# Advanced Models and Methods in Operations Research
## Heuristic tree search

Florian Fontan

November 14, 2023

# Table of contents

# Overview

Heuristic tree search is an optimization method based on the exploration of a search tree. It is made of two ingredients:

# Overview

Heuristic tree search is an optimization method based on the exploration of a search tree. It is made of two ingredients:

▶ Branching scheme: representing the search space as an implicit decision tree.

# Overview

Heuristic tree search is an optimization method based on the exploration of a search tree. It is made of two ingredients:

▶ Branching scheme: representing the search space as an implicit decision tree.

▶ Tree search algorithm: exploring this search tree in a smart way to visit the most promising regions in priority

# Table of contents

# Definition

Branching scheme: representing the search space as an implicit decision tree.

A branching scheme is defined by:
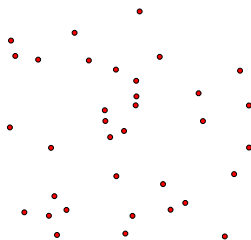
- Its root node
- How to generate the children of a node

# Example: travelling salesman problem

- Input:
    - $n$ locations
    - an $n \times n$ symmetric matrix containing the distances between each pair of locations
- Problem: find a tour such that each location is visited exactly once
- Objective: minimize the total length of the tour
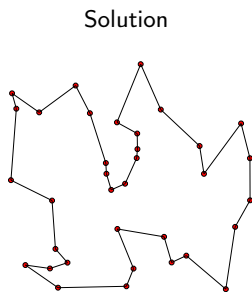
# Example: travelling salesman problem

- Input:
  - $n$ locations
  - an $n \times n$ symmetric matrix containing the distances between each pair of locations
- Problem: find a tour such that each location is visited exactly once
- Objective: minimize the total length of the tour

Instance
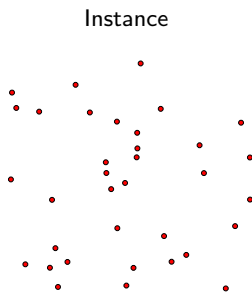
# Example: travelling salesman problem

- ▶ Input:
  - ▶ $n$ locations
  - ▶ an $n \times n$ symmetric matrix containing the distances between each pair of locations
- ▶ Problem: find a tour such that each location is visited exactly once
- ▶ Objective: minimize the total length of the tour

Instance

Solution

# Example: travelling salesman problem

Branching scheme

- ▶ A node corresponds to a partial tour
- ▶ Root node: contains only location 1
- ▶ Children of a node: append to the partial tour the next location to visit; generate one child for each remaining location to visit

# Example: travelling salesman problem

Branching scheme

- ▶ A node corresponds to a partial tour
- ▶ Root node: contains only location 1
- ▶ Children of a node: append to the partial tour the next location to visit; generate one child for each remaining location to visit

Example with 4 nodes:
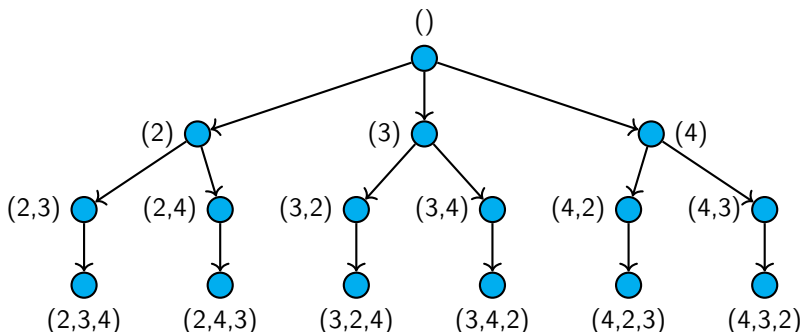
# Example: sequential ordering problem

- Input:
  - $n$ locations
  - an $n \times n$ matrix containing the distances between each pair of locations (not necessarily symmetric)
  - a directed acyclic graph $G$ such that each vertex corresponds to a location
- Problem: find a route from location 1 such that:
  - each location is visited exactly once
  - if there exists an arc from vertex $j_1$ to vertex $j_2$ in G, then location $j_1$ is visited before location $j_2$
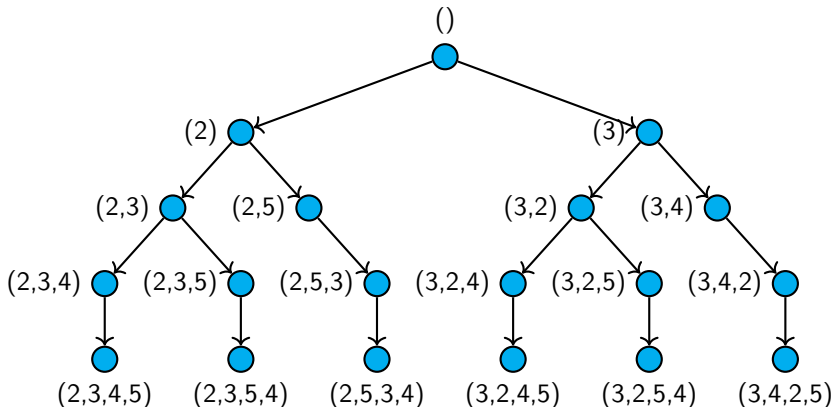- Objective: minimize the total length of the route

# Example: sequential ordering problem

- ▶ Same branching scheme as for the travelling salesman problem.

# Example: sequential ordering problem

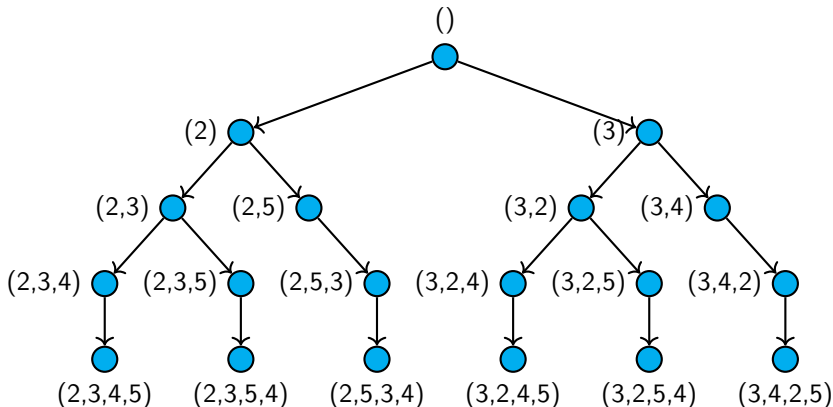▶ Same branching scheme as for the travelling salesman problem.

Example with 5 nodes, precedences: $2 \to 5$, $3 \to 4$:

# Example: sequential ordering problem

▶ Same branching scheme as for the travelling salesman problem.

Example with 5 nodes, precedences: $2 \to 5$, $3 \to 4$:



▶ Usually, more constraints $\implies$ less nodes.

# Transition

▶ These search trees usually become very large when the depth increases

# Transition

- These search trees usually become very large when the depth increases
- It is not possible to explore them exhaustively

# Transition

- These search trees usually become very large when the depth increases
- It is not possible to explore them exhaustively
- We need to find smart ways to explore the most promising nodes

# Table of contents

# Greedy algorithm

Select the best child until reaching a leaf.
The value next to the nodes is the criteria used to compare them. The lesser the value, the better the node.

0

# Greedy algorithm

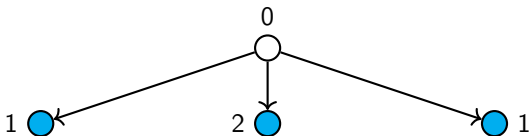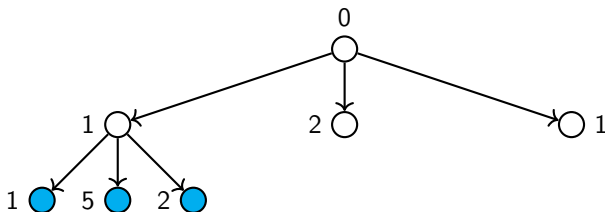Select the best child until reaching a leaf.
The value next to the nodes is the criteria used to compare them. The lesser the value, the better the node.

# Greedy algorithm

Select the best child until reaching a leaf.
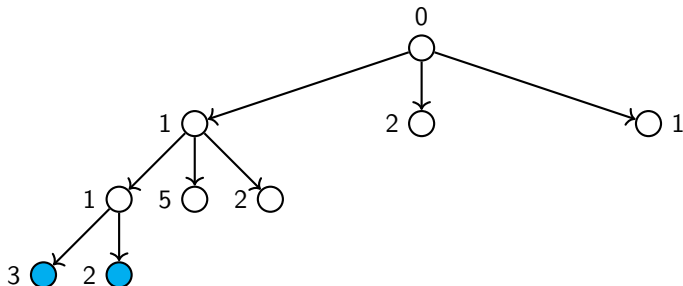The value next to the nodes is the criteria used to compare them. The lesser the value, the better the node.

# Greedy algorithm

Select the best child until reaching a leaf.
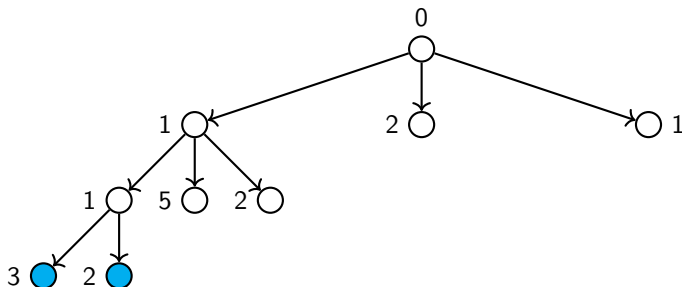The value next to the nodes is the criteria used to compare them. The lesser the value, the better the node.

# Greedy algorithm

Select the best child until reaching a leaf.
The value next to the nodes is the criteria used to compare them. The lesser the value, the better the node.



**function** Greedy(branching_scheme)
    node ← branching_scheme.root()
    **while** branching_scheme.children(node) is not empty **do**
        node ← "best" node from branching_scheme.children(node)

# Greedy algorithm

Advantages:

# Greedy algorithm

Advantages:

▶ Simple to understand and easy to implement

# Greedy algorithm

Advantages:

- ▶ Simple to understand and easy to implement
- ▶ Fast

# Greedy algorithm

Advantages:

- ▶ Simple to understand and easy to implement
- ▶ Fast

Drawbacks:

# Greedy algorithm

Advantages:

- ▶ Simple to understand and easy to implement
- ▶ Fast

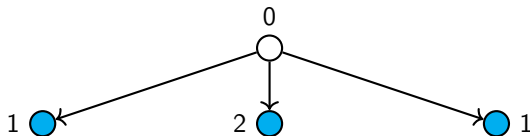Drawbacks:

- ▶ Low quality solutions

# A / best first search algorithm

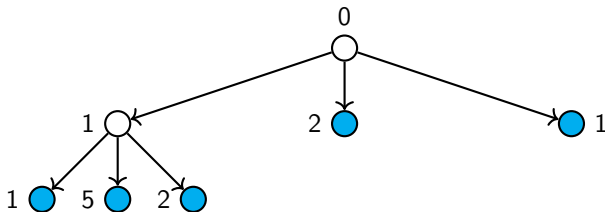At each iteration, we expand the "best" node.

0

# A / best first search algorithm

At each iteration, we expand the "best" node.

# A / best first search algorithm

At each iteration, we expand the "best" node.

# A / best first search algorithm

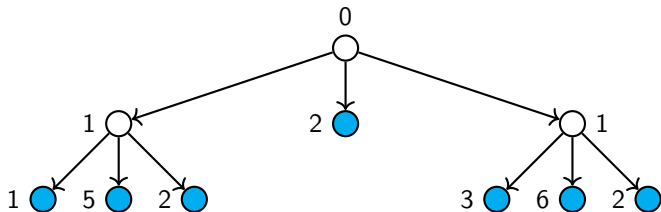At each iteration, we expand the "best" node.

# A / best first search algorithm

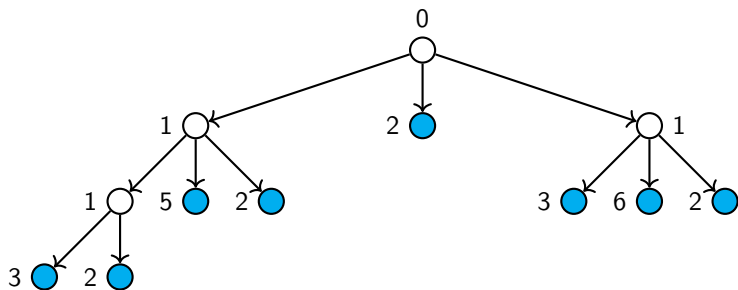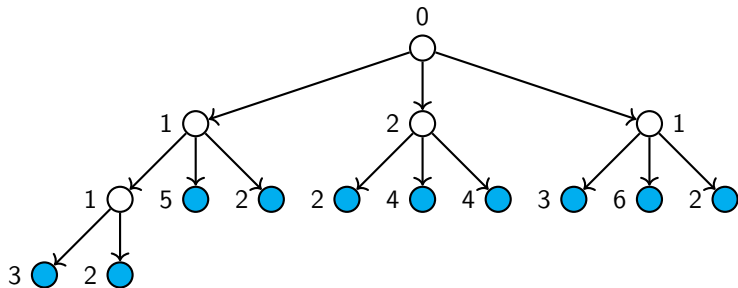At each iteration, we expand the "best" node.

# A / best first search algorithm

At each iteration, we expand the "best" node.

# A / best first search algorithm

At each iteration, we expand the "best" node.



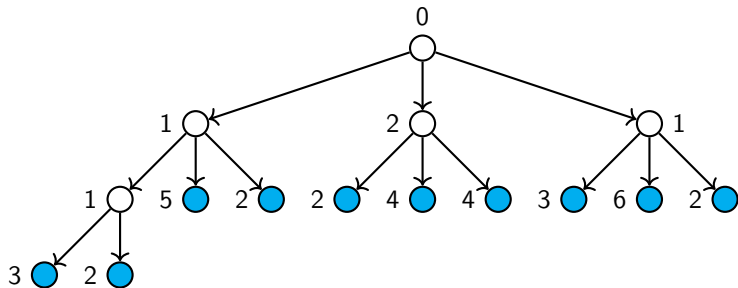**function** A(branching_scheme)
    queue ← {branching_scheme.root()}
    **while** queue is not empty **do**
        node ← extract "best" node from queue
        queue ← queue ∪ branching_scheme.children(node)

# A / best first search algorithm

Advantages:

# A / best first search algorithm

Advantages:

▶ Simple to understand and easy to implement

# A / best first search algorithm

Advantages:

▶ Simple to understand and easy to implement

▶ If the guide is a bound, then it minimizes the number of nodes explored

# A / best first search algorithm

Advantages:

▶ Simple to understand and easy to implement

▶ If the guide is a bound, then it minimizes the number of nodes explored

Drawbacks:

# A / best first search algorithm

Advantages:

- ▶ Simple to understand and easy to implement
- ▶ If the guide is a bound, then it minimizes the number of nodes explored

Drawbacks:

- ▶ It might take a long time to reach leaves (full solutions)

# A / best first search algorithm

Advantages:

- ▶ Simple to understand and easy to implement
- ▶ If the guide is a bound, then it minimizes the number of nodes explored

Drawbacks:

- ▶ It might take a long time to reach leaves (full solutions)
- ▶ The node queue quickly becomes too large

# Limited discrepancy search

Nodes are explored by increasing value of their discrepancy.

# Limited discrepancy search

Nodes are explored by increasing value of their discrepancy.
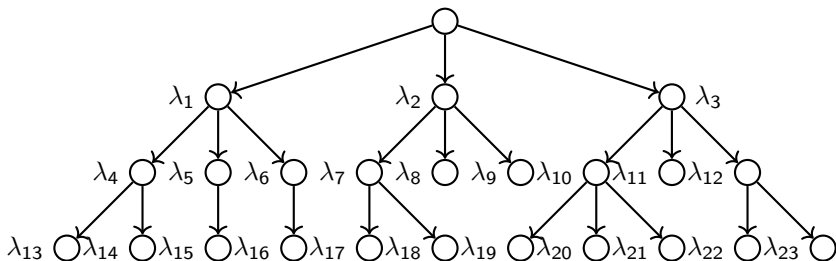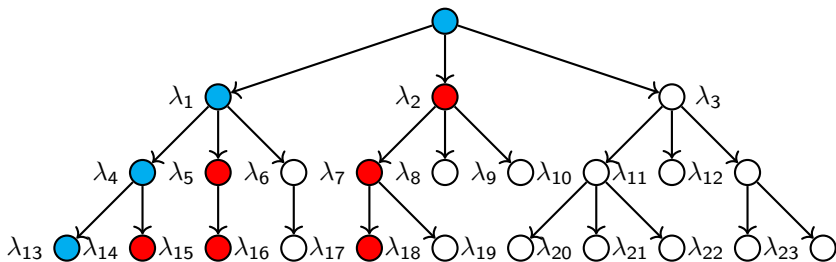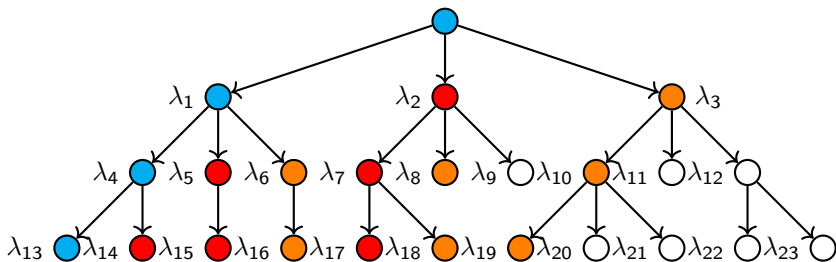
# Limited discrepancy search

Nodes are explored by increasing value of their discrepancy.

# Limited discrepancy search

Nodes are explored by increasing value of their discrepancy.

# Limited discrepancy search

Advantages:

# Limited discrepancy search

Advantages:
- ▶ Quickly reaches leaves

# Limited discrepancy search

Advantages:

- ▶ Quickly reaches leaves
- ▶ Works well with unbalanced trees

# Limited discrepancy search

Advantages:

- ▶ Quickly reaches leaves
- ▶ Works well with unbalanced trees
- ▶ A node is only compared with its brothers:
  $\implies$ very easy to design a guide

# Limited discrepancy search

Advantages:

▶ Quickly reaches leaves

▶ Works well with unbalanced trees

▶ A node is only compared with its brothers:
  $\implies$ very easy to design a guide

Drawbacks:

# Limited discrepancy search

Advantages:

- ▶ Quickly reaches leaves
- ▶ Works well with unbalanced trees
- ▶ A node is only compared with its brothers:
  $\implies$ very easy to design a guide

Drawbacks:

- ▶ Does not work as well with more balanced trees
  A node is only compared with its brothers:
  $\implies$ succession of decisions are never challenged

# Beam search

Breadth first search with a maximum width (called "beam width").
At each stage, the "worst" nodes are discarded.

Example with a beam width of 5

0

# Beam search

Breadth first search with a maximum width (called "beam width").
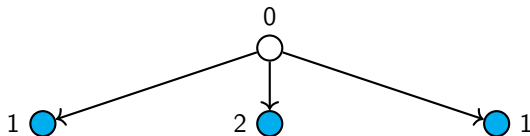At each stage, the "worst" nodes are discarded.

Example with a beam width of 5

# Beam search

Breadth first search with a maximum width (called "beam width").
At each stage, the "worst" nodes are discarded.
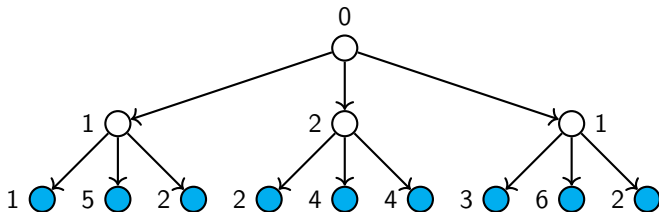
Example with a beam width of 5

# Beam search

Breadth first search with a maximum width (called "beam width").
At each stage, the "worst" nodes are discarded.

Example with a beam width of 5

# Beam search

Breadth first search with a maximum width (called "beam width").
At each stage, the "worst" nodes are discarded.
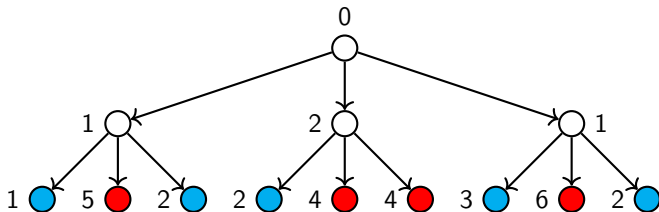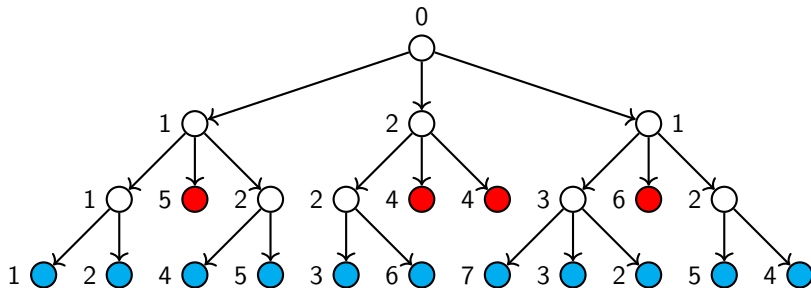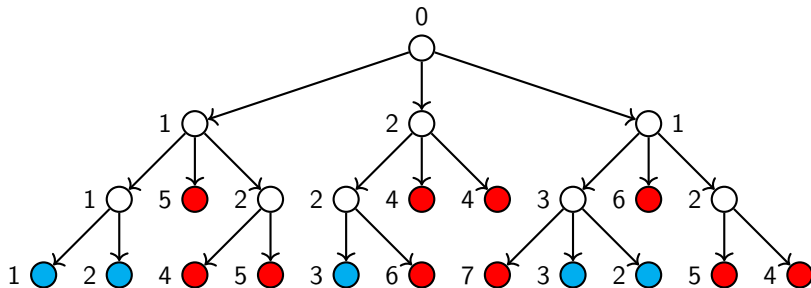
Example with a beam width of 5

# Beam search

Breadth first search with a maximum width (called "beam width").
At each stage, the "worst" nodes are discarded.

Example with a beam width of 5

# Beam search

Advantages:

# Beam search

Advantages:

- Good balance between the number of nodes explored at each depth

# Beam search

Advantages:

- ▶ Good balance between the number of nodes explored at each depth
- ▶ Only nodes at the same depth are compared with each other: easier to design a good guide

# Beam search

Advantages:

- ▶ Good balance between the number of nodes explored at each depth
- ▶ Only nodes at the same depth are compared with each other: easier to design a good guide

Drawbacks:

# Beam search

Advantages:

▶ Good balance between the number of nodes explored at each depth

▶ Only nodes at the same depth are compared with each other: easier to design a good guide

Drawbacks:

▶ How to choose the beam width?

# Iterative beam search

How to choose the beam width:

- ▶ small: close to greedy
- ▶ large: close to breadth first search

Iterative beam search:

- ▶ Succesive executions of a beam search while increasing the beam width: 1, 2, 4, 8, 16. . .
- ▶ Growth rate: between 1.25 and 2, small influence on the algorithm performances
- ▶ Anytime

# Dominances

Travelling salesman problem example:

- ▶ Node $N_1$: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$, length 10
- ▶ Node $N_2$: $1 \rightarrow 3 \rightarrow 2 \rightarrow 4$, length 11

We can safely prune Node $N_2$.

# Dominances

Travelling salesman problem example:

- Node $N_1$: $1 \rightarrow 2 \rightarrow 3 \rightarrow 4$, length 10
- Node $N_2$: $1 \rightarrow 3 \rightarrow 2 \rightarrow 4$, length 11

We can safely prune Node $N_2$.

More generally, let

- visited($N$) be the list of visited locations of node $N$.
- last($N$) be the last visited location of node $N$.
- length($N$) be the length of the partial tour of node $N$.

Consider two nodes $N_1$ and $N_2$. If

- visited($N_1$) $\supseteq$ visited($N_2$)
- last($N_1$) = last($N_2$)
- length($N_1$) $\leq$ length($N_2$)

then node $N_1$ dominates node $N_2$ and therefore node $N_2$ can be safely pruned.

# A / best first search + dynamic programming
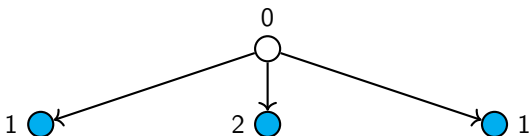
Each time a node is added to the queue:

▶ if it is dominated by another node which is in the queue, it is not added

▶ the nodes from the queue that it dominates are removed from the queue

# A / best first search + dynamic programming
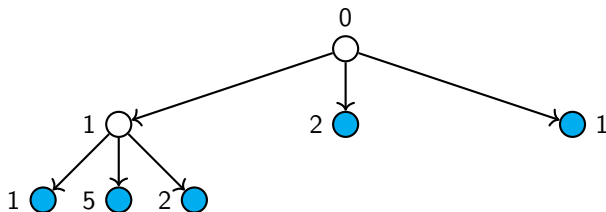
Each time a node is added to the queue:

▶ if it is dominated by another node which is in the queue, it is not added

▶ the nodes from the queue that it dominates are removed from the queue

# A / best first search + dynamic programming

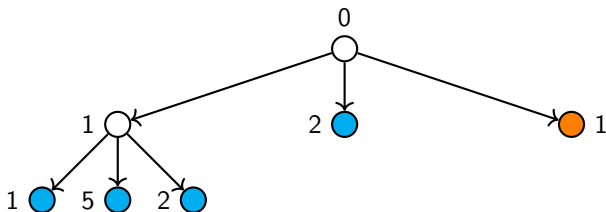Each time a node is added to the queue:

- ▶ if it is dominated by another node which is in the queue, it is not added
- ▶ the nodes from the queue that it dominates are removed from the queue

# A / best first search + dynamic programming

Each time a node is added to the queue:

▶ if it is dominated by another node which is in the queue, it is not added

▶ the nodes from the queue that it dominates are removed from the queue

# A / best first search + dynamic programming
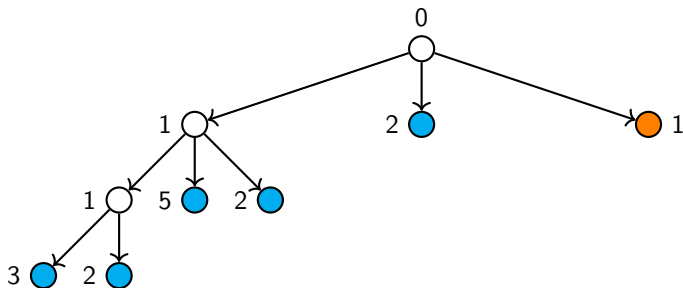
Each time a node is added to the queue:

▶ if it is dominated by another node which is in the queue, it is not added
▶ the nodes from the queue that it dominates are removed from the queue

# A / best first search + dynamic programming

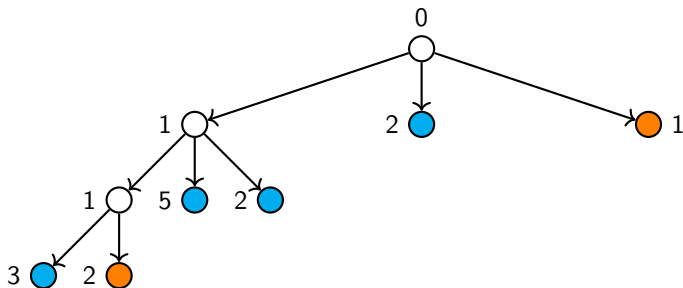Each time a node is added to the queue:
- ▶ if it is dominated by another node which is in the queue, it is not added
- ▶ the nodes from the queue that it dominates are removed from the queue

# A / best first search + dynamic programming

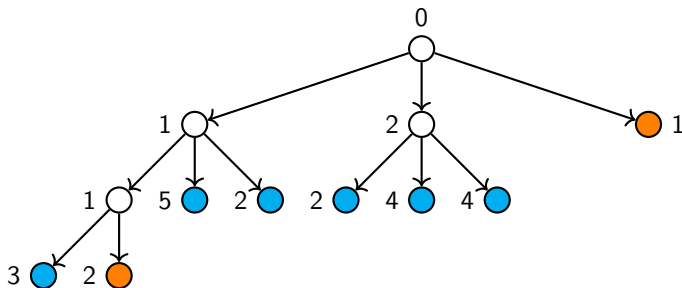Each time a node is added to the queue:

- ▶ if it is dominated by another node which is in the queue, it is not added
- ▶ the nodes from the queue that it dominates are removed from the queue

# Beam search + dynamic programming

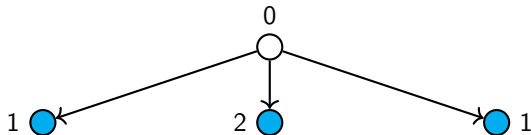Before discarding the "worst" nodes, the dominated ones are first removed.

Beam width: 5

0

# Beam search + dynamic programming

Before discarding the "worst" nodes, the dominated ones are first removed.

Beam width: 5

# Beam search + dynamic programming

Before discarding the "worst" nodes, the dominated ones are first removed.

Beam width: 5

# Beam search + dynamic programming

Before discarding the "worst" nodes, the dominated ones are first removed.

Beam width: 5

# Beam search + dynamic programming

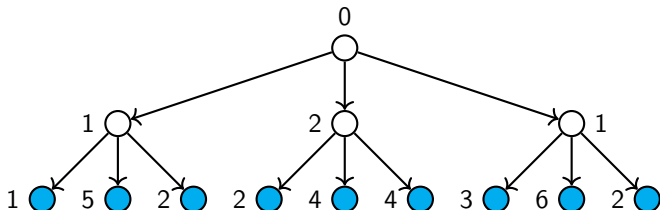Before discarding the "worst" nodes, the dominated ones are first removed.

Beam width: 5

# Beam search + dynamic programming

Before discarding the "worst" nodes, the dominated ones are first removed.

Beam width: 5

# Beam search + dynamic programming

Before discarding the "worst" nodes, the dominated ones are first removed.

Beam width: 5

# Beam search + dynamic programming

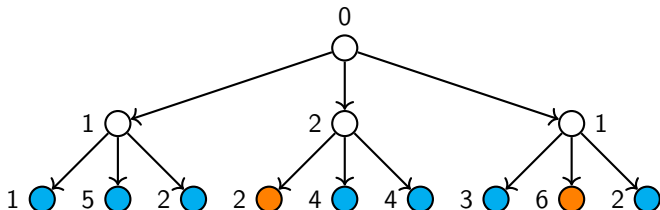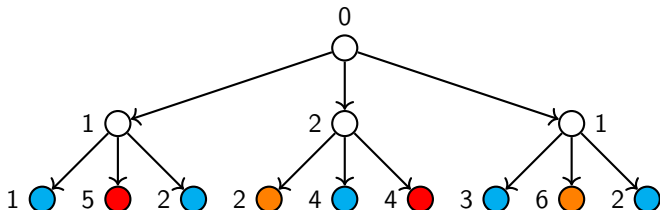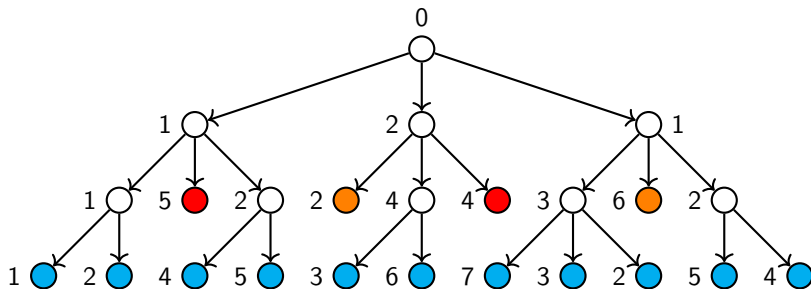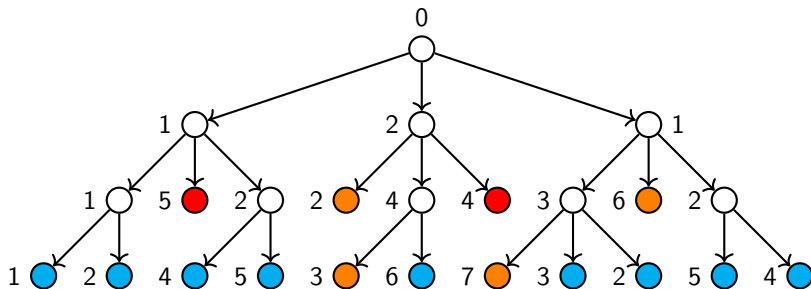Before discarding the "worst" nodes, the dominated ones are first removed.

Beam width: 5

# Table of contents

# Guides

All the previously presented algorithms require a criteria to compare nodes:

- ▶ First idea: define and use the objective of the partial solutions:
    - ▶ Examples:
        - ▶ Travelling salesman problem: length of the partial tour
    - ▶ Advantage: simple, might be good as a first approach
    - ▶ Drawbacks: does not take into account the rest of the solution
        - ▶ Travelling salesman problem: a forgotten location near the first ones

# Guides

All the previously presented algorithms require a criteria to compare nodes:

- ▶ First idea: define and use the objective of the partial solutions:
  - ▶ Examples:
    - ▶ Travelling salesman problem: length of the partial tour
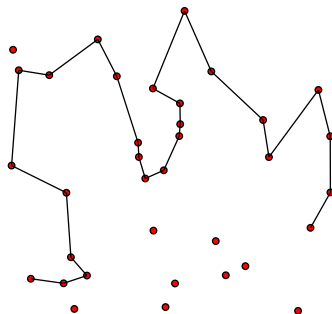  - ▶ Advantage: simple, might be good as a first approach
  - ▶ Drawbacks: does not take into account the rest of the solution
    - ▶ Travelling salesman problem: a forgotten location near the first ones

# Guides

- The criteria needs to take into account what is and what is not in the partial solution
- Other ideas: bound, expected value of the best reachable leaf...
- The complexity of computing the score needs to be taken into account. A better score with a greater complexity might not be worth
- In practice, heuristic tree search does not work well for the traveling salesman problem, but it provides state-of-the-art results on instances of the sequential ordering problem with a dense precedence graph

# Example: two-dimensional guillotine knapsack problem

- ▶ Input
  - ▶ A bin with width $W$ and height $H$
  - ▶ $n$ items; for each item $j = 1, \ldots, n$, a width $w_j$, a height $h_j$ and a profit $p_j$
- ▶ Problem: find a 3-staged guillotine cutting plan such that:
  - ▶ each item is cut at most once
- ▶ Objective: maximize the total profit of the item cut
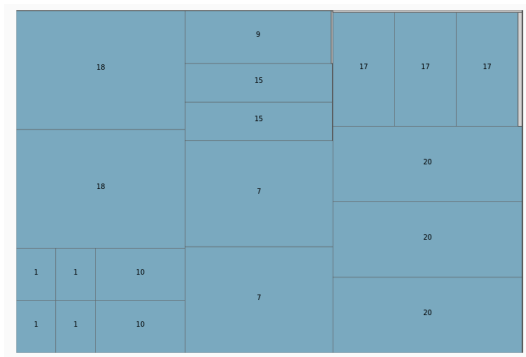
# Example: two-dimensional guillotine knapsack problem

Guillotine cutting plan: items can be extracted with only edge-to-edge cuts:
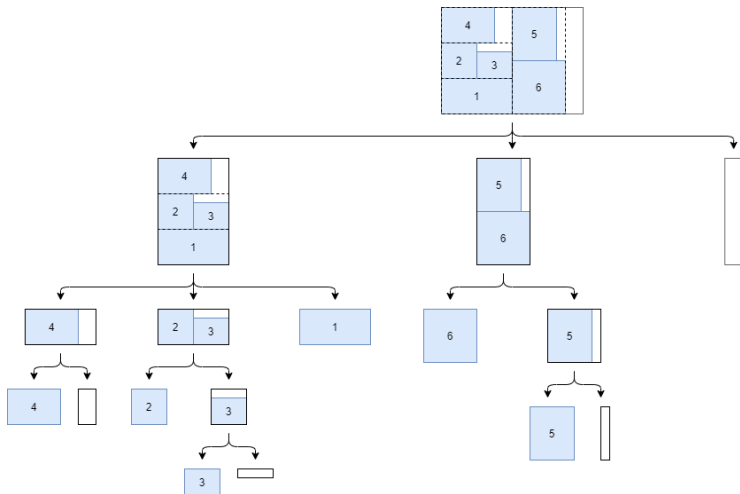
# Example: two-dimensional guillotine knapsack mroblem
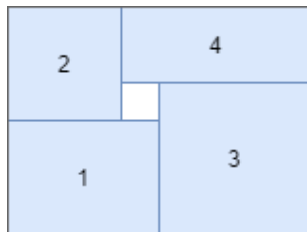


Non-guillotine pattern
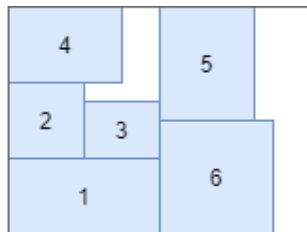
Guillotine pattern

# Example: two-dimensional guillotine knapsack problem

Order of the items in a cutting plan:
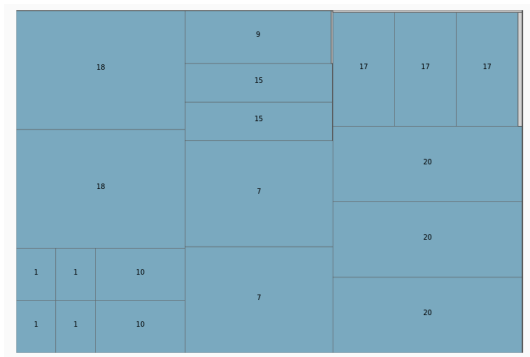
# Example: two-dimensional guillotine knapsack problem

Order of the items in a cutting plan:
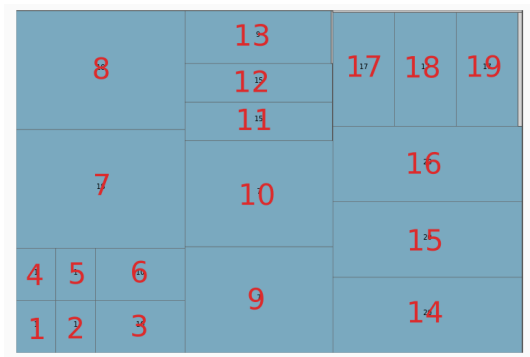
# Example: two-dimensional guillotine knapsack problem

Order of the items in a cutting plan:
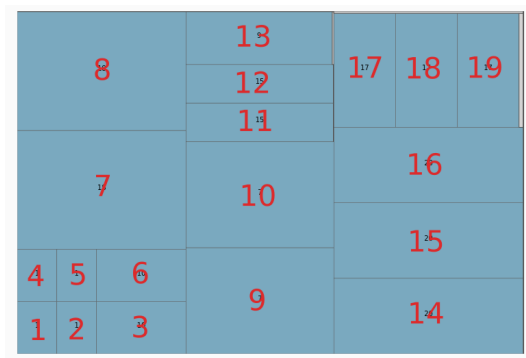


Branching scheme:
- Root node: empty solution, no item
- Add the next item (following the order defined above) to the partial solution; generate one child for each remaining item at each possible position

# Example: two-dimensional guillotine knapsack mroblem

There are at most 3 valid positions to pack a next item:

- ▶ In the current second-level subplate, in a new third-level subplate
- ▶ In the current first-level subplate, a new second-level subplate
- ▶ In a new first-level subplate
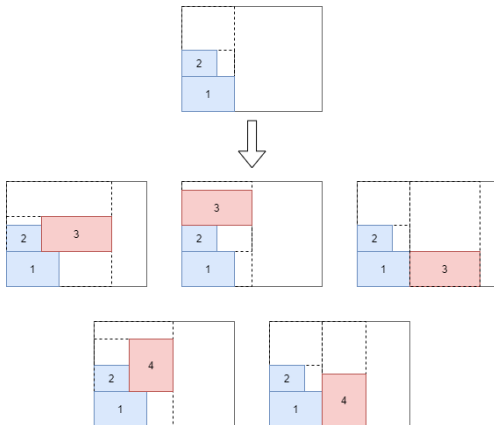
# Example: two-dimensional guillotine knapsack problem

Guiding the search:

▶ First idea, guide with the profit of the partial solution:

$$\frac{1}{\text{profit}(S)}$$

▶ This is not as bad as for the traveling salesman problem, since a wrong choice may less likely lead to a very bad final solution.

▶ Still, with this guide, we favor solutions containing high profit items, even if these packed items are not tightly packed
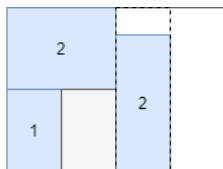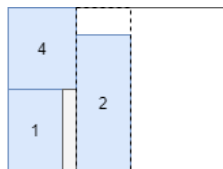


Partial solution A                    Partial solution B

If the profit of an item is equal to its area, then partial solution A is more profitable than partial solution B.
Still, partial solution B seems more favorable since it contains less waste

# Example: two-dimensional guillotine knapsack problem

One way to overcome this issue is to guide with:

$$\frac{\text{area}(S)}{\text{profit}(S)}$$

where $\text{area}(S)$ is the used area of a partial solution $S$ as illustrated below:
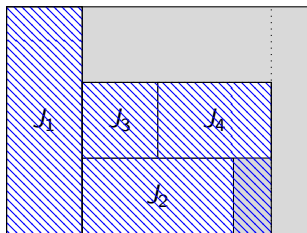
# Example: two-dimensional guillotine knapsack problem

- In some cases, this guide favors partial solutions with many small items since it is easier to generate partial solutions with less waste using small items

- Then, at the end, only large items remains and packing them generates a lot of waste

- One way to overcome this issue is to integrate the mean area of the packed items in the guide:

$$\frac{\text{area}(S)}{\text{profit}(S)} \frac{1}{\text{mean\_item\_area}(S)}$$

- In practice, we use the last two guide functions

# Example: ROADEF/EURO 2022/2023 challenge

3D knapsack subproblem:
- ▶ Input
    - ▶ A bin with length $L$, width $W$ and height $H$
    - ▶ $n$ items; for each item $j = 1, \ldots, n$, a length $l_j$, a width $w_j$, a height $h_j$, a quantity $d_j$, a profit $p_j$...
- ▶ Problem: find a packing plan such that each constraint is satisfied
- ▶ Objective: maximize the total profit of the items packed

# Packing constraints: stacks

The third dimensions is only possible through stacks

▶ A stack necessarily contains items with the same length and width
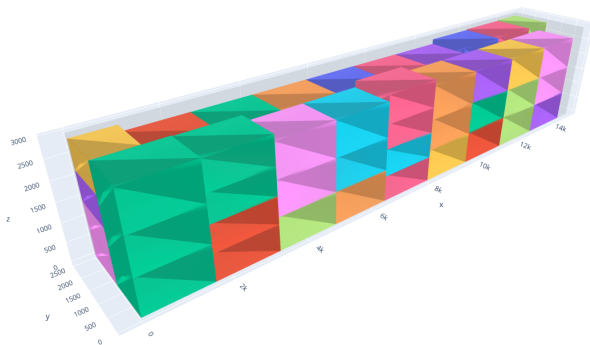
# Packing constraints: stacks

Any stack must be adjacent to another stack on its left on the X-axis, or to the front of the truck (adjacent to the truck driver)
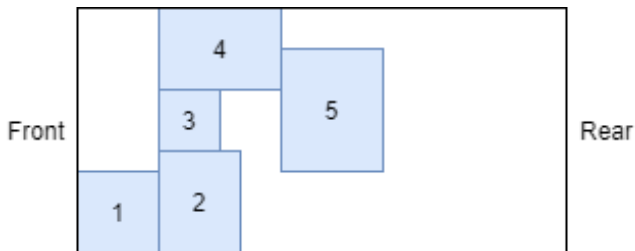


View from the top.
The placement of stacks 3 and 4 is not allowed.

# Packing constraints: nesting height

For some items, their height is reduced when they are packed above another item.
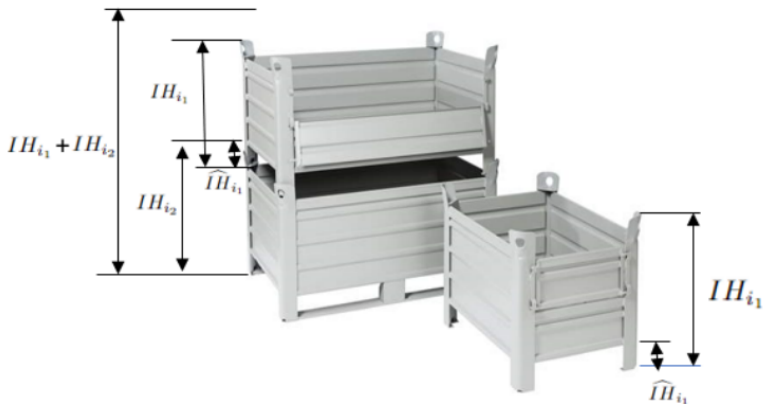
# Packing constraints: loading order

Loading order:

▶ Stacks must be placed in an increasing fashion from the front to the rear of the truck according to the suppliers' pickup order
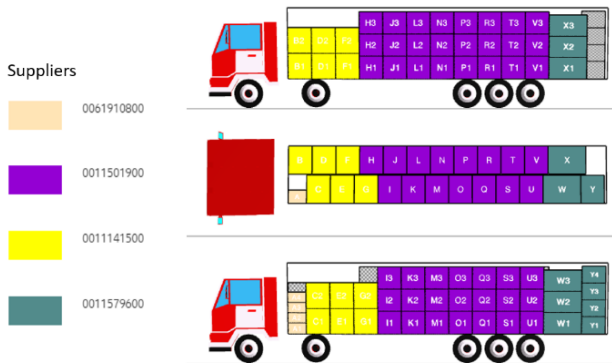


Figure 5: A truck with 4 picked-up suppliers

# Packing constraints: axle weights

Maximum weight on the middle and rear axles of the truck

▶ Formulas to compute axle weights:

$$ej^e = \frac{\sum_{s \in \widetilde{TS}_t} (sx_s^o + \frac{(sx_s^e - sx_s^o)}{2}) \times sm_s}{tm_t}$$

$$ej^r = EJ^{eh} + EJ^{hr} - ej^e$$

$$em^h = \frac{tm_t \times ej^r + EM \times EJ^{cr}}{EJ^{hr}}$$

$$em^r = tm_t + EM - em^h$$

$$em^m = \frac{CM \times CJ^{fc} + em^h \times CJ^{fh}}{CJ^{fm}}$$

# Packing constraints: axle weights
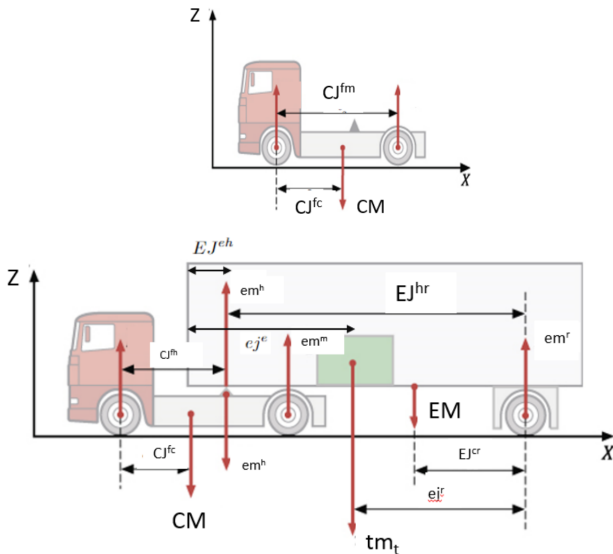
# Packing constraints: axle weights

The maximum axle weight constraints has some counter-intuitive properties:

- ▶ The solution on the right is infeasible
  - ▶ The weight on the middle axle weight is too high
- ▶ The solution on the right is feasible

Adding some items to a solution without removing or moving the current items might decrease the axle weights!
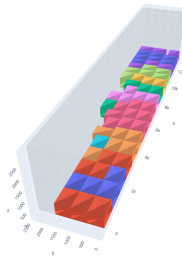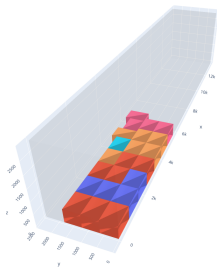
# Packing constraints: other constraints

Remaining constraints:

- Maximum weight allowed in the truck
- For each item, maximum weight allowed above
- For each item, maximum number of items in a stack containing this item
- Maximum density of a stack

# Solution method

To solve the 3D knapsack subproblems, we design two algorithms:

- ▶ An algorithm that decomposes the 3D problem into
  - ▶ A 1D problem that builds stacks
    - ▶ Build the least number of stacks with all the items
    - ▶ This is a 1D bin packing problem
  - ▶ A 2D packing problem that packs these stacks.
    - ▶ This is a 2D knapsack problem
  - ▶ This algorithm works very well when axle weight constraints are not critical
- ▶ An algorithm that directly builds 3D packings
  - ▶ This algorithm works very well when weight constraints are critical and therefore, sparse packings are needed

# 3D knapsack subproblem, sequential 1D 2D
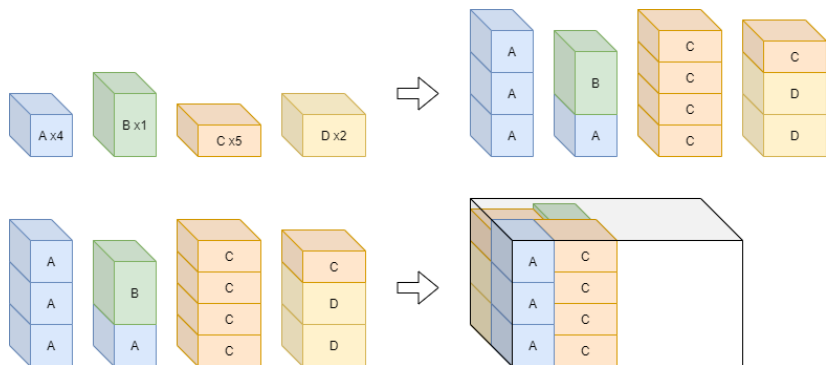
# 1D bin packing subproblem

- For a given set of items, we want to pack them into the least number of stacks
- Remaining constraints:
  - Maximum height of a stack (height of the truck)
  - Nesting height
  - Maximum number of items in a stack containing an item of a given type
  - Maximum weight of a stack (stack density)
  - Maximum weight allowed above an item of a given type
- We decompose the problem into a sequence of (1D) knapsack subproblems
- The 1D knapsack subproblems are solved with a tree search algorithm

# 1D knapsack subproblem, branching scheme

Branching scheme:

- ▶ Root node: empty stack
- ▶ Children of a node: we generate a child node for each item which is valid to add *on top* of the stack
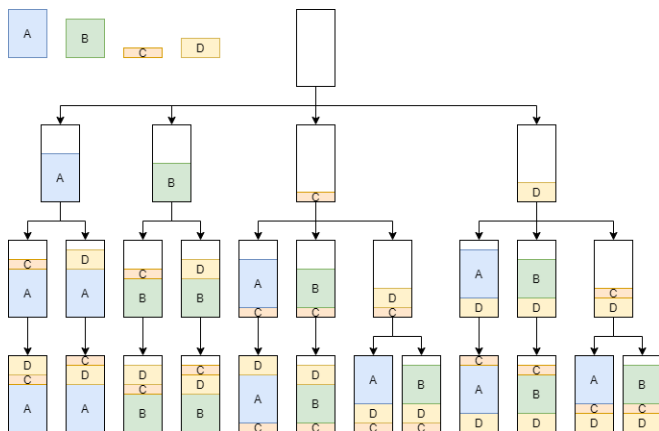
# 2D knapsack problem

- ▶ For a set of given stacks, we want to pack as many as possible of them inside a truck
- ▶ Remaining constraints:
  - ▶ Geometrical constraints
  - ▶ Maximum weight of the truck
  - ▶ Loading order
  - ▶ We ignore axle weight constraints

# 2D knapsack problem, branching scheme

We solve this 2D knapsack subproblem with a tree search algorithm based on a skyline
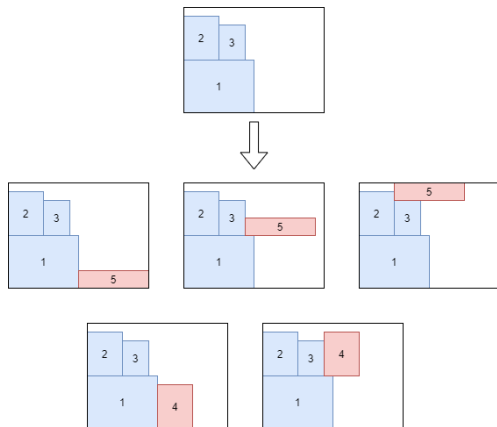
# 3D knapsack problem, direct approach

▶ Our second approach to solve the 3D knapsack problem is a tree search algorithm that directly builds 3D solutions

▶ The branching scheme is similar to the 2D case, except that we allow adding an item above one of the "last" stack
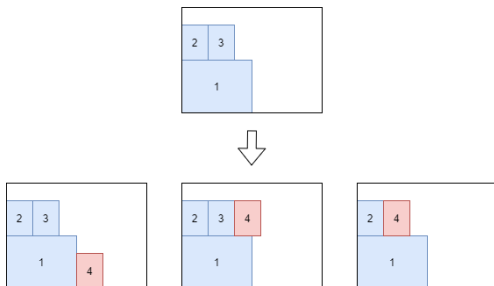
# 3D knapsack problem, direct approach

- ▶ We allow partial solutions that do not satisfy the weight distribution constraints, since the path to feasible full solutions often contains infeasible partial solutions
- ▶ But we guide the search towards full feasible solutions

# Guides for 3D packing

- The SOR algorithm fails to find a good enough solution when the middle axle weight constraint is so critical, that a sparse packing is required to get a good feasible solution
- The goal of the 3D packing algorithm is to generate such a sparse solution
- The solution must be sparse enough to be feasible, but dense enough to pack as many items as possible
- In particular, the sparsity is mostly important at the front of the truck, since the items packed at the front contribute more to the middle axle weight

# Guides for 3D packing

Here is an example of a sparse solution (from top):
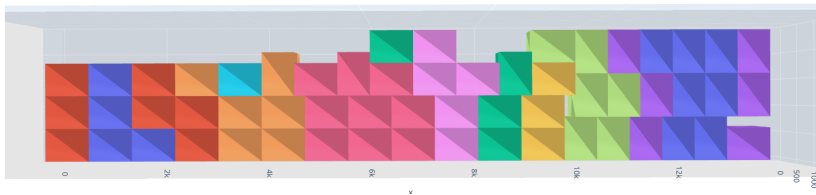
# Guides for 3D packing

- To generate dense solutions with the SOR algorithm, we guide the search with:

$$\frac{\text{space of the items packed}}{\text{space used}}$$

- To generate sparse solutions with the 3D tree search algorithm, we estimate in advance, depending on the current number of items packed
    - The maximum length that the partial solution should not exceed
    - The maximum middle axle weight that the partial solution should not exceed

  and the guide looks like

$$\text{length excess} + \text{middle axle weight excess}$$

  How do we generate these estimates?

# Guides for 3D packing

- We consider a one-dimensional problem with only the x-component
- We ignore item overlap
- Let $n$ be the number of items that we plan to pack
- Let $L$ be the length of the truck
- We consider that the position of the $j$th item is given by

$$x_j = L \sqrt[\alpha]{1 - \left(1 - \frac{j}{n}\right)^{\alpha}}$$

# Guides for 3D packing

Here is an example for $L = 12000$, $n = 50$ and different values of $\alpha$:
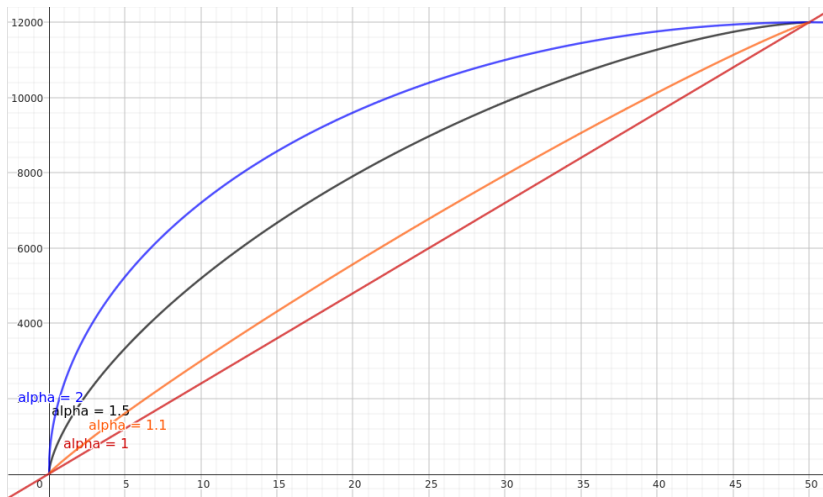
# Guides for 3D packing

- When $\alpha = 1$, items are uniformely distributed inside the truck
- When $\alpha$ increases the front of the truck becomes sparser of the rear of the truck becomes denser
- We look for the smallest $\alpha$ for which the solution satisfies the middle axle weight constraints
- We find this value with a dichotomic search
- Then we deduce the maximum length and maximum middle axle weight for each number of items

# Table of contents

# Heuristic tree search vs LP-based branch-and-bound

# Heuristic tree search vs LP-based branch-and-bound

- An LP-based branch-and-bound is also a tree search:
  - Root node: no variable bounds have been tightened
  - Children: compute the relaxation, select a fractional variable, divide its domain in two and generate one child for each part

# Heuristic tree search vs LP-based branch-and-bound

- ▶ An LP-based branch-and-bound is also a tree search:
  - ▶ Root node: no variable bounds have been tightened
  - ▶ Children: compute the relaxation, select a fractional variable, divide its domain in two and generate one child for each part
- ▶ The goal is to explore all nodes, or at least a good fraction of them

# Heuristic tree search vs LP-based branch-and-bound

- ▶ An LP-based branch-and-bound is also a tree search:
  - ▶ Root node: no variable bounds have been tightened
  - ▶ Children: compute the relaxation, select a fractional variable, divide its domain in two and generate one child for each part
- ▶ The goal is to explore all nodes, or at least a good fraction of them
- ▶ To reduce the number of nodes, an expensive bound is computed in each node

# Heuristic tree search vs LP-based branch-and-bound

- ▶ An LP-based branch-and-bound is also a tree search:
  - ▶ Root node: no variable bounds have been tightened
  - ▶ Children: compute the relaxation, select a fractional variable, divide its domain in two and generate one child for each part
- ▶ The goal is to explore all nodes, or at least a good fraction of them
- ▶ To reduce the number of nodes, an expensive bound is computed in each node
- ▶ Works better than a heuristic tree search approach if the bounds are strong
  - ▶ Example: arc-flow formulation of a bin packing problem

# Heuristic tree search vs LP-based branch-and-bound

- An LP-based branch-and-bound is also a tree search:
  - Root node: no variable bounds have been tightened
  - Children: compute the relaxation, select a fractional variable, divide its domain in two and generate one child for each part
- The goal is to explore all nodes, or at least a good fraction of them
- To reduce the number of nodes, an expensive bound is computed in each node
- Works better than a heuristic tree search approach if the bounds are strong
  - Example: arc-flow formulation of a bin packing problem
- Performs poorly if the bounds are weak (a lot of time is spent in the nodes, but the number of nodes remains too high)
  - Example: two-dimensional bin packing

# Table of contents

# treesearchsolverpy

- ▶ A package that simplifies the implementation of tree search based algorithms
- ▶ Written in Python3 (original version in C++)
- ▶ https://github.com/fontanf/treesearchsolverpy
- ▶ Install with: pip3 install treesearchsolverpy
- ▶ It includes an iterative beam search + dynamic programming
- ▶ To solve a problem, one needs to create a BranchingScheme class that implements the requried methods (about 100–200 lines of code). Then:
  iterative_beam_search(branching_scheme)

- ▶ For the branching scheme:
  - ▶ Node class with `__lt__(self, other)` (guide)
  - ▶ `root()` method
  - ▶ `next_child(father)` method
  - ▶ `infertile(node)` method
  - ▶ `leaf(node)` method
  - ▶ `bound(node_1, node_2)` method
- ▶ For the solution pool:
  - ▶ `better(node_1, node_2)` method (main objective, not guide)
  - ▶ `equals(node_1, node_2)` method (same solution, not same objective value)
- ▶ For the dominances:
  - ▶ `comparable(node)` method
  - ▶ Bucket class with `__init__(self, node)`, `__hash__(self)` and `__eq__(self, other)`
  - ▶ `dominates(node_1, node_2)` method (called only if both nodes are in the same bucket)
- ▶ `display(node)` method

# Table of contents

# Conclusion

- Heuristic tree search: branching scheme + tree search algorithm (+ guides, dominances)
- New optimization method to add to your toolbox
- As for all other methods, does not work well for all problems
- Works well for medium-sized problems
    - depth $\leq 1000$
- Works well for problems with many constraints
- Rather robust to the addition of new constraints
- Less robust to changes in the objective function

# Advanced Models and Methods in Operations Research
## Heuristic tree search

Florian Fontan

November 14, 2023