Learning a value-selection heuristic inside a constraint programming solver

ACP Summer School 2023 - Leuven
Human intelligence versus artificial intelligence

**Human intelligence**
- **Reasoning**
- **Learning**
- **Intuition**
- **Adaptation**

**Artificial intelligence**
- **Reasoning**
- **Learning**
- **Adaptation**
- **Intuition**

This connection is not yet established

**Long-term research plan:** building an AI with these connections

**Goal:** providing a better solving process for combinatorial problems
What is a combinatorial problem?

Combinatorial satisfaction problem (CSP)
- Finding a feasible solution from a finite set of solutions
  - Finding a needle in a haystack
    - Building a schedule satisfying a set of constraints
    - Servicing a set of customers without delays

Combinatorial optimization problem (COP)
- Finding the best feasible solution from a finite set of solutions
  - Finding the biggest diamond needle in a haystack
    - Scheduling a production while minimizing costs
    - Maximizing serviced customers during a day
Difficult of combinatorial problems

In practice: generally a huge amount of possible solutions!

In theory: interesting combinatorial problems are NP-complete or NP-hard

Consequence: there is no poly-time algorithms to solve them exactly

How can we solve them?

Idea 1: enumerate all the solutions and keep the best one (exhaustive search)

Idea 2: use a kind of intuition to build a solution (heuristic solving or greedy algorithm)

Idea 3: build or enumerate solutions in a clever way (search-based methods)

Search-based methods

- Local search
- Meta-heuristics
- Integer programming
- Constraint programming

Heuristic solving

Cheap
No guarantees

Exhaustive search

Expensive
With guarantees

Observation: there are many search-based methods, with a specific dependency to a heuristic
Great challenge: the efficiency of a method is often tightly linked with the quality of the heuristic

Greedy algorithm and local search: huge dependency

Constraint programming: high dependency for good performances (define how the search is directed)

Integer programming: less dependent - but the approach is limited to specific problems

Consequence: a bad heuristic can give very poor performances to most solving approaches

Option 1: hand-crafting the heuristic

Idea: design manually a heuristic, thanks to expert knowledge

Exemple (TSP): always visiting the closest available city

Difficulty 1: require a good understanding about the problem (e.g., LKH)

Difficulty 2: must be designed specifically for each problem
Search-based methods

Great challenge: the efficiency of a method is often tightly linked with the quality of the heuristic

Greedy algorithm and local search: huge dependency

Constraint programming: high dependency for good performances (define how the search is directed)

Integer programming: less dependent - but the approach is limited to specific problems

Consequence: a bad heuristic can give very poor performances to most solving approaches

Option 2: learning the heuristic

Observation: we do not leverage the fact that similar problems may be solved many times (e.g., routing)

Consequence: for each problem, the solving process repeatedly restart from scratch with no knowledge

Idea: use past experiments or historical data for learning a heuristic

This idea is actually quite old...
Back to the past...

A Hybrid Approach to Vehicle Routing using Neural Networks and Genetic Algorithms

Jean-Yves Potvin
Danny Dubé
Christian Robillard
Centre de recherche sur les transports
Université de Montréal
C.P. 6128, Succ. Centre-Ville,
Montréal (Québec),
Canada H3C 3J7

Using artificial neural networks to solve the orienteering problem

Qiwen Wang\textsuperscript{a)}, Xiaoyun Sun\textsuperscript{b)}, Bruce L. Golden\textsuperscript{b)} and Jiyou Jia\textsuperscript{a)}

\textsuperscript{a)} College of Business and Management, Beijing University,
Beijing 100871, PR China

\textsuperscript{b)} College of Business and Management, University of Maryland,
College Park, MD 20742, USA

Neural Networks for Automated Vehicle Dispatching

Yu Shen
Jean-Yves Potvin
Jean-Marc Rousseau
Centre de Recherche sur les Transports
Université de Montréal
C.P. 6128, Succ. "A"
Montréal (Québec)
Canada H3C 3J7

? When such papers were written and published?

Answer: In the nineties!

Fun-fact 1: papers with similar names are still published :-)

Artificial Intelligence

Neural large neighborhood search for routing problems

André Hottung, Kevin Tierney

Neurocomputing

Solve routing problems with a residual edge-graph attention neural network

Kun Lei, Peng Guo, Yi Wang, Xiuxing Song, Wenchao Zhao

Fun-fact 2: you may not know who is Jean-Marc Rousseau but you may know his son :-)

Observation: learning heuristics (with neural networks) is an old and still open research question!

Quentin Cappart
Search-based methods

How learning can be used to solve a combinatorial problem?

Three integrations have been identified in this survey:

1. End-to-end learning
2. Learning to configure algorithms
3. Machine learning within combinatorial solvers

Examples for each of them were proposed this week :-)

Related fields: learning to model & tackling uncertainty (predict-and-optimize, constraint acquisition, etc.)

End-to-end learning

Idea: the problem is directly solved using machine learning

- Supervised learning
- Reinforcement learning
- Unsupervised learning
Limitation of end-to-end learning

Fundamental limitation

**Machine learning:** set of tools dedicated to **predict** an output

**Observation:** machine learning can make mistakes! (100% accuracy is not achievable on test set)

Even in a very simple dataset (MNIST - standard dataset in ML)

**Best accuracy reported:** 99.87%

**Difficulty:** it can be an important bottleneck for combinatorial optimization!

**Reason:** we do not want solutions that are infeasible (or to lose optimality)

**Challenge:** how to handle arbitrary combinatorial constraints?

Related works on end-to-end learning (and analyses)

- Learning combinatorial optimization algorithms over graphs [Khalil et al., NeurIPS-2017]
- Neural combinatorial optimization with reinforcement learning [Bello et al., Arxiv-2016]
- Reinforcement Learning for solving the vehicle routing problem [Nazari et al., NeurIPS-2018]
- Attention, learn to solve routing problems! [Kool et al., ICLR-2019]
- Learning a SAT solver from single-bit supervision [Selsam et al., ICLR-2019]
- End-to-end constrained Optimization learning: A survey [Kotary et al., IJCAI-2021]
- Learning the TSP requires rethinking generalization [Joshi et al., Constraints-2022]

And many more!
Learning to configure algorithms

Idea: machine learning is used to augment a solver with valuable information

Exemple 1: selecting appropriate parameters for the solver (e.g., CPLEX has more than 70 parameters)

Exemple 2: selecting a specific configuration (e.g., simplex or interior-point method for a linear relaxation)

Exemple 3: deciding if a pre-processing step must be carried out before calling the solver

Related names: algorithm configuration, automated tuning, portfolio selector

Comment: these approaches are often complementary with other learning approaches

Related works

Sequential Model-Based Optimization for General Algorithm Configuration [Hutter et al., LION-2011]
The irace package: Iterated racing for automatic algorithm configuration [Lopez-Ibanez et al., ORP-2016]
Algorithm Selection for Combinatorial Search Problems: A Survey [Kotthoff, 2016]
Learning to schedule heuristics in branch and bound [Chmiela et al., NeurIPS-2021]
Automated dynamic algorithm configuration [Adriaensen et al., JAIR-2022]
Machine learning alongside combinatorial solvers

Traditional solver: can provide guarantee, but sometimes hard to make it efficient

Only learning: struggle to get guarantees, but easier to use (once trained)

Idea: use machine learning to speed-up the solving process inside the solver

Exemples: learning branching decisions or optimization bounds

Learning to search in branch and bound algorithms [He et al., 2014, NeurIPS]
Learning to branch in mixed integer programming [Khalil et al., 2016, AAAI]
Exact combinatorial optimization with graph convolutional neural networks [Gasse et al., 2019, NeurIPS]
Improving variable orderings of approximate decisions diagrams using reinforcement learning [Cappart et al., 2022, IJOC]
Towards a multimodal artificial intelligence

Can we integrate other aspects of artificial intelligence?

Search: intelligence by intuition (heuristic with trials-and-errors)

Machine learning: intelligence to learn from experiments

Reasoning: intelligence by logical reasoning

Generic: intelligence to adapt (or generalize) to new situations

Holy grail: making it easy to use (and efficient) for non-experts
Constraint programming as a unifying framework

Our research hypothesis

Constraint programming can be used as a hosting technology for building this hybrid AI

\[ \text{CP} = \text{model} + \text{propagation} + \text{search} \]

Observation 1: model, propagation and search are present in most standard CP solvers

Observation 2: the only new part is the integration of learning

Is it really difficult to achieve?
A first proof of concept

**Learning phase**
- Training instances (randomly generated)

**Combinational optimization problem**

**Dynamic programming Model**

**Solving phase**
- Evaluated instances

**Reinforcement learning**
- Deep Q-learning (DQN)
- Proximal policy optimization (PPO)

**Constraint programming**
- Depth-first search
- Restart-based search
- Limited discrepancy search

Combining reinforcement learning and constraint programming for combinatorial optimization [Cappart et al. 2021, AAAI]
CP search with a learned heuristic on TSP

Our current position

Remaining customers

Our current position

Remaining customers
A first proof of concept

Additional improvement: caching to avoid unnecessary call to the trained model

Additional improvement: leveraging dominance to prune the search space (redundant constraints)

Good news: signals of learning were observed and good branching decisions could be obtained

Main assumption: we need to cast the combinatorial problem into a dynamic program

Limitation: learning is disconnected with the CP solver (loss of relevant information - e.g., propagation)

Difficulty: we need to build a specific model for each problem (e.g., a neural network)

Difficulty: loss of performances with the back-and-forth between C++ and python
Motivation a new CP solver

There are a lot of drawbacks! Can we do something?

Idea: embed the learning directly inside the CP solver

Difficulty: there is no available solver allowing us to do that easily (and efficiently)

Reason: friction between the need of an efficient language, and the ML support mostly available in Python

Technical contribution
Introducing and building a new CP solver, making easy to integrate learning inside

Ok! but which programming language do you plan to consider?

(1) Simple to get on-board, yet fast and clean
(2) Active community in optimization and machine learning
(3) Quite young language (both a benefit and a drawback)
(4) No CP solver available (excepting one that seems no longer developed)

Moto of Julia: « Looks like Python... runs like C »
SeaPearl (Cee-Pee-Air-EI) - CP with RL

Next topics in this talk

1. Describing the architecture behind SeaPearl
2. Presenting few experiments on its performance
3. Identifying current challenges and possible future research directions in this field

We have a lot of research ideas if you would like to contribute :-)
Main components of SeaPearl

Module 1: a constraint programming solver
Module 2: a generic representation function
Module 3: a learning agent, based on reinforcement learning and graph neural network
Welcome and Intro to Constraint Programming

Prof. Guns will welcome all to the summer school with a general motivation of why machine learning is increasingly used with constraint solving. This will be followed by an overview of the interactions between constraint solving and machine learning and how the program of the summer school highlights many of these

General characteristics

- **Inspiration:** MiniCP solver (in Java)
- **Data structure:** trail-based solver
- **Modeling:** an interface with JuMP is planned

Search strategies

- **Search 1:** depth-first search with branch-and-bound (default strategy)
- **Search 2:** iterated limited discrepancy search (allow to leverage good heuristics)
- **Search 3:** restart-based search (allow to leverage probabilistic heuristics)

Propagation engine

- **Filtering:** constraint propagation at each node with fix-point execution
- **Constraints:** intension, extension, allDifferent, sum, element (+ few others)

The solver is currently compatible for XCSP3 mini-track competition (but not the learning)
Modeling example: graph-coloring

```
trailer = SeaPearl.Trailer()
model = SeaPearl.CPModel(trailer)
```

**Goal:** keeping the philosophy of CP and the ease in modeling

**Prototyping:** possible to write model directly in a jupyter notebook

https://github.com/corail-research/SeaPearlZoo.jl/blob/master/src/classic_cp/graph_coloring/graph_coloring.ipynb
Standard CP depth-first search (DFS)

Fix-point: no solution

Fix-point: branch pruned

Solution with cost 10 found

Branches pruned (cost > 10)
Iterated limited discrepancy search (ILDS)

**Principle:** explore the tree with no deviation from the left branch, then allow 1 deviation, then allow 2, etc.

**Convention:** the left branch is what is explored first (value recommended by the heuristic)
Restart-based search

Principle: follow the branch based on a weighted probability, and periodically restart

Restart schedule: Luby sequence in terms of number of failures (1, 1,2, 1,1,2,4, 1,1,2,1,1,2,4,8,...)
Architecture behind SeaPearl

Main components of SeaPearl

**Module 1:** a constraint programming solver

**Module 2:** a generic representation function

**Module 3:** a learning agent, based on reinforcement learning and graph neural network
Our goal: leverage learning algorithms to speed-up the solving process (e.g., value-selection heuristic)

Observation: CP solvers can handle many combinatorial problems (routing, scheduling, assignment, etc.)

Practical use: the learning component should work for any problem given as input

Idea: build a function able to encode any combinatorial problem into a structure suited for learning

What are the requirements of such a function?

Requirement 1: able to encode variables with different domains

Requirement 2: able to encode any kind of constraint

Requirement 3: able to handle problems regardless of the number of variables

Requirement 4: able to handle problems regardless of the number of constraints

Requirement 5: preserving the combinatorial structure of the problem

Requirement 6: the function must be bijective (1-to-1 mapping with a CSP and the encoding)
**Current proposition: encoding as a labeled tripartite heterogeneous graph**

**Heterogeneous graph:** graph where the vertices and edges can have a different meaning

**Tripartite:** there are three kinds of vertices

**Labeled:** each vertex is decorated with additional information (i.e., features)

\[
X_1 \in \{1, 2\}, \; X_2 \in \{1, 2\}, \; X_3 \in \{1, 2, 3\} \\
\quad \cdot \; C_1 = X_1 \leq X_2 \\
\quad \cdot \; C_2 = X_2 \leq X_3 \\
\quad \cdot \; C_3 = \text{AllDifferent}(X_1, X_2, X_3)
\]

**Encoding function**

(1) One vertex per variable
(2) One vertex per constraint
(3) One vertex per value
(4) One edge if a variable is involved in a constraint
(5) One edge if a value is on the domain of a variable

\[ V_1 : \text{set of vertices for variables} \]
\[ V_2 : \text{set of vertices for constraints} \]
\[ V_3 : \text{set of vertices for values} \]
\[ E_1 : \text{set of variable/constraint edges} \]
\[ E_2 : \text{set of value/variable edges} \]
Generic representation function

What about the features on vertices?

Features for variables
(1) Current domain size (integer)
(2) Initial domain size (integer)
(3) Is already assigned (binary)
(4) Is the objective to optimize (binary)

Features for constraints
(1) Constraint type (one-hot)
(2) Has reduced domains with propagation (integer)

Features for values
(1) Its numerical value (integer)

Conclusion
Goal: encoding any combinatorial problem in a generic way
Inspiration: bipartite encoding proposed by Gasse et al. for MIP
Extension: other information can be easily added as new features

Disclaimer: this representation is not perfect and has some drawbacks (discussed later)

Exact combinatorial optimization with graph convolutional neural networks [Gasse et al., 2019, NeurIPS]
Main components of SeaPearl

Module 1: a constraint programming solver

Module 2: a generic representation function

Module 3: a learning agent, based on reinforcement learning and graph neural network
Learning a value-selection heuristic

Branching on $x_1$ (e.g., first-fail)

Training data

Branching on $x_1=2$

Solver state and selected variable $s_t = (P_t, x_t)$

CP Encoder (1)

Tripartite Heterogeneous Graph

GNN Encoder (2)

GNN layers

GNN Encoder

Extract variable features

NN Decoder (3)

Extract value features

$\varphi_v$

$\varphi_q$

Predicted Q-Table

$\hat{Q}(S_t, X_1 = 3)$

$\hat{Q}(S_t, X_1 = 2)$

$\hat{Q}(S_t, X_1 = 1)$

Action-Selection Policy

With a training carried out by deep reinforcement learning
Learning a value-selection heuristic

**Neural architecture**

**Step 1:** encoding the current solving process as a labeled tripartite heterogeneous graph (previous slides)

**Step 2:** leveraging this graph thanks to a graph neural network and obtain an embedding for each node

**Step 3:** estimating the most promising value thanks to fully-connected neural networks

**Step 4:** selecting the branching value based on the estimated score of each value

**Learning algorithm**

**Paradigm:** training based on deep reinforcement learning

**Data:** require historical or synthetic data (i.e., other combinatorial problems) to train the model

**Benefit:** there is no need to solve the historical problems a priori (can be very costly)
Primer on fully-connected neural network (FCNN)

Input: vector of features \((x)\)

Layer 1: \(h^1 = g(\theta^1 x + b^1)\)

\(g\): non-linear function (e.g., ReLU)

\(\theta^1, b^1\): weights learned through backpropagation

Layer 2: \(h^2 = g(\theta^2 h^1 + b^2)\)

Layer 3: \(\hat{y} = \theta^3 h^2 + b^3\)

Output: real value (prediction)

Principle: each neuron computes a linear combination of the previous layer followed by a non-linearity

**Fondamental equation of FCNN:** \(h^{k+1} = g(\theta^{k+1} h^k + b^{k+1})\)

Learning aspect: trainable weights are involved at each layer

Main characteristic: the network is differentiable and can be trained by gradient descent algorithms

In practice: many variants exist (classification tasks, other activations, regularization mechanisms, etc.)

What about graph neural networks?
Primer on graph neural networks

**Fondamental equation of GNNs:**

\[ h_u^{k+1} = g \left( \theta_1^{k+1} h_u^k \| \bigoplus_{v \in N(u)} \theta_2^{k+1} h_v^k \right) \]

**Input:** graph with node features \((G)\)

**Layer 1:** 

\[ h_2^1 = g \left( \theta_1^1 v_2 \| (\theta_1^1 v_1 \oplus \theta_1^1 v_3) \right) \]

\(\oplus\): aggregation operation

\(\|\): merging operation

Idem for each vertex at layer 1

**Layer 2:** 

\[ h_2^2 = g \left( \theta_1^2 h_1^2 \| (\theta_1^1 h_1^1 \oplus \theta_1^1 h_3^1) \right) \]

Idem for each vertex at layer 2

**Output:** embedding for each node \((e)\)

**Principle:** at each layer, each node aggregates information from its neighbours (message passing)

**Learning aspect:** trainable weights are involved at each layer (biases \(b\) have been omitted for clarity)

**After few iterations:** the nodes have information from more distant node

**In practice:** many architectures are existing (with attention, other aggregations, etc.)
What are the benefits of graph neural networks?

In practice: many architectures are existing (with attention, other aggregations, etc.)

Last comment: architecture increasingly used in combinatorial optimization and worth to study

Content: survey on how GNNs can be used in combinatorial optimization and related challenges

Link: https://www.youtube.com/watch?v=uF53xsT7mjc
Our GNN module

But your graph is heterogeneous! How do you handle this?

Equation for variable nodes:
\[ h_x^{k+1} = g\left(\theta_1^k h_x^0 \parallel \theta_2^k h_x^k \parallel \bigoplus_{c \in N_c(x)} \theta_3^k h_c^k \parallel \bigoplus_{v \in N_v(x)} \theta_4^k h_v^k\right) \]

Equation for constraint nodes:
\[ h_c^{k+1} = g\left(\theta_5^k h_c^0 \parallel \theta_6^k h_c^k \parallel \bigoplus_{x \in N_x(v)} \theta_7^k h_x^k\right) \]

Equation for value nodes:
\[ h_v^{k+1} = g\left(\theta_8^k h_v^0 \parallel \theta_9^k h_v^k \parallel \bigoplus_{x \in N_x(v)} \theta_{10}^k h_x^k\right) \]

Idea: having specific parameters for each type of nodes

Keeping the initial features at each layer (skip connection as in ResNet)
Our FCNN module

Embedding $e^{x_2}$: vectorized representation of variable $x_2$ after GNN inference

Embedding $e^{v_3}$: vectorized representation of value $v_3$ after GNN inference

$\hat{Q}(x_2, v_3)$: prediction of how good $v_3$ is for variable $x_2$ (Q-value)

Final inference: $\hat{Q}(x_2, v_3) = \text{FCNN} \left( \text{FCNN}(e^{x_2}) \ || \ \text{FCNN}(e^{v_3}) \right)$
Summary of the architecture

GNN step: leveraging the labeled tripartite heterogeneous graph and obtain an embedding for each node

FCNN step: estimating the most promising value thanks to fully-connected neural networks

How do we select the final value to branch on a variable $x$?

Final selection: taking the value inside the domain of $x$ with the highest score
There is something missing...

But how to train this model?
Learning phase

Introduction to Machine Learning

General characteristics

**Paradigm:** training based on reinforcement learning

**Data:** require historical or synthetic data to train the model

**Benefit:** there is no need to solve the historical problems a priori

**Training algorithm:** deep Q-learning (support for proximal policy optimization -PPO- is on development)

Implementation

**Reinforcement learning algorithm:** based on ReinforcementLearning.jl package

**Neural network architecture:** based on Flux.jl package

**Note:** some modifications have been done from the initial implementation to fulfill our specific needs

**Novelty:** on the reinforcement learning environment (and not so much on the training algorithm)

https://fluxml.ai/Flux.jl/stable/

https://juliareinforcementlearning.org/
Reinforcement learning environment

Reinforcement learning in a nutshell

Goal of the agent: obtain the most reward as possible during an episode

Episode: sequence of states from an initial state to a final state

Action: move the agent in a new state (and update it through the transition)

Reward: score obtained after each action

Environment: formal definition of the set of states, possible actions, transition, and reward function

Solving a problem with RL require to define the environment (modeling step)
Reinforcement learning environment

Environment

Agent to train: a value-selection heuristic inside a CP solver for a specific problem

Episode: a path from in the tree search without backtracks

Initial state: the root node (unsolved combinatorial problem)

Final state: a leaf node (either a feasible or unfeasible solution)

Action: selecting the value to branch on the current variable (agent choice)

Transition: branching and executing all the related CP solver stuff (fix-point, propagation, etc.)

Reward function: not trivial! Explanation on the next slide :-)
Reward function

Main principles

Goal: finding good solutions (and not to prove optimality)

Intuitive idea: use the final objective cost as reward signal

Difficulty: this information is only often available at the end of an episode (sparse reward issue)

Proposition: rewarding scheme based on the domain reduction of the objective variable at each node

Propagation-based reward

Propagation scope: on the variable corresponding to the objective function (to minimize)

Principle 1: rewarding the propagation of largest values of the domain

Principle 2: penalizing the propagation of lowest values of the domain

Principle 3: penalizing episodes reaching an unfeasible solution

Final reward: accumulated reward from each transition

What do you think about this reward?

10 values (initial domain)

3 highest values pruned at state $t+1$

1 lowest value pruned at state $t+1$

Reward at state $t+1 = \frac{3 - 1}{10} = \frac{2}{10}$
Main components of SeaPearl

Module 1: a constraint programming solver

Module 2: a generic representation function

Module 3: a learning agent, based on reinforcement learning and graph neural network
Main components of SeaPearl

Module 1: a constraint programming solver
Module 2: a generic representation function
Module 3: a learning agent, based on deep reinforcement learning

DOES IT WORK?
Experimental setup

Experimental protocol

**Combinatorial problems:** graph coloring, maximum independent set, maximum cut

**Instance sizes:** graphs from 20 to 100 nodes

**Models trained:** one per configuration (problem/size pair)

**Training phase:** 72 hours on Nvidia Tesla V100 32Go GPU for the most difficult cases

**Baselines:** random selection, impact-based search, and activity-based search

**Implementation:** everything on SeaPearl (no comparisons yet with other solvers)

**Metrics:** optimality gap and execution time to reach a specific solution

*Question explored: what is the best solutions obtained given a limited budget of explored nodes?*
Performances of the approach

<table>
<thead>
<tr>
<th>Size</th>
<th>OPT</th>
<th>1st dive</th>
<th>Gap</th>
<th>Learned</th>
<th>Activity-Based</th>
<th>Impact-Based</th>
<th>Random</th>
<th>Budget</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Gap</td>
<td>Node</td>
<td>Time</td>
<td>Gap</td>
<td>Node</td>
</tr>
<tr>
<td>COL</td>
<td>20</td>
<td>5.05</td>
<td>0.06</td>
<td>0</td>
<td>27</td>
<td>&lt;1</td>
<td>0</td>
<td>378</td>
</tr>
<tr>
<td></td>
<td>40</td>
<td>7.90</td>
<td>0.08</td>
<td>0</td>
<td>104</td>
<td>&lt;1</td>
<td>0</td>
<td>1,664</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>8.75</td>
<td>0.06</td>
<td>0</td>
<td>120</td>
<td>1</td>
<td>0</td>
<td>7,051</td>
</tr>
<tr>
<td>MIS</td>
<td>30</td>
<td>9.90</td>
<td>0.08</td>
<td>0</td>
<td>88</td>
<td>&lt;1</td>
<td>0</td>
<td>215</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>15.00</td>
<td>0.09</td>
<td>0</td>
<td>539</td>
<td>1</td>
<td>0</td>
<td>5,807</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>21.70</td>
<td>0.20</td>
<td>0.02</td>
<td>28,392</td>
<td>253</td>
<td>0.09</td>
<td>35,536</td>
</tr>
<tr>
<td>MAXCUT</td>
<td>20</td>
<td>46.70</td>
<td>0.15</td>
<td>0.03</td>
<td>3,714</td>
<td>5</td>
<td>0.04</td>
<td>4,635</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>222.00</td>
<td>0.16</td>
<td>0.09</td>
<td>38,744</td>
<td>130</td>
<td>0.17</td>
<td>44,664</td>
</tr>
</tbody>
</table>

- **Configurations tested**: (20 instances per scenario)

- **Average value of the optimal cost**

- **Optimality gap obtained with a single dive**: (no backtrack)

- **Number of explored nodes to obtain a given gap**: (capped at 100,000)

**Observation 1**: a gap of 0.16 is obtained in a single dive while it 44,664 nodes for baselines to have 0.17

**Observation 2**: we are able to obtain good solutions in less explored nodes compared to baselines

**Observation 3**: the execution time of calling the NN is important
Zoom on the hardest scenarios - performance profiles

**Baselines:** each curve corresponds to a method

**Metric:** optimality gap

**Performance profiles:** each tick gives the proportion of instances able to achieve a given optimality gap

**Interpretation:** the upper is the curve, the better is the method

**Observation:** results obtained by the learned approach is robust among all the instances tested

**Conclusion of the experiments:** it seems that we are able to learn interesting branching decisions!
Second conclusion

? It seems great! Should I use this for solving my problems and get competitive results?

No!!!!

Explanation: I believe it is a promising research direction, but not mature yet to get competitive results.

Getting quickly competitive results: currently better to use problem-specific heuristics.

Take-home message: see this work as first building blocks to unlock new avenues in the mid-term.

Personal note: I have the same opinion for many works using ML for combinatorial optimization :-)

? I also like this idea of a hybrid paradigm! What kind of research can I carry out in this field?

Next slides: I will propose and discuss few challenges and related research questions.
Research idea: reducing the inference time

**Learned heuristic:** 130 seconds to explore 38,744 nodes (298 nodes/second)

**Random selection:** 19 seconds to explore 53,110 nodes (2795 nodes/second)

**Ratio:** roughly an exploration rate 10 times slower!

**Explanation:** calling the model (GNN + FCNN) is much more costly than simple branching heuristics

### What can we do?

**Idea 1:** caching Q-values and use them in similar states

**Idea 2:** reducing the inference time of the model (transfer learning, network pruning, etc.)

**Idea 3:** calling the model only in few nodes of the search tree (gave good results in another project)

---

Improving Variable Orderings of Approximate Decisions Diagrams using Reinforcement Learning [Cappart, Rousseau et al., IJOC-2022]
Research idea: rethinking the representation

**Encoding function**

$X_1 \in \{1, 2\}, X_2 \in \{1, 2\}, X_3 \in \{1, 2, 3\}$

- $C_1 = X_1 \leq X_2$
- $C_2 = X_2 \leq X_3$
- $C_3 = \text{AllDifferent}(X_1, X_2, X_3)$

**Challenge 1: scalability**

**Difficulty:** the size of the representation is growing fast

**Consequence:** the training phase is harder and more costly

**Idea:** curriculum learning from small instances

**Challenge 2: expressivity**

**Difficulty:** we may miss important relationships in the model

**Exemple:** inequalities with different constant values

**Consequence:** we either lose information on the constant, or that the constraint is similar

**Idea:** expend the representation with new information (as in an abstract syntactic tree)

- $c_1 : x \leq y + 3$
- $c_2 : x \leq y + 6$
- $c_3 : x \leq y^2$
Research idea: learning a double heuristic

Motivation: selecting the variable to branch on is also challenging

Idea: expend the architecture to learn a variable-selection heuristic at the same time

Possible option 1: adopting a methodology similar to cooperative multi-agent reinforcement learning

Possible option 2: allowing the agents to share information (a good value selection depends on the variable)

And many other ideas can also be leveraged and tested!
Conclusion

Our research hypothesis

Constraint programming can be used as a hosting technology for building an hybrid AI

CP = \textit{model} + \textit{propagation} + \textit{search} + \textit{learning}

Solver: https://github.com/corail-research/SeaPearl.jl

Zoo of models: https://github.com/corail-research/SeaPearl.jl


Other related projects: https://corail-research.github.io/publications/

Contact: quentin.cappart@polymtl.ca

Combining reinforcement learning and constraint programming for combinatorial optimization [Cappart et al., AAAI 2021]

Seapearl: A constraint programming solver guided by reinforcement learning [Chalumeau, Cappart et al., CPAIOR 2021]

Learning a generic value-selection heuristic inside a constraint programming solver [Marty, Cappart et al., CP 2023]
Contributors

Main Contributors: Quentin Cappart, Louis-Martin Rousseau, Tom Marty, Léo Boisvert


We are always open for new contributions :-)
Learning a value-selection heuristic inside a constraint programming solver

ACP Summer School 2023 - Leuven