Advanced Models and Methods in Operations Research
Heuristic Tree Search

Florian Fontan

November 15, 2022

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

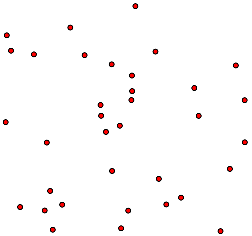
Example: Travelling Salesman Problem

- 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

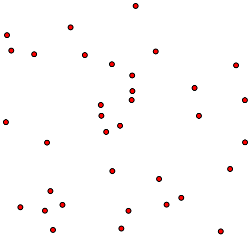
Instance



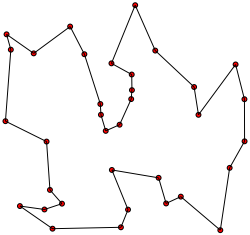
Example: Travelling Salesman Problem

- 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

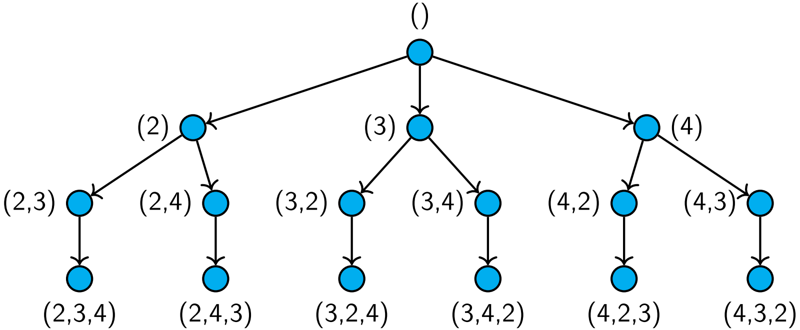
- A node corresponds to a partial tour
- Root node: contains only location 1
- Children of a node: append to the partial tour the next location to visit; generate one child for each remaining location to visit

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

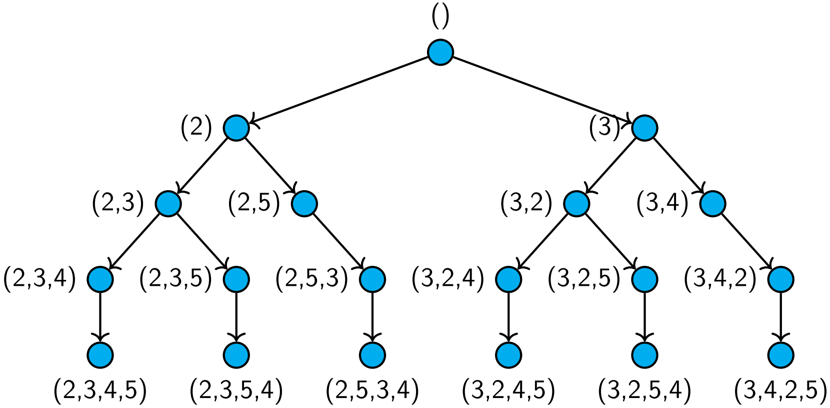
Example: Sequential Ordering Problem

- Same branching scheme as for the Travelling Salesman Problem.

Example: Sequential Ordering Problem

- Same branching scheme as for the Travelling Salesman Problem.

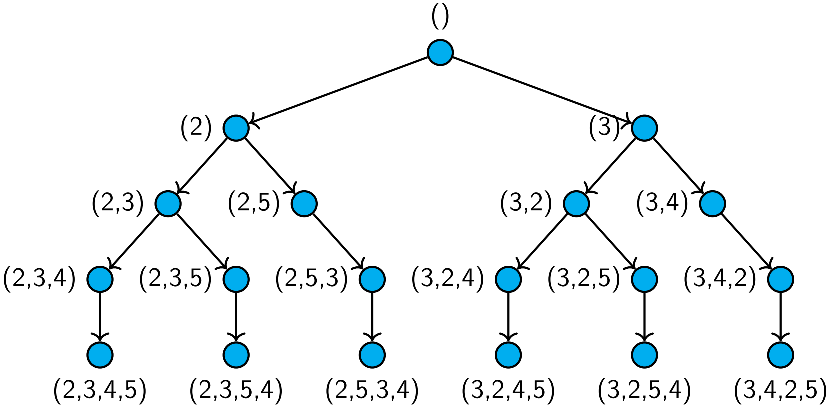
Example with 5 nodes, precedences: $2 \to 5$, $3 \to 4$:



Example: Sequential Ordering Problem

- Same branching scheme as for the Travelling Salesman Problem.

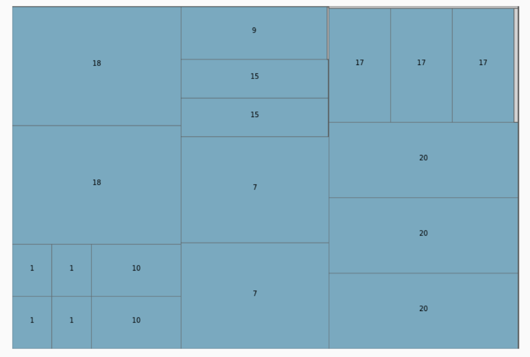
Example with 5 nodes, precedences: $2 \rightarrow 5$, $3 \rightarrow 4$:



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

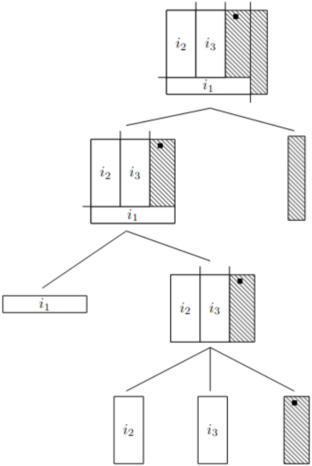
Example: Two-dimensional guillotine Knapsack Problem

- 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

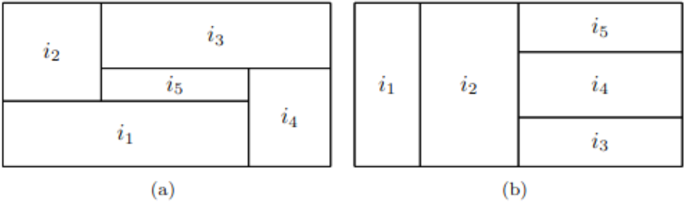


Example: Two-dimensional guillotine Knapsack Problem

Guillotine cutting plan: items can be extracted with only edge-to-edge cuts:



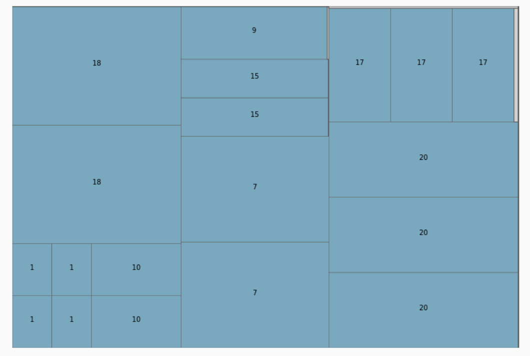
Example: Two-dimensional guillotine Knapsack Problem



- (a): non-guillotine cutting plan
- (b): guillotine cutting plan

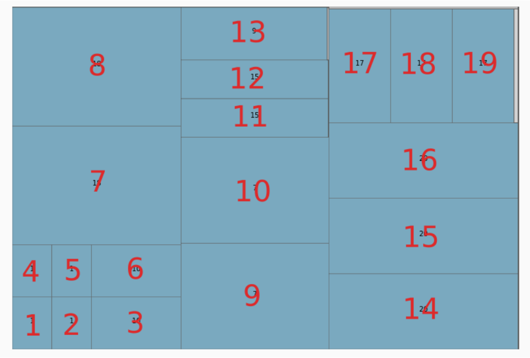
Example: Two-dimensional guillotine Knapsack Problem

Order of the items in a cutting plan:



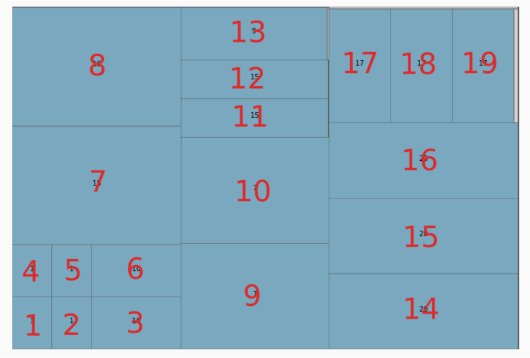
Example: Two-dimensional guillotine Knapsack Problem

Order of the items in a cutting plan:



Example: Two-dimensional guillotine Knapsack Problem

Order of the items in a cutting plan:

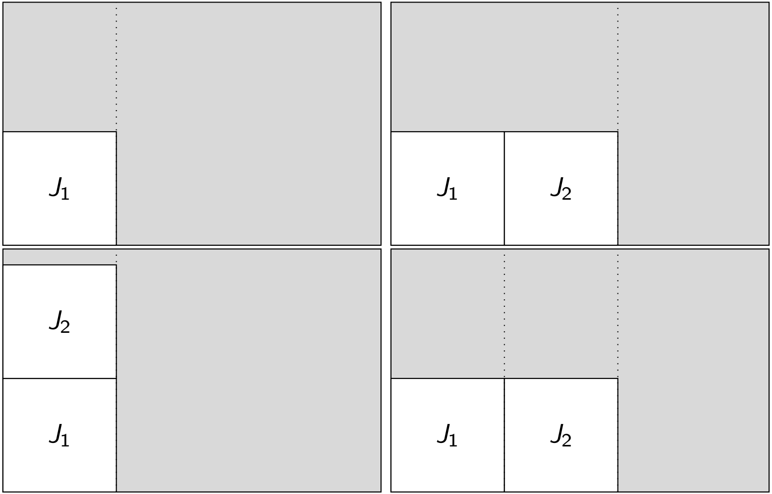


Branching scheme:

- 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 Problem

There are three ways to position a next item J_2:



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

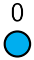
Transition

- 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

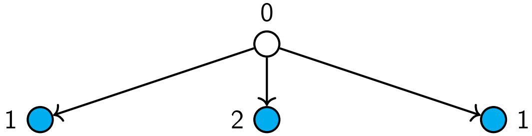
Greedy algorithm

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.



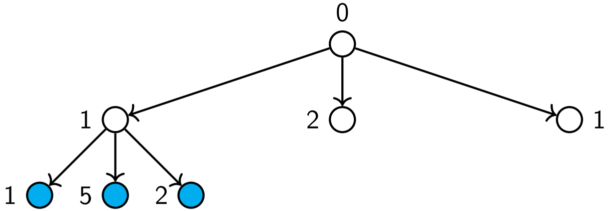
Greedy algorithm

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.



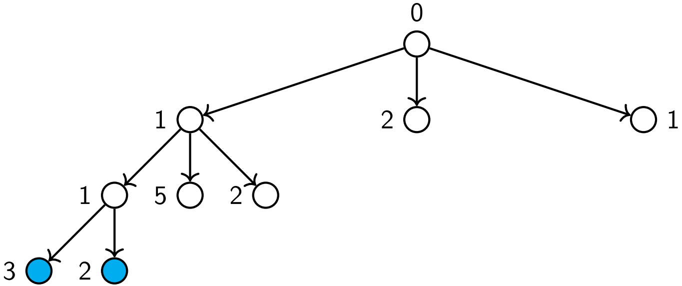
Greedy algorithm

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.



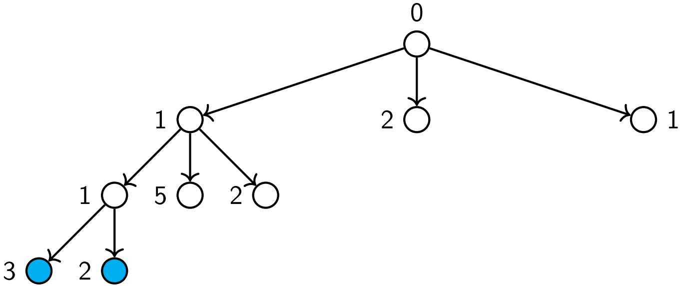
Greedy algorithm

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 $\leftarrow$ branching_scheme.root()
    while branching_scheme.children(node) is not empty **do**
        node $\leftarrow$ "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:

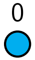
- Simple to understand and easy to implement
- Fast

Drawbacks:

- Low quality solutions

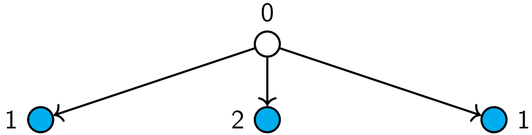
A / Best First Search algorithm

At each iteration, we expand the “best” node.



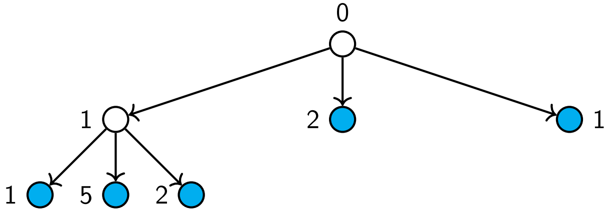
A / Best First Search algorithm

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



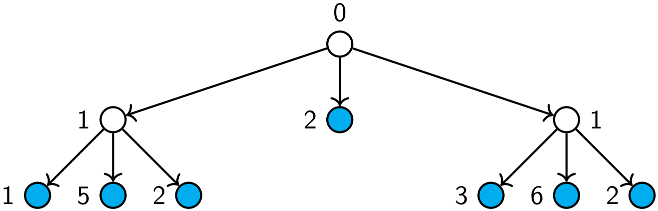
A / Best First Search algorithm

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



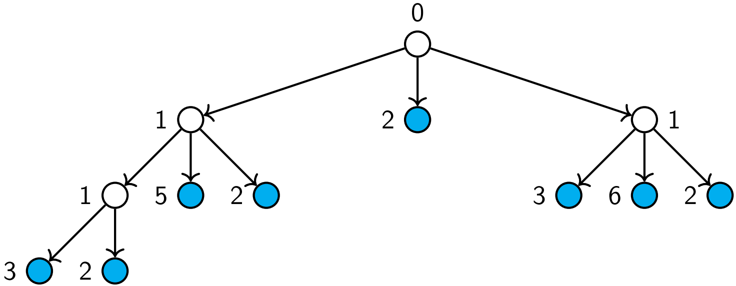
A / Best First Search algorithm

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



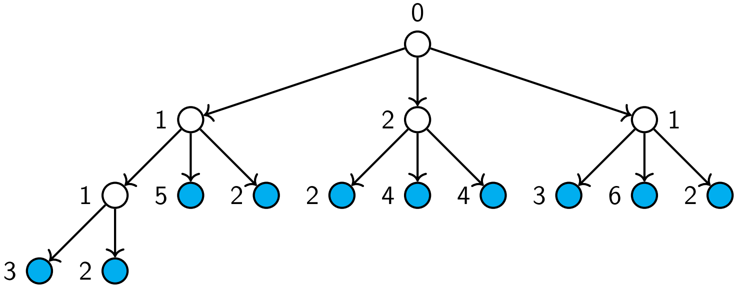
A / Best First Search algorithm

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



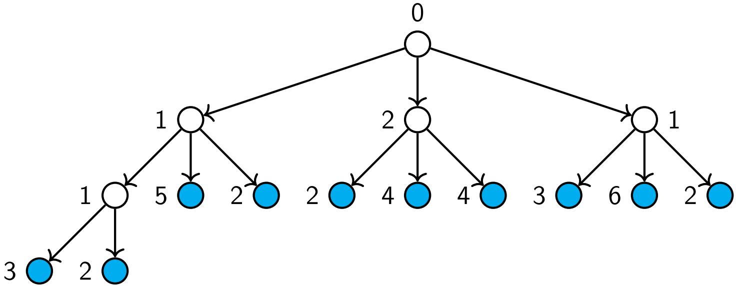
A / Best First Search algorithm

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 $\leftarrow$ {branching_scheme.root()}
    while queue is not empty **do**
        node $\leftarrow$ extract "best" node from queue
        queue $\leftarrow$ queue $\cup$ 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

Guides

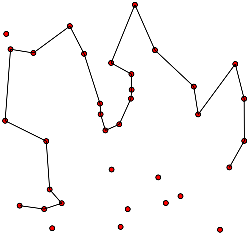
For both Greedy and A algorithms, a criteria is required to compare nodes:

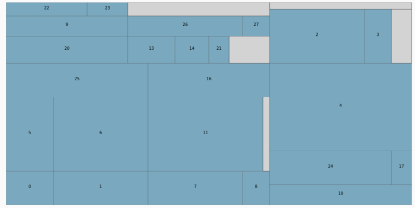
- Define and use the objective of the partial solutions:
  - Examples:
    - Travelling Salesman Problem: length of the partial tour
    - 2D Knapsack: total profit of the currently selected items
  - Advantage: simple, might be good as a first approach
  - Drawbacks: does not take into account the rest of the solution
    - Travelling Salesman Problem: a forgotten location near the first ones
    - 2D Knapsack: only big items remain

Guides

For both Greedy and A algorithms, a criteria is required to compare nodes:

- Define and use the objective of the partial solutions:
  - Examples:
    - Travelling Salesman Problem: length of the partial tour
    - 2D Knapsack: total profit of the currently selected items
  - 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
    - 2D Knapsack: only big items remain

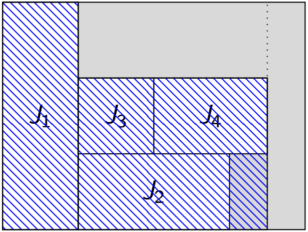




Guides

- Criteria which takes into account what is and is not in the partial solution

For the 2D guillotine Knapsack, first, instead of considering the profit $1/\mathrm{profit}(S)$ of the partial solution S, we consider the $\mathrm{area}(S)/\mathrm{profit}(S)$ with $\mathrm{area}(S)$ defined as illustrated below:

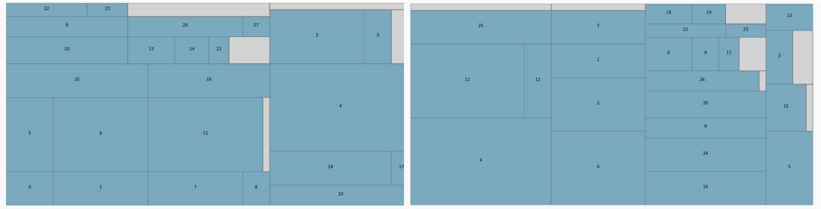


With this criteria, comparing nodes at different level of the tree makes more sense.

Guides

Then, to decrease the risk of packing all small items first, it is possible to introduce a bias in the guide:

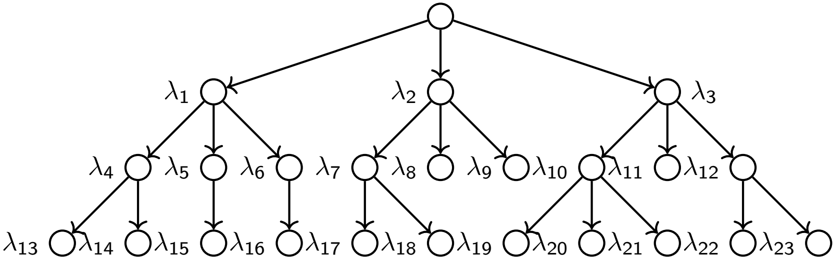
$$\frac{\mathrm{area}(S)}{\mathrm{profit(S)}} \qquad\qquad \frac{\mathrm{area}(S)}{\mathrm{profit(S)}}\frac{1}{\mathrm{mean\_item\_area}(S)}$$



Be careful about the computational complexity of the computation of the guide! In this example, it remains $O(1)$. More expensie guides may decrease the number of nodes explored.

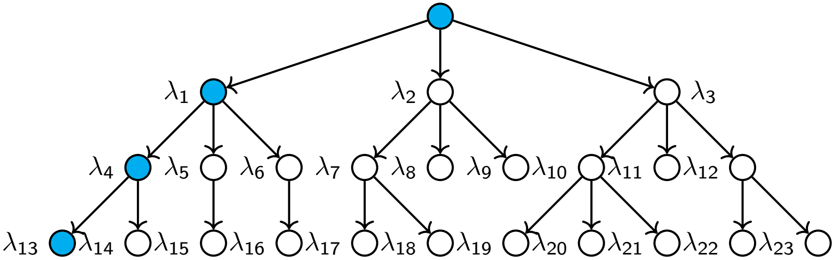
Limited Discrepancy Search

Nodes are explored by increasing value of their discrepancy.



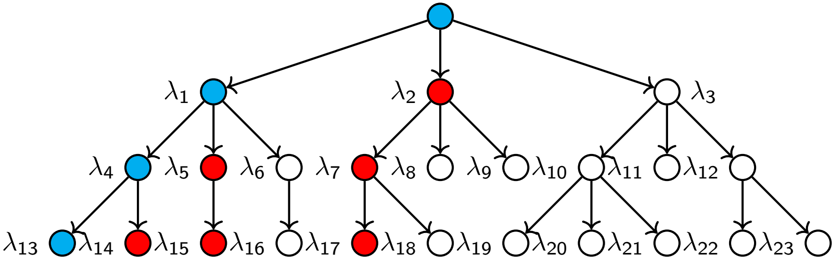
Limited Discrepancy Search

Nodes are explored by increasing value of their discrepancy.



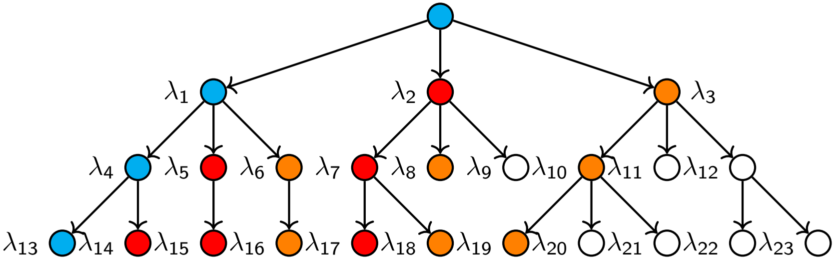
Limited Discrepancy Search

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:
  $\Longrightarrow$ 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:
  $\Longrightarrow$ 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:
  $\Longrightarrow$ very easy to design a guide

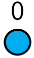
Drawbacks:

- Does not work as well with more balanced trees
  A node is only compared with its brothers:
  $\Longrightarrow$ 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.

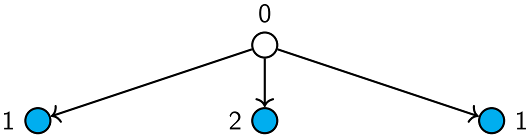
Example with a beam width of 5



Beam Search

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

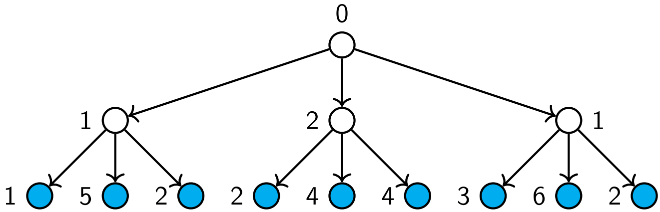
Example with a beam width of 5



Beam Search

Breadth First Search with a maximum width (called “beam width”).
At each stage, the “worst” nodes are discarded.

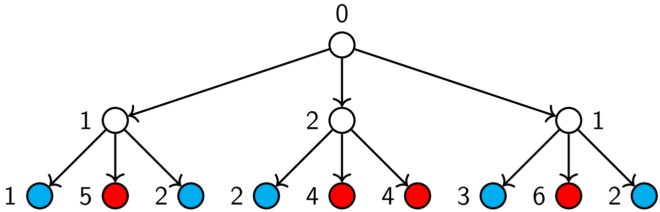
Example with a beam width of 5



Beam Search

Breadth First Search with a maximum width (called “beam width”).
At each stage, the “worst” nodes are discarded.

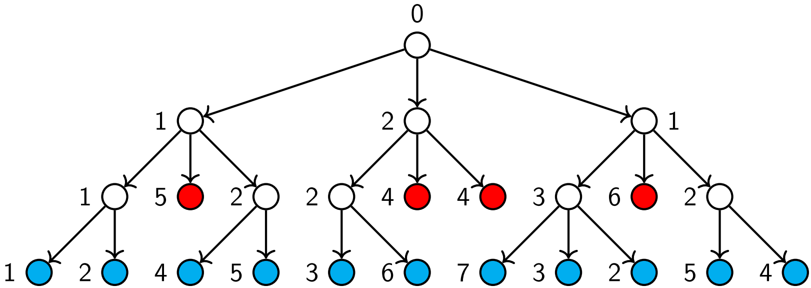
Example with a beam width of 5



Beam Search

Breadth First Search with a maximum width (called “beam width”).
At each stage, the “worst” nodes are discarded.

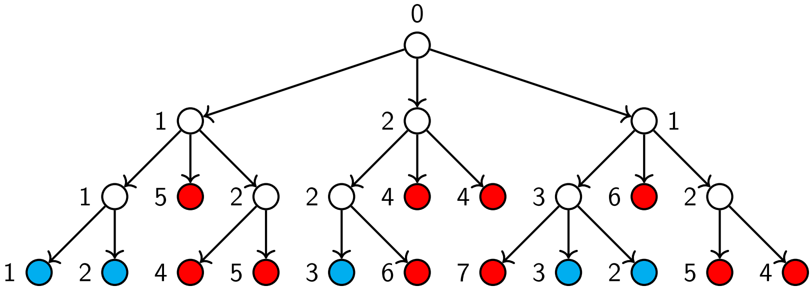
Example with a beam width of 5



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?

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

We can safely prune Node N_2.

More generally, let

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

Consider two nodes N_1 and N_2. If

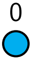
- $\mathrm{visited}(N_1) \supseteq \mathrm{visited}(N_2)$
- $\mathrm{last}(N_1) = \mathrm{last}(N_2)$
- $\mathrm{length}(N_1) \leq \mathrm{length}(N_2)$

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:

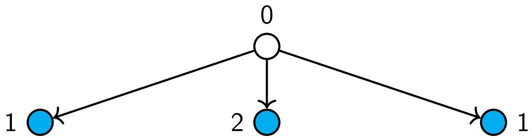
- if it is dominated by another node which is in the queue, it is not added
- the nodes from the queue that it dominates are removed from the queue



A / Best First Search + Dynamic Programming

Each time a node is added to the queue:

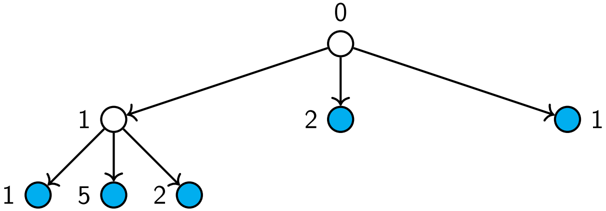
- if it is dominated by another node which is in the queue, it is not added
- the nodes from the queue that it dominates are removed from the queue



A / Best First Search + Dynamic Programming

Each time a node is added to the queue:

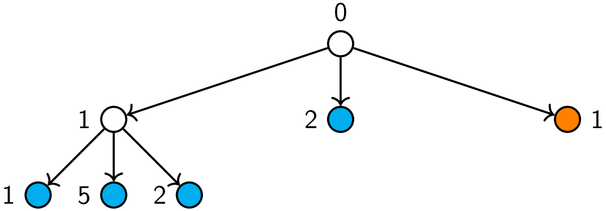
- if it is dominated by another node which is in the queue, it is not added
- the nodes from the queue that it dominates are removed from the queue



A / Best First Search + Dynamic Programming

Each time a node is added to the queue:

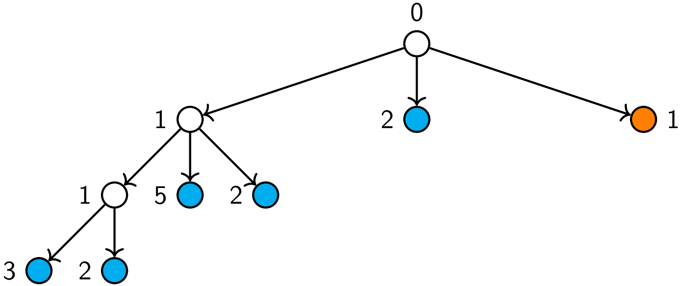
- if it is dominated by another node which is in the queue, it is not added
- the nodes from the queue that it dominates are removed from the queue



A / Best First Search + Dynamic Programming

Each time a node is added to the queue:

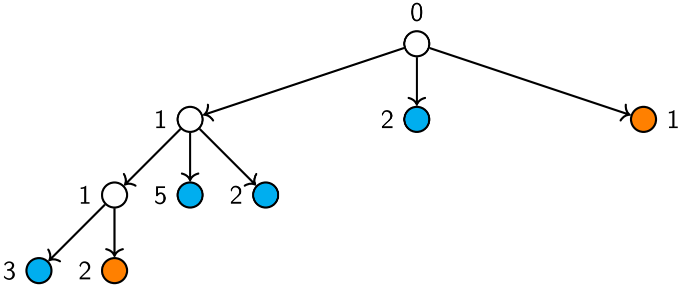
- if it is dominated by another node which is in the queue, it is not added
- the nodes from the queue that it dominates are removed from the queue



A / Best First Search + Dynamic Programming

Each time a node is added to the queue:

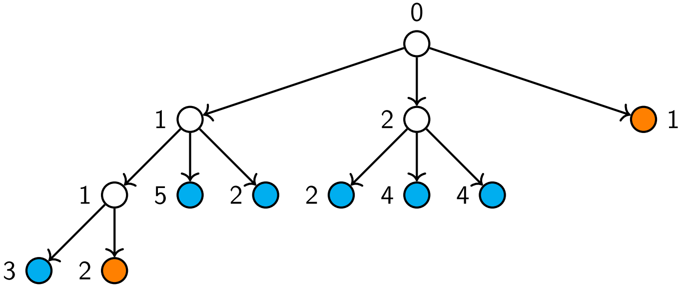
- if it is dominated by another node which is in the queue, it is not added
- the nodes from the queue that it dominates are removed from the queue



A / Best First Search + Dynamic Programming

Each time a node is added to the queue:

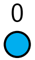
- if it is dominated by another node which is in the queue, it is not added
- the nodes from the queue that it dominates are removed from the queue



Beam Search + Dynamic Programming

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

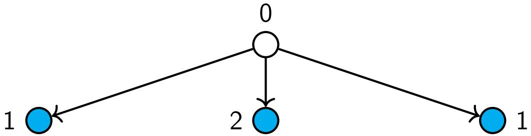
Beam width: 5



Beam Search + Dynamic Programming

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

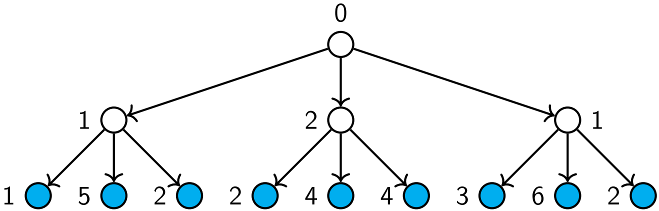
Beam width: 5



Beam Search + Dynamic Programming

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

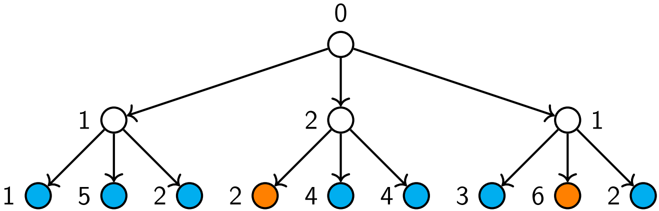
Beam width: 5



Beam Search + Dynamic Programming

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

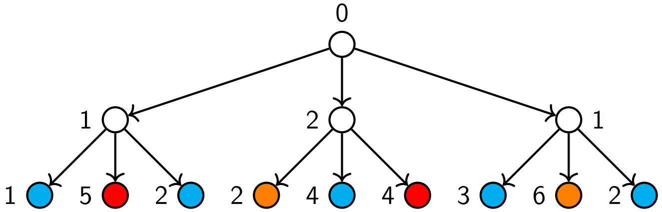
Beam width: 5



Beam Search + Dynamic Programming

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

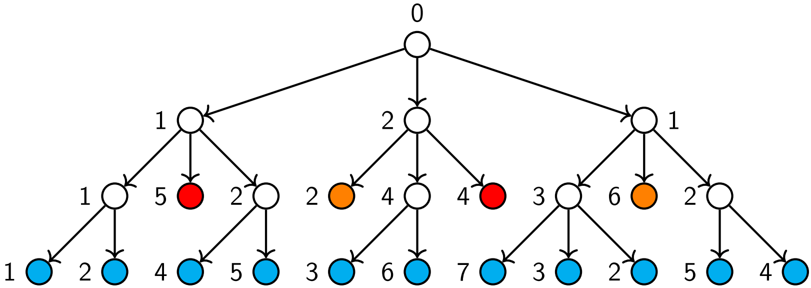
Beam width: 5



Beam Search + Dynamic Programming

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

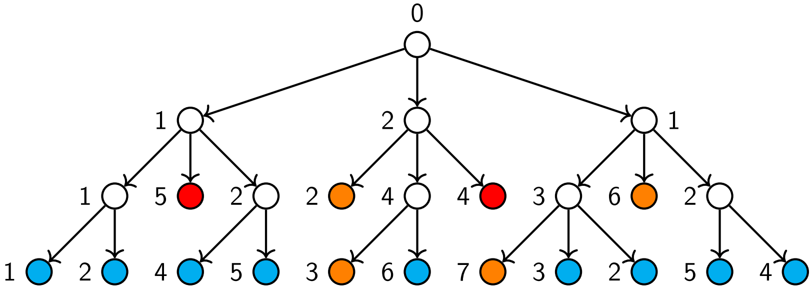
Beam width: 5



Beam Search + Dynamic Programming

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

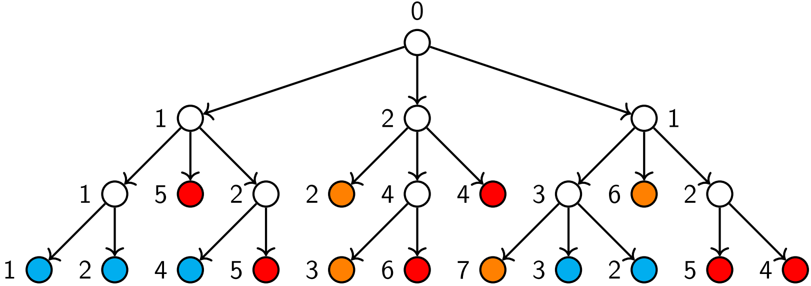
Beam width: 5



Beam Search + Dynamic Programming

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

Beam width: 5



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

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 ≤ 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 15, 2022