Constructive Search

1st ROAR-NET Training School

Marco Chiarandini marco@imada.sdu.dk

cost EUROPEAN COOPERATION IN SCIENCE & TECHNOLOGY

Funded by the European Union

Outline

Outline

Recap: Constrained Optimization Problems

Decision variables, $X = \{x_1, ..., x_k\}$, the unknowns that describe a solution to a problem

The **domain** of a variable x, denoted $D(x)$, is a finite set of elements that can be assigned to x.

A **constraint** C on X is a subset of the Cartesian product of the domains of the variables in X, i.e., $C \subseteq D(x_1) \times \cdots \times D(x_k)$. A tuple $(d_1, \ldots, d_k) \in C$ is called a **solution** to C.

Equivalently, we say that a **solution** $(d_1, ..., d_k) \in C$ is an assignment of the value d_i to the variable x_i for all $1 \leq i \leq k$, and that this assignment **satisfies** C.

Recap: Constrained Optimization Problems

Constraint Satisfaction Problem (CSP)

A CSP is a finite set of variables X with **domain extension** $\mathcal{D} = D(x_1) \times \cdots \times D(x_n)$, together with a finite set of constraints $\mathcal{C}$, each on a subset of X.
A **solution** to a CSP is an assignment of a value $d \in D(x)$ to each $x \in X$, such that all constraints are satisfied simultaneously.

Constraint Optimization Problem (COP)

A COP is a CSP $\mathcal{P}$ defined on the variables $x_1, \ldots, x_n$, together with an **objective function** $f : D(x_1) \times \cdots \times D(x_n) \to Q$ that assigns a value to each assignment of values to the variables.
An **optimal solution** to a minimization COP is a solution d to $\mathcal{P}$ that minimizes the value of $f(d)$.

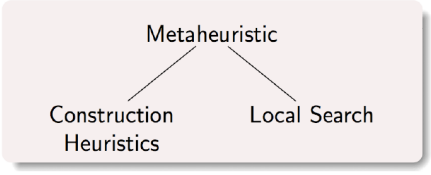
Partial *vs* **Complete** assignments

Recap: Randomized Optimization Algorithms (ROAs)

Two heuristic search paradigms:

- Constructive search (works on partial solutions)
- Local search (works on complete solutions)

plus high level guiding strategies (ie, metaheuristics), eg, evolutionary algorithms.



Recap: Constraint Handling

Constraints in heuristic methods are handled:

- as **one-way** constraints between variables
- **implicitly** in the definition of the **combinatorial structures** that constitute the search states:
  assignments, permutations, (sub)sets, partitions, (sub)graphs, sequences, set of sequences, ...
- as **soft constraints**
  ie, relaxed in the evaluation function as penalty components with large weights or as lexicographically more important componenents

Examples

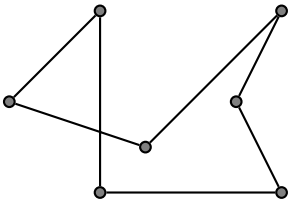
We will use the following examples to illustrate the concepts of solution representation and constructive search:

- Traveling Salesman Problem
- Single Machine Total Weighted Tardiness
- Knapsack Problem
- Graph Vertex-Coloring

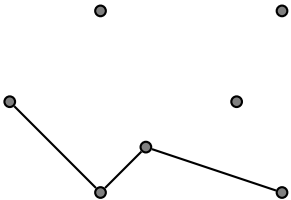
Traveling Salesman Problem

Given: A graph $G = (V, E)$ and a weight function $\omega : E \to \mathbb{R}$.

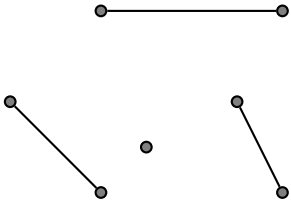
Task: Find the shortest Hamiltonian tour.



sol. representation:
permutation of vertices or
assignment of successors



sol. representation:
set of adjacent edges



sol. representation:
set of edges or
incidence binary vector

Single Machine Total Weighted Tardiness

Given: a set of n jobs $\{J_1, \ldots, J_n\}$ to be processed on a single machine and for each job J_i a processing time p_i, a weight w_i and a due date d_i.

Task: Find a schedule that minimizes the total weighted tardiness $\sum_{i=1}^{n} w_i \cdot T_i$, where $T_i = \max\{C_i - d_i, 0\}$ (C_i completion time of job J_i)

Example (Instance)

Job	J_1	J_2	J_3	J_4	J_5	J_6
Processing Time	3	2	2	3	4	3
Due date	6	13	4	9	7	17
Weight	2	3	1	5	1	2

Sol. representation:
Linear permutation

$\phi = [J_3, J_1, J_5, J_4, J_1, J_6]$

Job	J_3	J_1	J_5	J_4	J_2	J_6
C_i	2	5	9	12	14	17
T_i	0	0	2	3	1	0
$w_i \cdot T_i$	0	0	2	15	3	0

Knapsack Problem

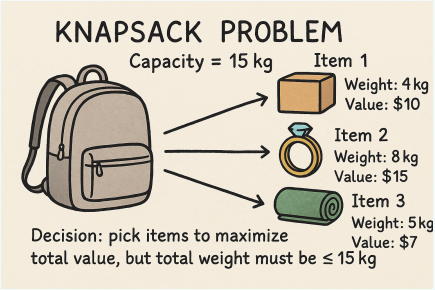
Given: A ground set of items with weights and values. A knapsack with a weight capacity.

Task: Find the subset that maximizes the total value

Example (Instance)

Capacity = 15

Item	Weight	Value
1	4	10
2	8	15
3	5	7
4	5	8
5	9	10
6	7	7
7	3	4
8	1	3

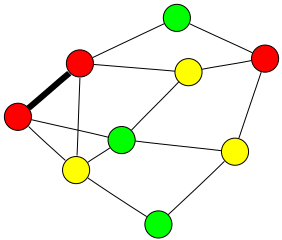


Sol. representation: Set: $\{1, 2, 7\}$ or Incidence vector: $[1, 1, 0, 0, 0, 0, 1, 0]$
Total weight: 15; Total value: 26

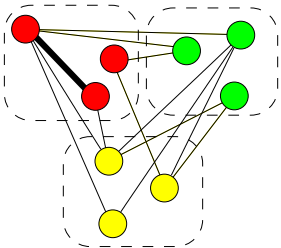
Graph Vertex-Coloring

Given: A graph G and a set of colors Γ.

A **proper coloring**: each vertex receives a color and no two adjacent vertices receive the same color.



Assignment representation



Partitioning representation

Task:

Find a proper coloring of G that uses the minimal number of colors (chromatic number).

Outline

A Search Problem

ROAR NET

Problem statement

Constrained Optimization Problem: $\min\{f(s) \mid s \in \mathcal{F}\}$

- $\mathcal{F} \subseteq \mathcal{S}$ set of **feasible solutions**
- $\mathcal{S}$ set of **candidate solutions** (**combinatorial structures**)

The concept of **feasibility** is flexible and a design choice.
Most typically, it implies satisfying the constraints of the problem.

Guiding rule: if it has an **objective function value**, then it is a **feasible solution**

Guiding rule: **constructive search algorithms** work with **partial, infeasible solutions**

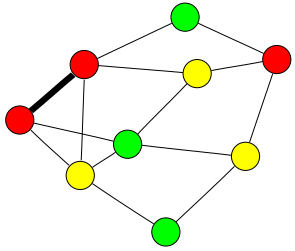
Example: a partial tour does not have an objective function value (ie, we do not have an Hamiltonian cycle yet, hence its length is not a valid objective function value).

Partial vs Complete and Infeasible vs Feasible

The previous guiding rules are often broken:

- Constructive search on partial sols and local search on complete sols not always true
- Partial *vs* Complete ≠ Infeasible *vs* Feasible

For example in the Graph Vertex-Coloring different choices for **candidate solutions** are used in local search algorithms (last column indicates the frequency in the literature):



k-fixed	complete	proper	+
k-fixed	partial	proper	+
k-fixed	complete	unproper	+ + +
k-fixed	partial	unproper	
k-variable	complete	proper	++
k-variable	partial	proper	+
k-variable	complete	unproper	++
k-variable	partial	unproper	

So don't take me too rigidly in this organization! :)

A Search Problem

Definition (Search or Optimization Algorithm)

Goal formulation: we want to find the minimum with respect to some criterion from a set of candidate elements.

Problem formulation: Given a description of the states, an initial state and actions necessary to reach the goal, find a sequence of actions to reach the goal.

Search: the algorithm simulates sequences of actions in the model of the goal, searching until it finds a sequence of actions that reaches the goal.
The algorithm might have to simulate multiple tentative answers that do not meet the goal, but eventually it reaches a solution, or it will find that no solution is possible.

Components of a Search Algorithm (1)

(valid for complete or heuristic and constructive or perturbative algorithms)

- **State or (Candidate) solution**: a definition of the states of the search
- **Search Space**: A set of possible **states** that the search can be in.
- **Initial State**: State the search starts in.
- **Goal**: A set of one or more goal states. Sometimes there is one goal state sometimes there is a small set of alternative goal states
- **Evaluation function** $f(s)$: assesses the distance from a potential goal. (note: different from ”objective”, it can also include penalties due to constraint violations).

Components of a Search Algorithm (2)

- **Action Type** t: available to the algorithm. Neighborhood
- A finite set of **actions** of type t that can be executed in s, $\text{Actions}(t, s)$. Move
- A **transition model** that describes what each action does. $\text{Result}(s, a)$ returns the state that results from doing action $a \in \text{Actions}(t, s)$ in state s. Apply Move
- An **action-cost function**, $\text{Action-Cost}(s, a, s')$, that gives the numeric cost of applying action a in state s to reach state s'. Increment

ROAR-NET Application Programming Interface (API)

(Note: Here not meant as Web API, network-based API, or REST API.)

The ROAR-NET API Specification is the definition of an interface or protocol between optimization problems seen as black box and their solvers in order to facilitate understanding, reusing and scaling of solution approaches.

We look for a model that

- … allows one to use off the shelf components to solve it.
- … assumes a separation between **problem specifics** and **solver**.
- .. is designed as a **software interface** offering a service to other pieces of software and is implemented by the user.
- … promotes **reusability** of software components and minimizes the user's effort to deploy a solution for the specific optimization problem at hand.
- … maximizes code extensability, reusability, and simplicity.

The Full API

```
# Types
Problem
Solution
Value
Increment
PreferenceModel
Neighborhood
Move
```

```
# Operations on Problem
empty_solution
random_solution
heuristic_solution

# Operations on Solution
objective_value
lower_bound
copy_solution

# Operations on Neighborhood
local_neighbourhood
construction_neighbourhood
destruction_neighbourhood
sub_neighbourhoods
```

```
# Operations on Move
moves
random_move
random_moves_without_replacement
lower_bound_increment
objective_value_increment
apply_move
invert_move

# Operations on PreferenceModel
scalarisation
better_or_indifferent  # preorder
better                 # strict preorder
indifferent            # equivalence
incomparable

compare_total_order
compare_partial_order

selection    # Indices
ranking      # Rank/class values
```

https://github.com/roar-net/roar-net-api-spec

Outline

Outline

Dynamic Programming (DP)

- Applied to problems with **optimal substructure** and **overlapping subproblems**
- **Optimal substructure** means an optimal solution can be constructed from optimal solutions to its subproblems
- **Overlapping subproblems** means solving each subproblem separately requires repeating certain operations
- **DP** begins with desired problem and recurses down to smaller and smaller subproblems, retrieving the value of previously solved problems as necessary
- Principle of Optimality (known as Bellman Optimality Conditions): Suppose that the solution of a problem is the result of a sequence of n decisions $D_1, D_2, \ldots, D_n$; if a given sequence is optimal, then the first k decisions must be optimal, but also the last $n - k$ decisions must be optimal

Knapsack Problem

- Let the $V(i, w)$ (the so-called **value function**) be the maximum value achievable using the first i items and a knapsack capacity w.

- Consider the ith item. You can either use it or not.
  - If you don't use it, then the value of your knapsack will be

    $$V(i-1, w)$$

  - If you use it, then the value of your knapsack will be

    $$V(i-1, w-w_i) + v_i$$

The Recursion

- For each item i and weight w:

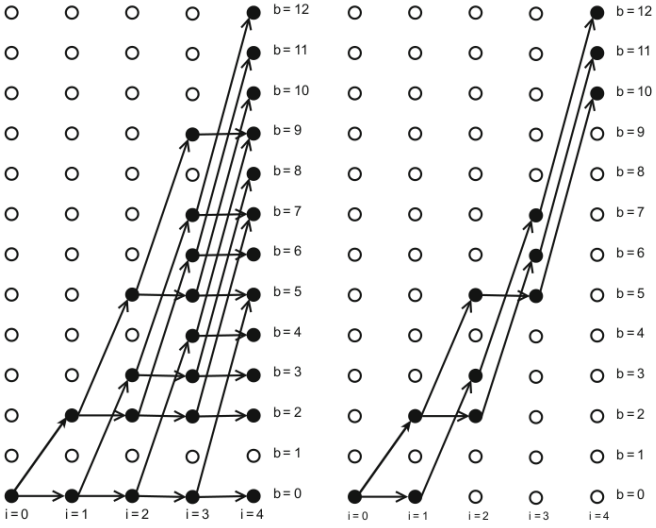
$$V(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ \max \begin{cases} V(i-1,w) & \text{(discard new item)} \\ V(i-1,w-w_i)+v_i & \text{(include new item)} \end{cases} & \text{if } w_i \le w \\ V(i-1,w) & \text{if } w_i > w \end{cases}$$

- Optimal solution:

$$z^* = V(n,W)$$

and trace back to find the items collected.

Visual Representation



SMTWT Problem

- $1 \mid\mid \sum h_j(C_j)$ generalization of $\sum w_j T_j$ (strongly NP-hard)
- **(Forward) dynamic programming** algorithm that runs in $O(2^n)$

J set of jobs already scheduled;

$V(J) = \sum_{j \in J} h_j(C_j)$ the value of a sequence of the jobs in J;

Step 1: Set $J = \emptyset$, $V^*(\{j\}) = V(\{j\}) = h_j(p_j), \quad j = 1, \ldots, n$

Step 2: $V^*(J) = \min_{j \in J} \left(V^*(J - \{j\}) + h_j \left(\sum_{k \in J} p_k \right) \right)$ best value of the set of jobs in J;

Step 3: If $J = \{1, 2, \ldots, n\}$ then $V^*(\{1, 2, \ldots, n\})$ is optimum,
otherwise go to Step 2.

Traveling Salesman Problem

The TSP asks for the shortest tour that starts from 0, visits all cities of the set $C = \{1, 2, ..., n\}$ exactly once, and returns to 0, where the cost to travel from i to j is c_{ij} (with $(i, j) \in A$)

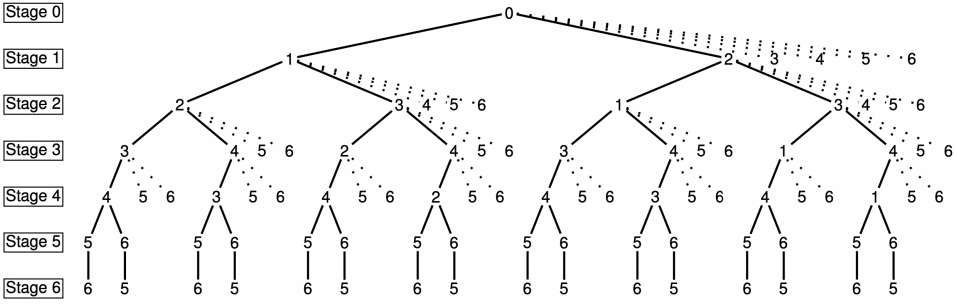
If the optimal solution of a TSP with six cities is $(0, 1, 3, 2, 4, 6, 5, 0)$, then...

- the optimal solution to visit $\{1, 2, 3, 4, 5, 6\}$ starting from 0 and ending at 5 is $(0, 1, 3, 2, 4, 6, 5)$
- the optimal solution to visit $\{1, 2, 3, 4, 6\}$ starting from 0 and ending at 6 is $(0, 1, 3, 2, 4, 6)$
- the optimal solution to visit $\{1, 2, 3, 4\}$ starting from 0 and ending at 4 is $(0, 1, 3, 2, 4)$
- the optimal solution to visit $\{1, 2, 3\}$ starting from 0 and ending at 2 is $(0, 1, 3, 2)$
- the optimal solution to visit $\{1, 3\}$ starting from 0 and ending at 3 is $(0, 1, 3)$
- the optimal solution to visit 1 starting from 0 is $(0, 1)$

The optimal solution is made up of a number of optimal solutions of smaller subproblems

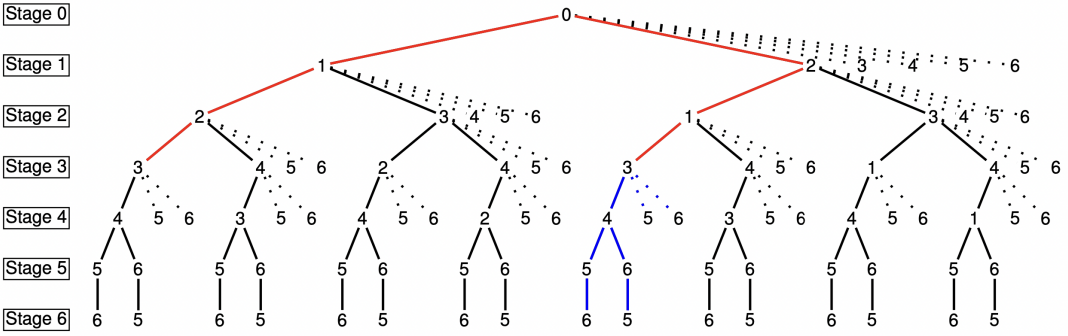
Enumerate All Solutions of the TSP

- A solution of a TSP with n cities derives from a sequence of n decisions, where the kth decision consists of choosing the kth city to visit in the tour



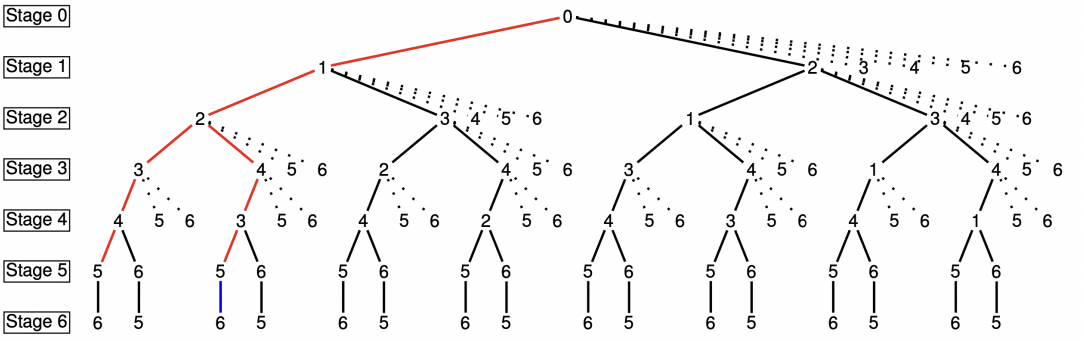
- The number of nodes (or states) grows exponentially with n
- At stage k, the number of states is $\binom{n}{k}k!$
- With $n = 6$, at stage $k = 6$, this yeilds 720 states

Are All States Necessary?



If path $(0, 1, 2, 3)$ costs less than $(0, 2, 1, 3)$, the optimal solution cannot be found in the blue part of the tree

Are All States Necessary?



If path $(0, 1, 2, 3, 4, 5)$ costs less than $(0, 1, 2, 4, 3, 5)$, the optimal solution cannot be found in the blue part of the tree

Are All States Necessary?

- At stage k $(1 \leq k \leq n)$, for each subset of cities $S \subseteq C$ of cardinality k, it is necessary to have only k states (one for each of the cities of the set S)

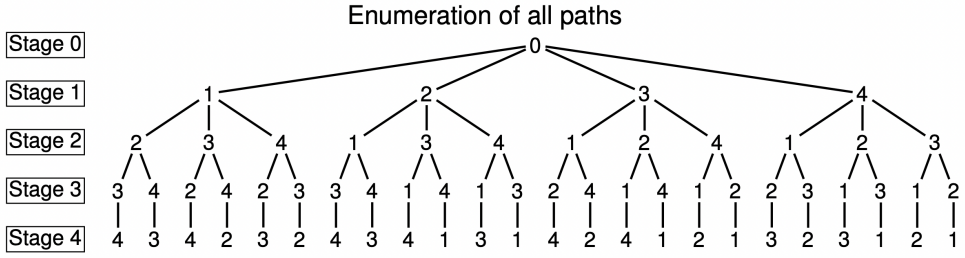
- At state $k = 3$, given the subset of cities $S = \{1, 2, 3\}$, three states are needed:
  - the shortest-path to visit S by starting from 0 and ending at 1
  - the shortest-path to visit S by starting from 0 and ending at 2
  - the shortest-path to visit S by starting from 0 and ending at 3

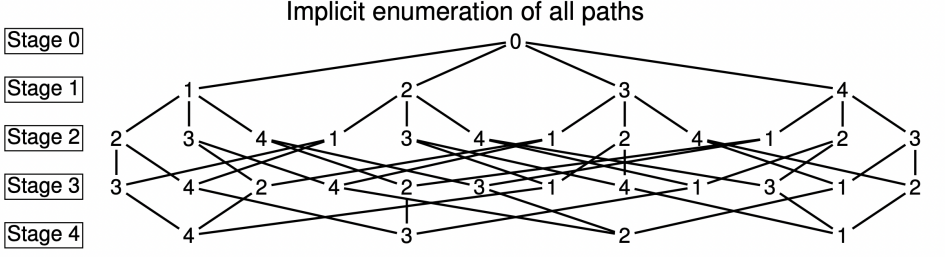
- At stage k, $\binom{n}{k}k$ states are required to compute the optimal solution (not $\binom{n}{k}k!$)

#States n = 6

Stage	$\binom{n}{k}k!$	$\binom{n}{k}k$
1	6	6
2	30	30
3	120	60
4	360	60
5	720	30
6	720	6

Complete Trees with n=4

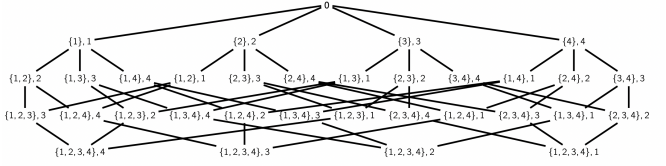




Dynamic Programming Recursion for the TSP (1/2)

- Given a subset $S \subseteq C$ of cities and $k \in S$, let $V(S,k)$ be the optimal cost of starting from 0, visiting all cities in S, and ending at k
- Begin by finding $V(S,k)$ for $|S| = 1$, which is $V(\{k\},k) = c_{0k}, \forall k \in C$
- To compute $V(S,k)$ for $|S| > 1$, the best way to visit all cities of S by starting from 0 and ending at k is to consider all $j \in S \setminus \{k\}$ immediately before k, and look up $V(S \setminus \{k\},j)$, namely

$$V(S,k) = \min_{j \in S \setminus \{k\}} \{V(S \setminus \{k\},j) + c_{jk}\}$$



- The optimal solution cost z^* of the TSP is $z^* = \min_{k \in C}\{V(C,k) + c_{k0}\}$

Dynamic Programming Recursion for the TSP (2/2)

More schematically, DP for TSP [Held and Karp (1962)]:

1. **Initialization**. Set $V(\{k\}, k) = c_{0k}$ for each $k \in C$
2. **Recursive Step**. For each stage $r = 2, 3, ..., n$, compute

$$V(S,k) = \min_{j \in S \setminus \{k\}} \{V(S \setminus \{k\}, j) + c_{jk}\} \quad \text{for all } S \subseteq C : |S| = r \text{ and for all } k \in S$$

3. **Optimal Solution**. Find the optimal solution cost z^* as

$$z^* = \min_{k \in C} \{V(C,k) + c_{k0}\}$$

- With DP, TSP instances with up to 25-30 nodes can be solved to optimality; other solution techniques (i.e., branch-and-cut) are able to solve TSP instances with up to 85,900 customers
- Nonetheless, DP represents the state-of-the-art techniques to solve a wide variety of search problems

Outline

Complete Graph Search Methods

Tree (or graph) search in

Uninformed settings

- Breadth-first search
- Uniform-cost search
- Depth-first search

Informed settings

- Greedy best-first search
- A* search

Complete Tree Search

For CSPs with n variables and d values, we can use a complete tree search with

Search Space

Tree with branching factor at the top level nd
at the next level the branching factor is $(n-1)d$.
The tree has $n! \cdot d^n$ leaves even if only d^n possible complete assignments.

- CSP is **commutative** in the order of application of any given set of action (i.e., we reach same partial solution regardless of the order)
- Hence, generate successors by considering possible assignments for only a single variable at each node in the search tree. The tree has now d^n leaves
- use information: look-ahead, best first, etc.

Outline

Exploiting Information

Assessment of partial, infeasible solutions

The priority assigned to a node x is determined by the function

$$f(x) = g(x) + h(x)$$

$g(x)$: cost of the path so far
$h(x)$: heuristic estimate of the minimal cost to reach the goal from x.

- **greedy best-first search** uses h to decide
- **A* best-first search** uses f and is cost-optimal when reaches the goal, if $h(x)$ is an
  - admissible heuristic: never overestimates the cost to reach the goal
  - consistent: $h(n) \leq c(n, a, n') + h(n')$
    (consistent $\Rightarrow$ admissible, only necessary in graph search)

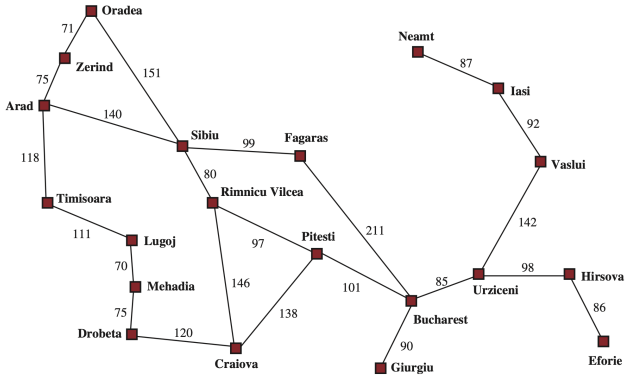
Best-First Search

```
Procedure TreeSearch(start, target)
add start to visited
add start to queue
while queue is not empty do
    current_node ← vertex of queue with min distance to target[a]
    remove current_node from queue
    for each neighbor n of current_node do
        if n not in visited then
            if n is target then
                return n
            else
                mark n as visited
                add n to queue
return failure
```

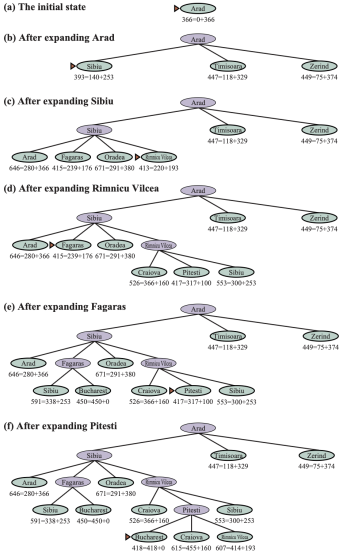
[a] Greedy best-first: min distance to target $= h(n)$;
A* best-first: min sum of distance so far and distance to target $= f(n)$

Example

Find shortest path from Arad to Bucharest:



Right: The tree explored by A* search



[P. Norvig and N. Russel, AI: A Modern Approach, 4th ed., 2020]

Heuristic Functions

Possible choices for admissible heuristic functions in A* search:

- optimal solution to an easily solvable **relaxed problem**
- optimal solution to an easily solvable **subproblem**
- learning from experience by gathering statistics on state features
- preferred heuristic functions with higher values (provided they do not overestimate)
- if several heuristics available $h_1, h_2, \ldots, h_m$ and not clear which is the best then:

$$h(x) = \max\{h_1(x), \ldots, h_m(x)\}$$

A Star Search

Drawbacks:

- Time complexity: In the worst case, the number of nodes expanded is exponential, (but it is polynomial when the heuristic function h meets the following condition:

$$|h(x) - h^*(x)| \leq O(\log h^*(x))$$

h^* is the optimal heuristic, the exact cost of getting from x to the goal.)

- Memory usage: In the worst case, it must remember an exponential number of nodes.
  Several variants: including iterative deepening A* (IDA*), memory-bounded A* (MA*) and simplified memory bounded A* (SMA*) and recursive best-first search (RBFS)

Outline

Dealing with Constraints

Backtracking search

Depth-first search that chooses one variable at a time and backtracks when a variable has no *feasible* values left to assign.

- No need to copy solutions all the times but rather extensions and undo extensions
- Since CSP is standard then the algorithm (initial state + successor function + goal test) is also standard ⇒ general purpose solvers
- Backtracking is **uninformed and complete**. Other search algorithms may use **information** in form of heuristics.

Informed Search with Constraints

Which **variable** should we assign next, and in which order should its **values** be tried?

- SELECT-INITIAL-UNASSIGNED-VARIABLE
- SELECT-UNASSIGNED-VARIABLE
  - most constrained first = fail-first heuristic
    = Minimum remaining values (MRV) heuristic
    (tend to reduce the branching factor and to speed up pruning)
  - least constrained last

  Eg.: max degree, farthest, earliest due date, etc.
- ORDER-DOMAIN-VALUES
  - greedy
  - least constraining value heuristic
    (leaves maximum flexibility for subsequent variable assignments)
  - maximal regret
    implements a kind of look ahead

Outline

Branch and Bound

```
LIST ← {S} ;                                          # S root of the branching tree
ub ← value of some heuristic solution;
current_best ← heuristic solution;
while LIST ≠ ∅ do
    Choose a branching node k from LIST;
    Remove k from LIST;
    Generate child_i of k, for i = 1, ..., n_k, and calculate corresponding lower bounds lb_i;
    for i = 1 to n_k do
        if lb_i < ub then
            if child_i consists of a feasible solution then
                ub ← lb_i;
                current_best ← solution corresponding to child_i;
            else
                add child_i to LIST;
        else
            prune;
```

Pruning happens: (i) by bound (ii) by optimality (iii) by infeasibility

Branch and Bound

Branch and bound *vs* Backtracking

- $=$ a state space tree is used to solve a problem.
- $\neq$ branch and bound does not limit us to any particular way of traversing the tree (backtracking is depth-first)
- $\neq$ branch and bound is used only for optimization problems.

Branch and bound *vs* A*

- $=$ In A* the admissible heuristic mimics bounding
- $\neq$ In A* there is no branching. It is a search algorithm.
- $\neq$ A* is best first

Branch and Bound: Strategies

[Jens Clausen (1999). Branch and Bound Algorithms - Principles and Examples.]

- **Eager Strategy**:
  1. select a node
  2. branch
  3. for each subproblem compute bounds and compare with incumbent solution
  4. discard or store nodes together with their bounds

  (Bounds are calculated as soon as nodes are available)

- **Lazy Strategy**:
  1. select a node
  2. compute bound
  3. branch
  4. store the new nodes together with the bound of the father node

  (often used when selection criterion for next node is max depth)

Branch and Bound: Components

1. Initial feasible solution (heuristic) – might be crucial!
2. Bounding function
3. Strategy for selecting
4. Branching
5. Fathoming (dominance test)

Branch and Bound: Bounding Techniques

$$\min_{s \in \mathcal{P}} g(s) \leq \left\{ \begin{matrix} \min_{s \in \mathcal{P}} f(s) \\ \min_{s \in \mathcal{F}} g(s) \end{matrix} \right\} \leq \min_{s \in \mathcal{F}} f(s)$$

P: candidate solutions; $\mathcal{S} \subseteq \mathcal{P}$ feasible solutions

Techniques:

- Linear programming
- Combinatorial relaxation
- Lagrangian relaxation
- Duality

should be polytime and tight (trade off)

Branch and Bound: Search Strategy

Strategies for selecting the next subproblem:

- best first
  (combined with eager strategy but also with lazy)

- breadth first
  (memory problems)

- depth first
  works on recursive updates (hence good for memory)
  but might compute a large part of the tree which is far from optimal
  (enhanced by alternating search in lowest and largest bounds combined with branching on the node with the largest difference in bound between the children)
  (it seems to perform best)

Branch and Bound: Branching

- dichotomic
- polytomic

Outline

Outline

Greedy Algorithms

- Algorithms based on incomplete versions of greedy best-first search perform a single descent in the search tree.
- They always expand the current state and does not maintain an open set (frontier)
- Strategy: always make the choice that is best at the moment
- Sometimes, they guarantee the optimal solution: Minimum Spanning Tree, Single Source Shortest Path, etc.)

Greedy Algorithms in the ROAR-NET API

```
def greedy_construction(problem, solution):
    neigh = problem.construction_neighbourhood()
    if solution is None:
        solution = problem.empty_solution()
    move_iter = iter(_valid_moves_and_increments(neigh, solution))
    move_and_incr = next(move_iter, None)
    while move_and_incr is not None:
        best_move, best_incr = move_and_incr
        for move, incr in move_iter:
            if incr < best_incr:
                best_move, best_incr = move, incr
                if incr == 0:
                    break
        solution = best_move.apply_move(solution)
        move_iter = iter(_valid_moves_and_increments(neigh, solution))
        move_and_incr = next(move_iter, None)
    return solution

def _valid_moves_and_increments(neigh, solution):
    for move in neigh.moves(solution):
        incr = move.objective_value_increment(solution)
        yield (move, incr) if incr < 0 else continue
```

Incomplete Search

On the backtracking framework
(beyond depth-first search)

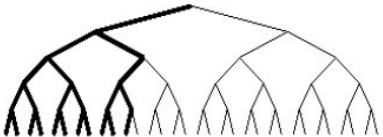
- Bounded backtrack
- Credit-based search
- Limited Discrepancy Search
- Barrier Search
- Randomization in Tree Search
- Random Restart

Outside the exact framework
(beyond greedy search)

- Random Restart
- Rollout/Pilot Method
- Beam Search
- Iterated Greedy
- GRASP
- (Adaptive Iterated Construction Search)
- (Multilevel Refinement)

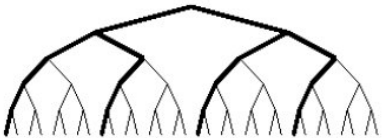
Bounded Backtrack

Bounded-backtrack search:



bbs(10)

Depth-bounded, then bounded-backtrack search:

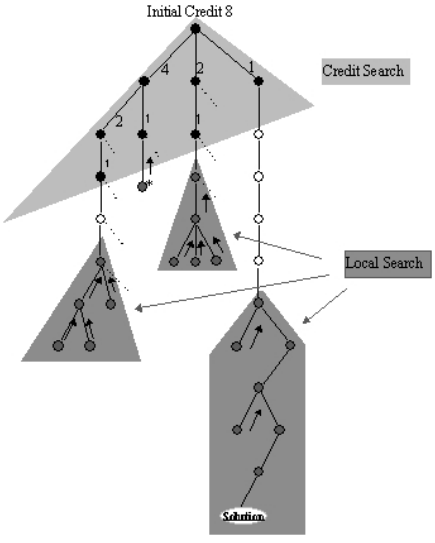


dbs(2, bbs(0))

Visualizations by Helmut Simonis

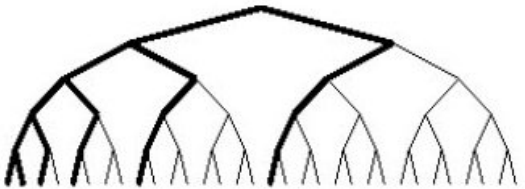
Credit-Based Search

- Key idea: important decisions are at the top of the tree
- **Credit** = backtracking steps
- Credit distribution: one half at the best child the other divided among the other children.
- When credits run out follow deterministic best-search
- In addition: allow limited backtracking steps (eg, 5) at the bottom
- **Control parameters**: initial credit, distribution of credit among the children, amount of local backtracking at bottom.



Limited Discrepancy Search

- Key observation that often the heuristic used in the search is nearly always correct with just a few exceptions.
- Explore the tree in increasing number of **discrepancies**, modifications from the heuristic choice.
- Eg: count one discrepancy if second best is chosen
  count two discrepancies either if third best is chosen or twice the second best is chosen
- **Control parameter**: the number of discrepancies



Randomization in Tree Search

Idea: important decisions are made up in the search tree (backdoors - set of variables such that once they are instantiated the remaining problem simplifies to a tractable form)
$\rightsquigarrow$ random selections + restart strategy

Random selections

- randomization in variable ordering:
  - breaking ties at random
  - use heuristic to rank and randomly pick from small factor from the best
  - random pick among heuristics
  - random pick variable with probability depending on heuristic value
- randomization in value ordering:
  - just select random from the domain

Restart strategy with different budget in backtracking (if kept running it becomes complete)

- Example: $B = (1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 4, 8, 1, \ldots)$

Outline

Rollout/Pilot Method

Derived from A*

- Master process adds components sequentially to a partial solution $S_k = (s_1, s_2, \ldots s_k)$
- At the kth iteration the master process evaluates feasible components to add based on a **heuristic look-ahead strategy**.
- The evaluation function $H(S_{k+1})$ is determined by sub-heuristics that complete the solution starting from S_k
- Sub-heuristics are combined in $H(S_{k+1})$ by
  - weighted sum
  - minimal value

Note: this evaluation does not need to be a lower bound!

Rollout/Pilot Method

Speed-ups:

- halt whenever cost of current partial solution exceeds current upper bound
- evaluate only a fraction of possible components

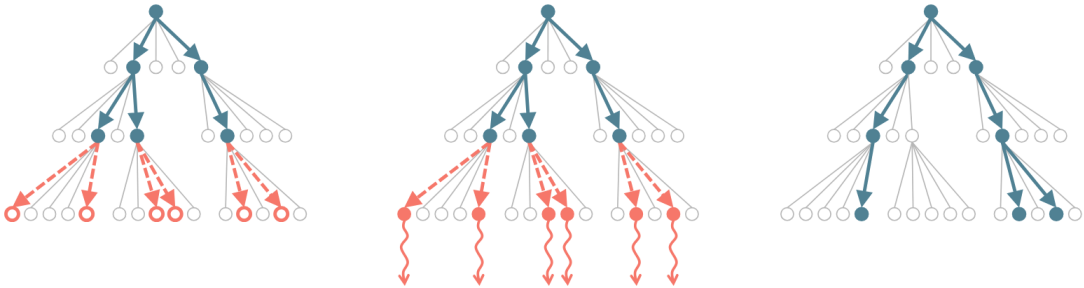
Beam Search

Based on the tree search framework:

- maintain a set B of bw (beam width) partial candidate solutions
- at each iteration extend each solution from B in fw (filter width) possible ways
- rank each $bw \times fw$ candidate solutions and take the best bw partial solutions
- complete candidate solutions obtained by B are maintained in B_f
- stop when no partial solution in B is to be extended

Visualization of Beam Search

Three phases of Beam Search illustrated with beam width $bw = 3$ and expansion factor $bf = 2$. [Choo, Jinho et al. 2022]



Greedy Randomized Adaptive Search Procedure

Greedy Randomized "Adaptive" Search Procedure, aka, GRASP

Key Idea: Combine randomized constructive search with subsequent local search.

Motivation:

- Candidate solutions obtained from construction heuristics can often be substantially improved by local search.
- Local search methods often require substantially fewer steps to reach high-quality solutions when initialized using greedy constructive search rather than random picking.
- By iterating cycles of constructive + local search, further performance improvements can be achieved.

GRASP

Procedure GRASP();
while termination criterion is not satisfied **do**
  generate candidate solution *s* using **subsidiary greedy randomized constructive search**;
  perform **subsidiary local search** on *s* ;

- Randomization in constructive search ensures that a large number of good starting points for subsidiary local search is obtained.
- Constructive search in GRASP is 'adaptive' (or dynamic): Heuristic value of solution component to be added to a given partial candidate solution may depend on solution components present in it.
- Variants of GRASP without local search phase (aka semi-greedy heuristics) typically do not reach the performance of GRASP with local search.

GRASP

- Each step of constructive search adds a solution component selected uniformly at random from a **restricted candidate list (RCL)**.
- RCLs are constructed in each step using a heuristic function h.
  - RCLs based on **cardinality restriction** comprises the k best-ranked solution components.
    (k is a parameter of the algorithm.)
  - RCLs based on **value restriction** comprise all solution components ℓ for which $h(\ell) \leq h_{min} + \alpha \cdot (h_{max} - h_{min})$,
    where h_{min} = minimal value of h and h_{max} = maximal value of h for any ℓ.
    (α is a parameter of the algorithm)
  - Possible extension: **reactive GRASP** (dynamic adaptation of α during search)

A GRASP Example: Squeaky Wheel

Key idea: solutions can reveal problem structure which maybe worth to exploit.

Use a greedy heuristic repeatedly by prioritizing the elements that create troubles.

Squeaky Wheel

- **Constructor**: greedy algorithm on a sequence of problem elements.
- **Analyzer**: assign a penalty to problem elements that contribute to flaws in the current solution.
- **Prioritizer**: uses the penalties to modify the previous sequence of problem elements. Elements with high penalty are moved toward the front.

Possible to include a local search phase between one iteration and the other

Iterated Greedy

Key idea: use greedy construction

- alternation of **construction** and **deconstruction** phases
- an acceptance criterion decides whether the search continues from the new or from the old solution.

Procedure Iterated Greedy;
determine initial candidate solution s;
while termination criterion is not satisfied **do**
  $r \leftarrow s$;
  (randomly or heuristically) **destruct** part of s;
  greedily **reconstruct** the missing part of s;
  based on **acceptance criterion**, keep s or revert to $s \leftarrow r$

[Hoos, Stützle, SLS FA, 2004]

Very Large Scale Neighborhood Algorithms

- Local search algorithms typically explore small neighborhoods of a solution. Very large scale neighborhood (VLSN) algorithms are a generalization to large neighborhoods that can be searched efficently.

- Algorithms that proceed by iteratively destroying and repairing a solution can be seen as performing a neighborhood search in a large neighborhood as well.

- Examples of this kind are:
  - Iterated greedy
  - Large Neighborhood Search (LNS) proposed by [Shaw, 1998] use constraint programming to reconstruct a solution after destroying part of it.
  - Adaptive Large Neighborhood Search (ALNS) [Røpke and Pisinger, 2006]

Large Neighborhood Search (LNS)

input: a feasible solution x
$x^b = x$;
repeat
  $x^t = r(d(x))$;
  if accept(x^t, x) **then**
    $x = x^t$;
  end if
  if $c(x^t) < c(x^b)$ **then**
    $x^b = x^t$;
  end if
until stopping criterion is met
return x^b

$d()$ destruction (destroy) method
$r()$ reconstruction (repair) method

The LNS metaheuristic does not search the entire neighborhood of a solution, but merely samples this neighborhood.

Large Neighborhood Search (LNS)

Acceptance criterion:

- always
- only if better
- record-to-record travel (accept if $f(s') \leq (1+\epsilon)f(s_b)$)
- threshold accepting (Metropolis criterion)
- simulated annealing criterion

Degree of destruction

- gradually increase
- randomly chosen from a specific range dependent on the instance size

To guarantee connectivity, it must be possible to destroy every part of the solution.

Repair method:

- problem-specific heuristic
- exact method
- general purpose mixed integer programming (MIP) (aka, fix and optimize)
- constraint programming solver (aka, fix and optimize)

It should allow diversification

Adaptive Large Neighborhood Search (ALNS)

Key Idea: allow multiple destroy and repair methods controlling with an adaptive weighting system how often a particular method is attempted during the search [Røpke, Pisinger, 2006]

input: a feasible solution x
$x^b = x$; $\rho^- = (1, \ldots, 1)$; $\rho^+ = (1, \ldots, 1)$;
repeat
  select destroy and repair methods $d \in \Omega^-$ and $r \in \Omega^+$ using ρ^- and ρ^+;
  $x^t = r(d(x))$;
  if accept(x^t, x) **then**
    $x = x^t$;
  end if
  if $c(x^t) < c(x^b)$ **then**
    $x^b = x^t$;
  end if
  update ρ^- and ρ^+;
until stopping criterion is met
return x^b

Adaptive Large Neighborhood Search (ALNS)

Selection mechanism: roulette wheel principle:

$$p(j) = \frac{\rho_j^-}{\sum_{k \in \Omega^-} \rho_k^-}$$

Update mechanism:

$$\Psi = \max \begin{cases} \omega_1 & \text{if the new solution is a new global best} \\ \omega_2 & \text{if the new solution is better than the current one} \\ \omega_3 & \text{if the new solution is accepted} \\ \omega_4 & \text{if the new solution is rejected} \end{cases}$$

with normally $\omega_1 \geq \omega_2 \geq \omega_3 \geq \omega_4 \geq 0$. Only accepted a and b are updated:

$$\rho_a^- = \lambda \rho_a^- + (1 - \lambda)\Psi, \qquad \rho_b^+ = \lambda \rho_b^+ + (1 - \lambda)\Psi$$

$\lambda \in [0, 1]$ is a decay parameter

ALNS: Design Choices

Destroy methods:

- diversification: random destroy method.
- intensification: remove q "critical" variables, i.e. variables having a large cost or variables that spoil the current structure of the solution (e.g. edges crossing each other in an Euclidean TSP). This is known as **worst destroy** or **critical destroy**.
- **related destroy**: select a set of customers that have a high mutual relatedness measure. Eg on the CVRP, relatedness measure between each pair of customers is distance between the customers (and it could include customer demand)
- **history based destroy**: q variables are chosen according to some historical information,

Repair methods:

- greedy heuristics, problem specific
- include local search
- exact algorithms
- mixed integer programming (aka, matheuristic)
- constraint programming

ALNS: Design Choices

Large multiple-neighborhood search (LMNS) heuristics: It may be sufficient to have a number of destroy and repair heuristics that are selected randomly with equal probability, that is, without the adaptive layer.

Same robustness as ALNS heuristics, while fewer parameters to calibrate.

ALNS: Other Relations

- Variable Neighborhood Search
- Portfolio Algorithms
- Hyperheuristics
- Reinforcement learning

Adaptive Iterated Construction Search (AICS)

Key Idea: Alternate construction and local search phases as in GRASP, exploiting experience gained during the search process.

Realisation:

- Associate weights with possible decisions made during constructive search.
- Initialize all weights to some small value τ_0 at beginning of search process.
- After every cycle (= constructive + local search phase), update weights based on solution quality and solution components of current candidate solution.

Adaptive Iterated Construction Search (AICS)

Procedure AICS();
initialise weights;
while termination criterion is not satisfied **do**
    generate candidate solution *s* using subsidiary randomized constructive search;
    perform subsidiary local search on *s*;
    adapt weights based on *s*;

AICS: Components

Subsidiary constructive search:

- The solution component to be added in each step of constructive search is based on:
  i) weights and
  ii) heuristic function *h*.
- *h* can be standard heuristic function as, e.g., used by greedy heuristics
- It is often useful to design solution component selection in constructive search such that any solution component may be chosen (at least with some small probability) irrespective of its weight and heuristic value.

AICS: Components

Subsidiary local search:

- As in GRASP, local search phase is typically important for achieving good performance.
- Can be based on Iterative Improvement or more advanced LS method (the latter often results in better performance).
- Tradeoff between computation time used in construction phase *vs* local search phase (typically optimized empirically, depends on problem domain).

AICS: Components

Weight updating mechanism:

- Typical mechanism: increase weights of all solution components contained in candidate solution obtained from local search.
- Can also use aspects of search history;
  e.g., current candidate solution can be used as basis for
  weight update for additional intensification.

Example: A simple AICS algorithm for the TSP (1/2)

[Based on Ant System for the TSP, Dorigo et al. 1991]

- Search space and solution set as usual (all Hamiltonian cycles in given graph G). However represented in a construction tree T.
- Associate weight τ_{ij} with each edge (i,j) in G and T
- Use heuristic values $\eta_{ij} := 1/w_{ij}$.
- Initialize all weights to a small value τ_0 (parameter).
- Constructive search start with randomly chosen vertex and iteratively extend partial round trip π by selecting vertex not contained in π with probability

$$\Pr(\pi_j) = \frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{l \in N'(i)} [\tau_{il}]^\alpha \cdot [\eta_{ij}]^\beta}$$

Example: A simple AICS algorithm for the TSP (2/2)

- Subsidiary local search = typical iterative improvement
- Weight update according to

$$\tau_{ij} := (1 - \rho) \cdot \tau_{ij} + \Delta(ij, s')$$

where $\Delta(i, j, s') := 1/f(s')$, if edge ij is contained in the cycle represented by s', and 0 otherwise.

- Criterion for weight increase is based on intuition that edges contained in short round trips should be preferably used in subsequent constructions.
- Decay mechanism (controlled by parameter ρ) helps to avoid unlimited growth of weights and lets algorithm forget past experience reflected in weights.
- (Just add a population of cand. solutions and you have an Ant Colony Optimization Algorithm!)

Summary

1. Introduction
2. The ROAR-NET API
3. Solving Problems by Complete Search
4. Solving Problems by Incomplete Search

Acknowledgments

This presentation is based upon work from COST Action Randomised Optimisation Algorithms Research Network (ROAR-NET), CA22137, supported by COST (European Cooperation in Science and Technology).

COST (European Cooperation in Science and Technology) is a funding agency for research and innovation networks. Our Actions help connect research initiatives across Europe and enable scientists to grow their ideas by sharing them with their peers. This boosts their research, career and innovation.