Learning a labeling function for Cyclic Best First Search with Reinforcement Learning

Ian Lin
Department of Infomation Management and Finance
National Yang Ming Chiao Tung University
ianlienfa.mg07@nycu.edu.tw

Abstract

Branch-and-Bound is a commonly-used algorithm to solve operation research and scheduling problems, such as the traveling salesman problem and single-machine scheduling problems. In Branch-and-Bound, there are three different phases for solving the problem: the branching, the bounding, and the pruning phase. Each phase can use various strategies to improve the algorithm's performance. A trending searching strategy is called the cyclic-best first search (CBFS) algorithm, which uses contours to help guide the searching process. CBFS uses a labeling function to determine the next contour to search. In this paper, we explore the use of reinforcement learning to learn a labeling function for CBFS and compare the performance of the newly learned searching strategy with other common-seen search strategies. First, we provide a framework that is designed on the Branch-and-Bound concept proposed in Morrison et al. [2016]. Then, with it we solve a single machine scheduling problem, $1|r_j|\sum w_jC_j$, to evaluate the performance of the new searching strategy. Additionally, to make learning a labeling function more practical, a new interpretation of the CBFS labeling function is proposed, which is a game-like wrapper that labels nodes given to guide the search. Numerical experiments are conducted on instances categorized by the number of jobs. The results have shown better performance in instances where the number of nodes visited for solving them is less than 150000. It can be added as an additional improvement on an existing Branch-and-Bound solver and will have no collision with pruning or branch strategies. Moreover, it provides 10 percent less node search than the other search strategies. Additional discussions on the max-number-of-contours show that setting its value high is discouraged in practice based on the experiment.

1 Keywords

Cyclic best-first search, Labeling Function, Branch-and-Bound, $1|r_j|\sum W_jC_j$, Reinforcement Learning, Optimization.

2 Introduction

Branch-and-Bound is an algorithm widely used in solving operation research, scheduling, and linear programming problems. It is a search algorithm that can solve problems with large search spaces. Branch-and-Bound grows the search tree and prunes the branches that are not promising to speed up the searching process. Morrison et al. [2016] discusses Branch-and-Bound technique and split it into three phases, respectively searching, branching, and pruning. The searching phase refers to how the search tree is traversed, branching refers to how the sub-nodes are generated upon the current node,

and pruning refers to how the search tree eliminates some branches that are not promising based on some given rules.

Except for the common-seen Depth-First Search (DFS), Breadth-First Search (BrFS), and Best-First Search(BFS), there is one exciting searching strategy called Cyclic-Best First Search (CBFS). CBFS maintains a set of labeled contours and stores the nodes in the contours. For each round, an evaluated-to-be-the-best node will be extracted from the current contour to proceed. Then, the following search will start from the next contour. When a new node is generated, a labeling function will evaluate the node and assign it to a corresponding contour. What is interesting is that the design of the labeling function can be very flexible and can be used to implement different search strategies. Moreover, it is shown in Morrison et al. [2017] that CBFS can be interpreted as a more generalized form of the above-mentioned search strategies. We want to know to what extent the design of the labeling function can have an impact on the searching performance and whether it is possible to use RL to learn a good labeling function. The goal of this paper is to dive more deeply into the CBFS strategy and its labeling function and see if it is plausible to learn a good labeling function.

3 Recent Research and Comparison

3.1 CBFS

Cyclic best-first search is a searching strategy first proposed in Kao et al. [2009] initially called distributed best-first search. Morrison et al. [2017] provides a more theoretical discussion on CBFS. Zhang et al. [2021] further discusses the number of nodes explored by CBFS on various instances compared to the Depth-first search and the best-first search.

3.2 Making decisions with machine learning

Bengio et al. [2021] surveys the recent attempts at leveraging machine learning to solve combinatorial optimization problems. We see that machine learning can give recommendations on variables to branch for or provide better heuristics. In Di Liberto et al. [2016], the authors recognize that the variable selection that functions well near the root of search trees do not necessarily perform well at the bottom of the tree, Hence, they learn a model that dynamically switches between predefined policies during the tree search based on the current search state.

3.3 Learning the labeling function

As mentioned, the labeling function is essential in determining the search order. With different labeling function set-ups, CBFS can mimic different search strategies.

Bengio et al. [2021] discusses several state-of-the-art solver engines for combinatorial optimization that improves problem-solving by incorporating a machine learning decision-maker into an operation research algorithm. However, to the best of our knowledge, only a few focus on search strategies, and none learn a CBFS labeling function.

We want to learn the CBFS labeling function that automatically affects the order of search. The benefit of learning the CBFS labeling function is that the labeling function makes complex decisions on how to order search nodes instead of making choices among predefined strategies, which could potentially create limitations on guiding an optimal search path. For example, Di Liberto et al. [2016] learns to choose among predefined strategies, and since predefined strategies have fixed operations, there might be some searching behavior that it is incapable of performing.

4 Background for Branch-and-Bound and CBFS

4.1 Branch-and-Bound

The goal of Branch-and-Bound is to find the best solution x that minimizes $f(x)$ for one root problem in which:

- searching strategy provides the next node (subproblem) to search for

- branching strategy split current node (subproblem) into several nodes (subproblems)
- prune strategies prune subproblems that are not likely to be the optimal solutions, decreasing the size of the search space

4.2 Cyclic best-first search

Cyclic best-first search (CBFS), discussed in detail in Morrison et al. [2017], is a searching strategy like Best-first Search or Depth First Search.

CBFS comprises three main components: contours, measure-of-best function, and labeling functions. Contours are groups of nodes to be searched; every contour has its label. The measure-of-best function orders nodes in a contour. In practice, we implement contours using priority queues. A labeling function takes a node as the input and outputs a corresponding label for that node. CBFS drives the search by visiting contours in the order of their labels and picks out one node to search for at each visit.

Cyclic Best first search is widely used to provide "one more chance" for nodes with similar lower bounds. For instance, for those problems where subproblems have close lower bounds, CBFS still visits some nodes even if they are not evaluated as the best ones. The other benefit of CBFS shines when solving scheduling problems. CBFS-level, which categorizes nodes by the number of jobs done, can guide the search path to visit feasible solution nodes periodically. For example, it is often the case that the measure-the-best function considers nodes that are closer to the tree root better. However, in that situation, the incumbent solution is seldom updated since fewer nodes close to the leaf are visited. CBFS-level guarantees a periodical visit to the leaf node; hence, the pruning part of the Branch-and-Bound can work much better and result in a faster solving process.

A Branch-and-Bound algorithm that uses the CBFS strategy works as follows: define set of contours $C = \{...C_{-3}, C_{-2}, C_{-1}, C_0, C_1, C_2...\}$, labeling function κ, measure-of-best function μ , the best solution has seen (incumbent solution) $\hat{x}$.

Algorithm 1: Branch-and-Bound algorithm that uses CBFS as searching strategy

$C = \{\}$, initialize incumbent solution $\hat{x}$
while do $C \neq 0$
  $C_i \leftarrow$ contour $\in C$ with smallest label
  $S \in \arg\min_{U \in C_i} \mu(U)$
  if *S can be solved to find* $\hat{x}' = \arg\min_{x \in S} f(x)$ **then**
    $\hat{x} = \hat{x}'$ if $f(\hat{x}') < f(\hat{x})$
  else
    Generate $S_1, S_2, ...S_r$ such that $\cup_i S_i \supseteq S$
    foreach $j \in 1, 2, 3, ...r$ **do**
      Insert S_j into $C_{\kappa(S_j)}$
      Insert $C_{\kappa(S_j)}$ into C if $C_{\kappa(S_j)} \notin C$
    end
  end
  Remove S from C_i
end
return $\hat{x}$

Morrison et al. [2017] shows that CBFS can perform exactly like best-first search, depth-first search, and breadth-first search with different labeling functions applied. For example, to emulate the behavior of best-first search, set the labeling function $\kappa_{best}(S) = 0$ that takes node(or subproblem) as input and outputs a constant value of 0. In this case, every node will be placed into one single contour, and the order of search fully depends on the measure-of-best function.

From the discussion made in Morrison et al. [2017], we know that CBFS mimics different strategies under different labeling functions:

4.2.1 Best-First Search Emulation

$$\kappa(S) = 0$$

This makes all node goes to the exact contour with label 0, so the search order follows the output of measure-of-best function μ

4.2.2 Depth-First Search Emulation

$$\kappa(S_j) = \kappa(S) + 1 + \sum_{k=1}^{j-1} \Delta(S_k)$$

where S_j are subproblems of S and $\Delta(S)$ is the upper bound on the number of subproblems contained in the tree rooted in S. Intuitively, the labeling function puts the first child into the next contour, so that the search emulates DFS.

4.2.3 Breadth-First Search Emulation

$$\kappa(S_j) = j$$

The intuition on this is that the labeling function distributes subproblems (children) of one node successively in the following contours, so that as the CBFS visits the following contours, the search is guided to mimic the Breadth-first search.

Morrison et al. [2017] also discusses the concept that: A good labeling function can prioritize the search of nodes that could potentially result in better incumbent solutions Moreover, lower the precedence of the searching for those nodes that have the potential to be pruned in the future.

Combining the information above, we follow two principles while designing the RL environment and state encoding for training the CBFS labeling function:

1. The labeling function should be elastic enough to provide contour insertion
2. The contour environment should be elastic enough to provide contour insertion from both the front and bottom
3. The labeling function should know that it has the ability to place nodes into the same contour

The first and second principle is to ensure that CBFS can mimic the Breadth- or Depth-first search, and the third principle should be complied with for CBFS to be capable of performing like Best-first Search. Practically, at training, we have to have awareness and avoid the agent behaving too similarly to DFS or BrFS. The former takes a relatively long period to supply feasible solutions, making pruning hard, and the latter often takes up too much memory space. Another thing to keep in mind is to make sure the agent is trained but not under too many limits, so it has elastic behavior to choose between all kinds of possible search paths. In practice, if a good pruning strategy is implemented and the rate of the node increase does not outnumber that of the node pruning too much, it will be acceptable for BrFS-like behavior to take place, as long as we provide it with slightly larger memory space.

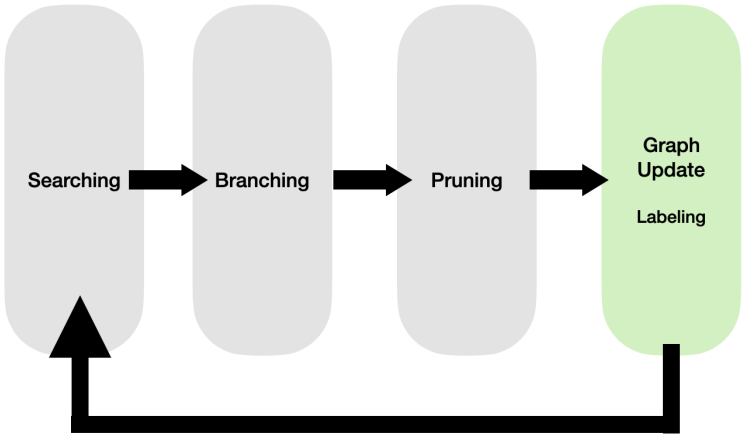
These principles are considered in the design of the game-like wrapper environment we use to help with the training on the CBFS labeling function.

5 Problem Formulation

The goal is to train a labeling function for CBFS. Since CBFS plays a part in providing search order for nodes in search space during the problem-solving process of Branch-and-Bound, a problem is called for to be solved, then improve with reinforcement learning. The problem chosen is a single machine scheduling problem, $1|r_j|\sum w_j C_j$.

$1|r_j|\sum w_j C_j$ is an NP-hard problem with a set of jobs $J = \{j_1, j_2, ...j_n\}$. For each job, R_j, w_j, p_j, and c_j respectively denotes the job's *release time* (the time point that a job is issued and allowed to be placed on a machine), *weight*, *processing time*, and *completion time*. A solution to this type of problem comes in a sequence of jobs, and the goal is to find an optimal job sequence that minimizes $\sum w_j c_j$.

Figure 1: The Branch-and-Bound architecture for BBGym-RL



To solve this problem, one can try out every permutation of job sequences and pick out the sequence with the minimum objective. However, this fashion cannot be applied in instances with a large number of jobs. For instance, the number of possible sequences of 15 jobs is 15! and has already come to the limit of the computation power of modern machine. This is where Branch-and-Bound comes into play.

With Branch-and-Bound, we maintain a tree-like structure where each node represents a partial sequence. For each iteration, the searching strategy indicates the next node to search. The branching strategy splits the node (subproblem) into several nodes (subproblems), then the pruning strategies prune some of these nodes out from the tree. The problem-solving process halts when the optimal solution is found and no more node is left to search.

Our primary goal is not about solving the problem but to *learn a good CBFS labeling function to guide the search better* so that the incumbent and optimal solution can be found with fewer node visitations. A good searching or branching strategy can speed up the process of finding a better incumbent solution and further improve the effect of pruning. Generally, a high-quality incumbent solution, which has a value close to the optimal solution, helps drive the pruning phase to prune out more nodes and leads to a quicker solve.

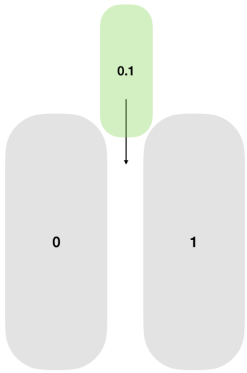
We choose $1|r_j|\sum w_jC_j$ as our problem to experiment for. The reason for choosing is as our problem is as follows:

1. $1|r_j|\sum w_jC_j$ is a scheduling problem on which, under a small number of jobs, all possible sequences can be easily enumerated to abusively find the optimal solution. Therefore, it is easier to validate
2. We want to solve a problem using Branch-and-Bound, and $1|r_j|\sum w_jC_j$ is rather a hard problem that the initial search space is large enough for searching and pruning to make an evident event. Besides, $1|r_j|\sum w_jC_j$ is well-surveyed and has many pruning and search strategies for us to apply.

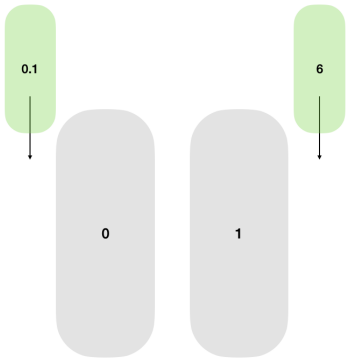
A Branch-and-Bound framework is established to learn the labeling function. The reinforcement learning model acts as a module that makes decisions for node searching order at the searching phase and is being trained while the Branch-and-Bound algorithm runs.

For this paper, a Branch-and-Bound framework – BBGym-RL is implemented.

As we can see in figure 1, BBGym-RL (our Branch-and-Bound framework) implements Branch-and-Bound following the idea of Morrison et al. [2016], which split Branch-and-Bound into searching, branching, and pruning phases. There is one extra phase called graph update, in which we update



(a) labeling function should allow insert between any given contours



(b) labeling function should allow large extension space for both front and bottom

Figure 2: Principles for designing labeling function for training

the tree with the branched node and place them into contour based on the label done by the labeling function.

6 Research methods

We want to know whether it is possible to improve the searching process by introducing a reinforcement learning agent as a labeler of CBFS. The improvement we discuss here is defined as less number of search nodes. We also want to know to what extent the improvement can be achieved if there are improvements. Additionally, we want to provide a practical design strategy of how to use this method to deal with real-world problems.

7 Implementation and experiments

For this section, we will discuss the implementation into four phases. The design of the Branch-and-Bound framework is provided first. Then we will briefly talk about the problem – $1|r_j|\sum W_jC_j$, and how we implement it with the framework. After that, we will elaborate on labeler design and how we apply the Proximal Policy Optimization (PPO) algorithm using it. Finally, we will discuss the experiments done and present the results.

7.1 Branch-and-Bound Framework – BBGym-RL

The framework aims to wrap a problem solver into a reinforcement learning environment for the model to learn in. Unlike other code that implements Branch-and-Bound, the Branch-and-Bound code is not implemented in one colossal function. Instead, the framework follows the concept of Branch-and-Bound described in Morrison et al. [2016] where Branch-and-Bound is split into searching, branching, and pruning phases. This design makes it easier the substitute different branching and prune strategies into the framework.

The framework is designed to solve a problem as follows:

1. A problem instance is sent into the problem parser, and the root of the search tree is filled.
2. The searching strategy picks up a not-yet-visited node from the tree and examines it
3. The branching strategy splits the node into several subproblems and holds them in a vector
4. The prune strategies prune some of the subproblems out of the vector

5. The search graph is updated, and the process starts from step 2 until there is no more node to be visit

This design makes it easier to precisely split different phases from each other and substitute different strategies into the framework.

Since the problem we are dealing with is $1|r_j|\sum W_jC_j$, the modules for different phases are coded to perform a Branch-and-Bound algorithm that solves this problem.

Our Branch-and-Bound algorithm is a simplified and slightly adjusted version of the one described in Kao et al. [2009]. We will first discuss the problem node configuration and how we implement the searcher, brancher, and pruner, and then we will talk about the labeler, namely, the RL agent.

7.1.1 The problem node configuration

The problem node includes the following information:

- The job index
- The earliest start time of unfinished jobs
- The completion time of current finished jobs
- The weighted completion time of current finished jobs
- The completion status of each job
- The current finished job sequence
- The lower bound of the objective

7.1.2 Searcher design for $1|r_j|\sum W_jC_j$

The searcher determines which node to visit next. For the Cyclic best-first search, several contours are maintained, and a pointer is used to keep track of the position of the current contour. The searcher picks out the top node of the current contour for each search phase. Recall that for the Cyclic best-first search, a priority queue is used to store the nodes in the contour, and the search goes from contour to contour, picking the node with the highest priority in each contour every time it goes to a new contour.

7.1.3 Brancher design for $1|r_j|\sum W_jC_j$

The brancher aims to split a node into several subnodes (subproblems). Since the problem we are solving is a scheduling problem, the brancher branch the node by choosing an unfinished job and assigning it to the end of the current finished job sequence, recomputing the earliest start time of the unfinished jobs, the weighted and unweighted completion time of finished jobs, lower bound, and updating the completion status of each job.

7.1.4 Pruner design for $1|r_j|\sum W_jC_j$

The pruner aims to prune unpromising subproblems out of the vector of subproblems. Here we implement two strategies:

1. Prune the node if the lower bound is larger than the incumbent(best-known) solution
2. If all earliest start times are the same (implying that every job is released), update the incumbent solution with a new feasible solution, calculated using the weighted shortest processing time (WSPT) rule given the current partial sequence. Then prune all subnodes of the currently inspecting node.

We only implemented some possible strategies since solving the problem is not the primary goal of this project. The second pruning strategy is provided in [Bianco and Ricciardelli, 1982]

7.1.5 Best first searching strategy to priority queue ordering

In [Bianco and Ricciardelli, 1982], the earliest-job-first strategy is used as the best-first searching strategy. In our implementation, instead, we set it to be the CBFS measure-of-best function and use it

to order the priority queue. A min-heap is used to implement the priority queue, and the job with the earliest start time always stays at the top of the heap.

7.1.6 The CBFS labeler

The labeler assigns nodes to different contours at the end of the pruning process, where the graph update is performed. Here we name the CBFS labeler being trained as CBFS-net. The RL agent for CBFS-net acts as a labeler to collect the state information and rewards after each node assignment, then updates its policy accordingly. Other CBFS labeling strategies are also implemented for comparisons, including CBFS-level, CBFS-best-first-search, and CBFS-random. For the CBFS-level strategy, the labeler simply assigns the node to contour depending on the length of the finished job sequence. For the CBFS-best-first-searching strategy, the labeler assigns all nodes to the same contour. And for the CBFS-random strategy, the labeler assigns the node to a random contour.

7.2 The Labeler Design

The environment is the whole Branch-and-Bound framework, called from a wrapper. The wrapper provides the RL network with the current state encoding and returns a reward based on the given action, which is the label of the current node. The RL network learns to place each node it sees into a contour that will lead to a better reward – correspondingly the less node explored, the better the reward is.

7.2.1 Inapplicable design: Discrete Heads

The intuitive idea for learning a labeling function might be wrapping a softmax layer outside the actor-network and use the output as the label, in other words, learning the labeling problem as classifying on contours. This design has the following problems:

1. The output size of the softmax layer should be set to be finite and in practice, should not be too large. However, the finite size of the contour brings a strong limit on node placements.
2. We cannot insert a contour between any two contours, which also largely limits the placement of nodes.

7.2.2 Inapplicable design: Continuous Action Space

Another possible practice, considering the definition in Morrison et al. [2017], the actor-network can be designed to output a floating point number as a label, namely, learning the labeler as a regression problem. However, the output of a neural network seldom produces the same value for multiple different inputs. It will be hard for CBFS to behave like a Best-first search (since the agent is unlikely to place different nodes into the same contour). Moreover, since the output is often unidentical, the number of new contour creations is high, resulting in a slow search. (Following the CBFS Branch-and-Bound algorithm proposed in Morrison et al. [2017], the set of contours is implemented as a binary search tree to do fast query given label)

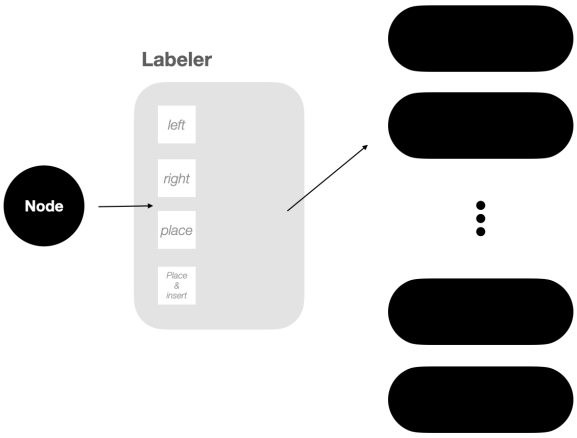
7.2.3 A new interpretation on CBFS that suits better for learning

The interpretation of CBFS in Morrison et al. [2017] is an overkill, considering that *contours* are actually a discrete concept, describing them using continuous labels makes it harder to train with a neural network in practice. There are two main problems:

1. What matters more is the order of contours being searched instead of the label

   For example, with contour configuration such as 1, 2, 3 and the node currently being searched is in contour 3, it does not matter if the subnode is labeled with 4 or 5 if we want it to be the next inspected node. If we label it as 4, the contour configuration will become 1, 2, 3, 4, and by labeling it as 5, the corresponding contour configuration will be 1, 2, 3, 5. However, in these two cases, the orderings of the node search are the same. This creates extra state space for RL, making it hard to train.
2. The CBFS interpretation requires the maintenance of a binary search tree to get the contour with the smallest label at each time.

Figure 3: The Labeler Design for CBFS labeling function



This creates an O(logn) overhead each time for node extraction and an O(nlogn) overhead for contour insertion.

Here we provide another interpretation for CBFS labeling. We can think of CBFS as a placement game with four actions, respectively *left, right, place, insert_and_place* the environment maintains a pointer to the current contour being searched, so we move the pointer to place the node each time.

Notice that we might spend more than one action to decide the contour for one node. However, we can immediately see the benefit:

1. the definite value of the label does not matter anymore
2. we can still insert a contour between any two contours, but we do not have to deal with the continuous action space
3. we can implement this structure with a simple linked list, no more search in binary search trees.

7.2.4 Final design: A Reinterpretation of the CBFS labeling function

In practice, 4 actions are provided, respectively, *left, right, place, insert_and_place*

Several actions can be seen as a breakdown of the original labeling process mentioned in Morrison et al. [2017]. For instance, moving several steps left and then placing it can be seen as labeling the node with a smaller label than the one being labeled last time under the Morrison et al. [2017] settings. Contours can also be created between any two contours, fulfilling the above requirement. When a piece of node information is sent into the labeler, the labeler will move the internal picker to point to the ideal contour. If the action "place" or "place and add" is chosen, the node is placed into the contour, and the labeling process ends.

7.2.5 Action Encoding

We define the action encoding as a vector of four boolean variables, respectively representing "place," "place and add contour," "move left," and "move right." We have mentioned that this design allows the labeler to output labels that fully utilize the flexibility of CBFS. However, we still need to limit the number of contours to a reasonable number. If not, the state encoding for the network will be too large, and the training will be slow.

7.2.6 State Encoding

The encoding for a search state include two parts, *node encoding*, *contour encoding* and *picker encoding* The *node encoding* includes:

- state processed rate (The ratio of completed jobs over the total number of jobs)
- lower bound (The lower bound of the objective of the current subproblem)
- weighted completion time (The current accumulated weighted completion time)
- current feasible solution (The incumbent solution withheld at the time the node is visited)

These values are further normalized before sending into the network.

The *contour encoding* aims to provide a snapshot of the current contour configuration for the network. We represent each contour with the top element of the underlying priority queue, so each contour is encoded using the above four values. The contour encoding is a vector of 4N, where N is the number of contours.

The *picker encoding* encodes the current position of the picker in the game-like labeler, this provides information for the network to decide the strategy for moving the picker.

7.3 Proximal Policy Optimization implementation

Since the Branch-and-Bound framework is written in C++ and training the RL model requires buffering and fast interaction between the environment and actor , it is discouraged to define the model in python and do number of calls from c++ to python in the training stage. (PyTorch and TensorFlow can turn trained models into binary or simple python function calls. However, the feature only supports trained models) , so we used libtorch API for implementation (The c++ version of PyTorch)

The state encoding is a concatenated vector of node encoding and contour encoding. The Deep Deterministic Policy Gradient algorithm (DDPG) was initially tested to train the labeling function. However, DDPG is used mainly for continuous state and action space, and as we mentioned above, continuous action space does not apply to our problem. In this paper, PPO is used to train the labeler.

There is a trade-off between providing ample extendable space for CBFS and making good use of memory. We have to decide on the variable *max_contour_size* to control the size of contour encoding. Setting *max_contour_size* larger can provide possible ordering or search paths, but it also strongly increases the state-action space. In practice, we set the *max_contour_size* at around 10 to avoid over-consumption of memory and to ease the training. In the vanilla setting, PPO is prone to fall into a local minimum. Therefore, we introduce the entropy loss to encourage exploration. Additionally, since steps in the beginning primarily affect the final result, we want to decrease the chance that the network starts with a few inferior actions and then provides uninformative samples to the network. This is done by decreasing the effect of the entropy term as training progresses. In our setting, there is an entropy control term multiplied by the entropy loss. The entropy control term is a linear function that starts at one and ends at zero.

Furthermore, we have tried reward shaping for the training process but only found it made the training more unstable. Therefore, we merely recap the whole process of training at the end of each episode and give a reward based on the final result, and each reward before the final stage is computed backward.

As we mentioned, reward shaping wobbles the training process, so we have to collect the whole trajectory for training, which implies that we have to store the whole trajectory in memory.

8 Performance and Results

To see the practicability of the proposed method, we trained the agent based on different difficulties of the problem instances. We also want to see the performance of the agent on different types of problems. The instances are grouped based on the number of jobs, namely 6-10, 6-20, and 6-25 jobs Here are some points we want to discuss:

- The state-of-the-art strategies for the $1|r_j|\sum W_jC_j$ problems can solve problems with 100+ jobs easily. However, the main goal of this paper is to lessen the number of node visitations, given that there are already a set of branching and pruning methods implemented. Hence, we choose to implement less complex methods to solve the problem.
- The number of jobs does not directly affect the problem's difficulty, and our problem's difficulty emerges from the number of search nodes. Nevertheless, in practice, the number of search nodes will not be known until the problem is solved, so we choose to use the number of jobs as a proxy for the problem's difficulty.

We train different models using the training set generated for different job groups, then test them on the corresponding test set generated likewise. The results are:

- Trained model provides better search strategies than other search strategies.
- The natures of the problems with different numbers of jobs are different. Though there is some generalization, it is still better to train a model for each difficulty
- The training difficulty and required training time grow as the instances' job number increases
- Statistically, using our model can provide 10 percent less node search on average (compared to the Originally used Best-first searching strategy)

Figure 4 shows the training curve of the 6-10 group instances. The x-axis shows the epochs, and the y-axis shows the average searched nodes as the training progresses. We can see that the searched nodes decrease as the training moves onward.

Then Figure 5 shows the CBFS-net results trained and tested on different groups of instances. It shows that CBFS-net outperformed all the other searching strategies when it was trained on the corresponding instance group with the same difficulty (same range on the number of jobs).

Figure 6 presents the experiment we did on the contour size. We trained the three settings under the same number of epochs and chose the model that performed best at validation. In the current training settings, we found that contour size does affect the capability and elasticity of a searching strategy when the maximum number of contours is too small. However, the result also shows the capability of CBFS-net where the maximum number of contours having values 10 and 50 are close. Another hidden disadvantage of setting the maximum number of contours too large is that it blows up the time needed for training. In general, we recommend setting this value small.

8.1 The training process of PPO

The proposed PPO is implemented using C++ and PyTorch-C++ on an M1 Mac-mini (2020 model) with 16GB RAM, only the CPU is used for training, and no GPU acceleration is applied. The problem instances are generated from a discrete uniform distribution with the help of numpy.random. The processing time, weight, and release time of the $1|r_j|\sum W_jC_j$ are limited to under 50. The parameter settings are listed.

8.2 Comparisons with other search strategies

We see from the result that, after training, the search strategy provides a higher winning rate than the other listed search strategies. It is also better to train different models for different difficulties.

9 Conclusion

9.1 Application of the method

In this paper, we have provided a working example of implementing CBFS Branch-and-Bound. Moreover, we have successfully observed decreases in the number of nodes visited following the node search ordering learned by the network. The value of this is that, often time, to decrease the number of nodes searched, computer scientists might dive into the problem and develop pruning methods, or use branching methods such as strong branching to improve the search further. Given a well-designed Best-first search Branch-and-Bound method, we can turn it into a CBFS Branch-and-Bound algorithm to seek a more efficient path to guide the search, using the original prioritization as the contour

Figure 4

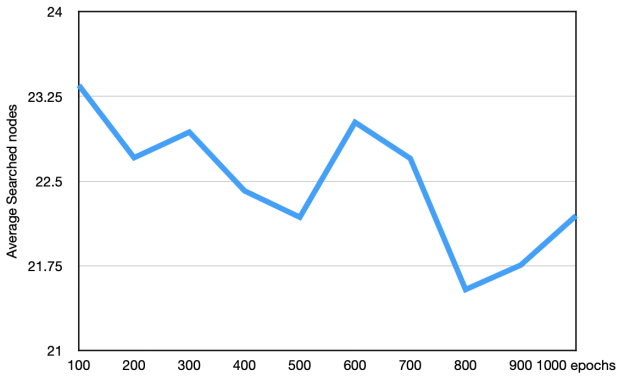


Figure 5

Winning Rate on different searching strategies w.r.p different training and testing combination

	Net	Best-first-search	level-CBFS	Rand-CBFS
	Small (6-10 jobs)			
Small-on-small	**46%**	18%	24%	12%
Small-on-semi-med	**34%**	8%	28%	30%
Small-on-med	28%	6%	32%	**34%**
Small-on-big	24%	10%	30%	**36%**
	Semi-med (6-20 jobs)			
Semi-med-on-small	**34%**	20%	28%	18%
Semi-med-on-semi	**52%**	0%	18%	30%
Semi-med-on-med	22%	6%	**38%**	34%
Semi-med-on-big	22%	10%	**34%**	34%
	Med (6-25 jobs)			
Med-to-small	**32%**	22%	30%	16%
Med-on-semi	28%	14%	**30%**	28%
Med-on-med	**34%**	16%	26%	24%
Med-on-big	18%	8%	**38%**	36%

Figure 6

Winning rate on different contour size (trained to convergence)

Contour size	Net	Best-first-search	level-CBFS	Rand-CBFS
5	**40%**	18%	26%	16%
10	**44%**	6%	34%	16%
50	**44%**	6%	34%	16%

Figure 7

Hyper-parameters

Hyper-Parameters	
max_num_contour	10
epoch_per_instance	10
epoch_per_update	5
entropy_lambda	1
lr_pi	0.000003
lr_q	0.00003
steps_per_epoch	100000
hidden_dimension	64
clip_ratio	0.2
target_kl	0.001

Figure 8

Winning Rate on different searching strategies w.r.p different training and testing combination-1

	CBFS-net	Best-first-search	CBFS-level	CBFS-rand	Best/Worst Gap
6-10 jobs	**46%**	8%	32%	14%	15.4%
11-15 jobs	**40%**	4%	38%	18%	14.3%
16-20 jobs	30%	12%	34%	24%	6%

measure-of-best function. The cost of this method is meager since no new theorem is needed to prove and the result is, on average better.

This method could be an additional improvement or a fine-tuning technique on an existing solver engine. Pruning or branching methods often require knowledge about the specific problem to come up with, while this method needs none of them. It is exciting to see if there's more to dig into from this perspective of improvement.

Another possible application of this model is for the exploration of CBFS strategies. There needs to be more research on solid and simple CBFS labeling functions. Since we now know that CBFS-net provides a better-searching strategy than CBFS-net and CBFS-random. Current research can focus on analyzing the internal behavior of CBFS-net and examining the reason for its winning, hopefully reducing it to a more elegant but powerful labeling function.

9.2 The Branch-and-Bound framework

The Branch-and-Bound framework we use in this project can be found on GitHub: BBGym-RL Though currently, there is only one problem type provided, we hope that it eventually supports more problems, such as Traveling salesman problems, parallel machine scheduling problems, or job-shop, flow-shop problems.

References

David R Morrison, Sheldon H Jacobson, Jason J Sauppe, and Edward C Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, 19:79–102, 2016.

David R Morrison, Jason J Sauppe, Wenda Zhang, Sheldon H Jacobson, and Edward C Sewell. Cyclic best first search: Using contours to guide branch-and-bound algorithms. *Naval Research Logistics (NRL)*, 64(1):64–82, 2017.

Gio K Kao, Edward C Sewell, and Sheldon H Jacobson. A branch, bound, and remember algorithm for the 1| ri| ti scheduling problem. *Journal of Scheduling*, 12(2):163–175, 2009.

Wenda Zhang, Jason J Sauppe, and Sheldon H Jacobson. Comparison of the number of nodes explored by cyclic best first search with depth contour and best first search. *Computers & Operations Research*, 126:105129, 2021.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

Giovanni Di Liberto, Serdar Kadioglu, Kevin Leo, and Yuri Malitsky. Dash: Dynamic approach for switching heuristics. *European Journal of Operational Research*, 248(3):943–953, 2016.

Lucio Bianco and Salvatore Ricciardelli. Scheduling of a single machine to minimize total weighted completion time subject to release dates. *Naval Research Logistics Quarterly*, 29(1):151–167, 1982.