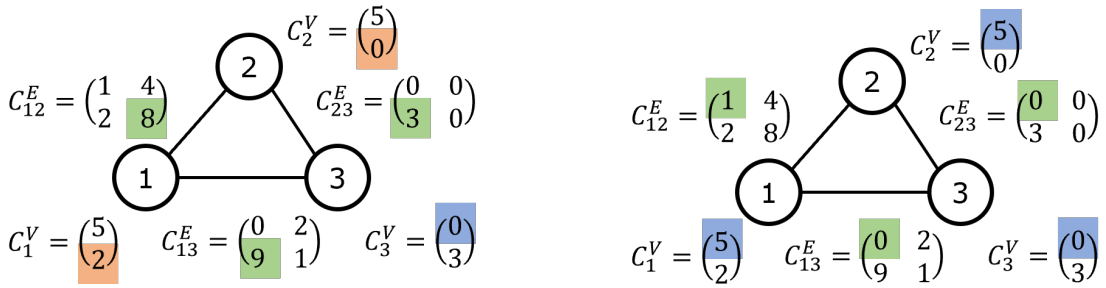
Scholz et al. proposed that register allocation can be mapped to a PBQP problem [15]. It is particularly useful for irregular architectures such as X86 where registers are not uniform with special constraints for register usages in addition to interferences. The problem is defined over a PBQP graph $G(V, E, C^V, C^E)$, an undirected graph with cost vectors C^V and cost matrices C^E, where there are m physical registers.

- $V := \{1, ..., n\}$: set of vertices numbered from 1 to $|V|$
- $E \in 2^{V \times V}$: set of edges
- $C^V \in {\mathbb{R}_\infty}^{n \times m}$: m-sized cost vector for each vertex
- $C^E \in {\mathbb{R}_\infty}^{n \times n \times m \times m}$: $m \times m$ cost matrix for each edge

The i^{th} entry of the cost vector of a vertex u, $C^V_u(i)$ means the cost when coloring the vertex with the color i, assuming the set of colors of $\{1, ..., m\}$. Similarly, the $(i, j)^{th}$ entry of the cost matrix of an edge $(u, v) \in V \times V$, $C^E_{uv}(i, j)$ means the cost when coloring vertex u with color i and vertex v with color j. Here, two vertices u and v are regarded as *disconnected* if and only if $C^E_{uv} = O$. Now, we introduce a *selection vector* $\mathbf{x}_u \in \{0, 1\}^m$ for each vertex u such that $\mathbf{x}^T_u \cdot \mathbf{1} = 1, \forall 1 \leq u \leq n$. In other words, each selection vector contains a single element of 1 while all others are 0, indicating that the vertex "selects" the corresponding physical register (i.e., coloring the vertex with the color i, if $\mathbf{x}_u(i) = 1$). To handle spill, we add one more selectable entry in C^V and C^E to express the spill cost. Then, a PBQP problem is defined as minimizing the *cost function* f, the sum of cost elements determined by the selections.

$$f = \sum_{1 \leq u \leq n} \mathbf{x}^T_u \cdot C^V_u + \sum_{1 \leq u < v \leq n} \mathbf{x}^T_u \cdot C^E_{uv} \cdot \mathbf{x}_v \qquad (1)$$

Figure 2 illustrates an example of a PBQP problem. There are three vertices, $\{1, 2, 3\}$, and we want to find the corresponding selection vectors $\vec{x}_1$, $\vec{x}_2$, and $\vec{x}_3$ that minimizes the cost sum of Equation 1. Figure 2a shows a set of selection vectors, $\vec{x}^T_1 = (0, 1)$, $\vec{x}^T_2 = (0, 1)$, and $\vec{x}^T_3 = (1, 0)$, which selects the highlighted costs, whose cost sum is $(2 + 0 + 0) + (8 + 9 + 5) = 24$. Figure 2b shows another set of selection vectors, $\vec{x}^T_1 = (1, 0)$, $\vec{x}^T_2 = (1, 0)$, and $\vec{x}^T_3 = (1, 0)$, whose cost sum is minimal, $(5 + 5 + 0) + (1 + 0 + 0) = 11$.



(a) A set of selections

(b) Another set of selections (optimal)

Fig. 2. An example of a PBQP graph with $n = 3$ and $m = 2$. Two arbitrary instances of selections are demonstrated.

Since a brute-force search for the optimal solution is intractable, Scholz et al. proposed a PBQP solver based on graph reduction [15], [16]. It repeatedly removes a vertex in increasing order of *degree* after propagating the costs to adjacent vertices until the graph is empty, then colors vertices in the reverse order. When the degree is low (≤ 2), the reduction produces an equivalent graph, but if the degree is high, it produces an approximate graph to lower complexity, which might lead to a fast but sub-optimal solution. For ATE programs, their PBQP graphs rarely have low-degree vertices, so this does not work properly.

Kim et al. proposed a different solver for ATEs [17]. It sorts the vertices in increasing order of *liberty*, the number of available colors (non-infinity costs in the cost vector), and colors vertices with a lower (≤ 4) liberty first by enumerating any colorable combinations for them with backtracking. In this way, they explore all possibilities for these hard vertices first. For easy vertices, they approximate as the original solver does.

B. PBQP Graphs for ATE Programs

ATE tests a DRAM chip by sending a bit vector to the pins of the chip every clock and decides pass/fail depending on its behavior. ATE is composed of processor units called ALPGs and irregularly-structured registers and is programmable. DRAM chipmakers often purchase ATEs from multiple manufacturers for a competitive price, and try to test a DRAM product using any ATE that can generate the bit vector with the same clock speed of the DRAM, for test productivity. One issue is that each ATE manufacturer provides its own programming language, mostly composed of assembly-like syntax using the instructions of the ALPG and accessing the registers directly. This makes programming not only complex, but not portable among the ATEs. Even different models of a manufacturer have a different number of ALPGs and registers, so they also have a portability issue.

One popular solution used by DRAM chipmakers is *translation*. For a given DRAM chip, test engineers develop and verify a program for one ATE, then translate it to a program for different ATEs that can test the same chip [27], [28]. For translation, we need to re-allocate registers for the new ATE, and the PBQP graph can elegantly express the irregular register constraints caused by pairing or major cycle [17]. Also, the PBQP graph can express the constraint that no spill is allowed in ATE by making every cost be either zero or infinity, which also makes the cost sum of a solution be either zero or infinity. This allows the enumeration-based approach somewhat simpler because all we need to find is any solution whose cost sum is zero (i.e., we do not have to find the minimum). Actually, finding a valid solution is a mission-critical task for translation because no solution means no translation, thus wasting testing resources. So, enumeration would be more preferred to approximation. However, liberty-based enumeration might easily suffer from an explosion of the search space; our ATE PBQP graphs have 28∼241 vertices, ∼40% of which have a liberty ≤ 4 on average, requiring 4^{19}∼4^{94} enumerations in the worst case. This paper proposes another enumeration-based solver using reinforcement learning to cut the enumeration space sharply, increasing the chances of finding a solution.

C. Deep Reinforcement Learning

We provide some background on deep reinforcement learning (Deep-RL), just enough for understanding the framework that we employ for PBQP, focusing on how *Monte Carlo tree search* (MCTS) works and what the deep neural network (DNN) does for MCTS. MCTS is for look-ahead planning before the agent makes an action. For a given state, such as a Go board positioned with stones, MCTS *simulate* the game in advance by building a *game tree* with the state as a root, where a node is a state and an edge (u, v) represents a state transition from u to v as a result of an action. Here, a *Q-value*, $Q(s, a)$, is defined as the expected reward when we make an action a at the state s. If we know the precise Q-value for any root state, thus the precise probability distribution of actions for any state (i.e., the best policy), we can make the right action that maximizes the cumulative reward, achieving the goal of RL. Unfortunately, we cannot know the precise Q-value since we cannot enumerate a complete game tree (e.g., its size is $\approx 10^{360}$ for Go), so we estimate the Q-value by building a *partial* game tree with the help of DNN.

When MCTS is invoked for the first time for a given game, the game tree will be added only with the root node, s (the initial state). Among the next possible children nodes from the root, MCTS will choose the most promising one and add

it to the tree. Then, among the children of this new node as well as the children of the root, MCTS will again choose the most promising one to expand the tree. MCTS will repeat this process of adding a new leaf node up to k times (as we saw in Figure 1), and estimate $Q(s,a)$ from which the agent will make an action to derive a new state. Then, MCTS is invoked again with the new state as a root, which must be a child of the root in the previous game tree, and MCTS will expand on top of the previous game tree, reusing the previous estimation values. This will repeat for n times, the game length (number of moves in Go), so at the end of the game, the game tree will have a maximum of $n \times k$ nodes.

More precisely, the MCTS algorithm repeats four phases: *selection*, *expansion*, *roll-out*, and *back-propagation*. In selection, MCTS will choose the most promising node among the undiscovered leaf nodes in the current game tree by estimating their *confidence* (see Equation 2 below). In expansion, MCTS will simply append the chosen node in the tree. In roll-out, MCTS will estimate the expected reward of the new node using the DNN. Finally, in back-propagation, MCTS will update all values used to estimate the confidence previously, for every node on the path from the new node back to the root based on the expected reward of the new node.

The selection phase works by recursively traversing a node from the root, following an edge that has the highest value of *upper confidence bound* (UCB) among the edges, until it reaches a leaf node that has never been discovered in the current game. A UCB, $U(s,a)$, for an edge (s,a) from a node s made by an action a is defined as follows:

$$U(s,a) = Q(s,a) + c_{puct} \cdot \hat{\mathbf{p}}(a|s) \cdot \frac{\sqrt{\epsilon + \sum_{a'} N(s,a')}}{1 + N(s,a)} \quad (2)$$

For a newly-added node s, the first term $Q(s,a)$ is initialized to zero. It then gets updated at the back-propagation phase as new nodes are added below the edge, becoming the average of $\hat{v}$ of all nodes encountered when following the edge (s,a). Here, $\hat{v}$ is the output of a DNN, called *V-Net*, which returns the estimation for the *value*, or the expected reward of a state. $N(s,a)$ is also initialized to zero and then incremented by one during the back-propagation phase if the edge (s,a) reaches to the newly added node, meaning that $N(s,a)$ is the number of times this edge has been selected. Therefore, if $N(s,a)$ is large enough, the corresponding $Q(s,a)$ would converge to the expected reward and get more accurate.

In Equation 2, c_{puct} is a constant, ϵ is a very small constant, and $\hat{\mathbf{p}}(a|s)$ indicates the *prior probability* of the edge (s,a), computed by a DNN, called *P-Net*. Since the second term becomes higher for the edges with a higher $\hat{\mathbf{p}}(a|s)$ and a lower $N(s,a)$, it tends to navigate to more probable yet less explored edges. That is, Equation 2 seeks a proper trade-off between *exploitation* and *exploration* using c_{puct} [29]–[31]. Because Q is the average of values collected in the current game, selecting higher Q exploits the current knowledge. Meanwhile, edges with lower N have had fewer chances to be explored, so their Q might not show their true value yet. So, we use the prior knowledge $\hat{p}$ to encourage exploration of those edges.

After repeating the phases k times, we construct a policy $\boldsymbol{\pi}$, the probabilistic distribution of actions for s, as follows.

$$\boldsymbol{\pi}(a|s) = \frac{N(s,a)}{\sum_{a'} N(s,a')} \quad (3)$$

This means that the agent would prefer an edge if it was selected more during the MCTS simulation because the edge is likely to have a higher $U(s,a)$, hence a higher $Q(s,a)$, indicating that the edge is promising in maximizing the reward.

Both training and inference perform MCTS-based actions, but for training, additional data are collected to train both P-Net and V-net. After MCTS builds a partial game tree, a tuple of ($\hat{\mathbf{p}}$, $\mathbf{p}$, $\hat{v}$, v) is generated for the *root node* s, where $\hat{\mathbf{p}}$ is the probability distribution of s estimated by P-Net, $\mathbf{p}$ is the collection of $\boldsymbol{\pi}(a|s)$ in Equation 3 computed after MCTS, $\hat{v}$ is the reward value of s estimated by V-Net, and v is the final reward value for this game which will be decided at the end of this game (all tuples of this game will have the same v value). These n tuples generated by this game will be merged with those tuples of other games in this *iteration*, in order to train P-Net and V-net to be used in the next iteration.

III. Formulation of Deep-RL for PBQP

This section formulates deep reinforcement learning (Deep-RL) for PBQP. First of all, we can easily confirm that register allocation on the PBQP graph is a Markov decision process, because we can make a coloring decision based only on the current state, a partially-colored PBQP graph, regardless of how it was colored in the past. So, we can apply Deep-RL, especially the one used in AlphaZero [20], seamlessly to PBQP. We now formulate action, reward, and state for PBQP.

A. Action

When we are given a PBQP graph of size n, we number the vertices in the graph from 1 to n, and we always color the vertices in increasing order of the vertex number. An action is defined as coloring the *next* uncolored vertex in the current, partially colored graph with the color $a \in \{1, ..., m\}$, where m is the number of colors. The next uncolored vertex should have the smallest vertex number among the uncolored vertices. This coloring order is also preserved when we simulate actions in MCTS such that if the root state is colored up to the vertex l, all its children states are for coloring the vertex $l+1$, all its grandchildren states are for coloring the vertex $l+2$, etc.

B. Reward

AlphaZero normally targets a two-player game such as chess, shogi, and Go, where two opponents take turns to make an action until the game ends. Then, the reward is determined as +1 for a win, 0 for a tie, or -1 for a loss immediately at the end of the game. By contrast, in the PBQP game, a single player repeatedly makes an action until the game is over. Therefore, the environment cannot judge how well this player did in this game. To handle this issue, we compare the current player with the best player from all previous iterations (here, the player means the DNN) for the same game. More specifically,

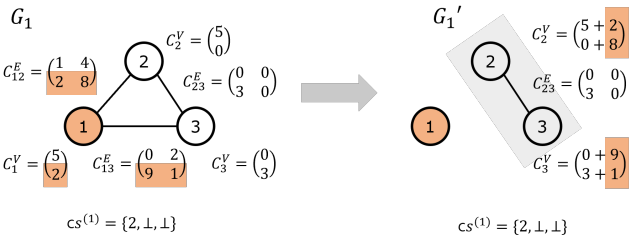


Fig. 3. G_1 is equivalent to G'_1 plus detached colored vertex 1.

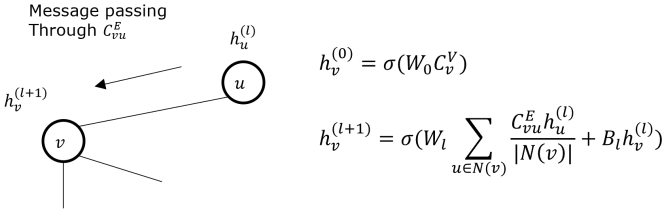


Fig. 4. Latent vector $h_v^{(l+1)}$ for v at layer l+1 is recursively defined by messages $h_u^{(l)}$ from its neighbors u in $N(v)$.

we compare the cost sum defined in Equation 1. If the cost sum produced by the current player is lower, the reward is +1 (win), if it is higher, the reward is -1 (loss), and in the case of a tie, the reward is 0. In this way, the PBQP game enables competition between the previously best player and the current player, and makes the current player incrementally learn how to beat the previously best player from self-plays.

C. State

In AlphaZero, the state of a game board is encoded on the spatial plane based on the game rule. The Go game, for example, has the state of a 19×19 matrix where each entry shows the existence of a stone at the corresponding position (e.g., 1 for a black stone, -1 for a white stone, or 0 for a blank). Likewise, the PBQP game has the state, a partially colored PBQP graph, and the player's action of coloring the next vertex makes a transition from one state to another. We devised an elaborate encoding of the PBQP graph state, which simplifies graph embedding, reduces the number of learnable parameters, and allows the DNN to learn more easily.

Conceptually, the player is given an initial state of the PBQP graph $G_0 := G(V, E, C^V, C^E)$. The player repeatedly makes a coloring action, which produces next states G_1, G_2, ..., G_t, ..., G_n when $|V| = n$. We define a *coloring state* of G_t, $cs^{(t)} \in \{\perp, 1, ..., m\}^n$, a list of length n where i-th entry shows the color assigned to vertex i ($\perp$ indicates that the corresponding vertex is not assigned yet). Since we are always coloring vertices in increasing order of the vertex number, $cs^{(t)} = \left[\{1, ..., m\}^t \| \{\perp\}^{n-t}\right]$. So, notionally, the PBQP state of G_t is defined by G_0 and $cs^{(t)}$.

Practically, we encode the PBQP state of G_t in a *reduced graph form* where we detach already-colored vertices, while propagating the matrix costs corresponding to a detached vertex to its neighbor vertices. In Figure 3, for example, given the vertex 1 is colored with color 2, i.e., $\mathbf{x}_1 = (0, 1)$, the graph G_1 with vertices $\{1, 2, 3\}$ is equivalent to G'_1 with vertices $\{2, 3\}$ with their vertex costs being increased by the matrix elements selected by $\mathbf{x}_1$, plus the detached vertex 1 (equivalency means that a cost sum for G_1 selected by $\mathbf{x}_2$ and $\mathbf{x}_3$ would be the same as the cost sum for G'_1 plus the cost of the vertex 1). Now, we can treat G'_1 as a reduced, equivalent state of G_1 with vertex 2 to be colored next, where the influence of the vertex 1 on other parts of the graph is already fully reflected. That is, after coloring vertex 1, we will actually invoke MCTS with G'_1 as a root (when we expand a child node of G'_1 in the game tree during MCTS, it will also be encoded as a reduced graph form). By encoding the PBQP state in this way, MCTS can now input the same kind of graph where no vertices are colored, instead of a graph with some vertices colored and others not. It decreases the number of learnable parameters, thus being easier to train. Also, the embedding of a PBQP state will be simplified, as described in the following section.

D. Graph Embedding

A key mechanism of AlphaZero is to train a deep neural network (DNN) parametrized by θ, which returns $\hat{\mathbf{p}}$ and $\hat{v}$ based on a board state s as an input: $(\hat{\mathbf{p}}, \hat{v}) = f_\theta(s)$. $\hat{\mathbf{p}}$ is a vector of the prior probabilities of actions with components $\hat{p}_a = Pr(a|s)$. Whereas, $\hat{v}$ is a scalar value of predicting the expected reward outcome v, such that $\hat{v} \approx \mathbb{E}[v|s]$.

Typical DNN require data representations in the form of numerical vectors. Our PBQP state s being in the graph form, we should convert it into a vector representation while maintaining both the local properties and the global structure of the graph. The performance of DNN highly depends on the quality of the vector representations. Therefore, a wide range of graph embedding methods proposed have been proposed for diverse applications. Most of the research is empirical and often geared towards specific application areas such as social network analysis, knowledge graphs, chemoinformatics, and computational biology [23]–[26].

We take the approach of message passing-based *graph convolutional network* (GCN) [23], [26] with some modifications. In this framework, a hidden feature vector $h_v^{(l)}$ denotes the l^{th}-layer latent vector for vertex v, as shown in Figure 4. It is defined for all vertices $v \in V$, in an inductive manner for all layers. That is, for each vertex v, the initial hidden feature vector $h_v^{(0)}$ is set based on the cost vector of v. We then calculate $h_v^{(1)}$ based on all *messages*, $h_u^{(0)}$, where u is a neighbor vertex of v ($u \in N(v)$). The messages are aggregated by using a function AGG, which can be a mean, max pooling, or learned model such as LSTM [26] (we use a mean function). Repeating this process until the final layer l, we can get n hidden feature vectors $\mu_v := h_v^{(l)}, \forall v \in V$. While an edge in general graphs simply shows a message on if two vertices interact or not, the edges in PBQP graphs have additional data, the cost matrix, which indicates the impact of each combination of coloring actions related to the connected vertices. Accordingly, we multiply the message by the cost matrix to convey the real implication of the edge when a message passes thru it, as shown

in Figure 4. So, we represent the graph as a concatenation of all final hidden features, $\mu := [\mu_1 \| ... \| \mu_n]$, an $m \times n$ matrix. It should be noted that we do not have to depict which vertices are colored with what colors for embedding, due to a reduced graph form of a state, as mentioned above. The weights and biases of GCN will also be trained along with P-Net and V-Net during training.

IV. Deep-RL Algorithm for PBQP

This section describes the Deep-RL algorithm for PBQP. As we saw previously in Figure 1, a self-play of a PBQP game is involved with the *graph manager*, the *MCTS planner*, and the *DNN*. We first overview the overall algorithm structure, then describes these components in detail.

A. Overview

One self-play of a PBQP graph with n vertices is called an *episode*, which collects n training data while MCTS explores $n \times k$ states. A fixed number (100 in our evaluation) of different episodes constitute one *iteration*, which is a unit to train the DNN with $100 \times n$ training data using the stochastic gradient descent optimization, and to update the DNN if the training result is better than the existing DNN's.

For the execution of an episode, an initial PBQP graph is given along with the current DNN θ and the previously best DNN θ^*. We first color the graph with θ^* and obtain the cost sum. We then color the graph with θ and obtain the cost sum. Both colorings work in the same way of invoking MCTS n times using its DNN, except that only the latter collects the training data during MCTS (i.e., the former is regarded as an *inference run* while the latter as a *training run*). By comparing the two cost sums, we will decide the reward for this episode.

To color the graph of this episode with f_θ, we first create a new MCTS instance, an empty MCTS game tree. Then, we repeat the following two things until the PBQP game is over. First, we invoke MCTS with the current PBQP state s as an argument, which will append up to k new nodes on a sub-tree rooted at s in the game tree. As a result of MCTS, we will have a tuple of (s, $\hat{\mathbf{p}}$, $\mathbf{p}$, $\hat{v}$, v) which we add to the queue of the training data. Second, we make a coloring action based on the policy $\mathbf{p}$, and the graph manager will make a new current PBQP state s'. This repetition ends when all vertices are colored. Then, we compare the two cost sums as mentioned above, decide the reward (+1, -1, or 0), and update v of all tuples to this reward. In this way, we can train the DNN to prefer or avoid those states.

B. Graph Manager

The role of graph manager is to manage the states in a reduced graph form as discussed in Section III-C. For the current state in a graph form G_t, the graph manager asks MCTS, the planner, to estimate $\mathbf{p}$, an $m \times 1$ vector containing the probability of each action. Then, the graph manager decides an action a_{t+1} based on $\mathbf{p}$, by either sampling $a_{t+1} \sim \mathbf{p}$ or selecting the highest-probability one a_{t+1}=argmax($\mathbf{p}$). Once the action a_{t+1} is decided at the state G_t, the transition $\mathcal{T}$ is applied to generate the next state G_{t+1}. The transition $G_{t+1} = \mathcal{T}(G_t, a_{t+1})$ is defined as follows.

- $V(G_{t+1}) = V(G_t) - \{t\} = \{t+1, ..., n\}$
- $E(G_{t+1}) = E(G_t) - \{(t,u) \in E(G_t), \forall u \in N(t)\}$
- $C_u^V(G_{t+1}) = C_u^V(G_t) + C_{tu}^E(G_t)[a_{t+1}, :], \forall u \in N(t)$
- $C^E(G_{t+1}) = C^E(G_t)$

where $N(t)$ is the set of vertices adjacent to vertex t. We saw an example of such transition when G_0 has 3 vertices and the action $a_1 = 2$ in Figure 3. Graph manager can detect a *dead-end* state where no more coloring is possible when it transits to a new reduced graph.

Algorithm 1: SIMULATE

```
   Input  : s (the board state); θ (parameters of the NN)
   Output : v̂ (the estimated value of s)
1  if s is terminal then
2      return GETGAMERESULT(s);
3  else if s ∈ tree then
4      a ← argmax_{a'∈A} U(s, a');
5      s' ← T(s, a), v̂ ← SIMULATE(s');
6      Q(s, a) ← (N(s,a)×Q(s,a)+v̂) / (N(s,a)+1);
7      N(s, a) ← N(s, a) + 1;
8      return v̂;
9  else
10     tree ← tree ∪ {s};
11     Q(s, ·) ← 0, N(s, ·) ← 0;
12     p̂(·|s), v̂ ← f_θ(s);
13     return v̂;
14 end
```

C. MCTS Planner

We now describe how the MCTS planner simulates the game tree. As mentioned previously, MCTS repeats four phases: selection, expansion, roll-out, and back-propagation. Algorithm 1 (SIMULATE) shows one cycle of such repetitions. The selection phase (lines 4-5) recursively selects an edge with the highest UCB value $U(s,a)$ from the root until it reaches a leaf node that has not been included in the game tree yet. $U(s,a)$ is calculated from $Q(s,a)$, $\hat{\mathbf{p}}(a|s)$, and $N(s,a)$ by Equation 2. The expansion phase (lines 10-11) appends the new leaf node to the partial game tree with the initialization of Q and N values by zero. The roll-out phase (line 12) predicts $\hat{\mathbf{p}}$ and $\hat{v}$ of the new node (state) using the DNN f_θ, or evaluates the state directly if it is a *terminal* state (lines 1-2), meaning that the coloring is done so $\hat{v}$ is computed by comparing its cost sum with the best player's ($\hat{\mathbf{p}}$ is undefined). Finally, the back-propagation phase (lines 6-7) is conducted in the reverse order of selection, by updating Q and N of all edges in the selected path. After k calls of SIMULATE on the current state s_{root}, the values of Q and N get more reliable if f_θ is accurate and k is large enough. When the simulation with MCTS is over, we can obtain the probabilistic distribution of the root state as in Equation 3, from which we make an action.

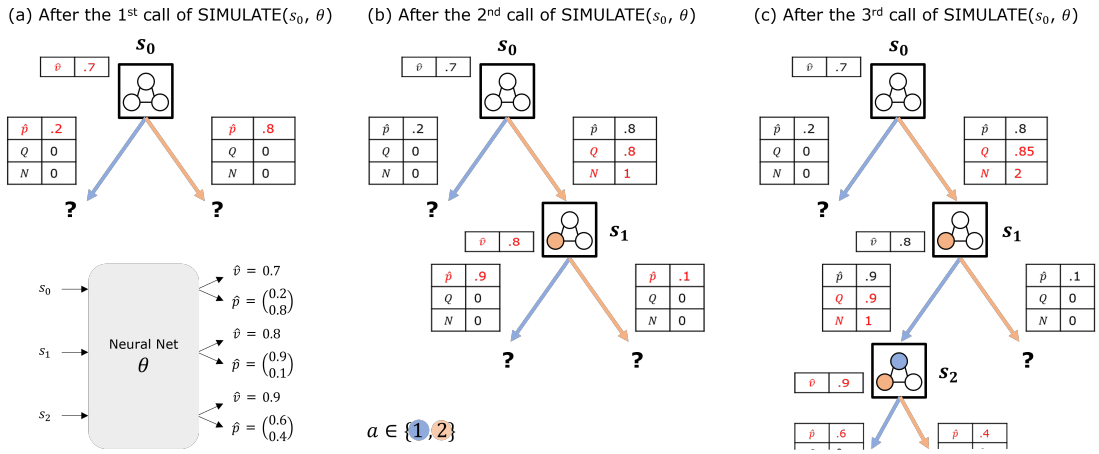


Fig. 5. Illustration of MCTS with three calls of SIMULATE in Algorithm 1.

Figure 5 illustrates how MCTS builds the partial game tree with the first three calls of SIMULATE. For simplicity, the input graph has three vertices and two colors: 1 (blue) and 2 (orange). The initial state s_0 means when no colors are assigned to any vertices. At the 1^{st} call of SIMULATE(s_0, θ), it appends the state s_0 to the tree root and evaluates s_0 using the neural network f_θ. The evaluation results, $\hat{v}$ and $\hat{\mathbf{p}}$, are recorded in the partial game tree, and the newly discovered edges have their Q and N initialized to zero, as in Figure 5 (a). At the 2^{nd} call of SIMULATE(s_0, θ), the edge to the state s_1 is selected because it has the highest U out of all two edges starting from s_0. The newly discovered state s_1 is also evaluated by f_θ and the results are recorded. Now, at its back-propagation phase, $Q(s_0, s_1)$ is updated to have $\hat{v}(s_1) = 0.8$, and $N(s_0, s_1)$ is incremented by one, as in Figure 5 (b). At the 3^{rd} call of SIMULATE(s_0, θ), the edges (s_0, s_1) and (s_1, s_2) are recursively selected because they have the highest U. The new state s_2 is evaluated by f_θ and the results are stored. The two edges on the path from $s2$ to the root are updated at the back-propagation phase. First, $Q(s1, s2)$ is updated to have $\hat{v}(s_2) = 0.9$ and $N(s1, s2)$ is incremented by one. Then, $Q(s0, s1)$ is updated to have the average of $\hat{v}(s_1) = 0.8$ and $\hat{v}(s_2) = 0.9$, resulting in 0.85, and $N(s0, s1)$ increases to 2, as in Figure 5 (c). In this way, we make k calls to SIMULATE and obtain $\mathbf{p}(a|s_{root})$, from which we make a coloring action and create a new state. Then, we make another k calls to SIMULATE with the new state as a root, say s_1 in Figure 5 (c), which must be a child of the previous root. The k calls to SIMULATE would append k new nodes at the subtree rooted at s_1, while reusing all previous nodes in the subtree and their Q and N values (actually, fewer than k nodes if encountering terminal nodes, especially in the latter stages of the episode). So, a maximum of $k \times n$ nodes will be added in the game tree during this episode. After making k calls to SIMULATE(s, θ), we obtain a tuple of $(s, \hat{\mathbf{p}}, \mathbf{p}, \hat{v}, v)$, where $\hat{\mathbf{p}}$ and $\hat{v}$ are predicted values for the root s by θ, while $\mathbf{p}$ is $\mathbf{p}(a|s)$ calculated for the root, which would work as a label for $\hat{\mathbf{p}}$. Also, as a label for $\hat{v}$, v will be decided later based on the reward of the final state.

D. Neural Networks

The structure of the neural network with which the MCTS module consults is as follows. There are GCN layers first, which produce the graph embedding μ for a given state in a reduced graph form, as discussed in Section III-D. It then goes through a standard ResNet [32], followed by two separate fully-connected (FC) layers. One FC layer is connected to softmax function and the other FC layer is connected to $tanh$ function, to generate classification $\hat{\mathbf{p}}$ (P-Net) and regression $\hat{v}$ (V-Net), respectively. The learnable parameters of the neural networks denoted by θ are trained by the gradient descent from the training data collected during self-plays. The loss function is designed to be the sum of cross entropy, MSE, and the L2 regularization term as $\mathcal{L}_\theta(\hat{\mathbf{p}}, \mathbf{p}, \hat{v}, v) = (v - \hat{v})^2 - \mathbf{p}^T \log \hat{\mathbf{p}} + c\|\theta\|^2$. A few batch normalization [33] layers are also inserted.

E. Backtracking and Liberty-Based Coloring Order

Register coloring based on the inference run of Deep-RL is an *one-way movement* guided by MCTS and DNN; it never *backtracks*. For ATE register allocation where the cost is either zero or infinity, coloring a vertex then propagating the cost to its neighbors might create a dead-end state where no colors are available for the next vertex (i.e., its cost vector includes only infinities). At a dead-end state, we normally give up and the register allocation fails. However, if we allow backtracking, we might pursue additional opportunities to color the graph. Unlike typical AlphaZero games such as Go, the PBQP game allows the agents to freely take back actions, since it is a single-player game for optimization.

Despite planning ahead using MCTS, coloring might end up with a dead-end state, even before reaching a terminal state because MCTS only performs a partial simulation based on the prediction of DNN. Simulation gets vulnerable with less trained DNNs and early stages of the game. At a dead-end state, we cancel the most recent coloring action and try another promising coloring action, after calling MCTS again for the previous state; the previous decision was wrong probably due

to a lack of enough "thinking time" (MCTS), so we extend the game tree further to make the right decision this time.

Another important issue to consider is the coloring order. As we mentioned in Section III-A, Deep-RL colors the vertices in a fixed order. Instead of using a random order, we propose coloring vertices in decreasing order of liberty, in an opposite order used in [17]. The reason is that within an episode the MCTS would get more accurate as the game tree is expanded more in the later stages of the game, so coloring hard, lower-liberty vertices later would be the right strategy, while coloring easy vertices in the early stage would avoid a dead-end state even if inaccurate MCTS guides a wrong decision.

V. Experiment

A. Experimental Setup

We trained the neural network on a server equipped with Intel Core i7-7700 CPU @3.60 GHz, 32 GB DDR4 RAM, and NVIDIA TITAN Xp with the CUDA 10.2 driver. Our Deep-RL is implemented in the software environment of Ubuntu 16.04 LTS, Python 3.8.3, and PyTorch 1.7.0. For each episode, we generate a random graph using Erdős-Rényi model [34], which takes two arguments n and p_{edge}, where n is the number of vertices and p_{edge} is the probability of generating an edge, so the expected number of edges becomes $p_{edge} \times \frac{n \cdot (n-1)}{2}$. Since we need to generate a PBQP graph, we also assign a random vector of size m to each vertex and a random matrix of size as $m \times m$ to each edge. These vectors/matrices must include some infinity costs, so we have another argument p_{inf}, a ratio of infinities, which we set to 1%. The hyperparameters found by a grid search are as follows: we experimented with 200 iterations, where the number of episodes in each iteration is 100, meaning that we trained using 20,000 random PBQP graphs. The number of vertices of a PBQP graph is normally distributed with an average of 100, so an iteration generates around 10,000 fresh training data. To avoid a radical update of the DNN, we enqueue these data to a queue of previous data whose size is fixed to 200,000 training data, which will be used to train a new DNN with the Adam optimizer (batch size is 64). Then, we compare its performance with the previous DNN for 10 new random graphs, and we keep the new model if it wins for more than 5 graphs, and discard it otherwise. We used a diverse number of k for MCTS: k_{train}=50 and k_{train}=100 for training runs which took one week and two weeks, respectively, and k_{infer} of 25, 50, 100, and 150 for inference runs (cf. AlphaZero uses 800 for both k's).

B. Evaluation of ATE Programs

For evaluation of PBQP register allocation for ATEs, we experimented with 10 product-level ATE programs (*PRO1-PRO10*). Each has a single function whose average code size is 1K lines. The irregular register constraints are expressed using the cost vectors/matrices as in [17], and we target the same m=13 physical registers. The original PBQP solver [15] failed for 9 programs, while the liberty-based enumeration solver succeeded for all programs [17], which we use as a basis for our evaluation. We first experimented with a Deep-RL solver *without* backtracking, for various pairs of (k_{train},k_{infer}) of MCTS, which means an inference run with MCTS of k_{infer}, using a DNN trained with MCTS of k_{train}. We found that (100,150) finds a solution for all programs; (50,25) failed for 7 programs, and (50,50) failed for one program, *PRO10*, which has the biggest PBQP graph with 250 vertices, making many pairs fail. It took ∼40 seconds for the pair (100,150) to run.

Next, we evaluate a Deep-RL solver *with* backtracking. This time we compare the total number of nodes (states) generated in the game tree during the inference run. It will show how efficient a Deep-RL solver is in narrowing down the search space, regardless of the GPU computation power. We experimented with a backtracking solver that colors the vertices in three different orders: (b) random order, (c) increasing liberty order, and (d) decreasing liberty order. We compare with the Deep-RL solver without backtracking, designated by (a).

Figure 6 shows the number of nodes generated for two MCTS pairs of the inference run: (50,25) and (50,50). The X marks on some bars indicate the cases where it fails to find a valid solution. As mentioned above, (a) failed for 7 programs for (50,25) and failed one program for (50,50). Surprisingly, all of (b), (c), and (d) found a valid solution for all programs even with (50,25) and (50,50); it should be remembered that the Deep-RL solver without backtracking required (100,150) to find a solution for *PRO10*. This means that backtracking is highly effective for complementing Deep-RL to find a solution, even with a modestly trained DNN and with around 1/3∼1/5 of the simulated nodes during inference.

When we compare the number of nodes generated, we can see that (d) generates much fewer nodes than (b) and (c). For ATE programs where (a) found a valid solution, the number is almost the same between (a) and (d). This confirms that coloring with a high-to-low liberty order, thus coloring hard vertices later when MCTS gets more accurate, would be the right strategy as conjectured in Section IV-E.

For (d) of (50,25), we also evaluated a case when we do not call MCTS for the parent of a dead-end state but simply choose the next highest-probability coloring action; if we can choose the same state without such MCTS, we might find a solution without expanding the game tree. However, we found that there is no tangible difference and the reason is that if we choose the same state, the nodes to be added by MCTS for the parent are likely to be added by MCTS for the chosen node anyway, eventually ended up with the same terminal nodes. Also, our UCB Equation 2 seems to prefer exploitation rather than exploration, expanding the tree deeply rather than widely.

If we compare with the number of states generated by the liberty-based enumeration [17], (d) reduces it by a factor of 1/3,500-1/13,000, e.g., 5.6K vs. 19.8M for PRO10. Although MCTS requires more expensive DNN computations to generate states, we may reduce them by using powerful GPUs or NPUs; it is not fair to compare our time with a (mediocre) GPU to their CPU time, but both are ∼15 seconds for PRO10, but our time would be smaller for other benchmarks considering the number of states and its correlation with running time. More importantly, the enumeration solver is fundamentally an

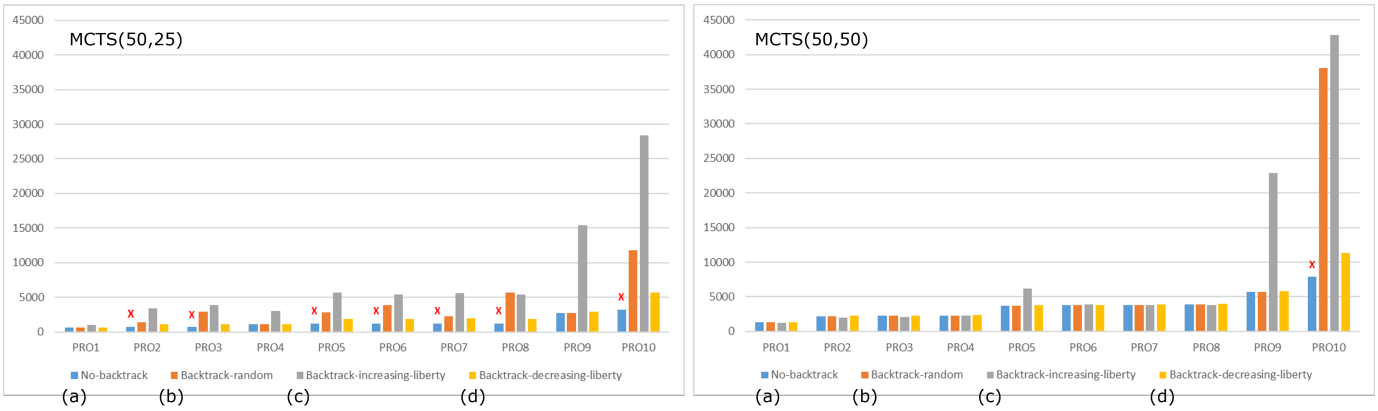


Fig. 6. The total number of nodes (states) generated in the partial game trees (lower is better) with k_{infer} 25 (left) 50 (right).

exponential algorithm whose running time can easily explode if the complexity increases (e.g., when m is smaller), while our Deep-RL solver which works more of a polynomial algorithm can readily absorb parts of the complexity by training with a higher k_{train} to produce a more precise DNN. For example, when we experiment with (d) of (100,20), we could find a solution with up to 10% fewer nodes than (50,25), requiring 10% fewer DNN computations. So, Deep-RL can work more efficiently as we increase k_{train} and decrease k_{infer}, thus thinking more during training but less during inference.

C. Evaluation of C programs on LLVM

Currently, LLVM has four options for register allocation. `FAST` is the baseline with local register allocation. `BASIC` is a linear-scan algorithm [6]. `GREEDY` (also known as "GRA") is the LLVM default, a linear-scan algorithm [6] with aggressive live ranges splitting. `PBQP` is PBQP-based allocator with the original solver [15]. We implemented the RL-based PBQP solver by modifying `PBQP` on top of LLVM 12.0.0, which we call `PBQP-RL`. We do not use backtracking for `PBQP-RL` since it does not help when searching for the minimum cost sum, unlike the zero cost sum in ATE PBQPs, since there is no dead-end state if the spill is allowed as in regular PBQPs.

The evaluation is conducted on three benchmark suites included in llvm-test-suite, which consists of 24 C/C++ programs. The programs are compiled to LLVM Bitcode with the clang -O3 option. We first compared `PBQP-RL` and `PBQP`, in terms of the cost sum computed. We found that `PBQP-RL`(50,150) achieved almost the same cost sum except for two benchmarks (Oscar.c and FloatMM.c) which has a slightly ($< 9\%$) higher sum. As is usually the case with DNN, the reason is not easily explainable. In terms of the speedup of the generated code compared to `FAST` as a baseline, `GREEDY`, `PBQP`, and `PBQP-RL` achieves 1.464x, 1.422x and, 1.416x, respectively, on x86 machine (Intel Core i7-9700K CPU @3.60 GHz, 64 GB DDR4 RAM, Ubuntu 18.04.2 LTS). This means that `PBQP` is already doing well compared to highly optimized `GREEDY`, and `PBQP-RL` is comparable to them in this allocation environment looser than ATE environment.

When we experiment with `PBQP-RL`(50,300) and `PBQP-RL`(50,650), we could obtain the same cost sum even for Oscar.c and FloatMMC.c, respectively. At these k_{infer}, `PBQP-RL` achieved a slightly ($\leq 3\%$) lower cost sum than `PBQP` for three benchmarks. These results indicate that `PBQP-RL` is also competitive for general programs.

VI. Related Work

There are a few ML-based techniques applicable to solve register allocation. *Meta-optimization* is an ML technique to automatically fine-tune compiler heuristics including the priority function of register spills [35], but it is not for building a register allocation algorithm itself using DNN, unlike ours.

There is a work to predict the chromatic number of a given graph, the minimum number of colors to color all vertices, using deep learning [36]. It does not produce the actual mapping of colors and sometimes predicts a wrong chromatic number, thus not directly applicable to register allocation.

Huang et al. employ Deep-RL of AlphaZero to color big graphs [37], which actually inspired our work. However, it is not straightforward to solve register allocation directly from vanilla graph coloring since register spills as well as the limited number of colors should be considered, yet they consider only coloring graphs with unlimited colors.

Das et al. [38] introduce a supervised learning technique to directly solve register allocation using LSTM (long short-term memory). They use a recurrent neural network to generate a sequence of allocation decisions one for each vertex in the interference graph. Due to LSTM bounding its output to a range of valid choices, it might color vertices incorrectly. Therefore, they require a follow-up correction phase, which fixes all pairs violating interference rules in a greedy manner. Although their result shows a promising performance, the correction phase might make sub-optimal decisions, limiting its learnability.

VII. Summary and Future Work

Register allocation is a complex combinatorial optimization problem that may benefit from machine learning (ML). Among ML techniques, reinforcement learning (RL) is more appropriate than supervised/unsupervised learning since it obviates the

labels, the optimal allocation results that are hard to obtain, but learns from as many self-plays as it wants. A Deep-RL approach with MCTS and DNN such as AlphaZero is a proven technology, applicable when the model is clear such as the register allocation game. Finally, PBQP is a better formulation than graph coloring for Deep-RL since it can represent spill as well as coloring in a unified framework, and work better for irregular register allocation. Motivated by these rationales, this paper proposed a PBQP solver based on Deep-RL. We solved how to decide the reward, how to represent the state and its transitions, how to embed PBQP graphs, and how to help Deep-RL with backtracking and liberty-based coloring orders. Our experiments show that Deep-RL can find solutions for ATE programs while searching 1/3,500∼1/13,000 fewer states than the liberty-based enumeration. It also shows the competitive result for general programs on LLVM.

We are working on another practical task with Deep-RL, solving a combined problem of scheduling *and* register allocation. When translating ATE programs for different-speed DRAM, we need to schedule instructions while meeting all register constraints, which is more challenging than PBQP.

ACKNOWLEDGMENT

This work was supported by the MSIT (Ministry of Science and ICT), Korea, under the ITRC (Information Technology Research Center) support program (IITP-2021-0-01835, 50%) supervised by the IITP (Institute of Information & Communications Technology Planning & Evaluation), by the IITP grant funded by the Korea government (MSIT) (No. 2021-0-00136, Development of Big Blockchain Data Highly Scalable Distributed Storage Technology for Increased Applications in Various Industries, 50%), and by Samsung Electronics.

REFERENCES

[1] F. Bouchez, A. Darte, C. Guillon, and F. Rastello, "Register allocation: What does the np-completeness proof of chaitin et al. really prove? or revisiting register allocation: Why and how," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2006, pp. 283–298.

[2] G. J. Chaitin, "Register allocation & spilling via graph coloring," *ACM Sigplan Notices*, vol. 17, no. 6, pp. 98–101, 1982.

[3] P. Briggs, K. D. Cooper, and L. Torczon, "Improvements to graph coloring register allocation," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, pp. 428–455, 1994.

[4] L. George and A. W. Appel, "Iterated register coalescing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 18, no. 3, pp. 300–324, 1996.

[5] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 21, no. 5, pp. 895–913, 1999.

[6] C. Wimmer and M. Franz, "Linear scan register allocation on ssa form," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, 2010, pp. 170–179.

[7] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, 2004, pp. 75–86.

[8] J. Olesen, "Register allocation in llvm 3.0," in *LLVM Developers' Meeting*, 2011.

[9] T. C. de Souza Xavier, G. S. Oliveira, E. D. de Lima, and A. F. da Silva, "A detailed analysis of the llvm's register allocators," in *2012 31st International Conference of the Chilean Computer Science Society*. IEEE, 2012, pp. 190–198.

[10] D. W. Goodwin and K. D. Wilken, "Optimal and near-optimal global register allocation using 0–1 integer programming," *Software: practice and experience*, vol. 26, no. 8, pp. 929–965, 1996.

[11] T. Kong and K. D. Wilken, "Precise register allocation for irregular architectures," in *Proceedings. 31st Annual ACM/IEEE International Symposium on Microarchitecture*. IEEE, 1998, pp. 297–307.

[12] A. W. Appel and L. George, "Optimal spilling for cisc machines with few registers," *ACM SIGPLAN Notices*, vol. 36, no. 5, pp. 243–253, 2001.

[13] D. Koes and S. C. Goldstein, "A progressive register allocator for irregular architectures," in *International Symposium on Code Generation and Optimization*. IEEE, 2005, pp. 269–279.

[14] F. M. Quintão Pereira and J. Palsberg, "Register allocation by puzzle solving," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2008, pp. 216–226.

[15] B. Scholz and E. Eckstein, "Register allocation for irregular architectures," in *Proceedings of the joint conference on Languages, compilers and tools for embedded systems: software and compilers for embedded systems*, 2002, pp. 139–148.

[16] L. Hames and B. Scholz, "Nearly optimal register allocation with pbqp," in *Joint Modular Languages Conference*. Springer, 2006, pp. 346–361.

[17] M. Kim, J.-K. Park, and S.-M. Moon, "Irregular register allocation for translation of test-pattern programs," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 18, no. 1, pp. 1–23, 2020.

[18] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[19] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[20] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017.

[21] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel *et al.*, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[22] T. Anthony, Z. Tian, and D. Barber, "Thinking fast and slow with deep learning and tree search," in *Advances in Neural Information Processing Systems*, 2017, pp. 5360–5370.

[23] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[24] M. Defferrard, X. Bresson, and P. Vandergheynst, "Convolutional neural networks on graphs with fast localized spectral filtering," *arXiv preprint arXiv:1606.09375*, 2016.

[25] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, "Neural message passing for quantum chemistry," in *International Conference on Machine Learning*. PMLR, 2017, pp. 1263–1272.

[26] W. L. Hamilton, R. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *arXiv preprint arXiv:1706.02216*, 2017.

[27] M. Kim, J.-K. Park, S. Kim, I. Yang, H. Jung, and S.-M. Moon, "Output-based intermediate representation for translation of test-pattern program," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 18, no. 5s, pp. 1–22, 2019.

[28] J.-G. Park, M. Kim, S.-M. Moon, S. Kim, I. Yang, and H. Jung, "Patran: Translation platform for test pattern program," in *2019 IEEE European Test Symposium (ETS)*. IEEE, 2019, pp. 1–2.

[29] P. Auer, "Using confidence bounds for exploitation-exploration trade-offs," *Journal of Machine Learning Research*, vol. 3, no. Nov, pp. 397–422, 2002.

[30] C. D. Rosin, "Multi-armed bandits with episode context," *Annals of Mathematics and Artificial Intelligence*, vol. 61, no. 3, pp. 203–230, 2011.

[31] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140–1144, 2018.

[32] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[33] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *International conference on machine learning*. PMLR, 2015, pp. 448–456.

[34] P. Erdős and A. Rényi, "On the evolution of random graphs," *Publ. Math. Inst. Hung. Acad. Sci*, vol. 5, no. 1, pp. 17–60, 1960.

[35] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, "Meta optimization: Improving compiler heuristics with machine learning," *ACM sigplan notices*, vol. 38, no. 5, pp. 77–90, 2003.

[36] H. Lemos, M. Prates, P. Avelar, and L. Lamb, "Graph colouring meets deep learning: Effective graph neural network models for combinatorial problems," in *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2019, pp. 879–885.

[37] J. Huang, M. Patwary, and G. Diamos, "Coloring big graphs with alphagozero," *arXiv preprint arXiv:1902.10162*, 2019.

[38] D. Das, S. A. Ahmad, and V. Kumar, "Deep learning-based approximate graph-coloring algorithm for register allocation," in *2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar)*. IEEE, 2020, pp. 23–32.