Efficient tree-search algorithms in Optimization and Operation Research

Abdel-Malik Bouhassoun, Luc Libralesso

July 11, 2019

G-SCOP

Table of contents

Glass Cutting Challenge ?

Constructive algorithm for the ROADEF challenge

- General methodology
- Branching Scheme
- Search Strategy
- Results

Towards a generic Tree Search framework

Sequential Ordering Problem

Glass Cutting Challenge ?

EURO/ROADEF Challenge

- Presented by the French and European *Operations Research* societies
- International competition

EURO/ROADEF Challenge

- Presented by the French and European *Operations Research* societies
- International competition

- A challenge every two years
  - 2012: Google
  - 2014: SNCF (state-owned railway company)
  - 2016: Air Liquide

EURO/ROADEF Challenge

- Presented by the French and European *Operations Research* societies
- International competition

- A challenge every two years
  - 2012: Google
  - 2014: SNCF (state-owned railway company)
  - 2016: Air Liquide
  - **2018: Saint Gobain**

2018 edition of the challenge - glass cutting

SAINT-GOBAIN

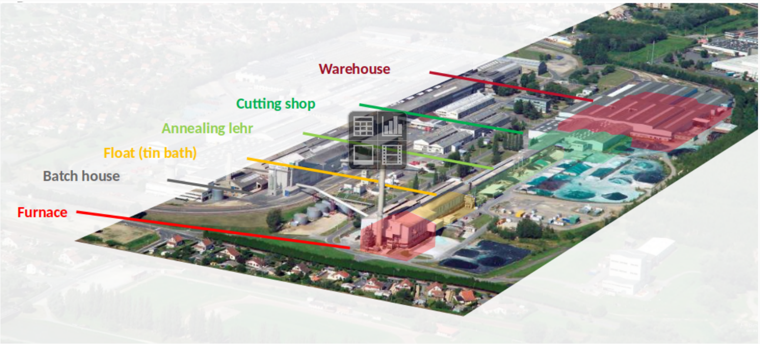
- Founded in 1665
- produces pipes, mirrors, mortars and glass

2018 edition of the challenge - glass cutting

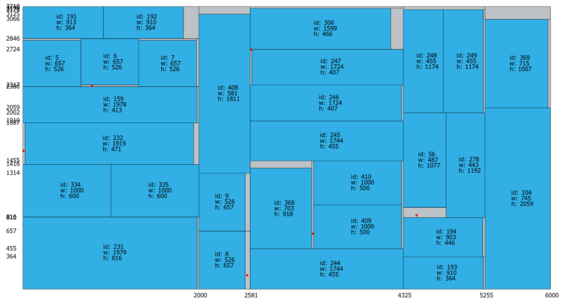
- Founded in 1665
- produces pipes, mirrors, mortars and glass

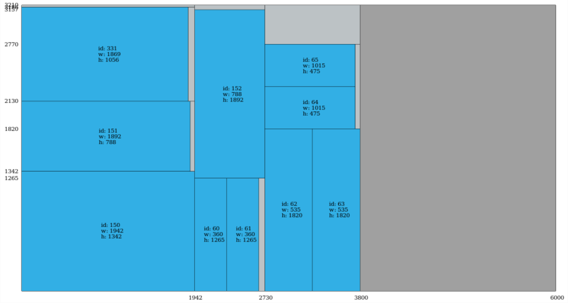
Cut rectangular glass items from big glass plates (Plates)

How to make glass



One of our solutions





Glass cutting Problem

OBJECTIVE:

minimize waste

Glass cutting Problem

OBJECTIVE:

minimize waste

DATA:

- Items (defined width and height, rotation possible)

Glass cutting Problem

Objective:

minimize waste

Data:

- Items (defined width and height, rotation possible)
- Stacks (chain precedence constraints)

Glass cutting Problem

OBJECTIVE:

minimize waste

DATA:

- Items (defined width and height, rotation possible)
- Stacks (chain precedence constraints)
- 100 Plates (6m x 3m) with defects

Glass cutting Problem

OBJECTIVE:

minimize waste

DATA:

- Items (defined width and height, rotation possible)
- Stacks (chain precedence constraints)
- 100 Plates (6m x 3m) with defects

CONSTRAINTS:

- **guillotine constraint**

Guillotine constraint

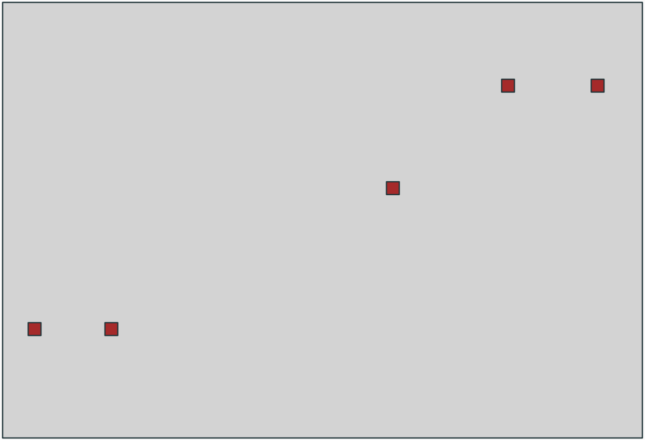


Figure 1: Example of a solution

Guillotine constraint

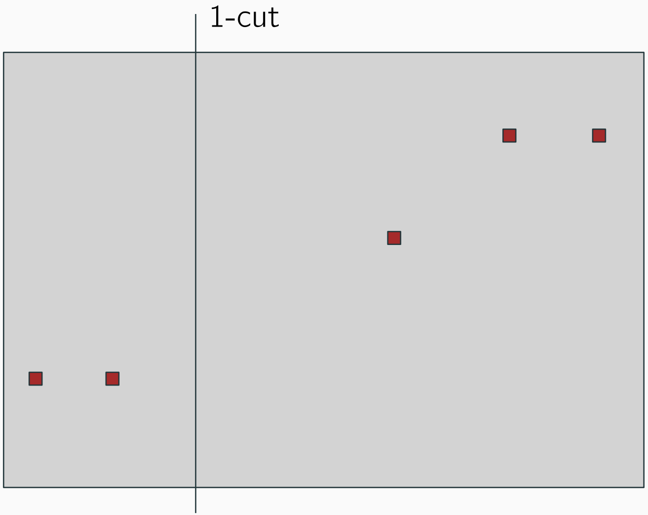


Figure 2: Example of a solution

Guillotine constraint

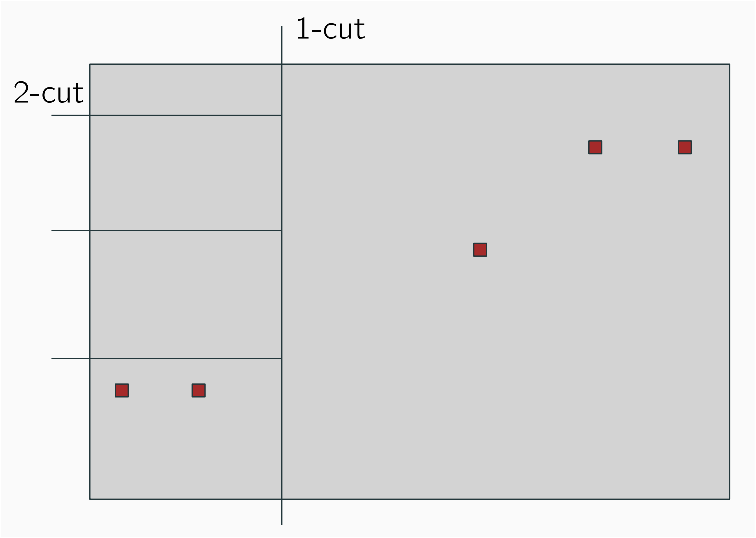


Figure 3: Example of a solution

Guillotine constraint

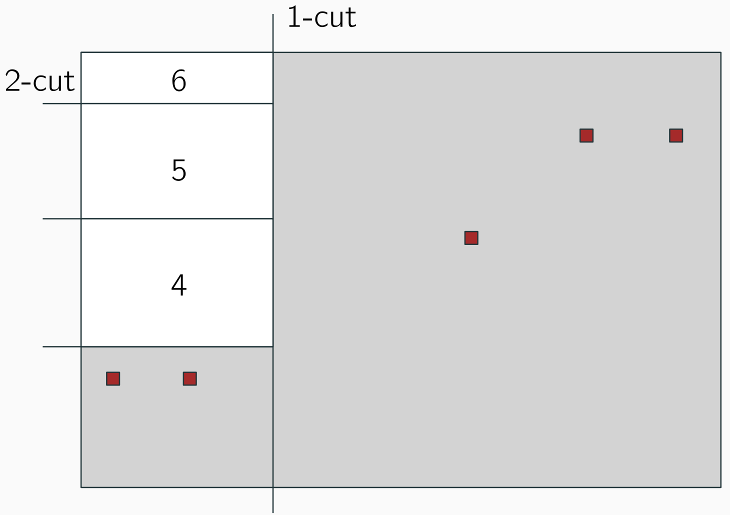


Figure 4: Example of a solution

Guillotine constraint

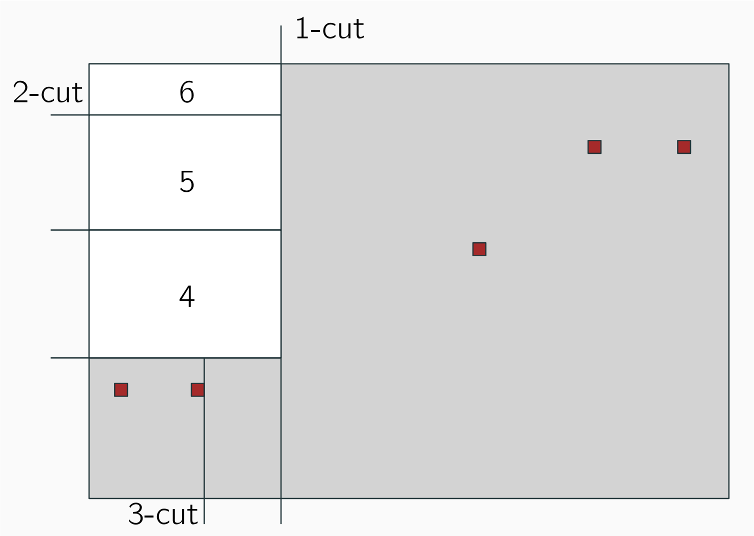


Figure 5: Example of a solution

Guillotine constraint

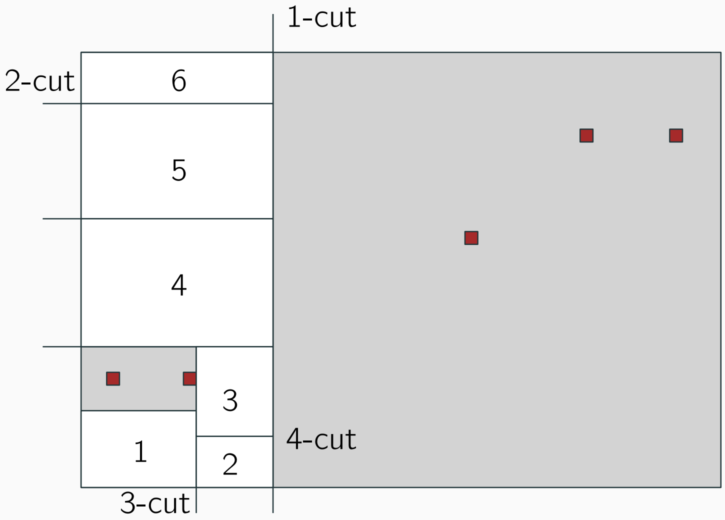
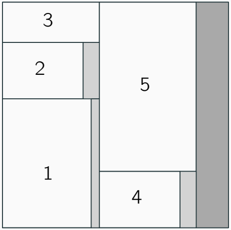


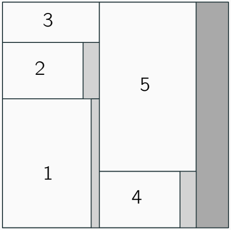
Figure 6: Example of a solution

guillotine cuts and not allowed cuts

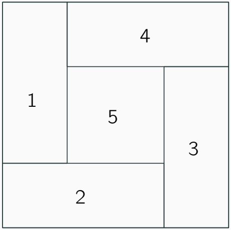


guillotine

guillotine cuts and not allowed cuts

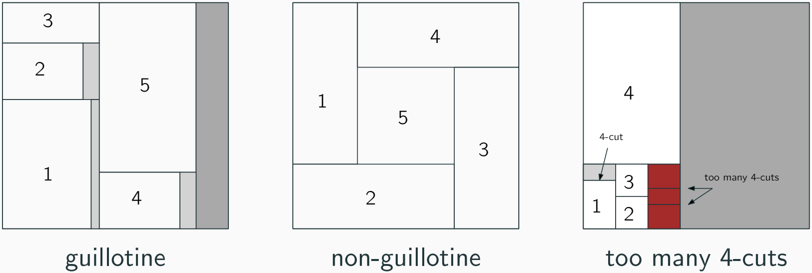


guillotine



non-guillotine

guillotine cuts and not allowed cuts



Precedence constraints

OBJECTIVE:

minimize waste

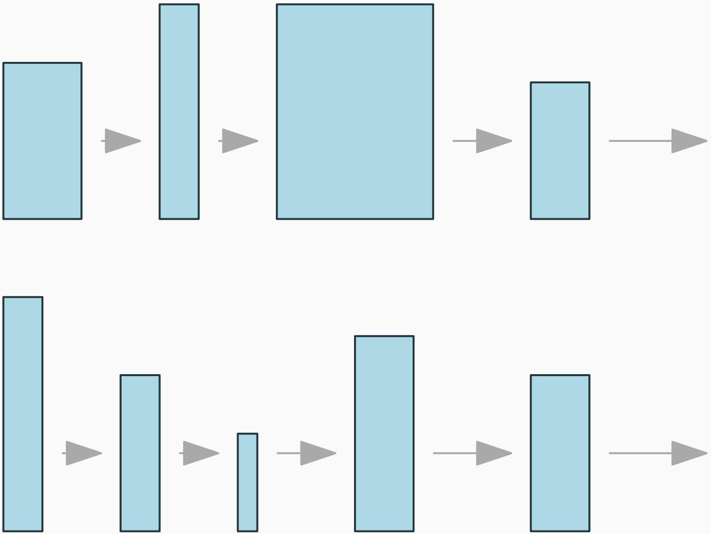
DATA:

- Items (defined width and height, rotation possible)
- Stacks (chain precedence constraints)
- 100 Plates (6m x 3m) with defects

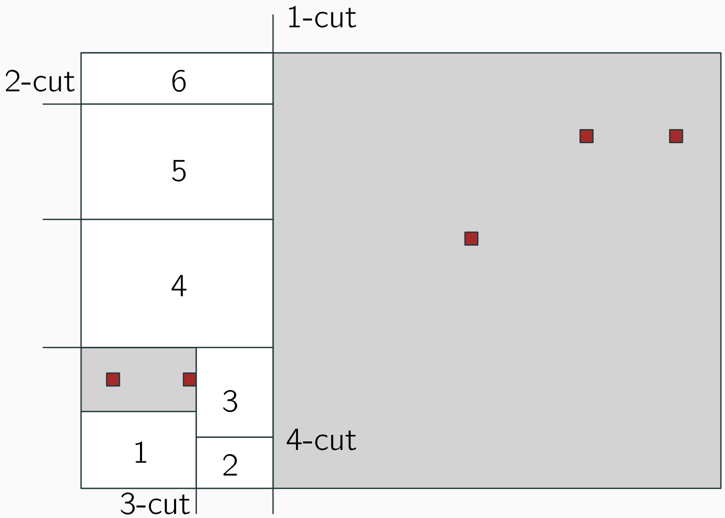
CONSTRAINTS:

- guillotine constraint
- **all items produced in a valid order**

Precedence Constraint



Precedence Constraint



Defect avoidance

OBJECTIVE:

minimize waste

DATA:

- Items (defined width and height, rotation possible)
- Stacks (chain precedence constraints)
- 100 Plates (6m x 3m) with defects

CONSTRAINTS:

- guillotine constraint
- all items produced in a valid order
- **no defects in items**
- **no cut on a defect**

minimum/maximum cut size

Objective:

minimize waste

Data:

- Items (defined width and height, rotation possible)
- Stacks (chain precedence constraints)
- 100 Plates (6m x 3m) with defects

Constraints:

- guillotine constraint
- all items produced in a valid order
- no defects in items
- no cut on a defect
- **min/max constraints on cuts and waste**

Min-waste constraint

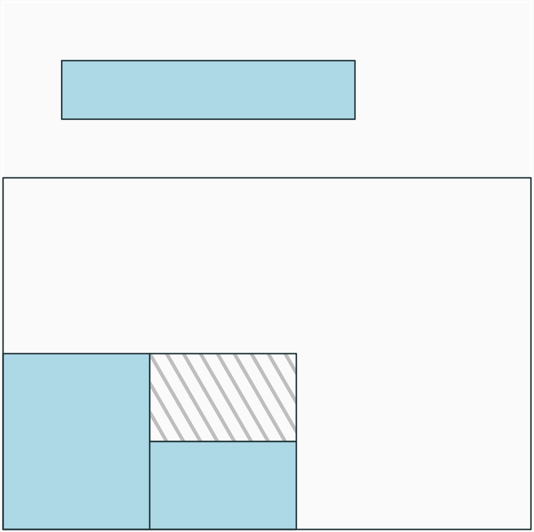


Figure 7: Min waste: easy case

Min-waste constraint

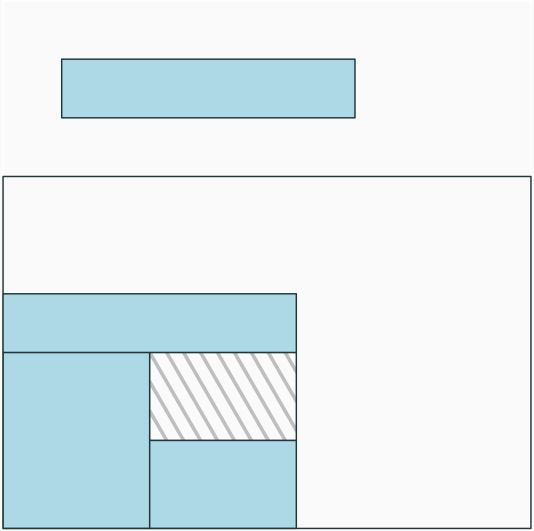


Figure 8: Min waste: easy case

Min-waste constraint

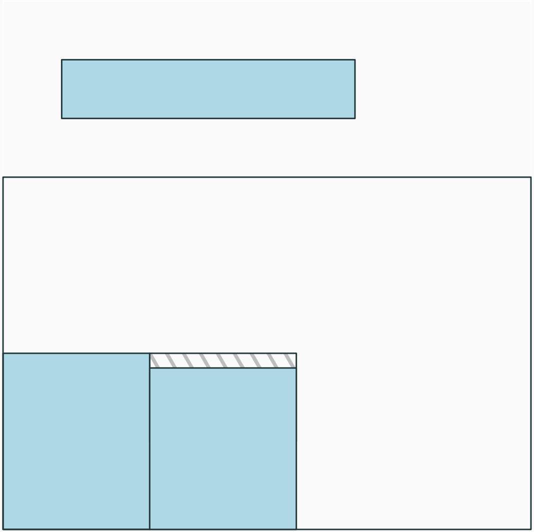


Figure 9: Min waste: more difficult

Min-waste constraint

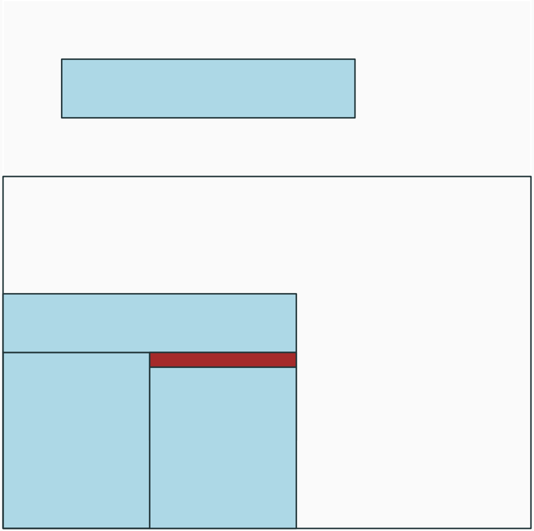


Figure 10: Min waste: more difficult

Min-waste constraint

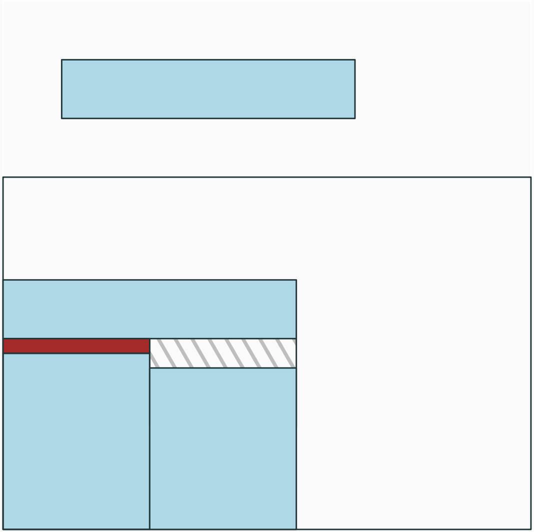


Figure 11: Min waste: more difficult

Min-waste constraint

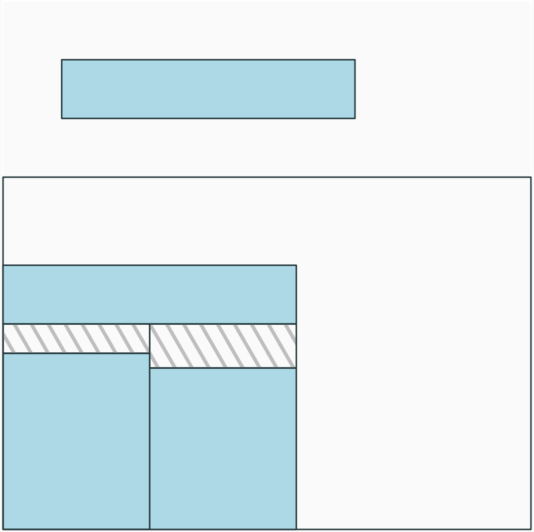


Figure 12: Min waste: more difficult

Glass cutting Problem

The problem is $\mathcal{NP}$-Hard.

Glass cutting Problem

The problem is $\mathcal{NP}$-Hard.

Difficult problem and big instances

Glass cutting Problem

The problem is $\mathcal{NP}$-Hard.

Difficult problem and big instances

We use anytime algorithms (meta-heuristics)

In this talk

We generate an implicit search tree. (next section)
It is called **Branching Scheme**

In this talk

We generate an implicit search tree. (next section)
It is called **Branching Scheme**

We explore this search tree cleverly (section after)
we use **anytime tree searches**

In this talk

We generate an implicit search tree. (next section)
It is called **Branching Scheme**

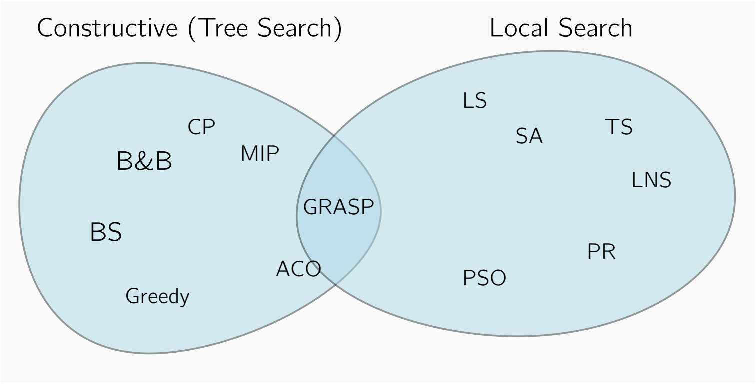
We explore this search tree cleverly (section after)
we use **anytime tree searches**

Work on a generic tree search framework

Application on the Sequential Ordering Problem

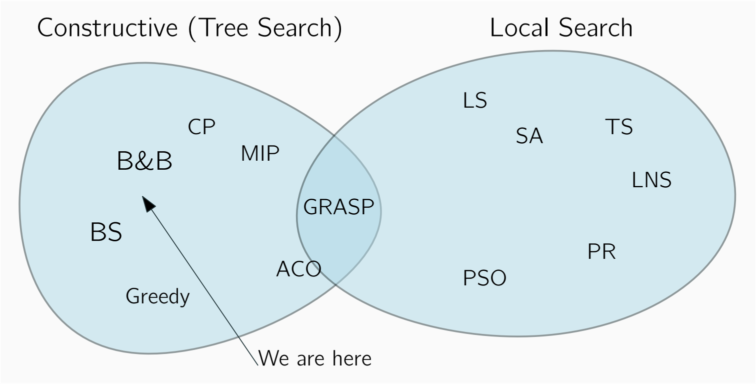
Constructive algorithm for the ROADEF challenge

Constructive vs Local Search



.

Constructive vs Local Search



.

Our method integrates parts of *Branch and bounds* and *Beam Search*

Anatomy of a Tree Search

Tree Searches are made of two parts:

Anatomy of a Tree Search

Tree Searches are made of two parts:

- the Branching Scheme (*i.e. problem specific*):
  - root definition
  - children generation
  - bounds
  - *etc.*

Anatomy of a Tree Search

Tree Searches are made of two parts:

- the Branching Scheme (*i.e. problem specific*):
  - root definition
  - children generation
  - bounds
  - *etc.*
- the Search strategy (*i.e. generic*):
  - DFS, A*, Best First, Beam Search, LDS, *etc.*

Anatomy of a Tree Search

Tree Searches are made of two parts:

- the Branching Scheme (*i.e. problem specific*):
    - root definition
    - children generation
    - bounds
    - *etc.*
- the Search strategy (*i.e. generic*):
    - DFS, A*, Best First, Beam Search, LDS, *etc.*
    - others known in AI/planning: SMA*, BULB, wA* *etc.*

Anatomy of a Tree Search

Tree Searches are made of two parts:

- the Branching Scheme (*i.e. problem specific*):
  - root definition
  - children generation
  - bounds
  - *etc.*
- the Search strategy (*i.e. generic*):
  - DFS, A*, Best First, Beam Search, LDS, *etc.*
  - others known in AI/planning: SMA*, BULB, wA* *etc.*

We developed our algorithm using this principle.

Our algorithm

- **the Branching Scheme** (*i.e. problem specific*)
- the Search strategy (*i.e. generic*)

Node inference trade-off

≥ 1 M nodes/s

Weak inference

≤ 1000 nodes/s

Strong inference

Node inference trade-off

Weak inference	Strong inference
≥ 1 M nodes/s	≤ 1000 nodes/s
Greedy	MIP
Beam Search	CP

Node inference trade-off

≥ 1 M nodes/s

≤ 1000 nodes/s

Weak inference

Strong inference

Greedy

Beam Search

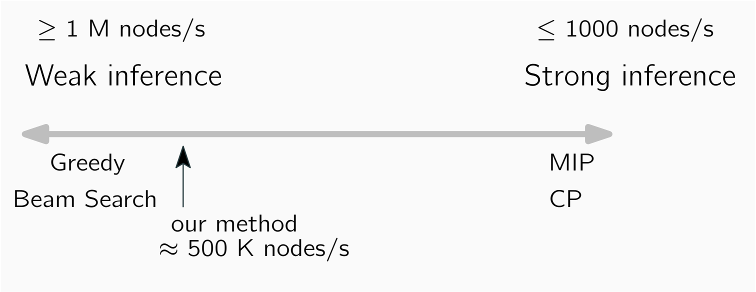
MIP

CP

our method

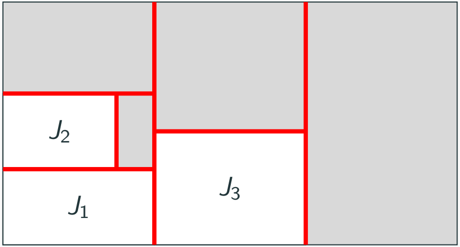
≈ 500 K nodes/s

Node inference trade-off

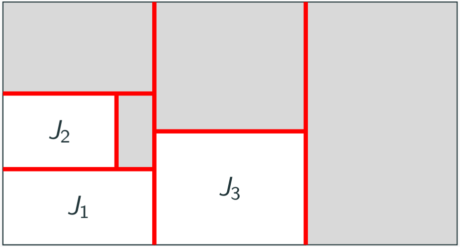


- We integrate quick bounds, symmetry breaking, dominance checking
- The idea of integrating Branch and Bound parts into Beam Searches can be found in [STDC18]

Packing in the bottom left corner



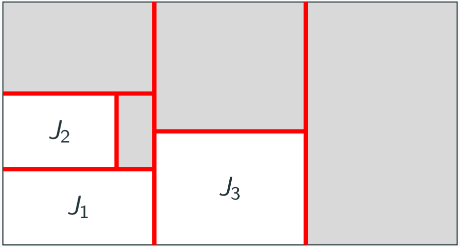
Packing in the bottom left corner



We prove that it is optimal if:

- guillotine and defects and precedence only
- guillotine and min waste only

Packing in the bottom left corner



We prove that it is optimal if:

- guillotine and defects and precedence only
- guillotine and min waste only

We prove it is not if:

- guillotine and min waste and precedences
- guillotine and min waste and defects

Not dominant in the challenge

Since guillotine and min waste and precedences and defects constraints.

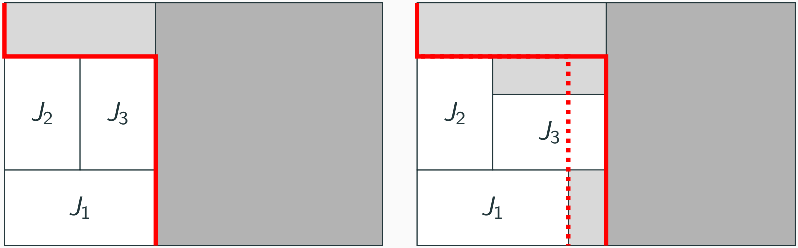
Good news - It still works very well !

We only need good solutions, so we make a heuristic Branch and Bound.

How to construct children

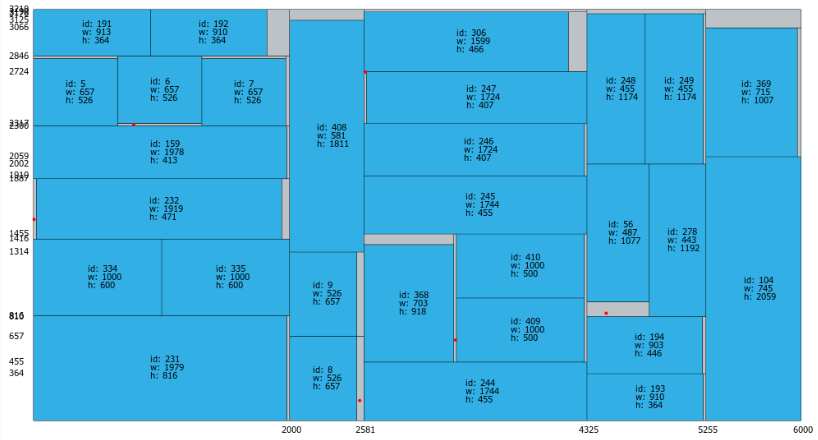
- Root node: empty solution
- Children: all possible items in all possible positions (*i.e.* new plate, new 1-cut, new 2-cut, new 3-cut or new 4-cut, rotations, defect avoidance)

Pseudo dominance



Symmetry breaking

- Symmetry breaking strategy: for two consecutive blocks, the one with the smallest minimum item id comes before.

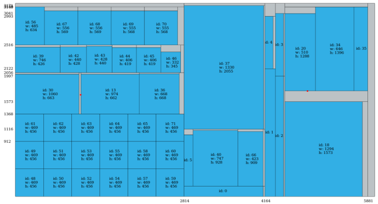


Node Goodness measure

Waste accumulated so far

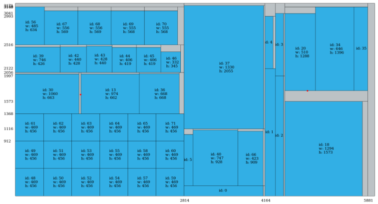
Node Goodness measure

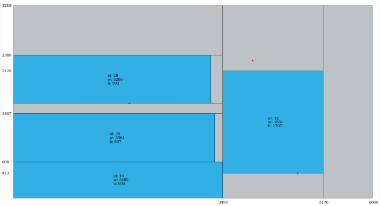
Waste accumulated so far



Node Goodness measure

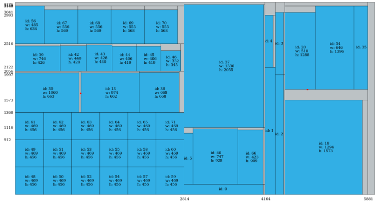
Waste accumulated so far

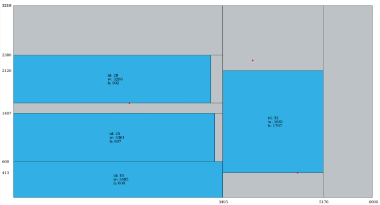




Node Goodness measure

Waste accumulated so far





Problem with waste:

- Small items at the beginning and big items at the end

A better node goodness measure

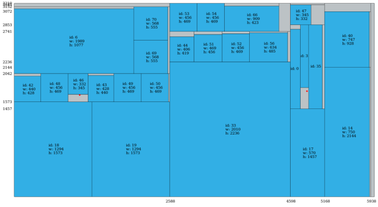
waste percentage

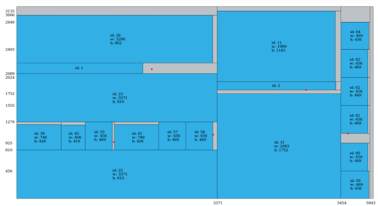
An even better node goodness measure

$$\frac{\text{waste}}{\text{total area} \cdot \text{mean area}}$$

An even better node goodness measure

$$\frac{\text{waste}}{\text{total area} \cdot \text{mean area}}$$





Our algorithm

- the Branching Scheme (*i.e. problem specific*)
- **the Search strategy** (*i.e. generic , DFS, Best First, Beam Search, ...*)

MBA*

Inspired from *Beam Search* and *SMA**

MBA*

Inspired from *Beam Search* and *SMA**

- Best First strategy
- Delete some bad nodes if too many at the same time
- If finished, Restart with a bigger node limit D ($D_{n+1} \leftarrow D_n \times 2$)

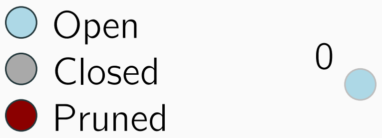
MBA*

Inspired from *Beam Search* and *SMA**

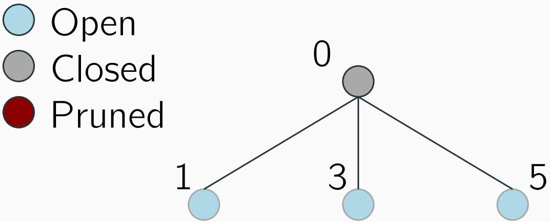
- Best First strategy
- Delete some bad nodes if too many at the same time
- If finished, Restart with a bigger node limit D ($D_{n+1} \leftarrow D_n \times 2$)

- at the beginning ($D = 1$), it behaves like a greedy algorithm
- at the end ($D \approx \infty$), it behaves like a Best First Search

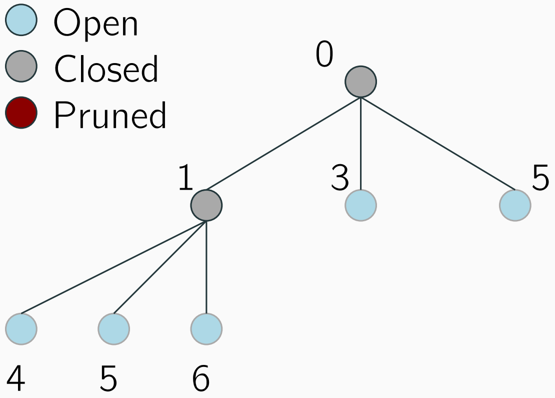
MBA* - an example



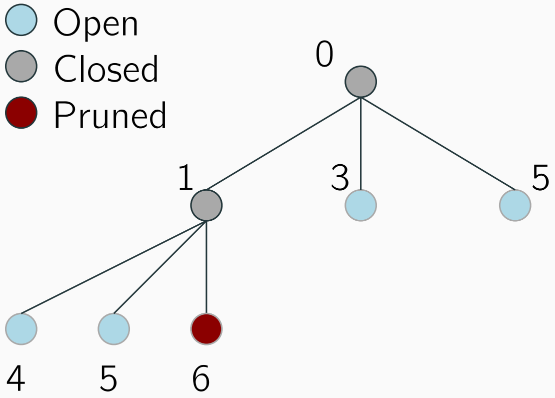
MBA* - an example



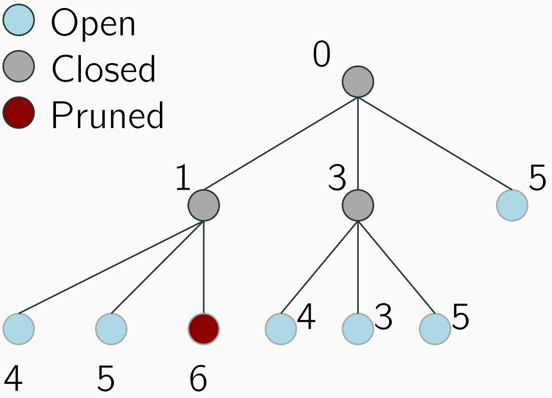
MBA* - an example



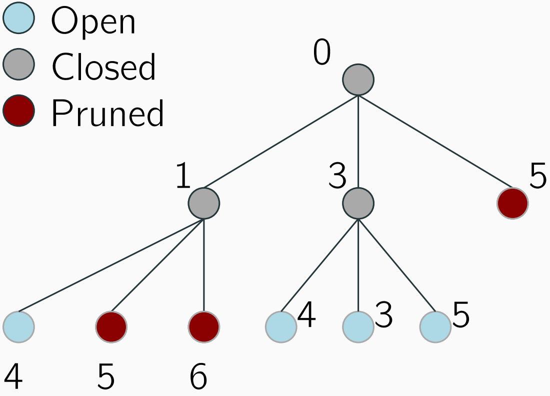
MBA* - an example



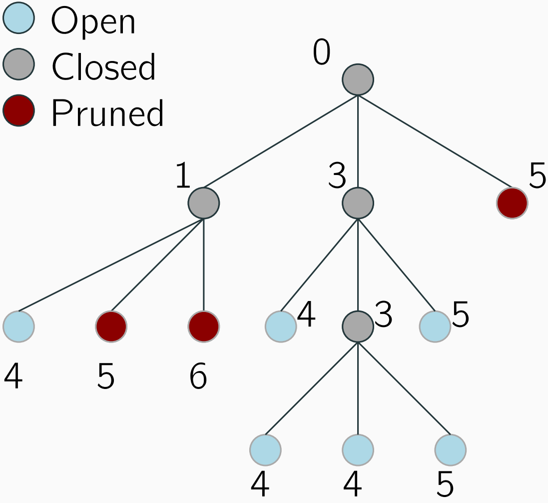
MBA* - an example



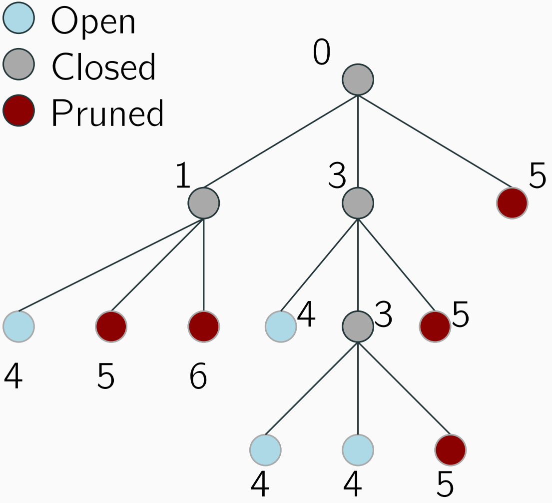
MBA* - an example



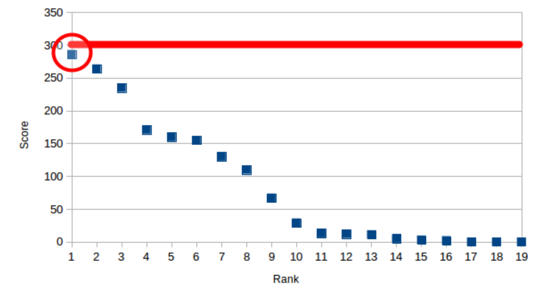
MBA* - an example



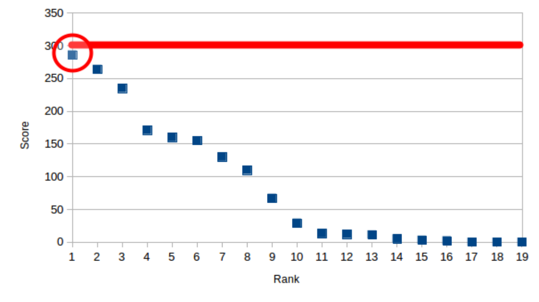
MBA* - an example



performance

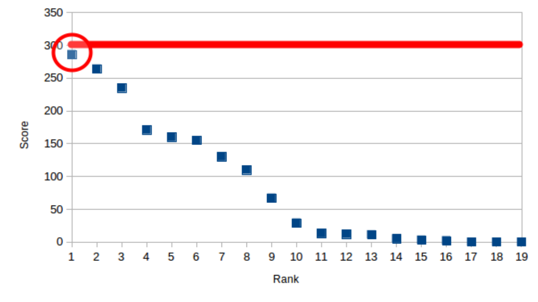


performance



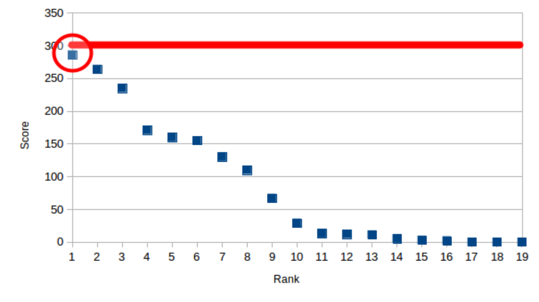
- Best solutions found on 20 over 30 instances.

performance



- Best solutions found on 20 over 30 instances.
- Total waste 2nd team: $506M$
- Total waste: $493M$ ($13M$ less)

performance



- Best solutions found on 20 over 30 instances.
- Total waste 2nd team: $506M$
- Total waste: $493M$ ($13M$ less)
- Total waste new version: $469M$ ($24M$ less than our submission)

Conclusions on the challenge

- Simple and effective algorithm
- Tree searches can be competitive with other methods
- Decomposing the algorithm helps to identify good (and bad) parts

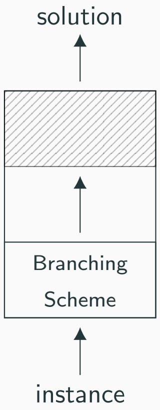
Conclusions on the challenge

- Simple and effective algorithm
- Tree searches can be competitive with other methods
- Decomposing the algorithm helps to identify good (and bad) parts

- We tried
  - several search strategies (DFS, Beam Search, LDS, and MBA*)
  - several guides
- Chose best combination

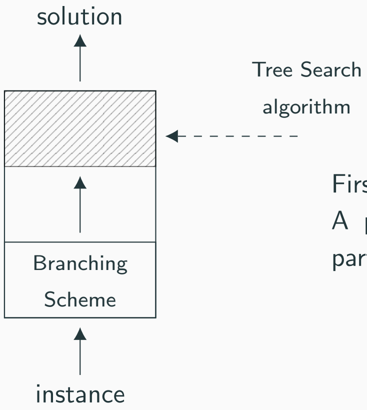
Towards a generic Tree Search framework

Starting Point



First Black-box decomposition.
A problem specific and a Tree Search part.

Starting Point



First Black-box decomposition.
A problem specific and a Tree Search part.

Tree Searches

Exhaustive Search
Enumerate all solutions of a problem to find the optimal one.

Exact Algorithm
For a long enough search, the optimal solution cannot be missed.

Heuristic Algorithm
Prunes nodes heuristically and could lead to missing the optimal solution.

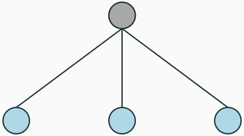
Anytime Algorithm
Can produce solutions during the search and not only at its end.

Breadth First Search

Breadth First Search

- Exhaustive Search.

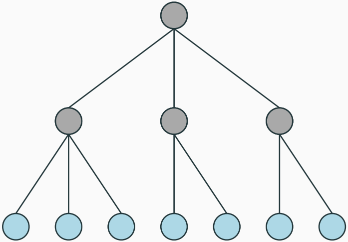
Breadth First Search



Breadth First Search

- Exhaustive Search.

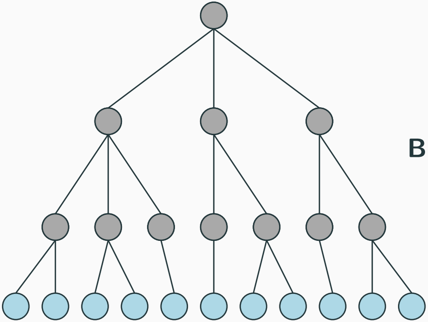
Breadth First Search



Breadth First Search

- Exhaustive Search.

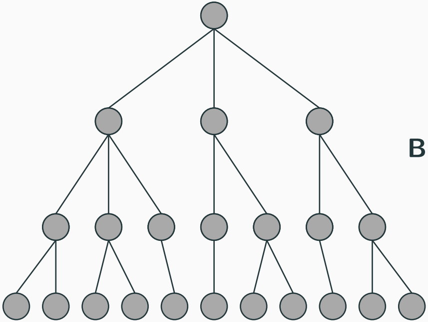
Breadth First Search



Breadth First Search

- Exhaustive Search.

Breadth First Search



Breadth First Search

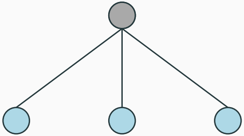
- Exhaustive Search.

Depth First Search

Depth First Search

- Exhaustive Search.
- Anytime Search.

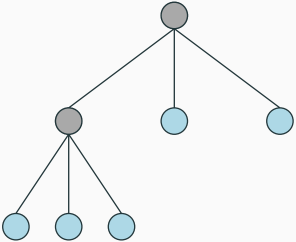
Depth First Search



Depth First Search

- Exhaustive Search.
- Anytime Search.

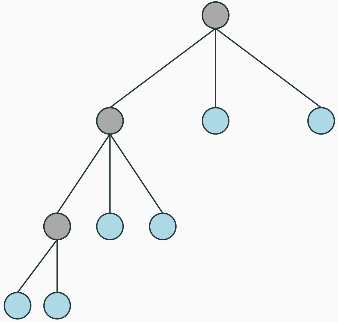
Depth First Search



Depth First Search

- Exhaustive Search.
- Anytime Search.

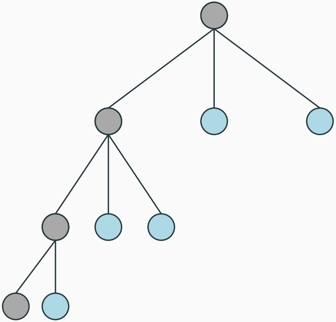
Depth First Search



Depth First Search

- Exhaustive Search.
- Anytime Search.

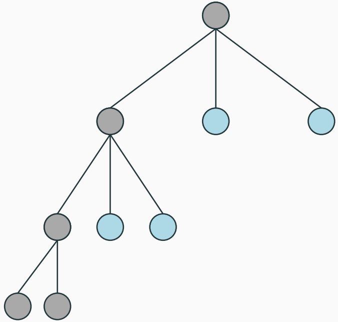
Depth First Search



Depth First Search

- Exhaustive Search.
- Anytime Search.

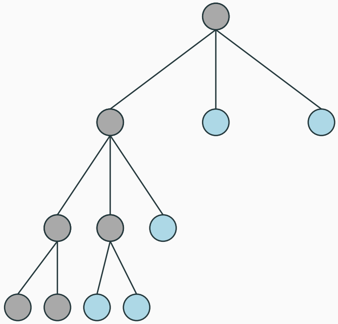
Depth First Search



Depth First Search

- Exhaustive Search.
- Anytime Search.

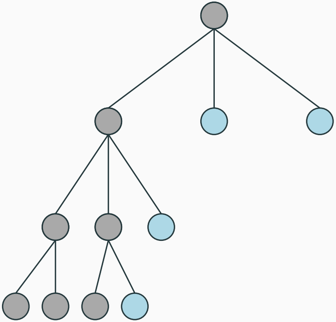
Depth First Search



Depth First Search

- Exhaustive Search.
- Anytime Search.

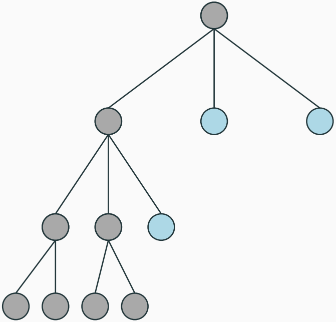
Depth First Search



Depth First Search

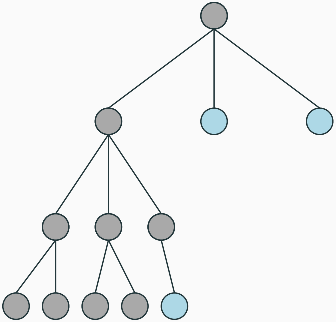
- Exhaustive Search.
- Anytime Search.

Depth First Search



Depth First Search

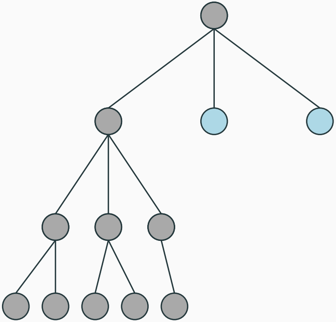
- Exhaustive Search.
- Anytime Search.



Depth First Search

- Exhaustive Search.
- Anytime Search.

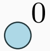
Depth First Search



Depth First Search

- Exhaustive Search.
- Anytime Search.

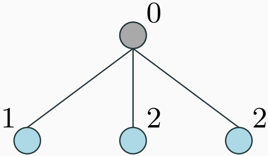
Beam Search ($D = 3$)



Beam Search

- Heuristic Search.
- Iterative Anytime version exists.

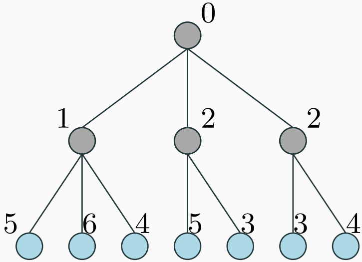
Beam Search ($D = 3$)



Beam Search

- Heuristic Search.
- Iterative Anytime version exists.

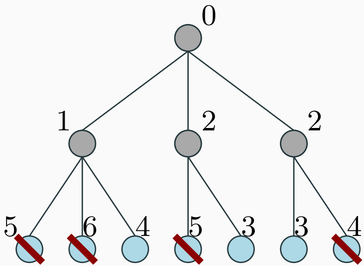
Beam Search ($D = 3$)



Beam Search

- Heuristic Search.
- Iterative Anytime version exists.

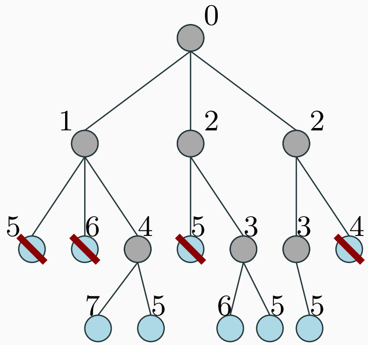
Beam Search ($D = 3$)



Beam Search

- Heuristic Search.
- Iterative Anytime version exists.

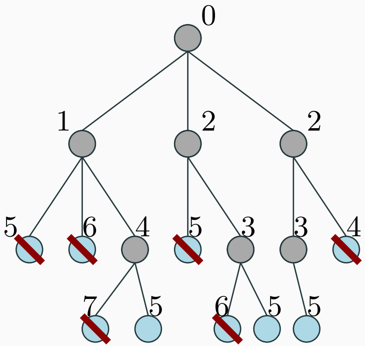
Beam Search ($D = 3$)



Beam Search

- Heuristic Search.
- Iterative Anytime version exists.

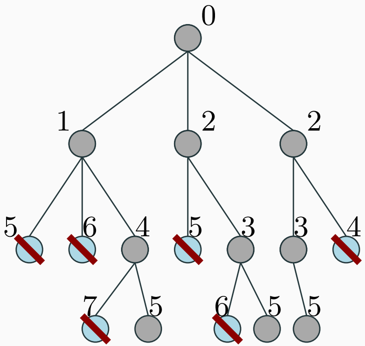
Beam Search ($D = 3$)



Beam Search

- Heuristic Search.
- Iterative Anytime version exists.

Beam Search ($D = 3$)



Beam Search

- Heuristic Search.
- Iterative Anytime version exists.

Limited Discrepancy Search

Limited Discrepancy Search

Parameters: A starting discrepancy d. An evaluation function f.

- Step 0: Open a node.
- Step 1: Sort the list of children nodes.
- Step 2: Apply discrepancy function.
- Step 3: Depth-First opening on children with discrepancy ≥ 0.

discrepancy function $d(\cdot)$

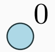
Let x_c a child of x, $d(x_c) = d(x) - k$ (with k the rank of the node x_c in the sorted list).

Limited Discrepancy Search

Limited Discrepancy Search

Parameters: A starting discrepancy d. An evaluation function f.

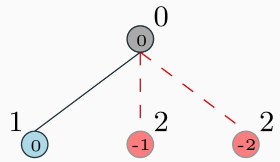
- Step 0: Open a node.
- Step 1: Sort the list of children nodes.
- Step 2: Apply discrepancy function.
- Step 3: Depth-First opening on children with discrepancy ≥ 0.



Limited Discrepancy Search

Parameters: A starting discrepancy d. An evaluation function f.

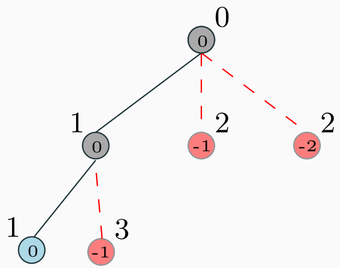
- Step 0: Open a node.
- Step 1: Sort the list of children nodes.
- Step 2: Apply discrepancy function.
- Step 3: Depth-First opening on children with discrepancy ≥ 0.



Limited Discrepancy Search

Parameters: A starting discrepancy d. An evaluation function f.

- Step 0: Open a node.
- Step 1: Sort the list of children nodes.
- Step 2: Apply discrepancy function.
- Step 3: Depth-First opening on children with discrepancy ≥ 0.

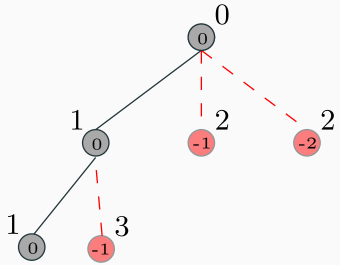


Limited Discrepancy Search

Limited Discrepancy Search

Parameters: A starting discrepancy d. An evaluation function f.

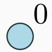
- Step 0: Open a node.
- Step 1: Sort the list of children nodes.
- Step 2: Apply discrepancy function.
- Step 3: Depth-First opening on children with discrepancy ≥ 0.



Limited Discrepancy Search

Parameters: A starting discrepancy d. An evaluation function f.

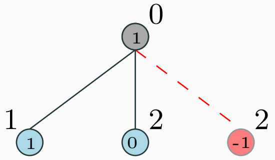
- Step 0: Open a node.
- Step 1: Sort the list of children nodes.
- Step 2: Apply discrepancy function.
- Step 3: Depth-First opening on children with discrepancy ≥ 0.



Limited Discrepancy Search

Parameters: A starting discrepancy d. An evaluation function f.

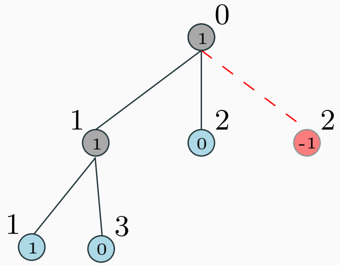
- Step 0: Open a node.
- Step 1: Sort the list of children nodes.
- Step 2: Apply discrepancy function.
- Step 3: Depth-First opening on children with discrepancy ≥ 0.



Limited Discrepancy Search

Parameters: A starting discrepancy d. An evaluation function f.

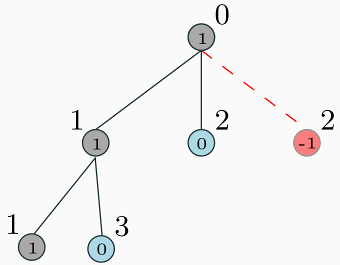
- Step 0: Open a node.
- Step 1: Sort the list of children nodes.
- Step 2: Apply discrepancy function.
- Step 3: Depth-First opening on children with discrepancy ≥ 0.



Limited Discrepancy Search

Parameters: A starting discrepancy d. An evaluation function f.

- Step 0: Open a node.
- Step 1: Sort the list of children nodes.
- Step 2: Apply discrepancy function.
- Step 3: Depth-First opening on children with discrepancy ≥ 0.

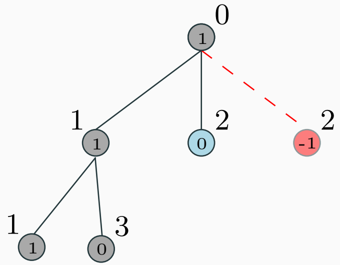


Limited Discrepancy Search

Limited Discrepancy Search

Parameters: A starting discrepancy d. An evaluation function f.

- Step 0: Open a node.
- Step 1: Sort the list of children nodes.
- Step 2: Apply discrepancy function.
- Step 3: Depth-First opening on children with discrepancy ≥ 0.

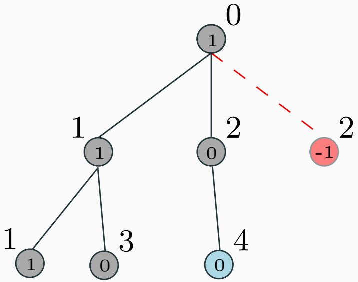


Limited Discrepancy Search

Limited Discrepancy Search

Parameters: A starting discrepancy d. An evaluation function f.

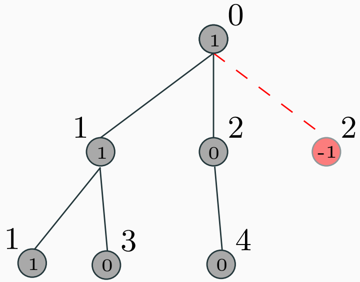
- Step 0: Open a node.
- Step 1: Sort the list of children nodes.
- Step 2: Apply discrepancy function.
- Step 3: Depth-First opening on children with discrepancy ≥ 0.



Limited Discrepancy Search

Parameters: A starting discrepancy d. An evaluation function f.

- Step 0: Open a node.
- Step 1: Sort the list of children nodes.
- Step 2: Apply discrepancy function.
- Step 3: Depth-First opening on children with discrepancy ≥ 0.

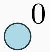


Limited Discrepancy Search

Limited Discrepancy Search

Parameters: A starting discrepancy d. An evaluation function f.

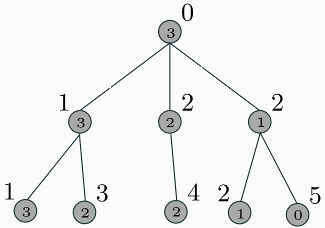
- Step 0: Open a node.
- Step 1: Sort the list of children nodes.
- Step 2: Apply discrepancy function.
- Step 3: Depth-First opening on children with discrepancy ≥ 0.



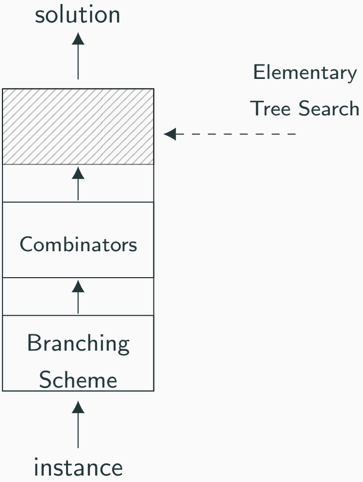
Limited Discrepancy Search

Parameters: A starting discrepancy d. An evaluation function f.

- Step 0: Open a node.
- Step 1: Sort the list of children nodes.
- Step 2: Apply discrepancy function.
- Step 3: Depth-First opening on children with discrepancy ≥ 0.



Improved Genericity - Combinators



Limited Discrepancy Search can be reproduced with : Limited Discrepancy Combinator & Depth First Search

Memorization Combinator

Memorization
Used to reproduce dynamic programming.
Can also be used to store remaining sub-problem lower bounds (memoization).

Memorization Example

Let $G = (V, E)$ be a complete graph where $V = \{0, 1, 2, 3, 4, 5, 6\}$.
We are looking for a Hamiltonian path.

Comparable partial solution: same set of chosen vertices and same last one.

[0,1,2] is comparable with [1,0,2] but not with [0,1,3] nor [5,0,2] .
The memorization combinator has to handle three different situations.

Memorization Example

Let $G = (V, E)$ be a complete graph where $V = \{0, 1, 2, 3, 4, 5, 6\}$.
We are looking for a Hamiltonian path.
Situation one *unknown solution* ; [0,1,2,3,4] value 10

Combinator memory	
partial solution	value

Memorization Example

Let $G = (V, E)$ be a complete graph where $V = \{0, 1, 2, 3, 4, 5, 6\}$.
We are looking for a Hamiltonian path.
Situation one *unknown solution* ; [0,1,2,3,4] value 10

Combinator memory	
partial solution	value
[0,1,2,3,4]	10

Memorization Example

Let $G = (V, E)$ be a complete graph where $V = \{0, 1, 2, 3, 4, 5, 6\}$.
We are looking for a Hamiltonian path.
Situation two *better solution known* ; [3,2,1,0,4] value 15

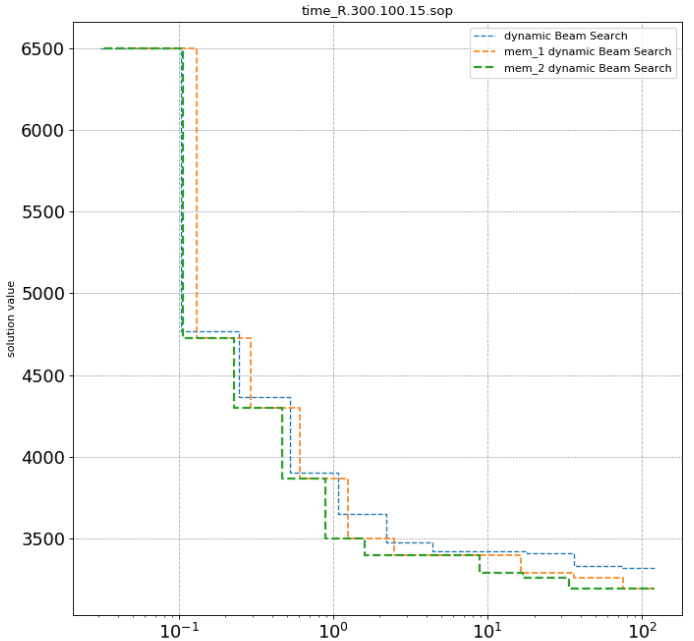
Combinator memory	
partial solution	value
[0,1,2,3,4]	10

Memorization Example

Let $G = (V, E)$ be a complete graph where $V = \{0, 1, 2, 3, 4, 5, 6\}$.
We are looking for a Hamiltonian path.
Situation three *new best solution* ; [1,0,3,2,4] value 7

Combinator memory	
partial solution	value
[1,0,3,2,4]	7

Memorization Efficiency



Sum up - Implemented Pieces

Search

(choose only one)

Beam-Search wA* SMA* Iterative Broadening

DFS MBA* Greedy Randomised BFS Best-First

Combinators

(ordered set)

Limited Discrepancy($f(.)$, n) Beam(n)

Memorization

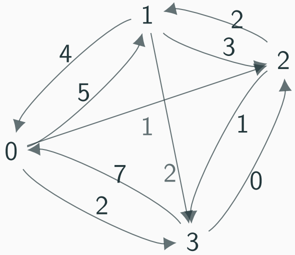
Pheromones Probing

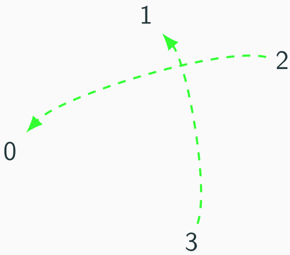
Sequential Ordering Problem

Problem Definition

Sequential Ordering Problem

It's a variant of classical Asymetric Traveling Salesman Problem which integrates precedency constraints. If a precedency constraint links *i* to *j* then *i* must be before *j* in any feasible solution.

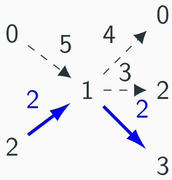


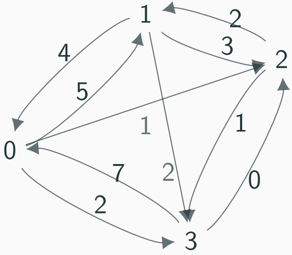


Bound

Static (in/out)-going Bound

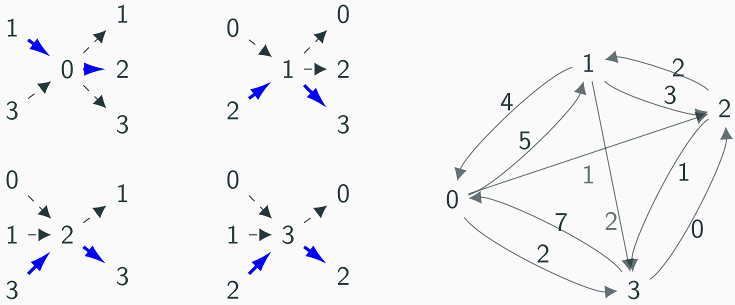
This bound stores for each vertex, its minimum weight arcs or the fixed ones.





Static (in/out)-going Bound

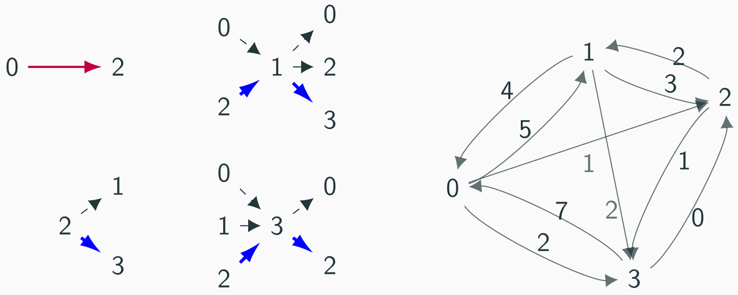
This bound stores for each vertex, its minimum weight arcs or the fixed ones.



Bound

Static (in/out)-going Bound

This bound stores for each vertex, its minimum weight arcs or the fixed ones.



Results

Instance	best known LB	best known UB	Beam Search	time to record (s)
R.500.100.1	4	4	281	-
R.500.100.15	4.628	5.284	**5.261**	61.5
R.500.1000.1	1.316	1.316	4.441	-
R.500.1000.15	43.134	49.504	**49.366**	79.2
R.600.100.1	1	1	307	-
R.600.100.15	4.803	5.493	**5.469**	75.5
R.600.1000.1	1.337	1.337	4.637	-
R.600.1000.15	47.042	55.213	**54.994**	99.5
R.700.100.1	1	1	315	-
R.700.100.15	5.946	7.021	**7.009**	439.3
R.700.1000.1	1.231	1.231	5.142	-
R.700.1000.15	54.351	65.305	**64.777**	46.7

avg [min ; max]	R.X.100.X	R.X.1000.X
R.X.X.30	1.2s [0.1 ; 3.6]	0.6s [0.1 ; 1.1]
R.X.X.60	0.0s [0.0 ; 0.0]	0.0s [0.0 ; 0.0]

Methods employed

Proposed method

Static bound - Memorization - Beam Search

Can be adapted to other problems.

State of the art

Ants - 3-exchange - Simulated Annealing.

It is a dedicated black-box.

Efficient tree-search algorithms in Optimization and Operation Research

Abdel-Malik Bouhassoun, Luc Libralesso

July 11, 2019

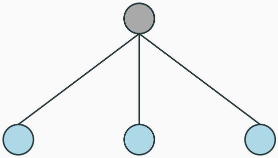
G-SCOP

Bibliography

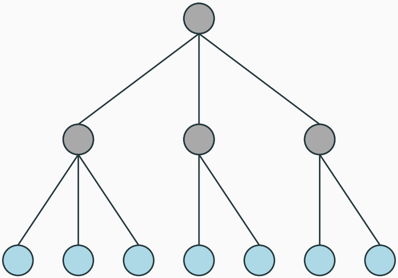
Lei Shang, Vincent T'Kindt, and Federico Della Croce.
The memorization paradigm: Branch & memorize algorithms for the efficient solution of sequencing problems.
2018.

Breadth First Search

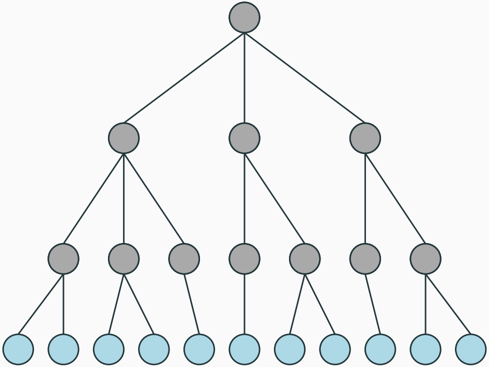
Breadth First Search



Breadth First Search

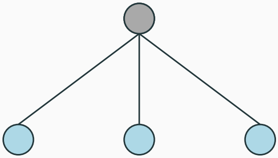


Breadth First Search

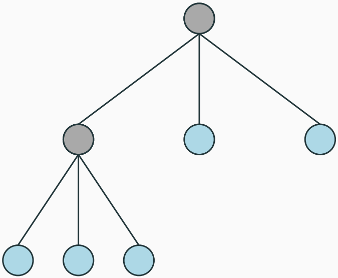


Depth First Search

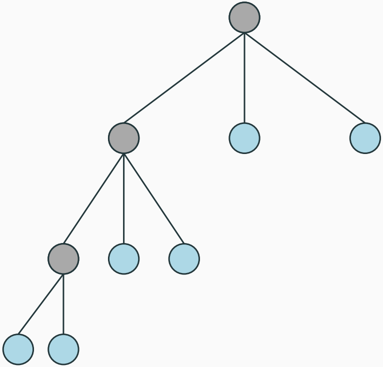
Depth First Search



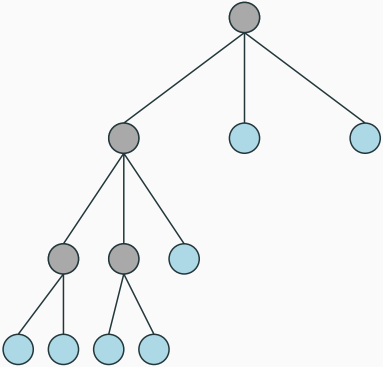
Depth First Search



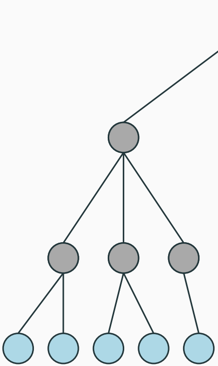
Depth First Search



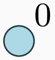
Depth First Search



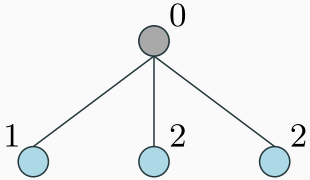
Depth First Search



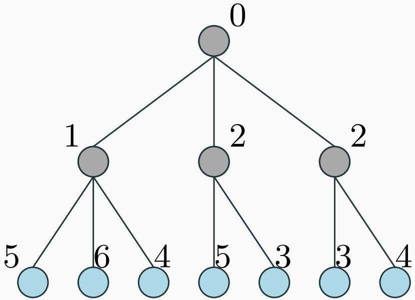
Beam Search ($D = 3$)



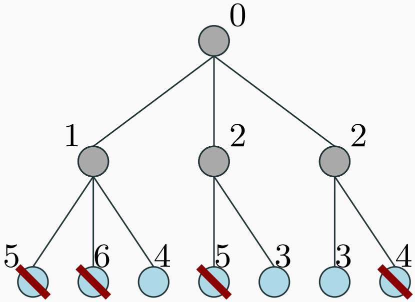
Beam Search ($D = 3$)



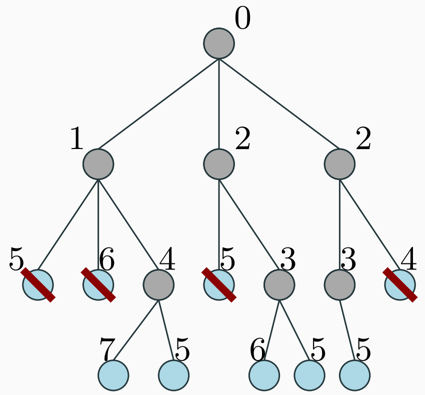
Beam Search ($D = 3$)



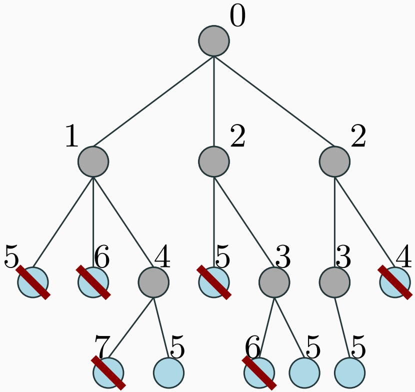
Beam Search ($D = 3$)



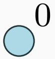
Beam Search ($D = 3$)



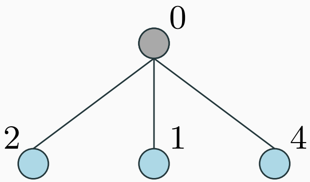
Beam Search ($D = 3$)



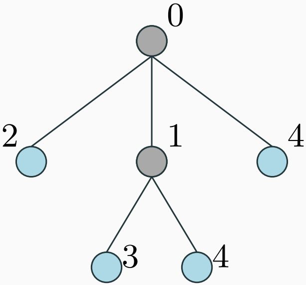
Best First



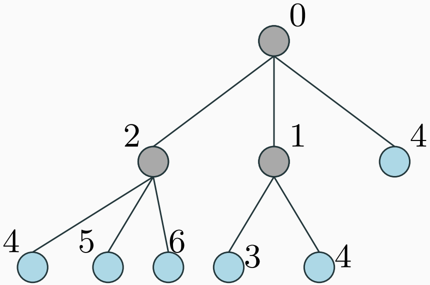
Best First



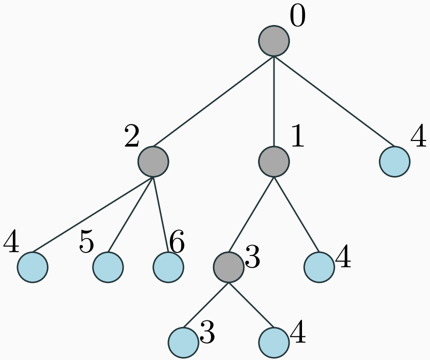
Best First



Best First



Best First



Best First