A Learning Based Branch and Bound for Maximum Common Subgraph Related Problems

Yanli Liu,[1,4,5] **Chu-Min Li,**[2*] **Hua Jiang,**[3†] **Kun He**[1‡]
[1]Huazhong University of Science and Technology, China
[2]MIS, Université de Picardie Jules Verne, France
[3]Yunnan University, China
[4]WuHan University of Science and Technology, China
[5]State Key Lab. for Novel Software Technology, Nanjing University, China
yanlil2008@163.com, chu-min.li@u-picardie.fr, huajiang@ynu.edu.cn, brooklet60@hust.edu.cn

Abstract

The performance of a branch-and-bound (BnB) algorithm for maximum common subgraph (MCS) problem and its related problems, like maximum common connected subgraph (MCCS) and induced Subgraph Isomorphism (SI), crucially depends on the branching heuristic. We propose a branching heuristic inspired from reinforcement learning with a goal of reaching a tree leaf as early as possible to greatly reduce the search tree size. Experimental results show that the proposed heuristic consistently and significantly improves the current best BnB algorithm for the MCS, MCCS and SI problems. An analysis is carried out to give insight on why and how reinforcement learning is useful in the new branching heuristic.

Introduction

A graph is a logic model to describe a set of objects and the relationship of the objects abstracted from real-world applications. Given two graphs G and H, it is often crucial to determine the similarity of G and H. The Maximum Common induced Subgraph (MCS) problem was introduced for this purpose, consisting in finding a graph with as many vertices as possible that is isomorphic to an induced subgraph in both G and H. If we require the induced subgraph to be connected, then it is called the Maximum Common Connected induced Subgraph (MCCS). They are NP-hard and widely occurs in applications such as image or video analysis (Bunke and Messmer 1995; Liu and Lee 2001), information retrieval (Cao, Yang, and Wang 2011), biochemistry (Giugno et al. 2013; Cootes, Muggleton, and Sternberg 2007; Faccioli et al. 2005), pattern recognition (Solnon et al. 2015; Conte et al. 2008; Cordella et al. 2004) and cheminformatics (Raymond, Gardiner, and Willett 2002). A related problem called induced Subgraph Isomorphism (SI) is to decide whether a small graph G occurs inside a graph H.

Many approaches have been designed to address MCS and MCCS, see, e.g, (McCreesh et al. 2016; Mcgregor 2010; Bahiense et al. 2012; Levi 1973; Hoffmann, McCreesh, and Reilly 2017; Lodi and Zarpellon 2017). Some techniques of constraint programming for filtering domain are used in SI algorithms (Solnon 2010; Cordella et al. 2004; Zampelli, Deville, and Solnon 2010; Bonnici and Giugno 2017). In this paper, we focus on the branch-and-bound (BnB) scheme for the MCS and extend it to the MCCS and SI. Examples of existing BnB algorithms for the MCS can be found in (Mcgregor 2010; Ndiaye and Solnon 2011; McCreesh, Prosser, and Trimble 2017).

It is well-known that the performance of a BnB algorithm crucially depends on its branching heuristic. The branching heuristic in McSplit (McCreesh, Prosser, and Trimble 2017), the current best BnB algorithm for the MCS, aims at minimizing the number of branches for the current branching point and uses vertex degree to rank the vertices. Unfortunately, due to the NP-hardness of the MCS, the BnB search tree size may not be predictable in general using vertex degree or any other static feature of a graph.

In this paper, we propose to use reinforcement learning (RL) to discover which branching choice yields the greatest reduction of the search tree by trying them during the search. Specifically, we consider the BnB algorithm as an agent and each branching choice as an action. When the agent takes an action, it receives a reward determined by the consequence of this action, which in our context is the reduction of the search space. The score of the action depends on the accumulated rewards it received in the past. Then, at every branching point, the agent selects an action with the greatest score to branch on.

We implement our branching heuristic on top of McSplit, and design a new algorithm called McSplit+RL. The new algorithm is extensively evaluated on 24,761 MCS instances from diverse applications (biochemical reaction, images analysis, 3D, 4D objects, complex networks), including the large instances used in (McCreesh, Prosser, and Trimble 2017) for evaluating McSplit. Since McSplit is already highly efficient, most instances are very easy for both McSplit and McSplit+RL, and can be solved by all tested algorithms within 10s. We exclude these easy instances that do not need learning to be solved, and the too hard instances that none of the tested algorithms can solve within the time

*The first two authors contribute equally to this paper.
†Corresponding author.
‡Corresponding author.

limit. There remain 2790 instances. Empirical results show that McSplit+RL solves 7.1% instances more than McSplit. Considering the high performance of McSplit and the notorious difficulty of MCS, these results show the effectiveness of combining reinforcement learning in designing branching heuristic for the BnB search. We also use the new branching heuristic to solve the SI and MCCS problems. Empirical results show that it is also effective.

We also carry out an empirical analysis to give insight on why and how the learning approach is effective, suggesting that the performance of McSplit+RL is due to its more diversified search allowing it to find an optimal solution earlier than McSplit.

Problem Definition

Consider a simple (unweighted, undirected), labelled graph $G = (V, E, \lambda)$,where V is a finite set of vertices, $E \subseteq V \times V$ is a set of edges, and λ is a label function that assigns to each vertex $v \in V$ a label value $\lambda(v)$. If the labels are the same for all vertices, then the labelled graph is reduced to an unlabelled graph. Two vertices u and v are adjacent iff $(u, v) \in E$. The degree of a vertex v is the number of its adjacent vertices. A subset $V' \subseteq V$ induces a subgraph $G[V'] = (V', E', \lambda')$ of G, where $E' = \{(u, v) \in E | u, v \in V'\}$, and $\forall v \in V'$, $\lambda'(v) = \lambda(v)$.

Given a pattern graph $G_p = (V_p, E_p, \lambda_p)$ and a target graph $G_t = (V_t, E_t, \lambda_t)$, the Maximum Common induced Subgraph (MCS) problem is to find a subset $V_p' \subseteq V_p$ and a subset $V_t' \subseteq V_t$ of the greatest cardinality and a bijection $\phi : V_p' \to V_t'$ such that: (1) $|V_p'| = |V_t'|$, (2) $\forall v \in V_p'$, $\lambda_p(v) = \lambda_t(\phi(v))$, and (3) given any $v, v' \in V_p'$, v and v' are adjacent in G_p if and only if $\phi(v)$ and $\phi(v')$ are adjacent in G_t. In other words, the MCS is to find a maximum subgraph $G_p[V_p']$ and a maximum subgraph $G_t[V_t']$ such that $G[V_p']$ and $G_t[V_t']$ are isomorphic. $G_p[V_p']$ or $G_t[V_t']$ is called a *maximum common induced subgraph* of G_p and G_t. The vertex pair $(v, \phi(v))$ is called a *match*.

Let $V_p' = \{v_1, v_2, \ldots, v_{|V_p'|}\}$, an optimal solution of the MCS is denoted as a set of matches $\{(v_1, \phi(v_1)), (v_2, \phi(v_2)), \ldots, (v_{|V_p'|}, \phi(v_{|V_p'|})\}$.

A variant of MCS called Maximum Common Connected induced Subgraph (MCCS) problem requires that the maximum common induced subgraph is connected. Another variant called induced Subgraph Isomorphism (SI) requires $V_p' = V_p$.

Branch and Bound for MCS

Given two graphs G_p and G_t, the BnB algorithm depicted in Algorithm 1 gradually constructs and proves an optimal solution using a depth-first search. During the search, the algorithm maintains two variables: $curSol$, the solution under construction; and $maxSol$, the best solution found so far. In addition, every vertex v of G_p is associated with a subset $\alpha(v)$ of vertices of G_t that can be matched with v. In the beginning, $curSol$ and $maxSol$ are initialized with $\emptyset$, and $\alpha(v)$ is initialized to be the set of all vertices of G_t having the same label as v. The call of MCS$(G_p, G_t, \alpha, \emptyset, \emptyset)$ returns a maximum common subgraph of G_p and G_t.

At each branching point, the algorithm first calls overestimate(G_p, G_t, α) to compute an upper bound UB on the cardinality of the best possible solution that can be found from this branching point. It then compares UB with $maxSol$. If UB $\leq |maxSol|$, a solution better than $maxSol$ cannot be found from this branching point, and the algorithm prunes the current branch and backtracks. Otherwise, it selects a not-yet matched vertex v from G_p and tries to match v with every vertex w in $\alpha(v)$, and updates α accordingly before it recursively calls MCS$(\cdot)$. As a consequence of matching v with w, (v, w) is added into $curSol$, and for each not-yet matched v' of G_p, $\alpha(v')$ is updated as follows: If v' is adjacent to v, remove all vertices non-adjacent to w from $\alpha(v')$; otherwise, remove all vertices adjacent to w from $\alpha(v')$.

Note that after updating $\alpha(v')$, $\forall w' \in \alpha(v')$, v is adjacent to v' in G_p iff w is adjacent to w' in G_t, so that the match (v', w') can be further added into $curSol$. If a solution better than $maxSol$ is found in a leaf of the search tree, the algorithm updates $maxSol$ by $curSol$ before backtracking.

Algorithm 1 MCS$(G_p, G_t, \alpha, curSol, maxSol)$

Input: $G_p = (V_p, E_p, \lambda_p)$, the pattern graph; $G_t = (V_t, E_t, \lambda_t)$, the target graph; α, a mapping $V_p \mapsto 2^{V_t}$; $curSol$, the solution under construction; $maxSol$, the best solution found so far.
Output: $maxSol$

```
1: if ∀v ∈ V_p, α(v) = emptyset then
2:    if |curSol| > |maxSol| then
3:       maxSol ← curSol;
4:    end if
5:    return maxSol;
6: end if
7: UB ← |curSol| + overestimate(G_p, G_t, α);
8: if UB ≤ |maxSol| then
9:    return maxSol;
10: end if
11: v ← a vertex from G_p such that α(v) ≠ ∅;
12: for each vertex w in α(v) do
13:    α' ← α; α'(v) ← ∅;
14:    for each vertex v' in G_p do
15:       remove w from α'(v');
16:       if v' is adjacent to v in G_p then
17:          remove the vertices non-adjacent to w in G_t
             from α'(v');
18:       else
19:          remove the vertices adjacent to w in G_t from
             α'(v');
20:       end if
21:    end for
22:    maxSol ← MCS(G_p, G_t, α',
                    curSol ∪ {(v, w)}, maxSol);
23: end for
24: α(v) ← ∅;
25: return MCS(G_p, G_t, α, curSol, maxSol);
```

An important issue for implementing Algorithm 1 is how

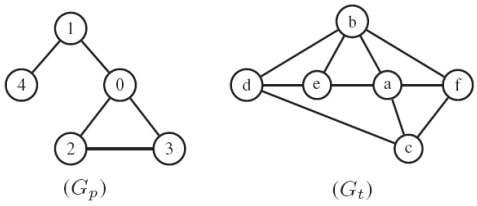

Figure 1: An example for the MCS problem.

to implement α, which determines how to select a vertex v in line 11 and how to design the overestimate$(\cdot)$ function. A natural way is to explicitly create a list of vertices of G_t for each vertex v of G_p. With this implementation, Ndiaye and Solnon (2011) represent each vertex of G_p as a variable whose domain is a set of vertices of G_t. Then, they select a vertex with the smallest domain in line 11, and use a soft all-different constraint in the overestimate$(\cdot)$ function to compute a bound. The difficulty in this implementation of α is that given a vertex w in G_t, it is not straightforward to know the number of variables whose domain contains w. If the domain of a vertex v is the smallest and contains a vertex w of G_t, but w also occurs in the domain of many other vertices of G_p, branching on v may not be the best choice to minimize the search tree size.

McCreesh *et al.* (2017) use an elegant way to represent α based on the fact that many vertices of G_p have the same domain during the search. Thus the vertices of G_p having the same α value should be put together to have a compact representation of α. The following example illustrates this representation and its use in Algorithm 1.

Example 1 *Figure 1 shows two undirected and unlabelled graphs G_p and G_t, V_p = $\{0, 1, 2, 3, 4\}$, V_t = $\{a, b, c, d, e, f\}$. Initially, $\alpha(v) = \{a, b, c, d, e, f\}$ for each vertex v of G_p, represented using the pair $\{\langle(0, 1, 2, 3, 4), (a, b, c, d, e, f)\rangle\}$.*

Then, vertex 0 is chosen in line 11 for branching. The first match added into $curSol$ is $(0, a)$. Consequently, $(1, 2, 3, 4)$ is split into $(1, 2, 3)$ and (4), and (b, c, d, e, f) is split into (b, c, e, f) and (d), because vertices 1, 2 and 3 are adjacent to the matched vertex 0, while vertex 4 is not; and vertices b, c, e and f are adjacent to the matched vertex a, while vertex d is not. The updated α is then represented by $\{\langle(4), (d)\rangle, \langle(1, 2, 3), (b, c, e, f)\rangle\}$, meaning $\alpha(4) = \{d\}$ and $\alpha(1) = \alpha(2) = \alpha(3) = \{b, c, e, f\}$.

Note that the splitting of $(0, 1, 2, 3, 4)$ and (a, b, c, d, e, f) is equivalent to removing b, c, e and f from $\alpha(4)$, and d from $\alpha(1)$, $\alpha(2)$ and $\alpha(3)$.

More generally, a pair $\langle V'_p, V'_t\rangle$ is called a label class in (McCreesh, Prosser, and Trimble 2017), where V'_p (V'_t) is a subset of V_p (V_t), meaning that $\alpha(v) = V'_t$ for each $v \in V'_p$. Let D be the set of label classes. When a new match (v, w) is added into $curSol$, Algorithm 1 splits each label class $\langle V'_p, V'_t\rangle$ in D into two label classes $\langle V'_{1p}, V'_{1t}\rangle$ and $\langle V'_{2p}, V'_{2t}\rangle$ in lines 14 – 21, so that the vertices in V'_{1p} (V'_{1t}) are all adjacent to v (w) and the vertices in V'_{2p} (V'_{2t}) are all non-adjacent to v (w). Note that V'_{1p} and V'_{2p}, as well as V'_{1t} and V'_{2t}, are disjoint.

This representation of α enables the following branching heuristic and bound computation in (McCreesh, Prosser, and Trimble 2017).

- Given a label class $\langle V'_p, V'_t\rangle$, there are $|V'_p| \times |V'_t|$ matches to try. So, McCreesh *et al.* first select a label class such that $\max(|V'_p|, |V'_t|)$ is the smallest and then select a vertex with the greatest degree in V'_p in line 11 for branching, which is similar in spirit to choosing a label class $\langle V'_p, V'_t\rangle$ with the smallest $|V'_p| \times |V'_t|$ and then breaking ties using vertex degrees. This heuristic is better than the heuristic in (Ndiaye and Solnon 2011) consisting in selecting a label class with the smallest $|V'_t|$.
- Let D be the set of label classes at a branching point. A label class $\langle V'_p, V'_t\rangle$ can offer at most $\min(|V'_p|, |V'_t|)$ matches to $curSol$. So, the overestimate(G_p, G_t, α) function in (McCreesh, Prosser, and Trimble 2017) computes and returns $\sum_{\langle V'_p, V'_t\rangle \in D} \min(|V'_p|, |V'_t|)$, which is equivalent to the bound given by the soft all-different constraint in (Ndiaye and Solnon 2011) but is simpler to compute.

Nevertheless, the branching heuristic in (McCreesh, Prosser, and Trimble 2017) depends heavily on vertex degrees and may not result in the smallest search tree. In the next section, we will propose a new branching heuristic inspired by reinforcement learning.

Learning Rewards for Branching

Reinforcement learning is recently a rapid growing research area. In reinforcement learning, we have agents in an uncertain environment. Each agent is a learner and decision maker who needs to discover which actions yield the most reward by trying them, and the goal is to maximize a numerical reward signal. Specifically, the agent interacts with the environment by taking actions and observing the impact of its actions to the environment. When the agent takes an action, it receives a reward related to its goal from the environment. It has a value function that transforms the cumulative rewards it received over time to a score of the action, representing a prediction of rewards in the future for this action. So, the agent should take an action with the maximum score among all available actions at each step so as to achieve its goal. See (Sutton and Barto 2018) for a comprehensive presentation of reinforcement learning.

Although the principle of reinforcement learning is simple and intuitive, it is not easy to define the reward of an action and the value function because a small change in the reward and the value function can have a drastic negative effect for the agent to achieve its goal. So, the definition of the reward and the definition of the value function are the key issues in applying reinforcement learning scheme to solve an NP-hard problem.

In order to solve the MCS and its related problems using the reinforcement learning scheme, we regard Algorithm 1 as an agent. It has a goal of reaching a search tree leaf as early as possible so as to reduce the search tree size as much as possible. The agent needs to successively select and add

a match (v, w) into $curSol$. However, it usually has many choices of (v, w) at each step and does not know which choice is better. So, we regard each choice of (v, w) as an action. Then, it remains to define the reward of the action (v, w) and the value function.

There are many possibilities to define the reward of the action (v, w) and the value function. Below we present three different definitions, each with a particular consideration.

Reward based on the reduction of UB

As can be seen in Algorithm 1, the algorithm reaches a leaf when UB $\leq |maxSol|$. So, reducing UB quickly allows to reach a leaf quickly. Thus, we define the reward R for an action (v, w) to be the reduction of UB after taking this action. Specifically, let D be the set of label classes before taking an action (v, w) and D' the set of label classes obtained by splitting the label classes in D according to their adjacency to v and w in lines 14 – 21 of Algorithm 1. $R(v, w)$ can be quickly computed as follows.

$$R(v,w) = \sum_{\langle V'_p, V'_t\rangle \in D} \min(|V'_p|, |V'_t|) - \sum_{\langle V''_p, V''_t\rangle \in D'} \min(|V''_p|, |V''_t|)$$

Our value function maintains a score $S_p(v)$ ($S_t(w)$) for each vertex $v \in V_p$ ($w \in V_t$), which are initialized to 0. Each time $R(v, w)$ is computed, $S_p(v)$ and $S_t(w)$ are updated as follows:

$$S_p(v) \leftarrow S_p(v) + R(v, w)$$
$$S_t(w) \leftarrow S_t(w) + R(v, w)$$

At each branching point (line 11 of Algorithm 1), we first select a label class $\langle V'_p, V'_t\rangle$ with the smallest $\max(|V'_p|, |V'_t|)$, and a vertex v in V'_p with the greatest score $S_p(v)$. Then, for each w in V'_t in the decreasing order of score $S_t(w)$, we match v and w, and recursively continue the search after adding match (v, w) into $curSol$. All ties are broken in favor of a vertex with the maximum degree.

Using $S_p(v)$ and $S_t(w)$ to select a branching match (v, w) in Algorithm 1 in this way, we obtain McSplit+RL, which is implemented on top of McSplit.

Reward based on the reduction rate of UB

The reward $R(v, w)$ defined in the previous subsection is based on the reduction of UB. For example, when UB is reduced by 10 by matching v to w, $R(v, w)$ is 10. However, a reduction of 10 does not has the same signification when UB is 100 or when UB is 1000. In order to take into account the magnitude of UB in the reward, we define $R_{Rate}(v, w)$ based on the reduction rate of UB.

$$R_{Rate}(v,w) = R(v,w) / \sum_{\langle V'_p, V'_t\rangle \in D} \min(|V'_p|, |V'_t|)$$

The value function is defined as follows.

$$S_{p\_Rate}(v) \leftarrow S_{p\_Rate}(v) + R_{Rate}(v, w)$$
$$S_{t\_Rate}(w) \leftarrow S_{t\_Rate}(w) + R_{Rate}(v, w)$$

As $S_p(v)$ and $S_t(w)$, $S_{p\_Rate}(v)$ and $S_{t\_Rate}(w)$ are initialized to 0. Using $S_{p\_Rate}(v)$ and $S_{t\_Rate}(w)$ instead of $S_p(v)$ and $S_t(w)$ to select a branching match in McSplit+RL, we obtain a variant McSplit+RL$_{Rate}$.

Computing the score of a match (v, w)

The reward $R(v, w)$ can also be used to compute the score of the match (v, w).

$$S_{Pair}(v, w) \leftarrow S_{Pair}(v, w) + R(v, w)$$

$S_{Pair}(v, w)$ is initialized to 0 for each pair (v, w). Using $S_p(v)$ to select v as in McSplit+RL, and for each w in V'_t in the decreasing order of score $S_{Pair}(v, w)$, instead of $S_t(w)$, select the branching match (v, w), we obtain another variant McSplit+RL$_{Pair}$.

Empirical Evaluation

All experiments were performed on Intel Xeon CPUs E5-2680 v4@2.40 gigahertz under Linux with 4G memory. The cutoff time is 1800 seconds for each instance. We first present the algorithms (also called solvers) and the benchmarks used in the experiments, then present and analyze the experimental results.

Solvers

• McSplit (McCreesh, Prosser, and Trimble 2017): An implementation of Algorithm 1 using the label class representation of the α mapping. It is an order of magnitude faster than the previous state-of-the-art for unlabelled and undirected MCS instances. It can also sovle labelled graphs.

• McSplit↓ (McCreesh, Prosser, and Trimble 2017): A variant of McSplit using a top-down strategy to call the main McSplit method to search for a solution of cardinality $k = |V_p|, k-1, k-2, \ldots$ and backtracks when the bound is strictly less than the required cardinality, and terminates when a solution of the required cardinality is found. This strategy is similar to the k↓ algorithm (Hoffmann, McCreesh, and Reilly 2017). McSplit↓ is specially designed for the large subgraph isomorphism instances for which McSplit is beaten by k↓.

• McSplit_SBS (Archibald et al. 2019): A variant of McSplit using biased value-ordering and nogood recording with restarts.

• McSplit+RL: Our implementation of Algorithm 1 on top of McSplit with reinforcement learning using $R(v, w)$, $S_p(v)$ and $S_t(w)$ to select a branching match (v, w)

• McSplit+RL↓: A variant of McSplit+RL using the top-down strategy of McSplit↓.

• McSplit+RL$_{Pair}$: A variant of McSplit+RL that uses $S_p(v)$ to select v and $S_{Pair}(v, w)$ to select w when selecting the branching match (v, w).

• McSplit+RL$_{Rate}$: A variant of McSplit+RL using reward R_{Rate} and value functions S_{p_Rate} and S_{t_Rate}, instead of R, S_p and S_t, to select the branching match (v, w).

• McSplit_SI: An implementation of McSplit adapted for the SI problem. Given a pattern graph G_p and a target graph G_t, it backtracks as soon as it discovers that a vertex of G_p cannot be matched with any vertex of G_t (i.e., $UB < |V_p|$).

• McSplit+RL_SI: An implementation of McSplit+RL adapted for the SI problem in the same manner as McSplit_SI.

• Glasgow (Archibald et al. 2019): A state-of-the-art algorithm for the SI problem using biased value-ordering and nogood recording with restart.

• McSplit+RL_SI_NG: An implementation of McSplit+RL_SI with nogood recording (Lee, Schulte, and Zhu 2016; Lecoutre et al. 2007) as in Glasgow for the SI problem.

Benchmark datasets

The benchmark datasets include 24,761 instances, divided into two sets.

• Biochemical reactions instances describing the biochemical reaction networks from biomodels.net [1]. All the 136 graphs are directed, unlabelled bipartite graphs having 9 to 386 vertices. Every pair of graphs gives an MCS instance, resulting in 9316 *Bio* instances (including 136 self-match pairs).

• Large subgraph isomorphism and MCS instances (Hoffmann, McCreesh, and Reilly 2017; McCreesh, Prosser, and Trimble 2017). This benchmark dataset includes real-world graphs and graphs generated using random models, such as segmented images, modelling 3D objects, and scale-free networks [1]. Pattern graphs range from 4 vertices to 900, and target graphs range from 10 vertices to 6,671. There are totally 15,445 instances, including: 6278 *images* instances from images-CVIU11 (43 pattern graphs and 146 target graphs, each pair of pattern graph and target graph resulting in an instance); 1225 *LV* instances given by each pair of graphs (including two identical graphs) among the 49 graphs selected in (McCreesh, Prosser, and Trimble 2017) from the LV dataset; 3430 *largerLV* instances from the above 49 LV graphs as the pattern and the remaining 70 graphs as the target in the LV dataset; 3018 *Mesh* (6 pattern graphs and 503 target graphs), 24 *PR15*, 200 *phase*, 100 *Scalefree* and 1170 *Si* instances used in (McCreesh, Prosser, and Trimble 2017).

Performance of the proposed approach

MCS problem. Figure 2 compares McSplit, McSplit_SBS, McSplit+RL, McSplit+RL$_{Pair}$, and McSplit+RL$_{Rate}$ on the 24,761 instances. We exclude the too easy instances that are solved by all the five solvers within 10s (The average runtimes of McSplit and McSplit+RL on these easy instances are 0.43s and 0.46s, respectively) and the too hard instances that cannot be solved by any solver within the time limit. There remain 2,790 instances in Figure 2 to show the ability difference of these solvers. Note that easy instances do not need any learning to be solved. So, it is natural that learning does not make difference for easy instances.

The three solvers with reinforcement learning (RL) solve more instances than the two solvers without RL, showing the robustness of RL in solving MCS. In particular, McSplit+RL solves 140 more instances than McSplit. Note that McSplit+RL and McSplit share the same implementation and their only difference is RL. Considering the high performance of McSplit and the notorious difficulty of MCS, the results indicate that RL is very effective to solve MCS.

Among the three solvers with RL, the simplest one, McSplit+RL, is the best. The fact that McSplit+RL is better than McSplit+RL$_{Pair}$ suggests that it is more effective to consider the individual impact of a vertex than the impact of a pair of vertices.

In order to test the compatibility of RL with the top-down strategy in McSplit↓, we compare McSplit+RL↓ and

[1] Available at http://liris.cnrs.fr/csolnon/SIP.html

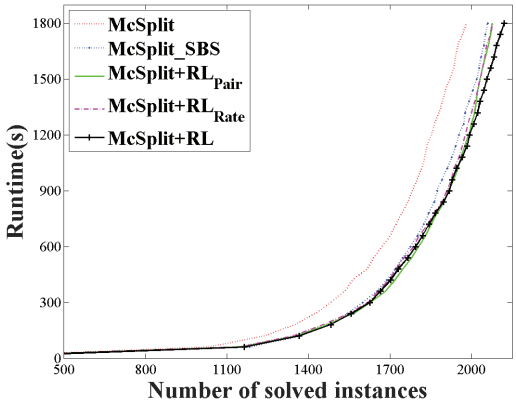

Figure 2: Cactus plot of the solvers of MCS on 2,790 MCS instances after excluding too easy and too hard instances.

McSplit↓ on the 24,761 MCS instances in Figure 3, by excluding the too easy instances that both solvers solve within 10s (The average runtimes of McSplit↓ and McSplit+RL↓ on these easy instances are 0.43s and 0.51s, respectively) and the too hard instances that none of the two solvers solves within the time limit. Note that McSplit↓ and McSplit+RL↓ also share the same implementation, but differs in the branching heuristic. McSplit+RL↓ solves 101 more instances than McSplit, showing that RL is compatible with the top-down strategy.

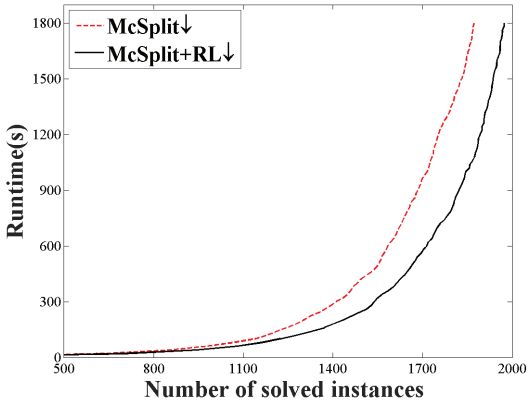

Figure 3: Cactus plot of two solvers of MCS on 2,288 instances after excluding too easy and too hard instances.

MCCS problem. We run McSplit, McSplit+RL, McSplit+RL$_{Pair}$, and McSplit+RL$_{Rate}$ on 15,445 MCS instances, but require the solvers to output a maximum common connnected induced subgraph (excluding the Biochemical reaction instances because McSplit does not support directed graphs for MCCS problem). McSplit+RL solves 3.0% more instances than McSplit, As illustrated in Figure 4 after excluding the too easy instances that are solved by all the four solvers within 10s (The average

runtimes of McSplit and McSplit+RL on these easy instances are 1.09s and 1.35s, respectively) and the too hard instances that none of the four solvers within the time limit. The results show the effectiveness in solving the MCCS problem.

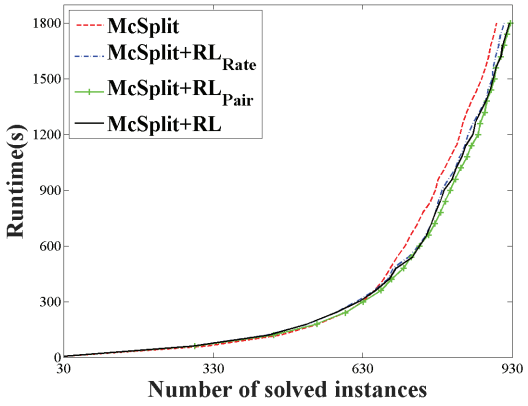

Figure 4: Cactus plot of four solvers of MCCS on 966 instances after excluding too easy and too hard instances.

SI problem. We applied our learning strategy to the SI problem to solve 14,220 instances by excluding the directed graph Bio and the too simple instance dataset LV from the 24,761 instances. We run McSplit_SI, McSplit+RL_SI, Glasgow and McSplit+RL_SI_NG to solve these instances. Figure 5 shows the results after excluding the too easy instances that all the four solvers solve within 10s (The average runtimes of McSplit_SI and McSplit+RL_SI on these easy instances are both 0.12s.) and the too hard instances that none of the four solvers solves within the time limit. McSplit+RL_SI (McSplit+RL_SI_NG) solved 7.6% (7.3%) more instances than McSplit_SI (Glasgow). Note that Glasgow is a solver dedicated for solving the SI problem using the nogood recording technique. The adding of RL and the nogood recording technique of Glasgow in McSplit_SI makes McSplit+RL_SI_NG substantially better than Glasgow, showing the effectiveness of our learning technique for the SI problem. This result also shows the compatibility of RL with the nogood recording technique of Glasgow.

Further Analysis

The experimental results presented in the previous subsection show that our reinforcement learning approach consistently improves McSplit for the MCS, MCCS and SI problems. We believe that the performance of McSplit+RL is due to the fact that it is able to find an optimal solution earlier than McSplit, so that it can prune the search more easily in line 8 of Algorithm 1. To confirm this, we divide the search of McSplit or McSplit+RL into two phases. In phase 1, McSplit or McSplit+RL finds an optimal solution s. Then, in phase 2, McSplit or McSplit+RL proves the optimality of s by showing that there exists no better solution. So, we distinguish the time for finding an optimal solution in phase 1 and the total solving time in the two phases.

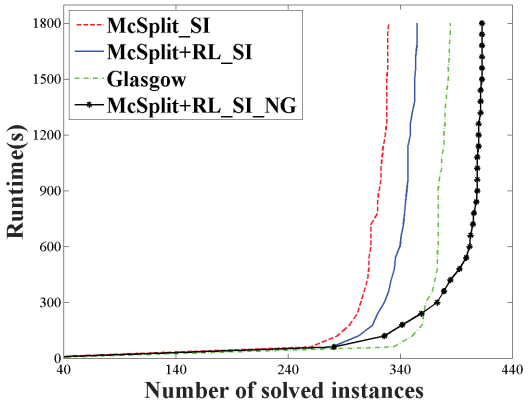

Figure 5: Cactus plot of four solvers of SI on 431 instances after excluding too easy and too hard instances.

Figure 6 compares McSplit and McSplit+RL in terms of the finding time and the total solving time. The two curves of McSplit form the same shape as the two curves of McSplit+RL, suggesting that the performance of McSplit+RL compared with McSplit is due to the ability of McSplit+RL to find an optimal solution quickly.

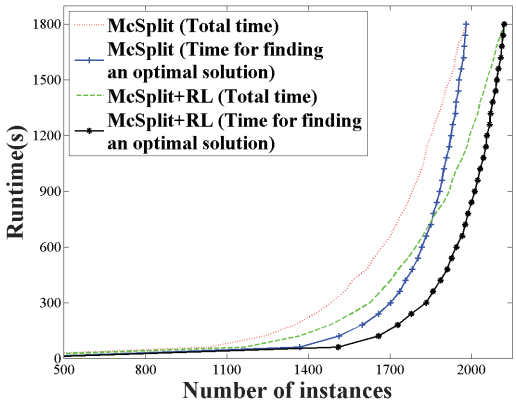

Figure 6: Cactus plot of McSplit and McSplit+RL for MCS on the same set of the 2,790 instances as in Figure 2. A point (x, y) in a curve means x instances for which an optimal solution is found within y seconds or for which the total solving time (phase 1 + phase 2) is within y seconds.

So, the question is: why McSplit+RL can find an optimal solution earlier than McSplit in general? An explanation is that reinforcement learning allows more diversified search. In fact, McSplit always selects the vertex v of the maximum degree in the label class $\langle V'_p, V'_t \rangle$ with the smallest $\max(|V'_p|, |V'_t|)$, concentrating the branching on a small subset of vertices with high degree. On the contrary, a vertex with low degree can have high score in McSplit+RL and can be selected in a branching point. Therefore, more vertices can participate in branching in McSplit+RL to allow more

diversified search.

Let $b_p(v)$ ($b_t(w)$) denote the number of times a vertex v in the pattern graph G_p (w in the target graph G_t) is used for branching at line 11 (line 12) of Algorithm 1. Table 1 compares the standard deviation of $b_p(v)$ ($b_t(w)$) when McSplit and McSplit+RL solves a set of instances randomly selected from the subsets. This standard deviation with McSplit+RL is significantly smaller than with McSplit in most cases, confirming that more vertices participate in branching in McSplit+RL than in McSplit.

Table 1: Comparison of the standard deviation of $b_p(v)$ (and $b_t(w)$ in parentheses) in McSplit and McSplit+RL, where $b_p(v)$ is the number of times (in 10^5) a vertex v in G_p is selected for branching. $|V_p|$ ($|V_t|$) denotes the numbers of vertices of pattern (target) graphs.

SubSet-G_p-G_t	$\|V_p\|$	$\|V_t\|$	McSplit	McSplit+RL
Bio-030.txt-061.txt	50	73	623.74(204.04)	**187.68(50.20)**
Bio-022.txt-046.txt	38	31	**309.68(79.55)**	1227.46(458.42)
Bio-001.txt-018.txt	46	79	5214.98(545.06)	**1720.57(214.56)**
Images-p11-t10	15	3506	**0.02(0.00)**	**0.02(0.00)**
Images-p43-t113	89	2877	83.54(0.19)	**16.49(0.07)**
Images-p24-t119	21	5376	2.81(0.00)	**0.95(0.00)**
Images-p29-t120	22	4301	1.28(0.00)	**0.61(0.00)**
LV-g10-g18	41	64	**1002.25(1.28)**	1419.91(1.78)
LV-g12-g19	48	64	953.03(180.30)	**13.22(5.98)**
LV-g11-g21	42	64	4.10(1.00)	**3.26(0.99)**
LV-g10-g17	41	64	**241.36(20.52)**	271.18(5.43)
LargerLV-g11-g78	42	627	2.33(0.27)	**0.43(0.07)**
LargerLV-g12-g55	48	256	3580.15(73.53)	**345.12(30.99)**
LargerLV-g13-g70	49	501	234.17(0.61)	**187.58(1.66)**
LargerLV-g6-g72	19	561	33.90(0.09)	**0.02(0.00)**
LargerLV-g6-g71	19	501	**0.02(0.00)**	0.14(0.00)
PR15-p1-t	83	4838	**0.27(0.02)**	0.54(0.02)
PR15-p9-t	68	4838	0.24(0.00)	**0.16(0.00)**
Si-si2_b03m_m200.05	40	200	3.19(0.94)	**0.16(0.03)**
Si-si2_m4Dr2_m256.02	51	256	1169.12(127.65)	**460.27(70.89)**

Related Work

Solving NP-hard problems based on reinforcement learning belongs to a class of look-back heuristics that share some similarities in spirit but are very different in conception when applied to solve different problems.

The branching heuristic proposed in (Boussemart et al. 2004) is for solving the CSP problem. In this heuristic, no action is rewarded. Instead, the number of times each constraint is violated is recorded during the search. At a branching point, the variable involved in the constraints that were most frequently violated in the past is chosen as the branching variable.

The branching heuristic VSIDS proposed in (Zhang et al. 2001) is for solving the SAT problem. When a clause is violated during the search, a conflict analysis is carried out and each variable encountered in the conflict analysis is rewarded by a constant. Most awarded variables are not assigned by decision in VSIDS but by unit propagation, and the award of an assignment does not depend on its individual consequence, which is different from our approach where only decisions are awarded and the award of a decision is not a constant but depends on its immediate consequence.

The branching heuristic LRB proposed in (Liang et al. 2016) is also for solving the SAT problem. LRB explicitly uses the reinforcement learning scheme. When a variable x is assigned a value, an interval of time I begins for x and ends when backtracking cancels this assignment. The ratio of the number of dead-ends involving x to the total number of dead-ends during I is the reward to x. The score of x is the exponential recency weighted average of all rewards x received.

The branching heuristic proposed in (Refalo 2004) is tested on the multiknapsack, magic square, and Latin square completion problems in their decision version. In these problems, the impact of an assignment, i.e., the ratio of the reduction of the search space size due to the assignment to the search space size, does not vary much during the search. So, the average impact of an assignment in the past is the score of this assignment. The score of a variable is computed by summing the scores of all possible assignments this variable can receive at the current branching point.

In our approach for MCS, MCCS and SI, we can consider a match (v, w) as an assignment to v and the reduction of UB as the impact of (v, w). There are two differences with the approach of (Refalo 2004).

- The approach of (Refalo 2004) is designed to solve decision problems. So, many assignments are made due to unit propagation (i.e., when a variable has only one value in its current domain, the assignment has to be made). However, our approach is designed for optimization problems. So, almost all assignments are made by branching decision. In other words, the branching heuristic in our approach is more frequently used than in (Refalo 2004).
- The impact of an assignment for MCS, MCCS and SI varies much, so that the average impact is not very meaningful for estimating the importance of a vertex at a branching point. So, the score of a vertex in our approach is the sum (instead of the mean) of the impacts of all assignments it received in the past. A consequence of this heuristic is that vertices with greater score have more chance to be branched on, making their score greater and greater. This allows a natural intensification as search proceeds for MCS, MCCS and SI, in addition to the diversification allowing more vertices to participate in branching explained in the previous section.

Conclusion

In this paper, we propose a reinforcement learning based branching heurstic, and introduce a new branch and bound algorithm for the maximum common subgraph (MCS) problem and its two variant problems, maximum common connected induced subgraph (MCCS) problem and induced subgraph isomorphism (SI) problem. Our method of selecting the branching node differs to existing heuristics as it dynamically learns the effectiveness of the branching node on the search tree. The weights of vertices are the rewards of the searching used to guide the search. Extensive experimental results show that the proposed method is more broadly applicable to maximum common subgraph related problems.

In the future, we plan to apply our approach to solve other combinatorial optimization problems such as MaxSAT. In fact, branching heuristics in a BnB procedure for MaxSAT remain rudimentary since longtime, that might be greatly improved using a similar learning based approach.

Acknowledgements

This work is supported by the National Natural Science Foundation of China (Grants 61472147, 61671338 and 61473213) and the Matrics platform of Université de Picardie Jules Verne.

References

Archibald, B.; Dunlop, F.; Hoffmann, R.; McCreesh, C.; Prosser, P.; and Trimble, J. 2019. Sequential and parallel solution-biased search for subgraph algorithms. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*, 20–38.

Bahiense, L.; Manic, G.; Piva, B.; and de Souza, C. C. 2012. The maximum common edge subgraph problem: A polyhedral investigation. *Discrete Applied Mathematics* 160(18):2523–2541.

Bonnici, V., and Giugno, R. 2017. On the variable ordering in subgraph isomorphism algorithms. *IEEE/ACM Trans. Comput. Biology Bioinform.* 14(1):193–203.

Boussemart, F.; Hemery, F.; Lecoutre, C.; and Sais, L. 2004. Boosting systematic search by weighting constraints. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*, 146–150. IOS Press.

Bunke, H., and Messmer, B. T. 1995. Efficient attributed graph matching and its application to image analysis. In *Image Analysis and Processing, 8th International Conference, ICIAP '95, San Remo, Italy, September 13-15, 1995, Proceedings*, 45–55.

Cao, N.; Yang, Z.; and Wang, C. 2011. Privacy-preserving query over encrypted graph-structured data in cloud computing. *IEEE Computer Society* 6567(6):393–402.

Conte, D.; Foggia, P.; Sansone, C.; and Vento, M. 2008. Thirty years of graph matching in pattern recognition. *Pattern Recognition & Artificial Intelligence* 18(03):265–298.

Cootes, A. P.; Muggleton, S. H.; and Sternberg, M. J. 2007. The identification of similarities between biological vetworks: application to the metabolome and interactome. *Molecular Biology* 369(4):1126–1139.

Cordella, L. P.; Foggia, P.; Sansone, C.; and Vento, M. 2004. A (sub)graph isomorphism algorithm for matching large graphs. *Communications in Computer and Information Science* 26(10):1367–1372.

Faccioli, P.; Provero, P.; Herrmann, C.; Stanca, A. M.; Morcia, C.; and Terzi, V. 2005. From single genes to co-expression networks: Extracting knowledge from barley functional genomics. *Plant Molecular Biology* 58(5):739–750.

Giugno, R.; Bonnici, V.; Bombieri, N.; Pulvirenti, A.; Ferro, A.; and Shasha, D. 2013. Grapes: A software for parallel searching on biological graphs targeting multi-core architectures. *PLOS ONE* 8(10):1–11.

Hoffmann, R.; McCreesh, C.; and Reilly, C. 2017. Between subgraph isomorphism and maximum common subgraph. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, February 4-9, 2017, San Francisco, California, USA.*, 3907–3914.

Lecoutre, C.; Sais, L.; Tabary, S.; and Vidal, V. 2007. Recording and minimizing nogoods from restarts. *JSAT* 1(3-4):147–167.

Lee, J. H. M.; Schulte, C.; and Zhu, Z. 2016. Increasing nogoods in restart-based search. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA.*, 3426–3433. AAAI Press.

Levi, G. 1973. A note on the derivation of maximal common subgraphs of two directed or undirected graphs. *CALCOLO* 9(4):341–352.

Liang, J. H.; Ganesh, V.; Poupart, P.; and Czarnecki, K. 2016. Learning rate based branching heuristic for SAT solvers. In *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, 123–140.

Liu, J., and Lee, Y. T. 2001. Graph-based method for face identification from a single 2d line drawing. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 23(10):1106–1119.

Lodi, A., and Zarpellon, G. 2017. On learning and branching: a survey. *TOP* 25(2):207–236.

McCreesh, C.; Ndiaye, S. N.; Prosser, P.; and Solnon, C. 2016. Clique and constraint models for maximum common (connected) subgraph problems. In *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, 350–368.

McCreesh, C.; Prosser, P.; and Trimble, J. 2017. A partitioning algorithm for maximum common subgraph problems. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, 712–719.

Mcgregor, J. J. 2010. Backtrack search algorithms and the maximal common subgraph problem. *Software Practice & Experience* 12(1):23–34.

Ndiaye, S. N., and Solnon, C. 2011. CP models for maximum common subgraph problems. In *Principles and Practice of Constraint Programming - CP 2011 - 17th International Conference, CP 2011, Perugia, Italy, September 12-16, 2011. Proceedings*, 637–644.

Raymond, J. W.; Gardiner, E. J.; and Willett, P. 2002. RASCAL: calculation of graph similarity using maximum common edge subgraphs. *Comput. J.* 45(6):631–644.

Refalo, P. 2004. Impact-based search strategies for constraint programming. In *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*, 557–571. Springer.

Solnon, C.; Damiand, G.; de la Higuera, C.; and Janodet, J. 2015. On the complexity of submap isomorphism and maximum common submap problems. *Pattern Recognition* 48(2):302–316.

Solnon, C. 2010. Alldifferent-based filtering for subgraph isomorphism. *Artif. Intell.* 174(12-13):850–864.

Sutton, R. S., and Barto, A. G. 2018. *Reinforcement Learning: An introduction second edition*. MIT Press, Cambridge.

Zampelli, S.; Deville, Y.; and Solnon, C. 2010. Solving subgraph isomorphism problems with constraint programming. *Constraints* 15(3):327–353.

Zhang, L.; Madigan, C. F.; Moskewicz, M. H.; and Malik, S. 2001. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '01, 279–285. Piscataway, NJ, USA: IEEE Press.