AI505

Optimization

Discrete Optimization
Heuristic Methods, Randomized Optimization Algorithms

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

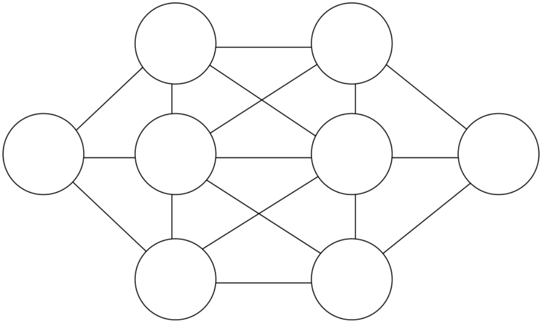
Outline

Outline

1. Constraint Programming

2. Randomized Optimization Heuristics

Number Circle Puzzle



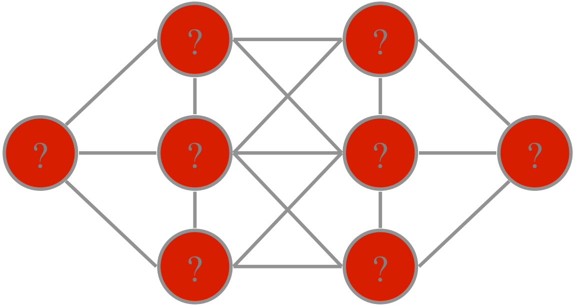
You have 8 minutes

Put a different number (1 to 8) in each circle such that adjacent circles do not have consecutive numbers.

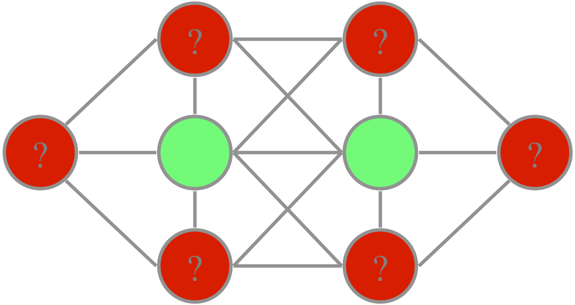
Example by Patrick Prosser with the help of Toby Walsh, Chris Beck, Barbara Smith, Peter van Beek, Edward Tsang, ...

Heuristic Search

Which nodes are hardest to number?

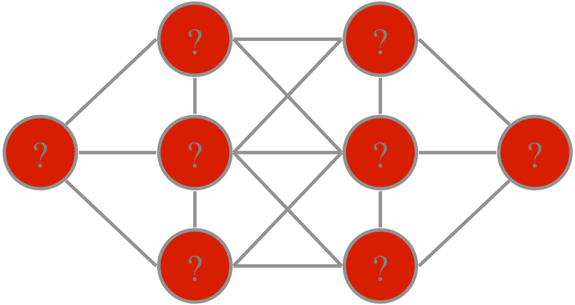


Heuristic Search



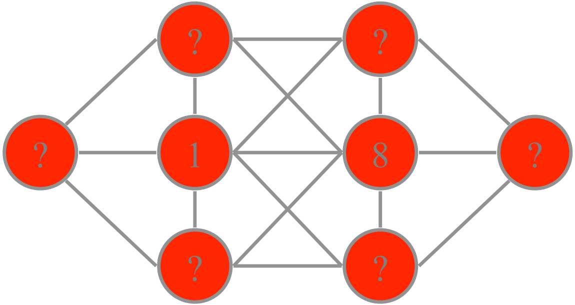
Heuristic Search

Which are the least constraining values to use?



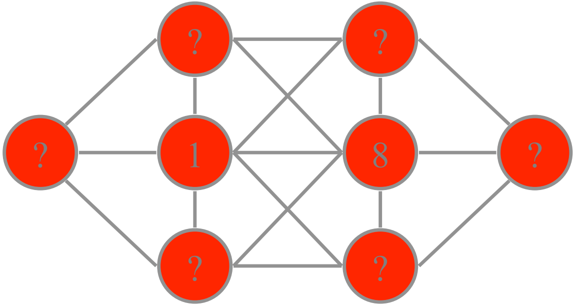
Heuristic Search

Values 1 and 8



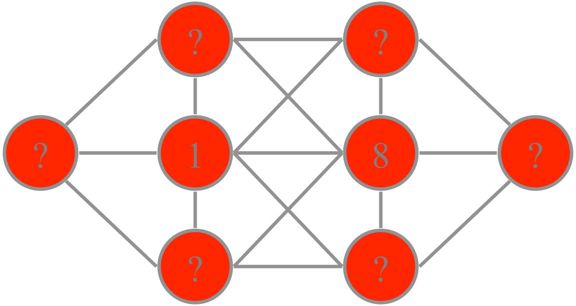
Heuristic Search

Values 1 and 8



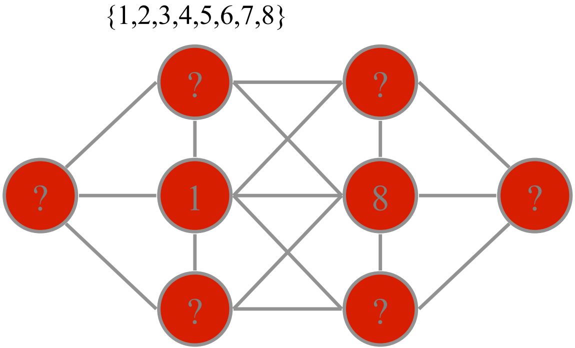
Symmetry means we don’t need to consider: 8 1

Inference/propagation

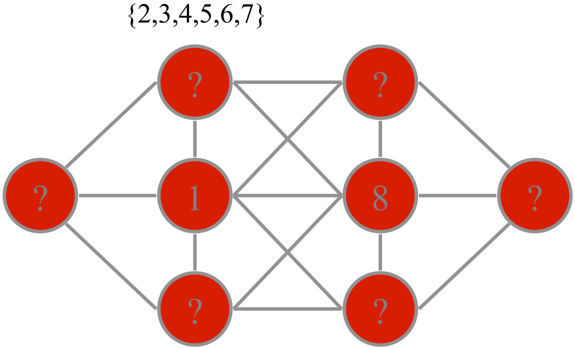


We can now eliminate many values for other nodes

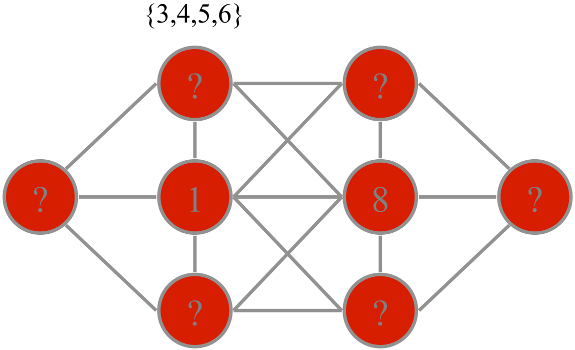
Inference/propagation



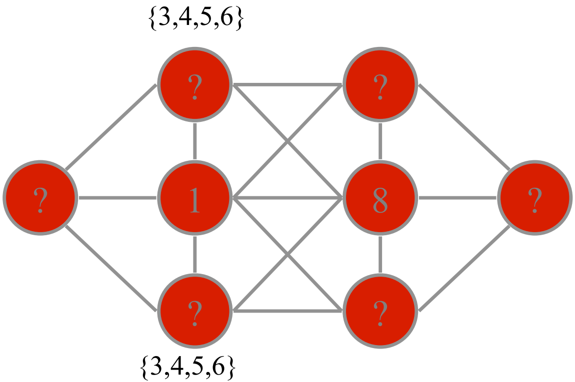
Inference/propagation



Inference/propagation

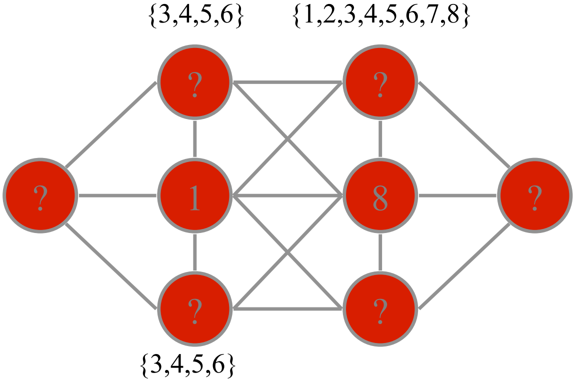


Inference/propagation

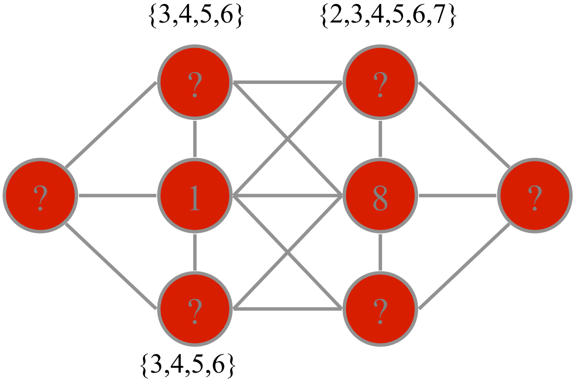


By symmetry

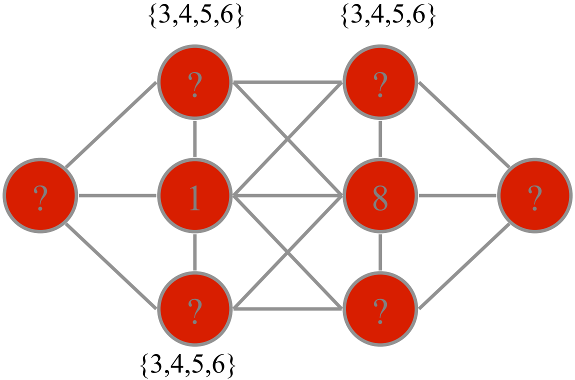
Inference/propagation



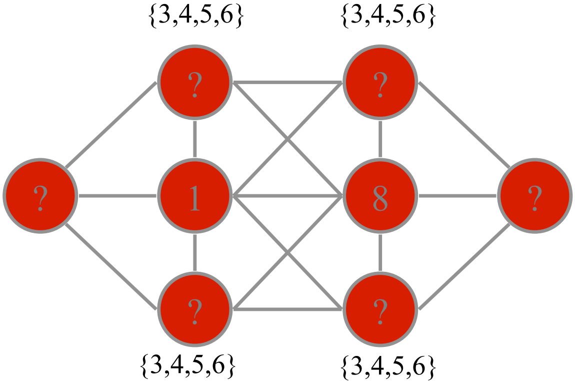
Inference/propagation



Inference/propagation

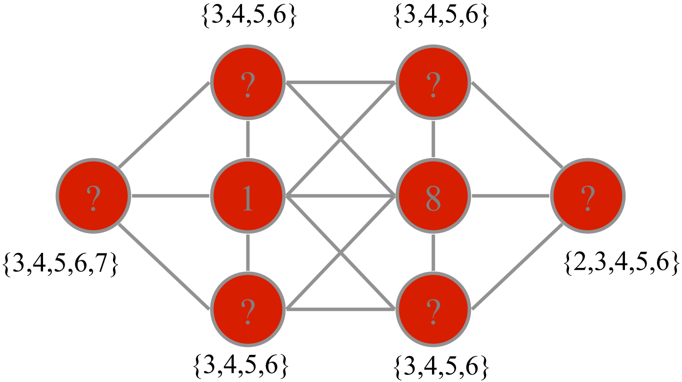


Inference/propagation

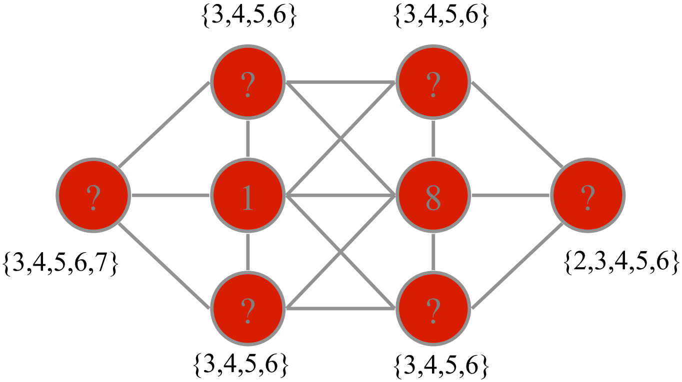


By symmetry

Inference/propagation

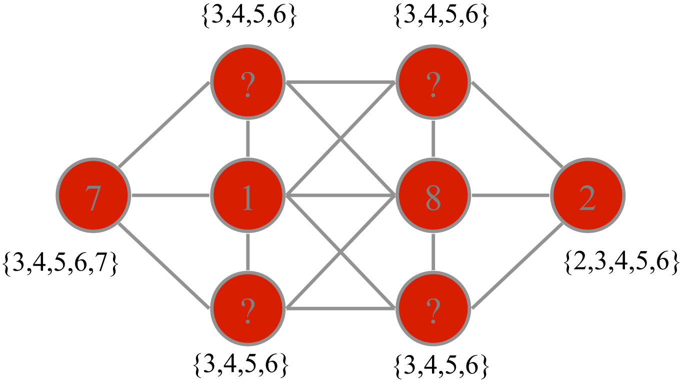


Inference/propagation



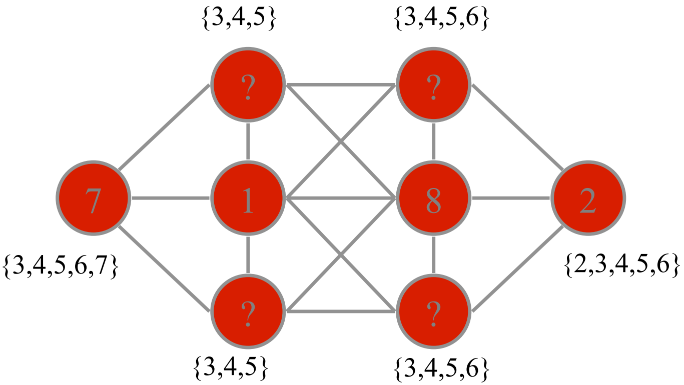
Value 2 and 7 are left in just one variable domain each

Inference/propagation



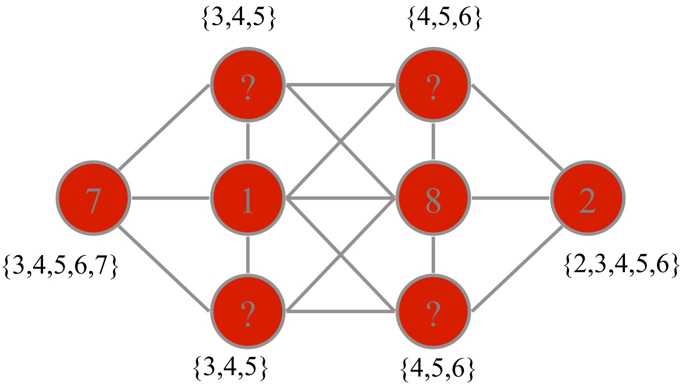
And propagate …

Inference/propagation



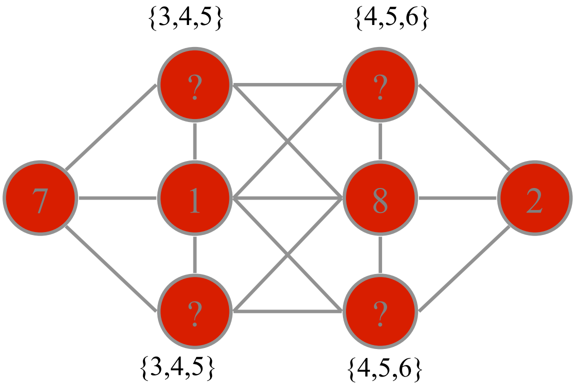
And propagate …

Inference/propagation



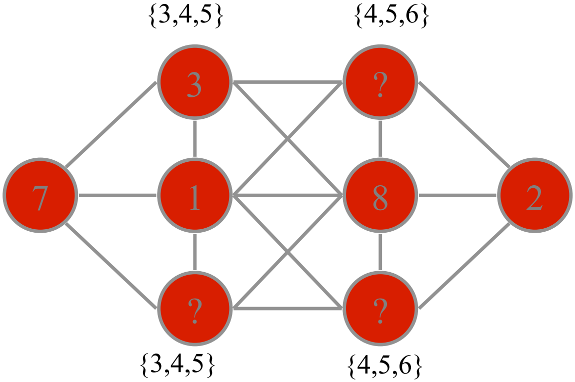
And propagate …

Inference/propagation



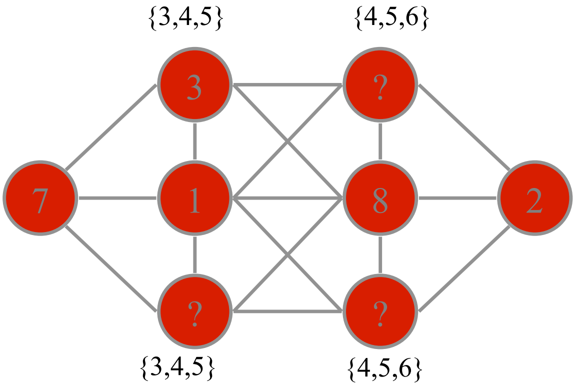
Guess a value, but be prepared to backtrack …

Inference/propagation



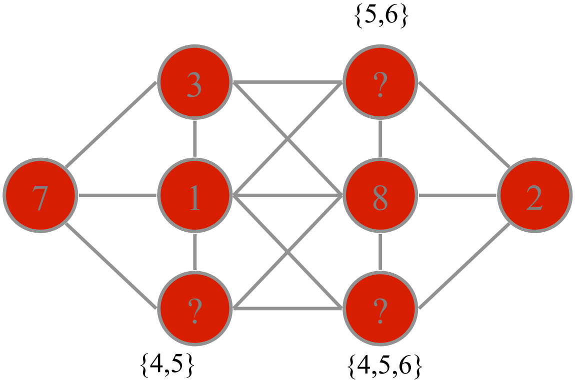
Guess a value, but be prepared to backtrack …

Inference/propagation



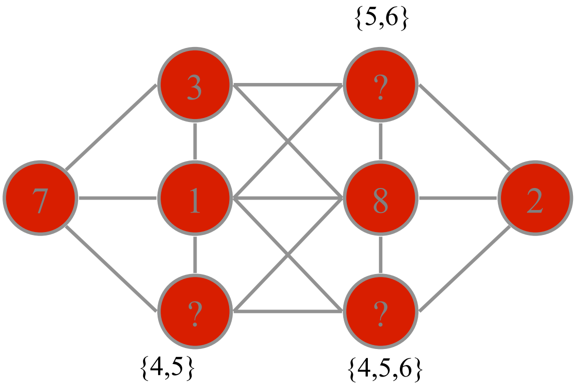
And propagate …

Inference/propagation



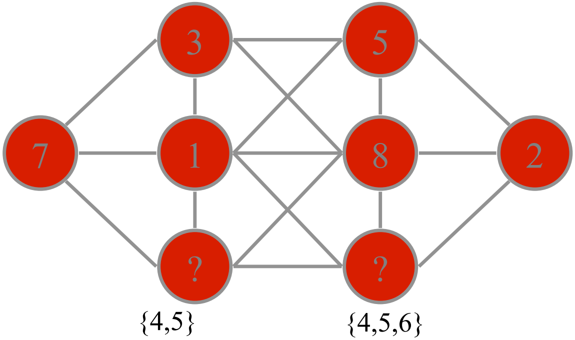
And propagate …

Inference/propagation



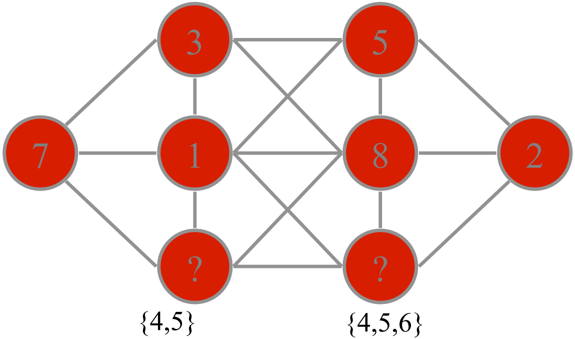
Guess another value …

Inference/propagation



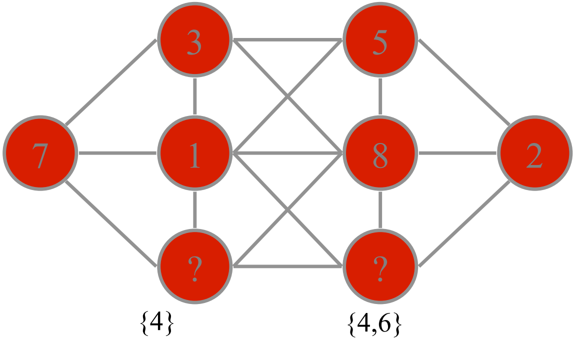
Guess another value …

Inference/propagation



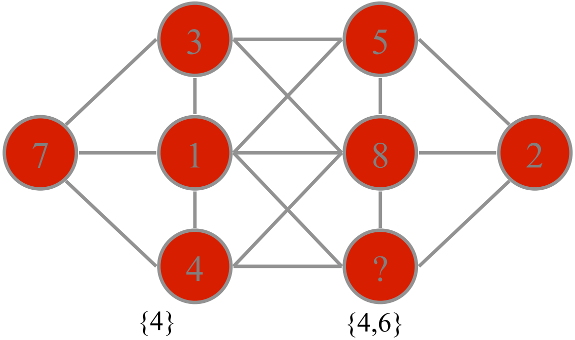
And propagate …

Inference/propagation



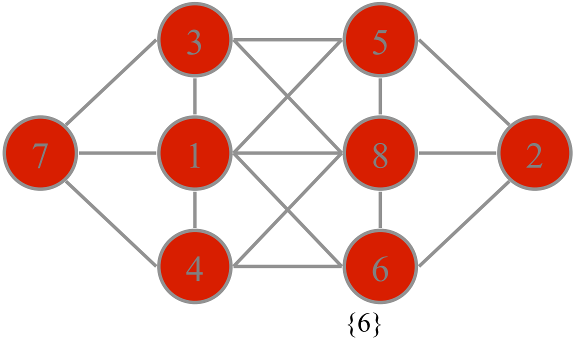
And propagate …

Inference/propagation

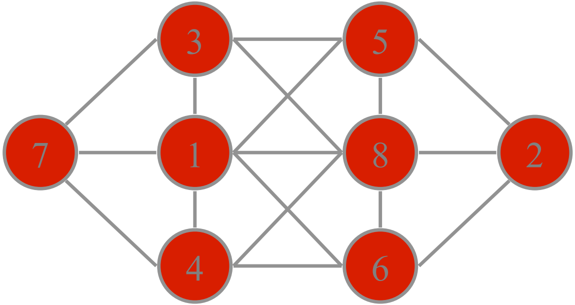


One node has only a single value left …

Inference/propagation



Solution

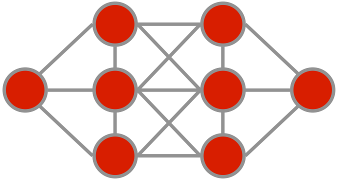


The Core of Constraint Computation

- Modelling
  - Deciding on variables/domains/constraints
- Heuristic Search
- Inference/Propagation
- Symmetry
- Backtracking

Hardness

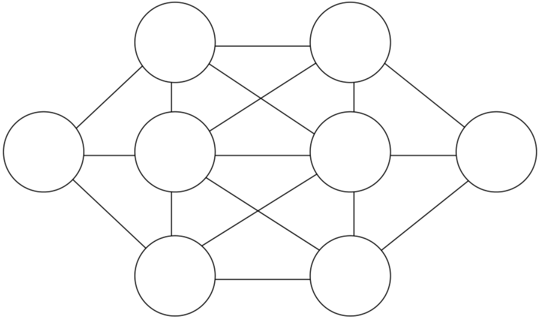
- The puzzle is actually a hard problem
  - NP-complete



Constraint programming

- **Model** problem by specifying constraints on acceptable solutions:
  - define variables and domains
  - post constraints on these variables
- Solve model
  - choose algorithm
    - incremental assignment / backtracking search
    - complete assignments / stochastic search
  - design heuristics

Constraint Satisfaction Problem



- Variable x_i for each node $i = 1, \ldots, 8$
- Domain $\{1, \ldots, 8\}$ for each variable x_i
- Constraints:

$\texttt{allDifferent}([x_1, x_2, x_3, x_4, x_5, x_6, x_7, x_8])$
$|x_1 - x_2| > 1$
$|x_2 - x_3| > 1$
$|x_3 - x_4| > 1$
$\vdots$
$|x_7 - x_8| > 1$

Modeling in Constraint Programming

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.

If $C = \emptyset$, we say that it is **inconsistent**.

Modeling in Constraint Programming

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 (maximization) COP is a solution d to $\mathcal{P}$ that minimizes (maximizes) the value of $f(d)$.

Another Example: Social Golfers

Example (Social Golfer Problem (Combinatorial Design))

- 9 golfers: 1, 2, 3, 4, 5, 6, 7, 8, 9
- wish to play in groups of 3 players in 4 days
- such that no golfer plays in the same group with any other golfer more than just once.

Is it possible?

	Group 1	**Group 2**	**Group 3**
Day 0	? ? ?	? ? ?	? ? ?
Day 1	? ? ?	? ? ?	? ? ?
Day 2	? ? ?	? ? ?	? ? ?
Day 3	? ? ?	? ? ?	? ? ?

This is an instance of a **constrained satisfaction problem**. Adding an optimizing criterion we get a **constrained optimization problem**.

Solution Paradigms

1. Dedicated algorithms
   (eg.: enumeration, branch and bound, dynamic programming)
2. Constraint Programming
3. Integer Linear Programming
4. Other modeling (SAT, SMT, etc.)
5. Randomized Search/Optimization Heuristics

Common to 2-5: **Representation (modeling) + reasoning (search + inference)**

Constraint Programming: Representation

Golfers

	Group 1	Group 2	Group 3
Day 0	? ? ?	? ? ?	? ? ?
Day 1	? ? ?	? ? ?	? ? ?
Day 2	? ? ?	? ? ?	? ? ?
Day 3	? ? ?	? ? ?	? ? ?

Integer variables:

`assign[i,j]` variables whose value is from domain $\{1, 2, 3\}$

Constraints:

C1: each group has exactly groupSize players

C2: each pair of players only meets once

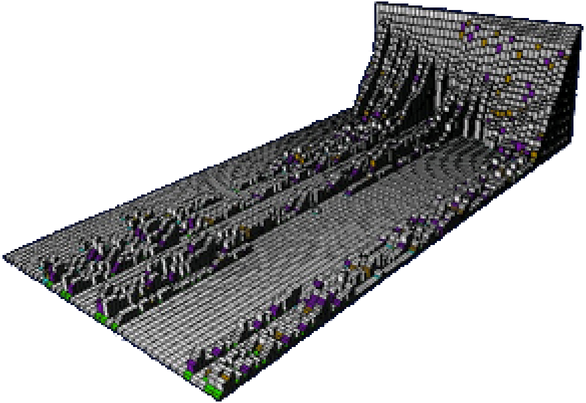
Groups

	Day 0	Day 1	Day 2	Day 3
Golfer 0	1	{1,2,3}	{1,2,3}	{1,2,3}
Golfer 1	1	{1,2,3}	{1,2,3}	{1,2,3}
Golfer 2	1	{1,2,3}	{1,2,3}	{1,2,3}
Golfer 3	{2,3}	{1,2,3}	{1,2,3}	{1,2,3}
Golfer 4	{2,3}	{1,2,3}	{1,2,3}	{1,2,3}
Golfer 5	{2,3}	{1,2,3}	{1,2,3}	{1,2,3}
Golfer 6	{2,3}	{1,2,3}	{1,2,3}	{1,2,3}
Golfer 7	{2,3}	{1,2,3}	{1,2,3}	{1,2,3}
Golfer 8	{2,3}	{1,2,3}	{1,2,3}	{1,2,3}

Constraint Programming: Representation

```
int: golfers = 9;
int: groupSize = 3;
int: days = 4;
int: groups = golfers/groupSize;
set of int: Golfer = 1..golfers;
set of int: Day = 1..days;
set of int: Group = 1..groups;

array[Golfer, Day] of var Group: assign; % Variables

constraint
  % C1: Each group has exactly groupSize players
  forall (gr in Group, d in Day) ( % c1
   sum (g in Golfer) (bool2int(assign[g,d] = gr)) = groupSize
  ) /\
  % C2: Each pair of players only meets at most once
  forall (g1, g2 in Golfer, d1, d2 in Day  where g1 != g2 /\ d1 != d2) (
   (bool2int(assign[g1,d1] = assign[g2,d1]) + bool2int(assign[g1,d2] = assign[g2,d2])) <=1);

solve :: int_search([assign[i,j] | i in Golfer, j in Day ],
                   first_fail, indomain_min, complete)  satisfy;
```

Constraint Programming: Reasoning



The solution process proceeds by propagating the constraints on the domanins of the variables (ie, removing values) and tentatively assigning variables until only feasible values are left or backtracking.

MILP: Representation

System of linear equalities and inequalities. Example (diet problem):

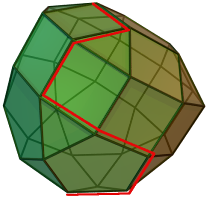
$$\begin{aligned}
\min \quad & \sum_{j \in F} c_j x_j \\
\sum_{j \in F} a_{ij} x_j &\geq N_{min,i}, && \forall i \in N \\
\sum_{j \in F} a_{ij} x_j &\leq N_{max,i}, && \forall i \in N \\
x_j &\geq F_{min,j}, && \forall j \in F \\
x_j &\leq F_{max,j}, && \forall j \in F
\end{aligned}$$

Mixed Integer Linear Programming: Representation

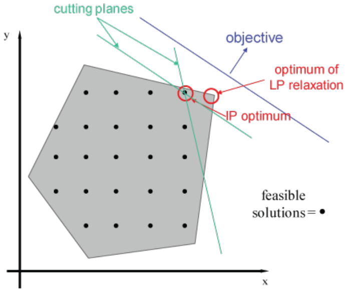

```
m = Model(``diet'')

buy = {}
for f in foods:
    buy[f] = m.addVar(obj=cost[f], name=f)


for c in categories:
    m.addConstr(
      quicksum(nutritionValues[f,c] * buy[f] for f in foods) <= maxNutrition[c],
                name=c+'max')
    m.addConstr(
      quicksum(nutritionValues[f,c] * buy[f] for f in foods) >= minNutrition[c],
                name=c+'min')

m.optimize()
```

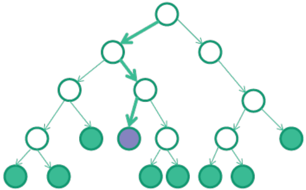
Mixed Integer Linear Programming: Reasoning

Linear Programming:



Integer Programming:





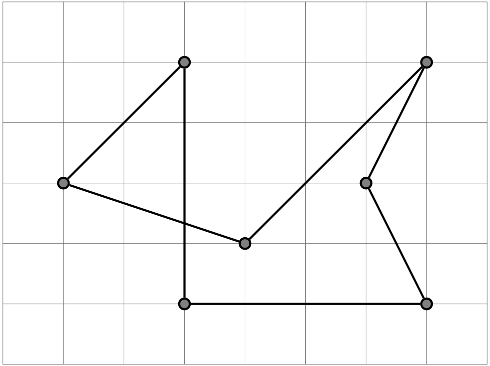
Each node in branch-and-bound is a new MIP

Polynomial run-time

NP-hard (exponential worst-case run-time)

Yet Another Example: TSP

Example (Traveling Salesman Problem)



Can you find a better solution?

Outline

1. Constraint Programming

2. Randomized Optimization Heuristics

Heuristics

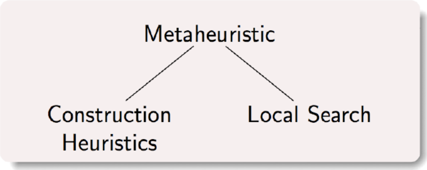
- Get inspired by approach to problem solving in human mind
  [A. Newell and H.A. Simon. "Computer science as empirical inquiry: symbols and search." Communications of the ACM, ACM, 1976, 19(3)]
  - effective rules without theoretical support
  - trial and error
- Applications:
  - Optimization
  - But also in Psychology, Economics, Management [Tversky, A.; Kahneman, D. (1974). "Judgment under uncertainty: Heuristics and biases". Science 185]
- Basis on empirical evidence rather than mathematical logic. Getting things done in the given time.

Randomized Optimization Heuristics (ROHs)

Two main search paradigms:

- Constructive search
- Local search

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



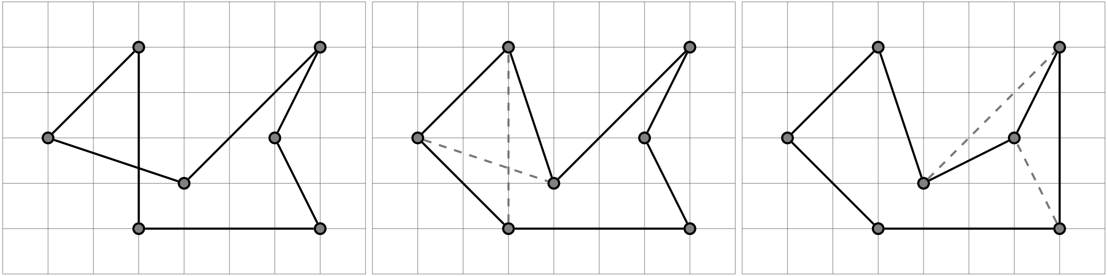
ROHs: Representation

	Group 1	Group 2	Group 3
Day 0	0 1 2	3 4 5	6 7 8
Day 1	**0 4** 6	1 **3 7**	2 5 8
Day 2	**0 4** 8	1 5 6	2 **3 7**
Day 3	0 5 7	1 3 8	2 4 6

- Variables = solution representation, tentative solution
- Constraints: relaxed = soft
- Evaluation function to guide the search

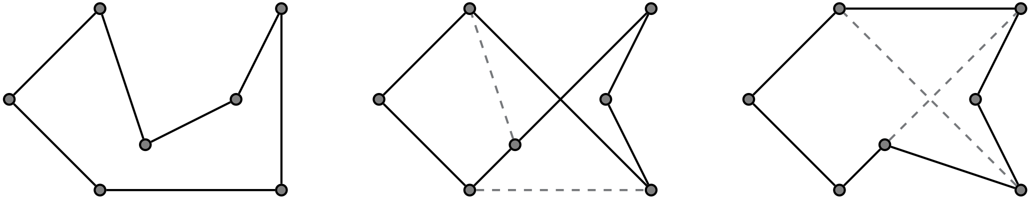
ROHs: Reasoning, Local Search

Example on Traveling Salesman Problem:

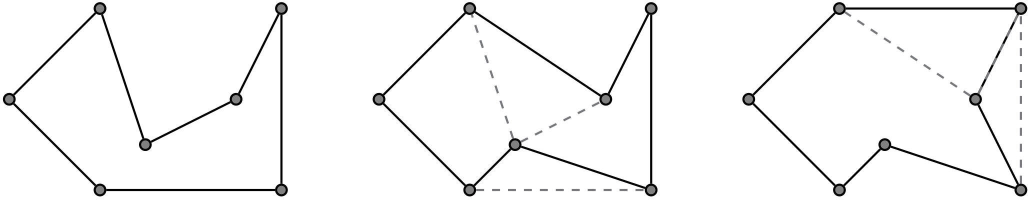


ROHs: Reasoning, Metaheuristics

Accepting worsening changes



Trying different changes



ROHs: Reasoning, Metaheuristics

- Stochastic Local Ssearch
- Simulated Annealing
- Iterated Local Search
- Tabu Search
- Variable Neighborhood Search
- Adaptive Large Neighborhood Search
- Evolutionary Algorithms
- Ant Colony Optimization
- Estimation-of-Distribution Algorithms
- Artificial Immune Systems
- ...
- Evolutionary Computation Bestiary `http://fcampelo.github.io/EC-Bestiary/`
- **Super**natural inspired [Maturana, Fouhey, 2013]

A Classification

- White box optimization:
  models can be expressed mathematically
- Grey box optimization:
  internal information about objective function computation is often available
  models that have a mathematical expression but may need data to determine them (eg, neural networks)
- Black box optimization:
  no mathematical expression is available

Approaches to ROHs

- White/Grey box: representation (modelling) + reasoning (search)
  constraint based local search, local solver (Hexaly)

- Black Box: a different approach, framework separating problem from solvers and defining the interface specification
  EasyLocal, ...,

$\Longrightarrow$ Cost Action: ROAR-NET

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

A Search Problem (the Abstraction from AI Course)

Definition (Problem statement)

Assume we want to solve a **constrained optimization problem**: $\min f(x) \mid x \in F$ where is a set of **feasible solutions** and f an objective function. All parameters of the problem are known and deterministic.

Definition (Search or Optimization Algorithm)

Goal formulation: we want to find the minimum with respect to some criteria 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 reach a solution, or it will find that no solution is possible.

A Search Problem (the Abstraction from the AI Course)

Ingredients of a Search Algorithm:

- **State or Search Space** A set of possible **states** that the search can be in.
- **State**: a definition of the states of the search, Candidate solutions.
- **Initial State** that the search starts in. For example: an empty set of actions or a complete set of actions.
- A set of one or more goal states. Sometimes there is one goal state sometimes there is a small set of alternative goal states
- **ACTION TYPES** t available to the algorithm.
- For a given ACTION TYPE t and a state/solution s, $ACTIONS(t, s)$ returns a finite set of actions of type t that can be executed in s. We say that each of these actions is applicable in s. A transition model, which describes what each action does.
- RESULT(s, a) returns the state that results from doing action a in state s.

Application Programming Interface (API)

(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 which

- ... 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.
- ... aims at maximizing code extensability, reusability, and simplicity.

Types

- `Problem` the problem instance
- `Solution` implements the representation of a **tentative** solution
- `Value` represents points in objective space
- `Neighborhood` a function that given a solution, gives another solution $neighbor : S \to S$, based on a neighborhood, compute a `Move`
- `Move` applied to a solution to get the novel solution
- `Increment` represents points in objective space

Simplification in single objective cases: `Value` and `Increment` are Reals (ie, `Double` or `Integer`)

Operations: Problem Representation

... on Problem (P):

- `empty_solution(P): Solution`
- `random_solution(P): Solution`
- `heuristic_solution(P): Solution[0..1]`

... on Solution S:

- `copy_solution(S): Solution`
- `lower_bound(S): Value[0..1]`
- `evaluation_value(): Value[0..1]`

Operations: Search

... on Neighborhood (N):

- `construction_neighbourhood(Problem): Neighbourhood`
- `destruction_neighbourhood(Problem): Neighbourhood`
- `local_neighbourhood(Problem): Neighbourhood`
- `moves(N, Solution): Move[0..*]`
- `random_move(N, Solution): Move[0..1]`
- `random_move_without_replacement(N, Solution): Move[0..*]`

Operations: Search

... on Move:

- `lower_bound_increment(Move, Solution): double[0..1]`
- `objective_value_increment(Move, Solution): double[0..1]`
  `apply_move(M, Solution): Solution` applies the move to a solution, to get a novel solution
- `invert_move(M): Move` computes the inverse of a move to revert it

The Full API

```
# Types
Problem
Solution
Value
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
```

```
# Operations on Move
moves
random_move
random_moves_without_replacement

lower_bound_increment
objective_value_increment

apply_move
invert_move
```