Introduction for Combinatorial DM

Combinatorial Decision making is decision making with many combinations of possibilities subject to **restrictions = CONSTRAINTS**.

- Any solution (meets all constraints) $\rightarrow$ satisfactions problems.
- Optimal solution (best solution according to an objective) $\rightarrow$ optimization problems.

Diagnostic analytics are all about understanding the past
Optimization refers to prescriptive analytics

- Properties
  - Computationally difficult problems (NP-hard)
  - Can only be solved by *intelligent search*.
  - Experimental in nature, because we have more different types and modes and there is not a clear better one.
  - Finding good/optimal solution can same time, money and reduce environmental impact.

Why Constraint Programming?

- Important and growing area of AI.
- Different applications: logistics, scheduling, planning.
- Useful asset.

What is Constraint Programming?

A **declarative programming paradigm** for stating and solving combinatorial optimization problems.

- User **models** a decision problem by formalizing:
  - **the unknowns** of the decision $\rightarrow$ **decision variables** (X_i)
  - **possible values** for unknowns $\rightarrow$ **domains** ($D(X_i) = \{v_i\}$)
  - **relations** between the unknowns $\rightarrow$ **constraints** ($r(X_i , X_i'$)
- A constraint **solver** finds a solution to the model (or proves that no solution exists) by assigning a value to every variable via a **search algorithm**.

Why Constraint Programming?

CP provides a **rich language for expressing constraints** and **defining search procedures**.

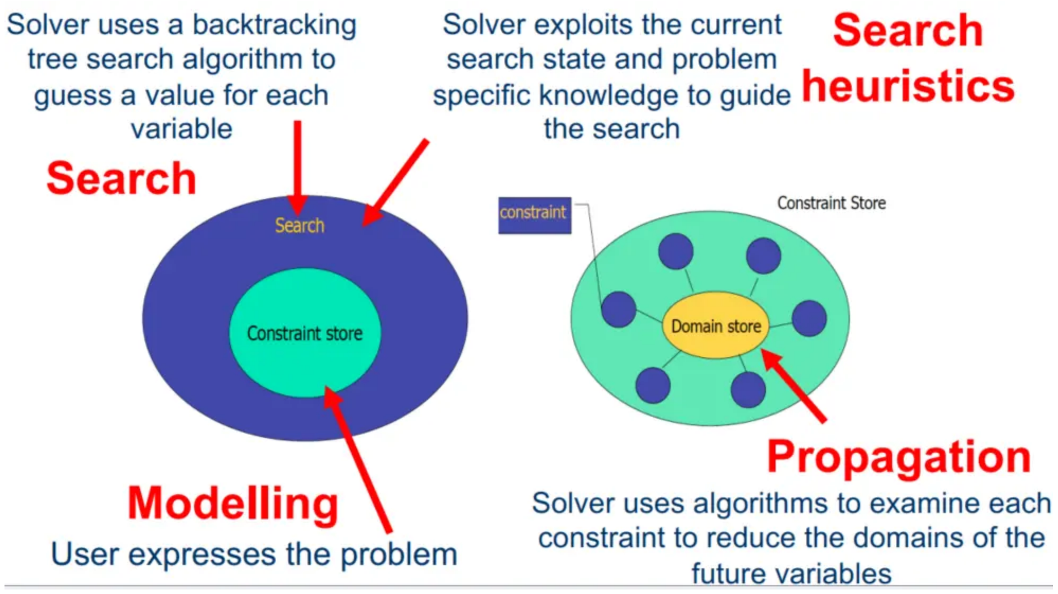
- Easy modelling.
    - fast prototyping with a variety of constraints.
    - easy to maintain programs.
    - extensibility.
- Easy control of search.
    - experimentation with advanced search strategies.
- Main focus on **constraints** and **feasibility**. (thanks to *propagation*)
    - constraints $\rightarrow$ reductions in the search space. (exploits local structure)
    - interest on tightly constrained problems.
    - more constraints $\rightarrow$ more domain reductions $\rightarrow$ problem easier to solve.
- Strengths of CP
    - timetabling, sequencing, scheduling, rostering.
    - can handle messy constraints non-linear in nature and similar things.
- Weaknesses of CP
    - no special focus on objective function and optimality (better alternatives for optimality)
    - best optimality approaches are often hybrids (CP + ILP and HS), and CP is a suitable framework for it

Overview of CP

Constraint solver

Enumerates all possible variable-value combinations via a **systematic backtracking tree search**.(guesses a value for each variable)
During search, examines (doesn't check (pure search does that)) the constraint to **remove incompatible values** from the domain of the **future (unexplored) variables**, via **propagation**. (shrinks the domain of the future variables)

Constraint Programming


Solver uses a backtracking tree search algorithm to guess a value for each variable
Search
Solver exploits the current search state and problem specific knowledge to guide the search
Search heuristics
Search
Constraint store
constraint
Constraint Store
Domain store
Propagation
Solver uses algorithms to examine each constraint to reduce the domains of the future variables
Modelling
User expresses the problem

A model captures combinatorial substructures and enables solver to reduce the search space.

- constraints = propagation algorithms.
- variables = communication mechanism.
  In fact, search decisions and propagation are interleaved: propagation $\rightarrow$ decision $\rightarrow$ propagation $\rightarrow$ decision $\rightarrow \dots$
  Modelling is critical as the user has to use advanced modelling techniques for strong propagation.
  The default search of the solver isn't usually enough so the user often has to program the search strategy (algorithms, heuristics, ...)

Some problems in CP

- Bad choice of variables, bad assignment of values $\rightarrow$ good heuristic choice is very important.
- Good heuristics aren't always possible.
- We can try to apply stronger form of propagation during search.
- Better modelling can result in stronger form of propagation.

Efficient CP programming

We need:

- effective propagation algorithms.
- model with effectively propagating constraints.
- effective search algorithm and heuristics.

Additional notes

- Our search is a complete systematic search
- Every time I use a value for a variable it defines some impossible values, so that I can remove those values from the constraint domain
- This propagation mechanism makes constraint programming different from a simple search
- During the search we examine almost all constraint(some are ignored thanks to some optimization)
- The constraint domain interact with each others thanks to the shrinking
- Propagation comes before search so you can already shrink the search space
- The solver does this firstly removing inconsistent values before the first search
- Solvers looks and propogates constraints on the future variables, not only on the current ones
- With a better and more complete model we can improve the efficiency of this process

PART 1 - Modeling

User **models** a decision problem by formalizing:
- **the unknowns** of the decision $\rightarrow$ **decision variables** (X_i)
- **possible values** for unknowns $\rightarrow$ **domains** ($D(X_i) = \{v_i\}$)
- **relations** between the unknowns $\rightarrow$ **constraints** ($r(X_i , X_i')$)

Formalization as CSP

A CSP (Constraint Satisfaction Problem) is a triple <X, D, C> where:

1. X is a set of decision variables $\{X_1, ..., X_n\}$;
2. D is a set of domains $\{D_1, ..., D_n\}$ for X:
    1. D_i is a set of possible values for X_i ;
    2. usually non-binary and assume finite domain;
3. C is a set of constraints $\{C_1, ... , C_m \}$:
    1. C_i is a relation over $X_j, ..., X_k$, denoted as $C_i(X_j, ..., X_k)$;
    2. C_i is the set of combination of allowed values $C_i \subseteq D(X_j) \times \ldots \times D(X_k)$
    A solution to a CSP is an assignment of values to the variables which satisfies all constraints simultaneously.

CSP enhanced with an optimization criterion (minimum cost, shortest distance, fastest route, maximum profit) are called Constraint Optimization Problems.
Formally, <X, D, C, f> where f is the formalization of the optimization criterion as an objective variable. Goal = minimize f (or maximize -f).

Variables and Domains

- Variable domains include the classical:
    - binary, integer, continuous.
- May take a value from any finite set (e.g. X in {a, b, c, d, e}).
- Special structured variable types:
    - set variables (a set of item as value)
    - activities or interval variables (for scheduling)

Constraints

- Any kind of constraint can be expressed by listing all allowed combinations. (e.g. $C(X_1 , X_2) = \{(0,0), (0,2), (1,3), (2,1)\}$) (also called TABLE representation)
    - extensional representation.

    - General but inconvenient and inefficient with large domains.
- Declarative (invariant) relations among objects. (e.g X > Y)
    - intensional representation.
    - Compact, clear but less general.

Properties of constraints

- The order of imposition does not matter.
- Non-directional. (a constraint can be user to infer domain information (and propagation) in both verses)
- Rarely independent. (shared variables as communication mechanism between constraints)

Modeling

- **Choice of variable and domains defines the search space**.
- **Choice of constraints defines:**
    - how search space can be reduces.
    - how search can be guided.

Symmetry in CSPs

- Creates many symmetrically equivalent search states:
    - a state will have many symmetrically equivalent states whether it leads to a solution or a failure.
- Bad when proving optimality, infeasibility or looking for all solutions.
- We focus on variable and value symmetry

Symmetries and Permutation

- **Permutation**:
    - Defined over a discrete set S as a 1-1 function $\pi: S \rightarrow S$.
    - Intuitively: re-arrangement of a set of elements
- **Variable Symmetry**:
    - A **permutation π of the variable indices** such that for each (un)feasible (partial) assignment, we can re-arrange the variables according to π and obtain another (un)feasible (partial) assignment
    - Intuitively: **permuting variable assignments**.
- Value Symmetry:
    - A **permutation π of values** such that for each (un)feasible (partial) assignment, we can re-arrange the values according to π and obtain another

(un) feasible (partial) assignment.
- Intuitively: **permuting values**.

Symmetry breaking constraints

Added constraint to break the symmetry.
Those are not logically implied by existing constraints.
Serves to try and speed up the solver.
If you don't do it well you end up missing solutions (at least one solution from each set of symmetrically equivalent solutions must remain).
A common technique is to impose an ordering to avoid permutations.

Dual Viewpoint

- Viewing a problem P from different perspectives may result in different models for P.
- Each model yields the same set of solutions.
- Each model is a different representation of P with different variables, domains and constraints $\rightarrow$ different size of the search space.

Combined Model

Can be useful to use different representation of a problem (models) with a **Combined Model**.
Keep both models and use **channeling constraints** to maintain consistency between the variables of the models.
Benefits:

- Facilitation of the expression of constraints.
- Enhanced constraint propagation. (more constraints = more opportunities for propagation)
- More options for search variables. (search on more variables and so more flexibility)

PART 2 - Constraint Propagation & Global Constraint

Part A - Consistency & Propagation

Constraint Solver definition.
Search decisions and propagation are interleaved: propagation $\rightarrow$ decision $\rightarrow$ propagation $\rightarrow$ decision $\rightarrow$ …

Local Consistency

- A form of inference which detects inconsistent (that do not take part in any solution) partial assignments. (subset of the variables assigned to a certain value - as long as not all variables have an a assignment)
- Local $\rightarrow$ we examine individual constraints.
- Popular local consistencies are domain-based (because they can directly change the domain of the variable):
  - GAC (Generalized Arc Consistency)
  - BC (Bounds Consistency)
  - domain-based $\rightarrow$ they detect inconsistent partial assignments of the form X_i = j:
    - j can be removed from D(X_i) via propagation.
    - propagation implemented easily.

Generalized Arc Consistency (GAC)

Basic constraint definition.
A constraint C defined on k variables C(X_1,..., X_k) gives the set of allowed combinations of values (i.e. allowed tuples).
Each allowed tuple (d_1,..., d_k) ∈ C where d_i ∈ X_i is a **support** for C.
C(X_1,..., X_k) is GAC iff:

- for all X_i in {X_1,..., X_k}, for all v ∈ D(X_i), v belongs to a support for C. (it's a tuple that satisfies the constraint)
  Called Arc Consistency (AC) when k = 2.
  A CSP is GAC iff all of its constraints are GAC.

Bounds Consistency (BC)

Generally weaker than GAC: everything detected by BC can be detected by GAC but not vice versa.

- Defined for totally ordered (e.g. integer) domains.
- Relaxes the domain of X_i from $D(X_i)$ to [$\min(X_i)$ ... $\max(X_i)$]
- A **bound support** is a tuple (d_1, ... , d_k) $\in$ C where $d_i \in$ [$\min(X_i)$ ... $\max(X_i)$]
- $C(X_1,..., X_k)$ is BC iff:
  - for all X_i in $\{X_1,..., X_k\}$, $\min(X_i)$ and $\max(X_i)$ belongs to a bound support.
- Disadvantage : BC might not detect all CAG inconsistencies in general $\rightarrow$ need to search more.
  - For some constraints (alldifferent for example) there are efficient algos to achieve GAC so BC isn't needed.
- Advantages :
  - Easier to look for a support in a range (opposed to a domain).
  - Achieving BC is often cheaper than GAC. (large domains)
  - Achieving BC is enough for GAC for monotonic constraints.

Constraint Propagation

- A **local consistency** notion defines properties that a constraint C must satisfy **after constraint propagation**.
  - propagation is the action, consistency is the notion that comes after the action.
- Propagation = takes inconsistent values and removes them from the variable domain, effectively making the constraint consistent.
  - The operational behaviour is left open; we can create algorithms to maintain consistency, the only requirement is to achieve the required property on C.

Propagation Algorithms & Constraint propagation

A propagation algorithm achieves a certain level of consistency on a constraint C by **removing the inconsistent values (detecting them first) from the domains of the variables** in C.

- The level of consistency depends on C.
  - GAC if an efficient propagation algorithm can be developed.
  - Otherwise BC or a lower level of consistency.
- Solving a CSP with multiple constraint, propagation algorithms interact, waking up other constraint to be propagated again; in the end propagation reaches a fixed-point and all constraints reach a level of consistency $\rightarrow$ this process is referred as ***constraint propagation***.

Properties of Propagation Algorithms

- It may be not enough to remove inconsistent values from domains once.
- A P.A. must wake up again when necessary $\rightarrow$ if not we may not achieve local consistency properly.
- Events that can trigger a constraint propagation:
  - domain changes (GAC)
  - domain bounds changes (BC)
  - variable is assigned a value

Complexity of Propagation Algorithms

Assuming |D(Xi)| = d $\rightarrow$ one time AC propagation on a C(X1,X2) takes O(d^2) time.
We can do better, depending on the semantics of the constraint and using the triggers for constraint propagation.

Specialized Propagation

Propagation specific to a given constraint.

- Exploits the constraint semantics $\rightarrow$ potentially much more efficient than a general propagation algorithms.
- Limited use (depends on the constraint); not always easy to develop one.
  Worth developing for recurring constraints.

Part B - Global Constraints

Global Constraints and benefits

- Capture complex, non-binary and recurring combinatorial substructures arising in a variety of applications
- Embed specialized propagation which exploits the substructure.
- Modelling benefits:
  - Reduce the gap between the problem statement and the model.
  - May allow the expression of constraints that are otherwise not possible to state using primitive constraints (semantic).
- Solving benefits:
  - Strong inference in propagation (operational).
  - Efficient propagation (algorithmic)

Types of global constraint

- Counting constraints = Restrict the number of variables satisfying a condition or the number of times values are taken.
  - alldifferent
  - nvalue $\rightarrow$ Constrains the number of distinct values assigned to the variables; useful in resources allocation.
  - gcc $\rightarrow$ Constrains the number of times each value is taken by the variable; useful in resources allocation.
  - among $\rightarrow$ Constrains the number of variables taken from a given set of values; useful in sequencing problems. (how many variable are taking the values in the domain)
- Sequencing constraints = Ensure a sequence of variable obey certain patterns.
  - sequence/AmongSeq $\rightarrow$ Constrains the number of values taken from a given set in any subsequence of q variables.
- Scheduling Constraints = Help schedule task with respective release times, duration, deadlines, using limited resources in a time interval.
  - disjunctive (noOverlap) $\rightarrow$ tasks don't overlap in time; one resource can execute at most one task at time.
  - cumulative $\rightarrow$ constraints the usage of a shared resource; used when a resource can execute multiple tasks at time.
- Ordering Constraints = Enforce an ordering between the variables or the values.
  - lex $\rightarrow$ Requires a sequence of variables to be lexicographically less than or equal to another sequence of variables; useful in symmetry breaking (avoids permutation of (groups of) variables)
  - value_precede $\rightarrow$ Requires a value to precede another value in a sequence of variables; useful in symmetry breaking (avoids permutation of values)

Specialized Propagation for Global Constraints

Happens with two main approaches:

- constraint decomposition
- dedicated ad-hoc algorithm

Constraint decomposition

A global constraint is decomposed into smaller and simpler constraints, each of which has a known propagation algorithm.
Propagating each of the constraints gives a propagation algorithm for the original global constraint.
This can be a very effective and efficient method for some global constraints.
Decomposing a constraint can be useful to reduce the search space and time search.

- Often GAC on the original constraint is stronger than (G)AC on the constraints in the decomposition.
  Often propagating a constraint via an ad-hoc algorithm is faster than propagating the (many) constraints in the decomposition, thanks to incremental computation.
- A propagation algorithm is often called multiple times, recomputing everything each time. Incremental computation can improve efficiency by caching some partial results at the first call and exploiting the cached data in later invocations.
  - This requires access to more details of the propagations (variables and values pruned)

Part C - Dedicated Propagation Algorithms

They provide effective and efficient propagation.
Advantages:

- GAC maintained in polynomial time
- many more inconsistent values detected (compared to decomposition)
- computation is done incrementally (less cost of propagation)

We need something to exploit (some similarities) between the constraint and for example a theory in order to easily develop a dedicated algorithm.
Sometimes maintaining GAC can be NP-hard.

Global constraints for Generic Purposes

Help us propagate a wide range of constraints.
Useful when we can't decompose or create an algorithm.

- Table (extensional) Constraint can be used to propagate constraint that we don't know how to express but we know the combination of values the variables can take.
  - Used when the constraint have no mathematical or logical formulas to apply.
    - Configuration problems, formal language-based constraint (fsa)
- Regular Constraint utilised with deterministic finite state automaton
  - rostering problems (scheduling shift with rules) and sequencing problems
  - many constraints are instances of regular (among, lex, prece, ecc)

PART 3 - Search

Constraint Solver definition.

BTS - Backtracking Tree Search

- Node $\rightarrow$ variable X$_i$
- Branch $\rightarrow$ Decision on X$_i$
- Variables are instantiated sequentially
- By default **depth-first traversal**

Definition of search

Enumerates all possible variable-value combinations via a **systematic backtracking tree search**.

- Guesses a value for each variable.

Propagation

During search, examines the constraints to remove inconsistent values from the domains of the future (unassigned) variables, via propagation.

- Shrinks the domains of the future variables $\rightarrow$ reduction of the tree search size.
- It happens on the constraints that monitor variables whose domains change.

Without Propagation

Whenever all the variables of a constraint is instantiated, the validity of the constraint is checked. In case of dead-end, the most recently posted branching decision is retracted (chronological backtracking).
Systematic search.

- Eventually finds a solution or proves unsatisfiability.
- Complexity O(d^n), exponential.

Depth-first Search (DFS)

Branching decision

We have a variable and we create branches to give a decision on that variable

- Enumeration (or labelling) with single values from D(X$_i$)
  - d-way branching
    - one branch generated by X$_i$ = v$_j$ for each v$_j$ $\in$ D(X$_i$)
  - 2-way branching
    - 2 branches generated by X$_i$ = v and X$_i$ $\neq$ v for some v $\in$ D(X$_i$)
- Domain partitioning of D(X$_i$)
  - k-way branching
    - One branch is generated by Xi ∈ S$_j$ for each partition S$_j$ of D$_i$
  - 2-way branching
    - 2 branches are generated by X$_i$ ∈ S and X$_i$ ∉ S for some S ⊆ D$_i$

Branching/Search Heuristics

- Guide the search.
  - For a branching decision, need to choose a variable X$_i$ and a (set of) value v$_j$.
  - Which variable next? Which value(s) next?
- Known also as variable and value ordering (vvo) heuristics.
- Static vs dynamic heuristics.
- Problem specific vs generic heuristics.

Static Variable Ordering Heuristics

- A variable is associated with each level.
- Branches are generated in the same order all over the tree.
- Calculated once and for all before search starts, hence cheap to evaluate
  Some examples: lexicographic; top-down, left to right, row by row ...

Dynamic Variable Ordering Heuristics

- At any node, any variable & branch can be considered.
- Decided dynamically during search, hence costly.
- Takes into account the current state of the search tree.

Search Heuristics (Dynamic)

We want to choose variables and values that are likely to yield a solution, this is easy for a feasible problem, but not if we don't know.
We could encounter an infeasible sub-problem so we could get stuck until we go back to the first decision that brought us in the subtree (because we need to explore the whole subtree before backtracking).

Heuristics for Infeasible Problems (Fail first)

- **Fail-first (FF) principle**: Try first where you are most likely to fail.
  - Aims at proving, as soon as possible, that the search is in a subtree with no feasible solutions.
- Trade-off:
  - choose next the variable that is most likely to cause failure;
  - choose next the value that is most likely to be part of a solution (least constrained value).
- Main focus on Variable Ordering Heuristics (VOHs).
  - To backtrack from an infeasible sub-problem, we need to explore all the values in the domain of a variable.

Generic Dynamic VOHs based on FF

- Minimum domain (**dom**)
  - Choose next the variable with **minimum domain size**.
  - Idea: minimize the search tree size (if propagation happens at higher levels, it has a much stronger effect)
- Most constrained (**deg**)
  - Choose next the variable involved in **most number of constraints**.
  - Idea: maximize constraint propagation.
- Combination
  - Minimize **dom / deg**

Weighted Degree Heuristic

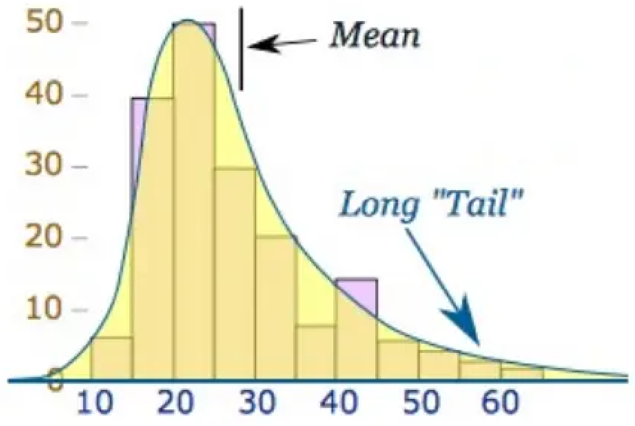
- Constraints are weighted.
  - Initially set to 1.
- During the propagation of a constraint c, its weight w(c) is incremented by 1 if the constraint fails.
- The weighted degree of a variable X_i:

$$w(X_i) = \sum_{c \text{ s.t. } X_i \in X(c)} w(c)$$

- Domain over weighted degree heuristic (domWdeg):
  - Choose the variable X_i with minimum $dom(X_i) / w(X_i)$.

Heavy Tail behaviour

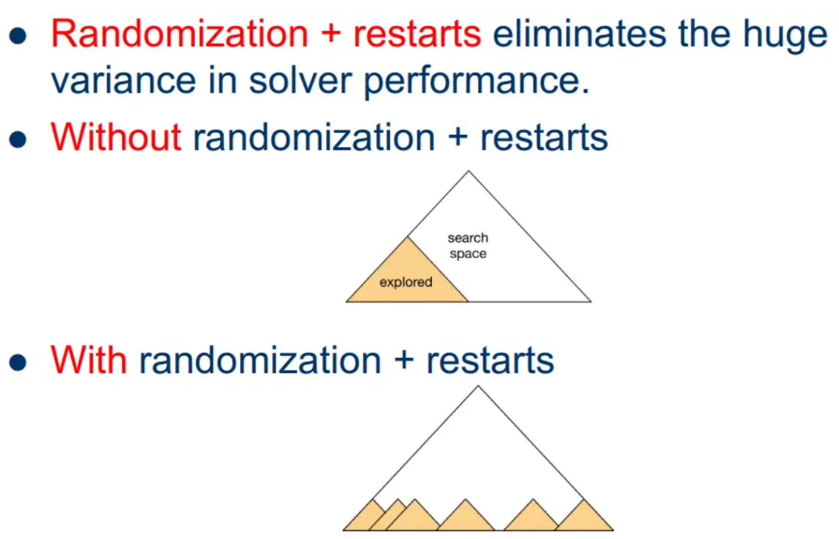
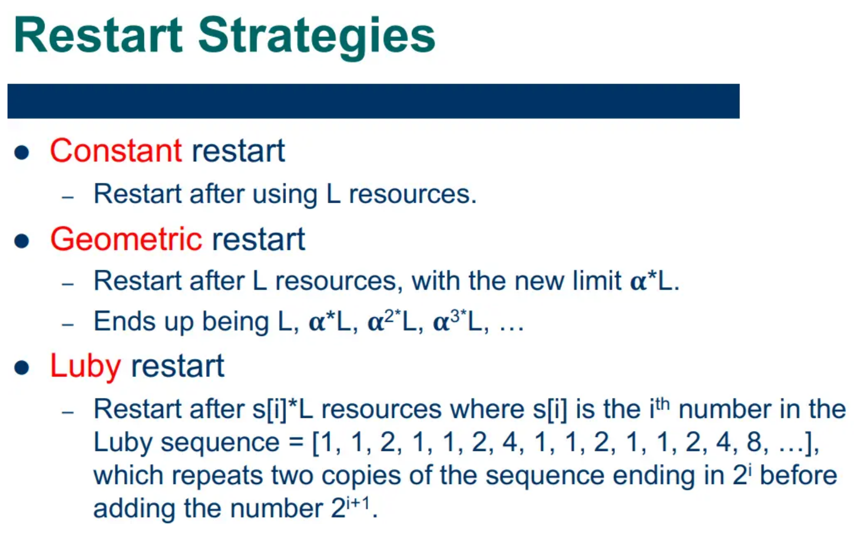
Given a collection of instances of a problem, we often observe some exceptionally hard instances that take exceptionally longer time to solve.



It's not a characteristic of the instance, but some combination of **instance + solver parameters**(search heuristics for example).
If we make a mistake early $\rightarrow$ stuck.
Observation $\rightarrow$ some mistakes seems **random**.

Solutions for HTB - RANDOMIZATION AND RESTARTS

- **Randomization**
  - Add some **randomized parameter in our search (heuristics**):
    - Pick (some) variables/values at random.
    - Break ties randomly.
  - Given the same random seed, the solver will explore the same tree, however it will never explore two identical subproblems in the same way
- **Restarting**
  - Restart the search, after certain amount of resources are consumed.
    - Usually in the form of search steps, such as the number of visited nodes.
  - In the **subsequent runs, search differently**.
  - Introduce **randomization**.
  - Learn from the **accumulated experiences of previous runs**

- 
  - Randomization + restarts eliminates the huge variance in solver performance.
  - Without randomization + restarts
  search space
explored
  - With randomization + restarts
- 
  ## Restart Strategies
  - Constant restart
    - Restart after using L resources.
  - Geometric restart
    - Restart after L resources, with the new limit α*L.
    - Ends up being L, α*L, α^{2}*L, α^{3}*L, …
  - Luby restart
    - Restart after s[i]*L resources where s[i] is the ith number in the Luby sequence = [1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, …], which repeats two copies of the sequence ending in 2^{i} before adding the number 2^{i+1}.

domWdeg heuristic works well with restart.

- Collected fail counts can be carried over to subsequent runs.
- domWdeg combined with random choice of values can be very effective

Problems with DFS

DFS puts tremendous burden on the heuristics early in the search and light burden deep in the search;

- consequently mistakes made near the root of the tree can be costly to discover and undo (it takes a lot of time to finally backtrack to the root of the tree)

Best-First Search (BFS)

Limited Discrepancy Search (LDS)

- A **discrepancy** is any decision in a search tree that does not follow the heuristic (any right branch out of a node) (left is the recommendation of the heuristic).
- **LDS**

- Trusts the heuristic and gives priority to the left branches.
- Iteratively searches the tree by increasing number of discrepancies.
  - On the 0th iteration, explore the leftmost branches.
  - On the nth iteration, explore all left branches except n branches.
- Problems with LDS:
  - all discrepancies are alike, irrespective of their depth.
  - heuristics are less informed $\rightarrow$ more mistakes at the top
  - it's worth exploring discrepancies at the top, before than those at the bottom

Depth-bounded Discrepancy Search

We can consider DDS like LDS but "up to a level".
Discrepancies below a certain depth are prohibited.

- 0th iteration $\rightarrow$ DDS = LDS.
- ith iteration $\rightarrow$ DDS explores those branches on which discrepancies occur at a depth of i or less
- At lesser depths, DDS explores more discrepancies. – At greater depths, DDS follows the heuristic

Constraint Optimization Problems

CSP enhanced with an optimization criterion (minimum cost, shortest distance, fastest route, maximum profit) are called **Constraint Optimization Problems**.
Formally, <X, D, C, f> where f is the formalization of the optimization criterion as an objective variable. Goal = minimize f (or maximize -f).

We want our search to go as fast as possible towards the optimal solution.
Possible ways of solving COPs:

- Enumeration: print all solution and then pick the best ones (not good when you have a lot of solutions)
- Search over the domain of f (we search the smallest f (if minimizing))
- Branch and Bound

Searching over D(f)

Destructive lower bound

- Iterate over the values v ∊ D(f), starting from min(D(f)).
- At each iteration, post the constraint f ≤ v and solve the CSP.
- The first feasible solution is guaranteed to be optimal.

- Why destructive? → Intermediate computation results are discarded.

Destructive upper bound

- Iterate over (some of) the values v ∊ D(f), starting from max(D(f)).
- At each iteration, post the constraint f ≤ v and solve the CSP.
- For the next iteration, set v = f -1.
- When the problem is infeasible, the last solution is proven optimal.

Lower vs Upper

- **Destructive lower bound**
    - CON: not an any time algorithm → until it finds the optimal solution we never have a solution
    - CON: small steps → one value at the time
    - PRO: tighter constraints (only on f) → more propagation (stronger propagation)
    - PRO: provides lower bounds
- **Destructive upper bound**
    - PRO: anytime algorithm
    - PRO: larger steps → we can skip values
    - CON: less propagation because we have lots of possibilities for the values
    - CON: no lower bounds

Binary search

- Binary search over D(f)
- Main idea:
    - keep both a (feasible) upper bound *ub* and an (infeasible) lower bound *lb*;
    - solve by posting *lb < f < (lb + ub)/2*
        - if feasible, update *ub*; if unfeasible, update *lb*
        - stop if a solution with *f = lb+1* is found (because *lb* is unfeasible)
- A compromise between destructive lower and upper bounding since it combines the advantages of both Lower and Upper:
    - Anytime algorithm.
    - Lower bounds.
    - Tight(ish) constraints on *f* → good propagation.
    - Large steps.

Main disadvantage: almost all information is discarded between each attempt, so there is a lot of repeated work. (also uses several search trees)

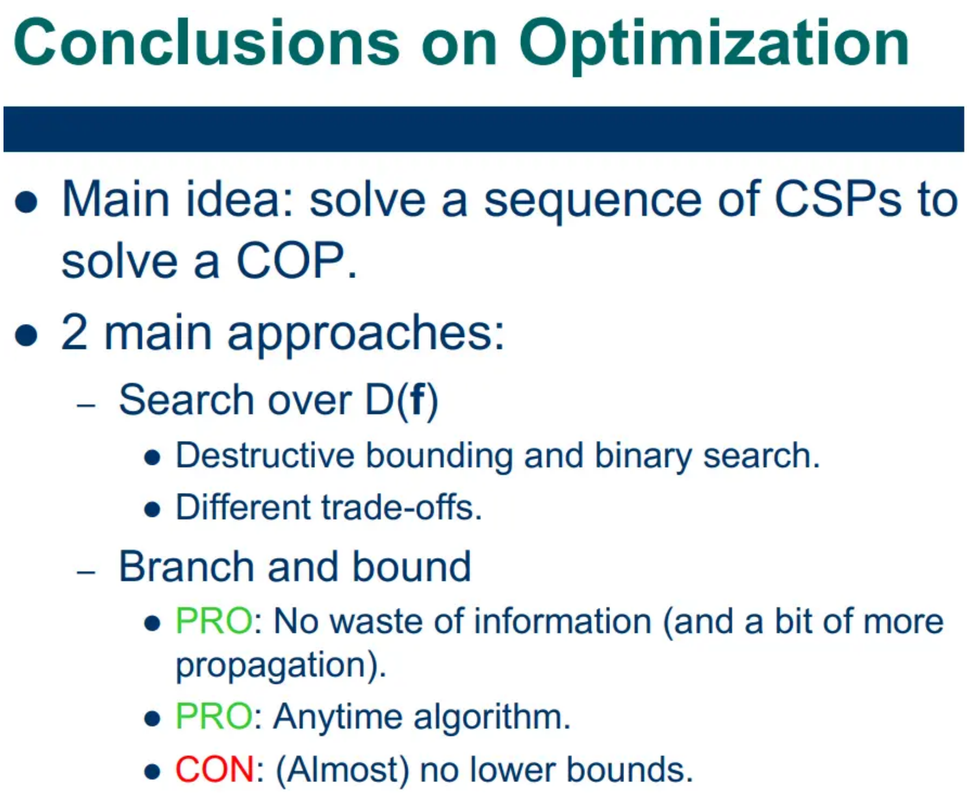
Branch & Bound Algorithm

Solves a sequence of CSPs via a single search tree and incorporates bounding in the search.

- Each time a feasible solution is found, posts a new **bounding constraint** which ensures that a future solution must be better than it. (solution $\rightarrow$ backtrack + new bound)
- Backtracks and looks for a new solution with the additional bounding constraint, using the same search tree.
- Repeats until infeasible: the last solution found is optimal.

Optimality proof refers to the fact that the whole search tree need to be explored to be certain that a solution is optimal.
We can measure:

- How many backtracks / failures it took to find the optimal solution
- How many backtracks / failures it took to prove that it was an optimal solution



Conclusions on Optimization

- Main idea: solve a sequence of CSPs to solve a COP.
- 2 main approaches:
  - Search over D(**f**)
    - Destructive bounding and binary search.
    - Different trade-offs.
  - Branch and bound
    - PRO: No waste of information (and a bit of more propagation).
    - PRO: Anytime algorithm.
    - CON: (Almost) no lower bounds.

PART 4 - Constraint-Based Scheduling

Scheduling = ordering resource-requiring tasks over time, a very important and tough problem class.

Resource Constrained Project Scheduling Problem - RCPSP

Given:

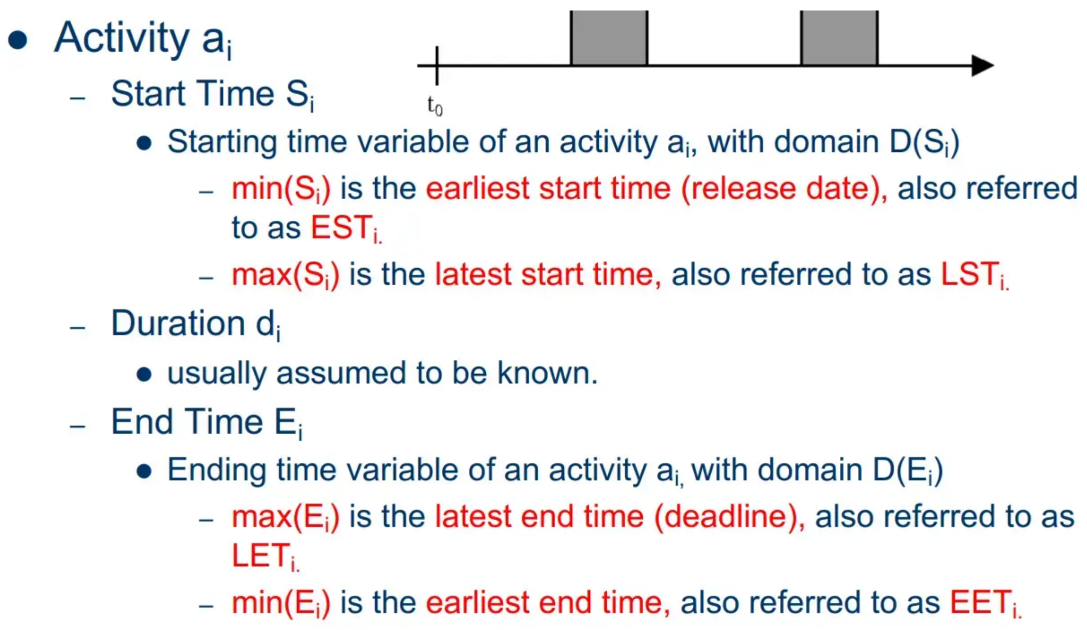
- a set of resources with fixed capacities,
- a set of tasks with given durations and resource requirements,
- a set of temporal constraints between tasks,
- and a performance metric
  RCPSP consists of deciding **when to execute** each task so as to **optimize the performance** metric, subject to **temporal and resource constraints**.

Constraint-based Model

- Tasks $\rightarrow$ (activity) variables.
- Resource constraints $\rightarrow$
  - unary/disjunctive/sequential resource;
  - cumulative/parallel resource.
- Temporal constraints.
- Performance metric $\rightarrow$ schedule dependent cost function (this is our objective variable (min or max this variable))

Activity variables

Variable = activity
Those are the main variables of the scheduling problem.
We need to decide the operation positions in the timeline of the schedule.



- Activity a_i
  - Start Time S_i
    - Starting time variable of an activity a_i, with domain $D(S_i)$
      - $\min(S_i)$ is the earliest start time (release date), also referred to as EST_i.
      - $\max(S_i)$ is the latest start time, also referred to as LST_i.
  - Duration d_i
    - usually assumed to be known.
  - End Time E_i
    - Ending time variable of an activity a_i, with domain $D(E_i)$
      - $\max(E_i)$ is the latest end time (deadline), also referred to as LET_i.
      - $\min(E_i)$ is the earliest end time, also referred to as EET_i.

Types of activities:

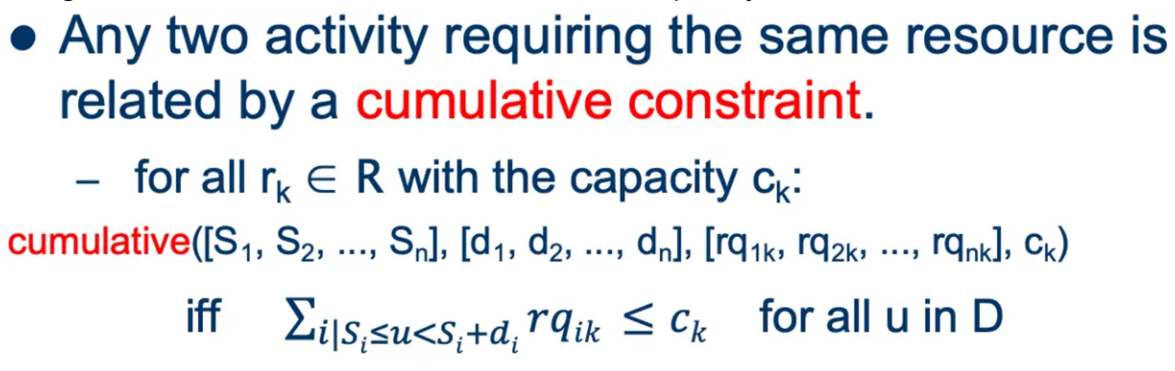
- Preemptive → can be interrupted at any time (S + d <= E)
- Non-preemptive → cannot be interrupted at any time (S + d = E)

Resources

A resource corresponds to an asset available to execute the operations.

Cumulative / Parallel Resource

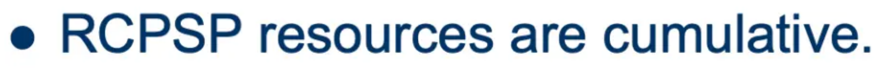
Can execute multiple activities in parallel → activities can overlap in time.
Each activity must express the required amount of resources needed, so that the total usage of the resource doesn't exceed the capacity.



- Any two activity requiring the same resource is related by a cumulative constraint.
  - for all $r_k \in R$ with the capacity c_k:

cumulative($[S_1, S_2, ..., S_n], [d_1, d_2, ..., d_n], [rq_{1k}, rq_{2k}, ..., rq_{nk}], c_k$)

$$\text{iff} \quad \sum_{i|S_i \leq u < S_i + d_i} rq_{ik} \leq c_k \quad \text{for all u in D}$$



- RCPSP resources are cumulative.

The **cumulative constraint** is posted once for each resource. (start, duration, resource requirements of running activities, max capacity).

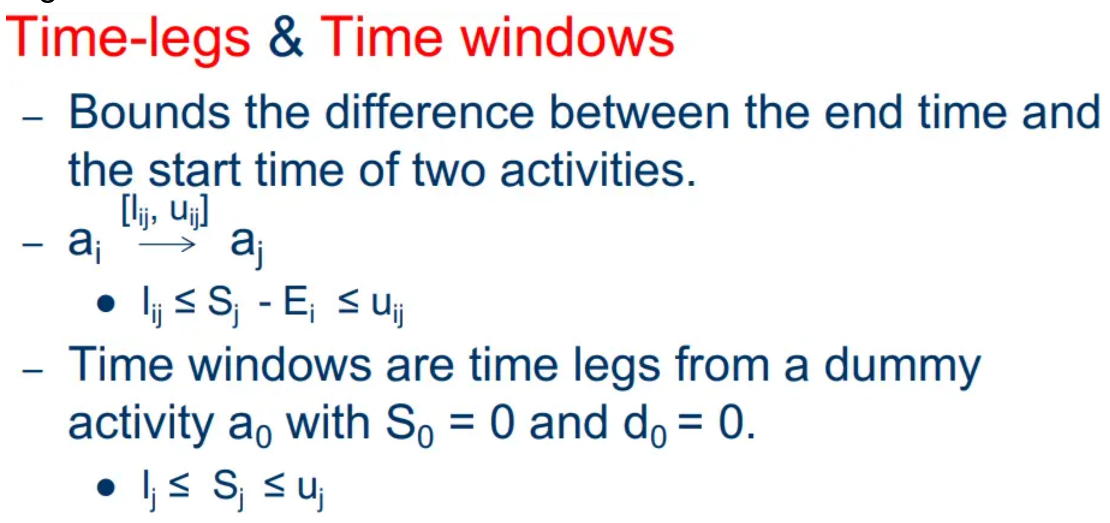
Unary / Disjunctive / Sequential Resource

Can execute one activity at a time → activities cannot overlap in time independently of the resource capacity. (therefore the capacity variables isn't needed anymore)
Any two activity is related by a disjunctive (noOverlap) constraint.

disjunctive($[S_1, S_2, \ldots, S_n], [d_1, d_2, \ldots, d_n]$) iff

$$S_i + d_i \leq S_j \vee S_j + d_j \leq S_i \quad \text{for all } 1 \leq i < j \leq n$$

Temporal Constraints

- Precedence constraints
    - Forces one activity to end before another starts.
    - ai → aj (ai come before aj)
        - Ei ≤ Sj
- Time legs & time windows
    - 

      Time-legs & Time windows

      – Bounds the difference between the end time and the start time of two activities.

      – $a_i \xrightarrow{[l_{ij}, u_{ij}]} a_j$

      • $l_{ij} \leq S_j - E_i \leq u_{ij}$

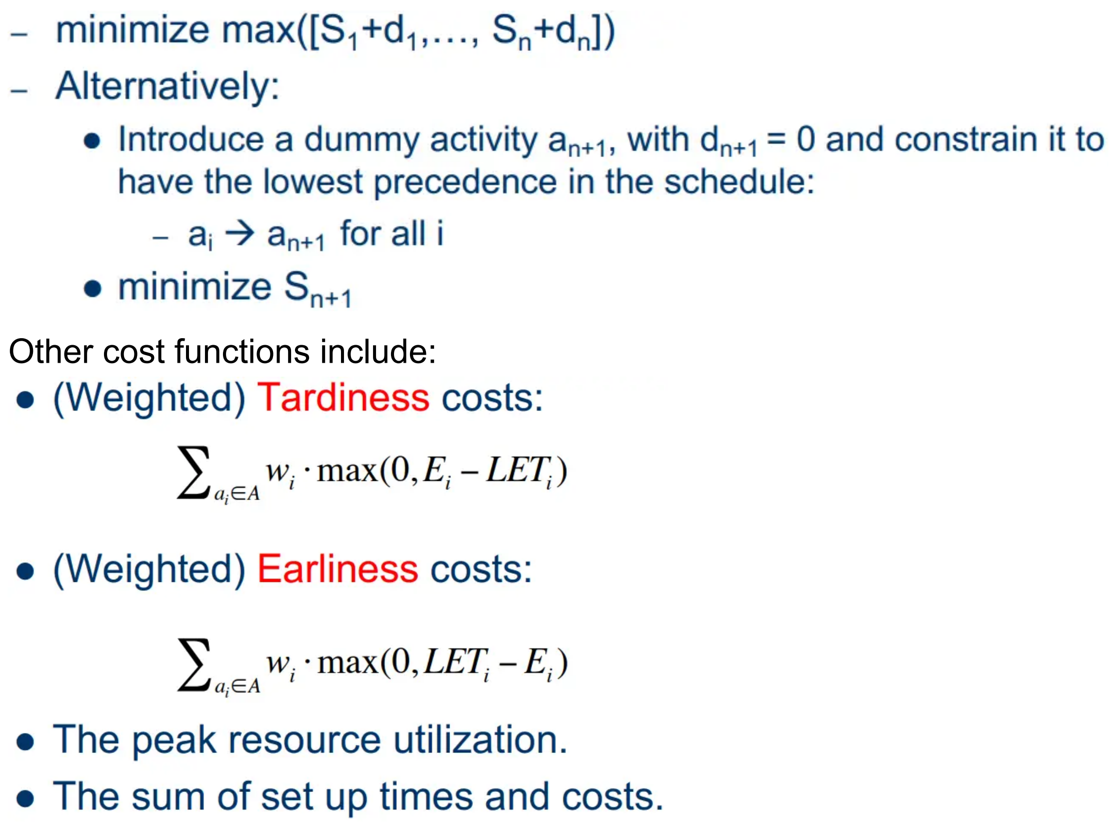
      – Time windows are time legs from a dummy activity a_0 with $S_0 = 0$ and $d_0 = 0$.

      • $l_j \leq S_j \leq u_j$
    - says that some time must pass between one activity and another
- Sequence-dependent set up times
    - Defined for unary resources.
    - If ai and aj are scheduled in a sequence, then they must obey a separation constraint.
        - Ei ≤ Sj → Ei + dij ≤ Sj
    - similar to the previous but relative to a single resource

Cost Function

A common cost function: **makespan**

- Completion time of the last activity.
- Optimum makespan is the minimum makespan.
- RCPSP and job shop scheduling cost functions are makespan.
  Minimum makespan can be modelled in different ways



- minimize $\max([S_1+d_1,\ldots, S_n+d_n])$
- Alternatively:
  - Introduce a dummy activity a_{n+1}, with $d_{n+1} = 0$ and constrain it to have the lowest precedence in the schedule:
    - $a_i \rightarrow a_{n+1}$ for all i
  - minimize S_{n+1}

Other cost functions include:

- (Weighted) Tardiness costs:

$$\sum_{a_i \in A} w_i \cdot \max(0, E_i - LET_i)$$

- (Weighted) Earliness costs:

$$\sum_{a_i \in A} w_i \cdot \max(0, LET_i - E_i)$$

- The peak resource utilization.
- The sum of set up times and costs.

Search Heuristics

Since typical instances have large domains, choosing which variable to pick next and which value to assign is difficult.

Value selection for RCPSP

- The objective is to minimize the makespan.
- Increasing an Si value (with other Sj untouched) cannot improve the makespan, so we should select ESTi (we can achieve this by using indomain_min)
- This is true not only for the RCPSP.
  - Many scheduling problems have so-called regular cost metrics.
  - Regular = increasing a single Si cannot improve the cost.

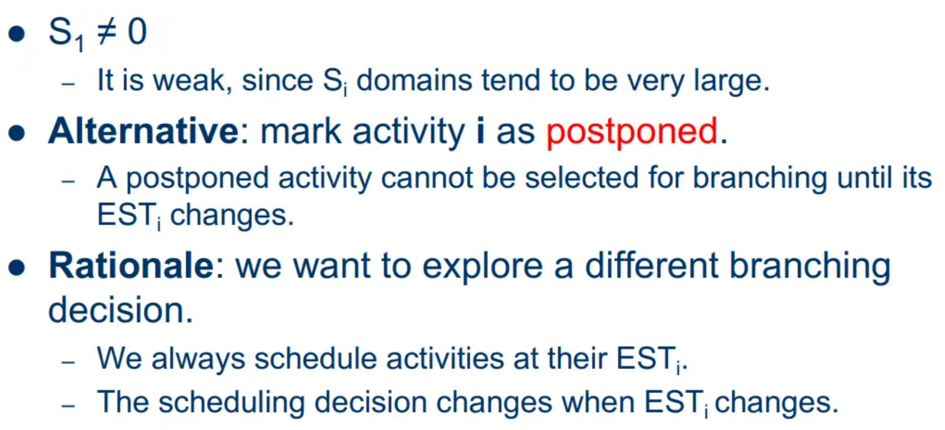
Variable selection for RCPSP

A greedy and simple approach (PRB) works well in many cases but is not guaranteed to give an optimal solution. (we can achieve this by using smallest)

Proving optimality

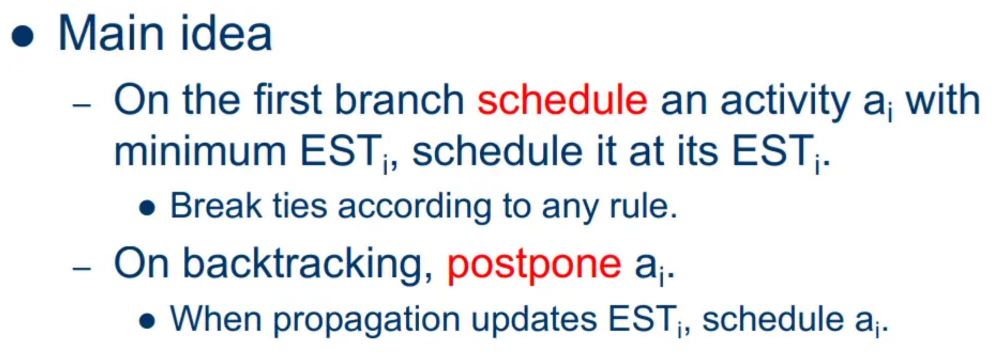
Once a makespan is found, we need to go back to the root node after posting new constraint to prove optimality or not.

Schedule or postpone / SetTimes Search Strategy



- $S_1 \neq 0$
  - It is weak, since S_i domains tend to be very large.
- **Alternative**: mark activity **i** as postponed.
  - A postponed activity cannot be selected for branching until its EST_i changes.
- **Rationale**: we want to explore a different branching decision.
  - We always schedule activities at their EST_i.
  - The scheduling decision changes when EST_i changes.

Basically we wait for the domain of the variable (activity) to change and then schedule it.
This is important to find an alternative branch that could lead to the optimal solution.



- Main idea
  - On the first branch schedule an activity a_i with minimum EST_i, schedule it at its EST_i.
    - Break ties according to any rule.
  - On backtracking, postpone a_i.
    - When propagation updates EST_i, schedule a_i.

- Very effective search strategy:
  - based on PRB (greedy heuristic) scheduling $\rightarrow$ starts with good solutions
  - effective branching choices
- Incomplete search strategy:
  - we don't partition the search space
  - we could find but not prove it's an optimal solution because of the delays
- Cost function is regular (increasing a single Si cannot improve the cost)
- No point in not scheduling activities at their ESTi, unless delayed by previous ones

PART 5 - Heuristic Search

Combinatorial Optimization

Complete methods

- Guarantee to find for every finite size instance an (optimal) solution in bounded time.
- E.g., constructive tree search in CP and branch & bound, branch & cut in ILP, other tree search methods.
- Might need exponential computation time.

Approximate methods

- Cannot guarantee optimality, nor termination in case of infeasibility.
- Obtain good-quality solutions in a significantly reduced amount of time.

1. Constructive heuristics
2. Local search
3. Metaheuristics

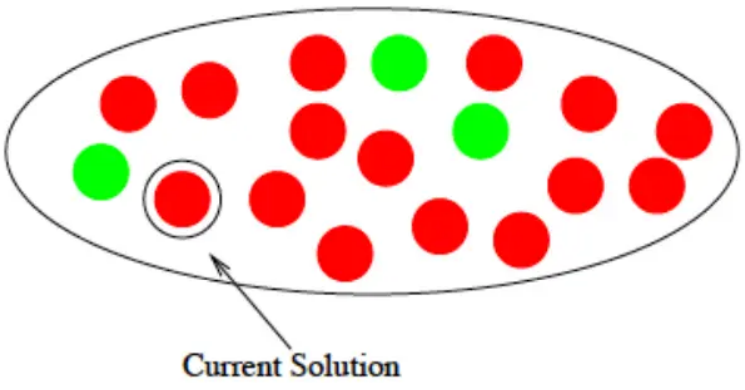
Constructive Heuristics

- Fastest approximation methods.
- No propagation. (?)
- Generate solutions from scratch by repeatedly extending the current partial assignments until a solution is found or stopping criteria are satisfied.
- Use problem-specific knowledge (heuristic) to construct a solution.
- A well-know class is greedy heuristics. (e.g. Priority rule-based scheduling)
    - Make the locally optimal choice at each stage!
- Simple, quick and often give good approximations. Solutions maybe far from optimal! Widely used together with other methods. (E.g., for initialization for local search and metaheuristics.)

Local Search

- Often returns solutions of superior quality when compared to constructive heuristics.

- Starts from some **initial solution** and iteratively tries to **replace the current solution** with a better one in an appropriately defined **neighbourhood** by applying **small (local) modifications**.
- Can also start from an unfeasible assignment of all the variables.
- Optimality isn't guaranteed, but it's good to replace an initial solution with a better one in a short amount of time.



Given an initial solution, a neighbourhood is defined and a better solution is searched inside the neighbourhood, comparing the objective function. (they are all solutions)

- So we need to refine our definition of combinatorial optimization:
  - Given <X, D, C, f> , find a feasible solution s *∊ S such that f(s*) ≤ f(s) for all s ∊ S.
  - basically we need to find the solution with the better objective function.

Neighbourhood Structure

- A function N : S → 2^S that assigns to every s ∊ S a set of neighbours N(s) ⊆ S. N(s) is called the **neighbourhood** of s.
- Often implicitly defined by specifying the modifications that must be applied to s in order to generate its neighbours N(s).
- The application of such an operator to s that produces a neighbour is commonly called a **move**.

Local Minimum

- A **locally minimal solution (or local minimum)** with respect to a neighbourhood structure N is a solution s' such that f(s') ≤ f(s) for all s ∊ N(s').
- It's the best solution in the neighbourhood.
- It's not guaranteed that it's globally optimal.

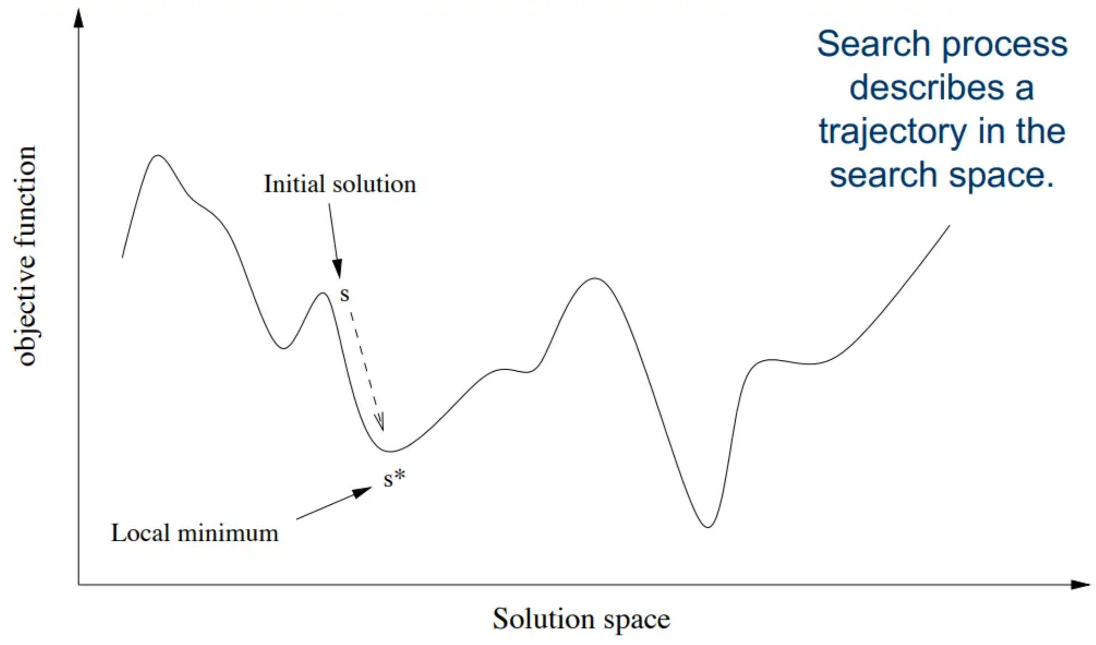
A Simple Local Search Algorithm

Algorithm 1 Iterative improvement local search

```
1: $s \leftarrow$ GenerateInitialSolution()
2: while $\exists\, s' \in \mathcal{N}(s)$ such that $f(s') < f(s)$ do
3:     $s \leftarrow$ ChooseImprovingNeighbor($\mathcal{N}(s)$)
4: end while
```

- Initial solution can be generated randomly or heuristically.
- A move is only performed if the resulting solution is better than the current solution (also called hill climbing).
- ChooseImrovingNeighbor: first improvement, best improvement.
- Stops as soon as it reaches a local minimum.
  - Performance highly depends on the neighbourhood structure.

A Pictorial View of Iterative Improvement



(solution space = all the solution visited)
As we can see from the graph, we have to find a fay to move out of the local minimum to go to the global minimum to find the optimal solution.

Metaheuristics

- **High level strategies to increase performance**.
  - Use of a priori knowledge (heuristics).
  - Exploitation of search history – adaptation.
  - General strategies to balance intensification and diversification.

    - Randomness and probabilistic choices.
- Aim: **not to get trapped in local minima and search for better and better local minima**.
- Encompass and combine:
    - constructive methods (e.g. random, heuristic, adaptive, etc);
    - local search-based (trajectory) methods;
    - population-based methods.

Intensification & Diversification

- Driving forces of metaheuristic search.
- **Intensification**: exploitation of the accumulated search experience (e.g., by concentrating the search in a confined, small search space area).
- **Diversification**: exploration “in the large” of the search space.
- (I&D) Contrary and complementary:
    - need to quickly identify regions in the search space with high quality solutions, without wasting too much time in the regions already explored or not containing high quality solutions;
    - their dynamical balance determines the effectiveness of the metaheuristics.

LS-based Methods

- Similarly to LS:
    - A single solution is used at each algorithm iteration.
    - Search process describes a trajectory in the search space.
- Differently from LS:
    - Add a diversification component to iterative improvement for escaping from local minima.
        - Allow worsening moves.
        - Change neighbourhood structure during search.
        - Change the objective function during search.
    - Termination criteria: maximum CPU time, maximum number of iterations (without improvement), a solution of sufficient quality, etc.
        - This is required because otherwise, escaping the local minimum, the computation could go on for a very long time.

Simulated Annealing (SA)

- **Accept worsening (up-hill) moves**, i.e., the search moves toward a solution with a worse objective function value.

- **Intuition**: **climb the hill** and go downward in another direction in the same search landscape.
- The probability of doing such a move is decreased during search, **favouring intensification to diversification**.

Variable Neighbourhood Search (VNS)

- **Change neighbourhood structure during search**.
- **Intuition**: different neighbourhoods generate different search landscapes.
- A neighbourhood N$_i$ is substituted by a neighbourhood N$_j$ as soon as local minima is reached.

Tabu Search (TS)

- **Change neighbourhood structure during search by exploiting the search history**.
- It changes the neighbourhood dynamically without changing its structure.
- **Tabu list**: keeps track of recently visited solutions/moves and forbids them.
    - it changes the neighbourhood by effectively "reducing" its size since only remaining unforbidden local minimum are available.
- Storing solutions is often inefficient:
    - better **store the moves**;
        - but that could eliminate good yet not visited solutions.
- **Aspiration criteria**: accept a forbidden move towards a solution better than the current one.
- Tabu list size determines the size of exploration, **favouring diversification to intensification as the size increases**.
    - Dynamic tabu size is of interest!
    - Increase in case of repetitions (thus diversification is needed).
    - Decrease in case of no improvements (thus intensification should be boosted).

Guided Local Search (GLS)

- **Change the objective function during search** so as to “fill in” local minima.
- **Intuition**: modify the search landscape with the aim of making the current local optimum less desirable. (by making it more expensive)
- Penalize solution features that occur frequently in visited solutions.
    - E.g., certain arcs in a tour in TSP.
- New objective function takes into account these penalties.

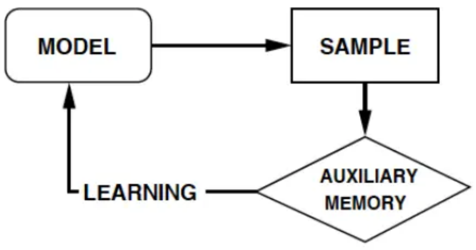
Population-based Methods

- At each algorithm iteration, a set – population – of solutions are used. (instead of starting from a single one like LS-based methods)
- Search process is the evolution of a set of points or a probability distribution in the search space.
- Majority are inspired by natural processes, such as natural evolution and social insects foraging behaviour.
- **Basic principle**: learn correlations between good solution components.

  Candidate solutions are generated using a parametrized probabilistic model.

  The model is updated using the previously seen solutions in such a way that the search will concentrate in the regions containing high quality solutions.

  E.g., Evolutionary Computation (EC), Ant Colony Optimization (ACO).



Ant Colony Optimization

- EXPLANATION: Inspired by the foraging behaviour of ants which enables them to find the shortest path between the nest and a food source. While walking ants deposit a substance called pheromone on the ground. When they decide about a direction to go, they choose with higher probability paths that are marked by stronger pheromone concentrations. This behaviour is the basis for a cooperative interaction which leads to the emergence of shortest paths
- Pheromone trails are simulated by a parametrized probabilistic model $\rightarrow$ **pheromone model**.
  - Consists of a set of parameters whose values are called pheromone values.
  - Pheromone values act as the memory to keep track of the search process so as to intensify search around the best solution components.
    - E.g., a pheromone value T(Xi ,vi) for all Xi ∊ X and vi ∊ D(Xi) can represent the desirability of assigning vi to Xi .
  - Bounding pheromone values between T$_{min}$ and T$_{max}$ can balance intensification and diversification.
  - Initially pheromone values are all set to τmax.

Artificial ants employ constructive heuristics for probabilistically constructing solutions using the pheromone values.

- Iteratively choose a variable X_i according to the heuristic and a value $v_i \in D(X_i)$ according to the probability:

Heuristic factor

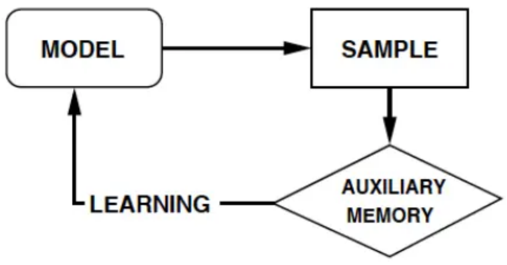
$$p(x_i, v_i) = \frac{[\tau(x_i, v_i)]^\alpha \cdot \boxed{[\eta(x_i, v_i)]^\beta}}{\sum_{v_j \in D(x_i)} [\tau(x_i, v_j)]^\alpha \cdot [\eta(x_i, v_j)]^\beta}$$

- Parameters α and β help to balance the influence of pheromone and heuristic factors.

- Pheromone values are updated.
- All pheromone values are decreased by an evaporation factor $\rightarrow$ allows ants to progressively forget older solutions and to emphasize to more recent ones (**diversification**).
- Pheromone values associated to the assignments taking part of good solutions are increased proportionally to the quality of the solutions. $\rightarrow$ the goal is to increase the probability of selecting such assignments in the future constructions (**intensification**).

ACO Algorithm

1. Initialize pheromone values.
2. Ants construct solutions using heuristics and a pheromone model.
3. The constructed solutions are used to update the pheromone values in a way to bias the future sampling towards high quality solutions.



Termination criteria: maximum CPU time, maximum number of iterations (without improvement), a solution of sufficient quality, etc.

LS or Population Metaheuristics?

- LS-based methods are preferable when:
    - neighbourhood structures create a correlated search graph;
    - inventing moves is easy;
    - computational cost of moves is low.
- Population-based methods are preferable when:

        - solutions can be encoded as composition of good building blocks;
        - computational cost of moves in LS is high;
        - it is difficult to design effective neighbourhood structures;
        - coarse grained exploration (e.g., huge search spaces) is preferable
- Complementary Strengths
    - LS-based methods
        - A promising area in the search space is searched in a more structured way.
        - Danger of being close to good solutions but “missing” them is low.
        - Intensification ability!
    - Population-based methods
        - New solutions are obtained by recombining earlier solutions.
        - Search process performs a guided sampling of the search space, usually resulting in a coarse grained exploration.
        - Diversification ability!

Hybrid Metaheuristics

- Goal: Exploit complementary strengths of the individual strategies (synergy).
- The use of LS-based methods inside population-based methods.
- Population-based iterated local search.
- Multilevel techniques.

Combinatorial Optimization

Complete methods.
Approximate methods.

Complementary strengths

- CP (a complete method)
    - Focus on constraints and feasibility.
    - Easy modelling and control of search.
    - Poor in optimization with loose bounds on the objective function.
- Metaheuristics (an approximation method)
    - Effective in finding good-quality solutions quickly.
    - Constraints are handled inefficiently, i.e. often by penalizing infeasible assignments in the objective function.

Metaheuristics + Complete Methods

Main approaches:

- Metaheuristics are applied before complete methods providing a valuable input, or vice versa.
- A complete method method applies a metaheuristic in order to improve a solution.
- **Metaheuristics use a complete method to efficiently explore the neighbourhood.**
    - **Large Neighbourhood Search**
    - **ACO + CP**
- Metaheuristic concepts can also be used to obtain incomplete but efficient tree exploration strategies.

Main issues in LS:

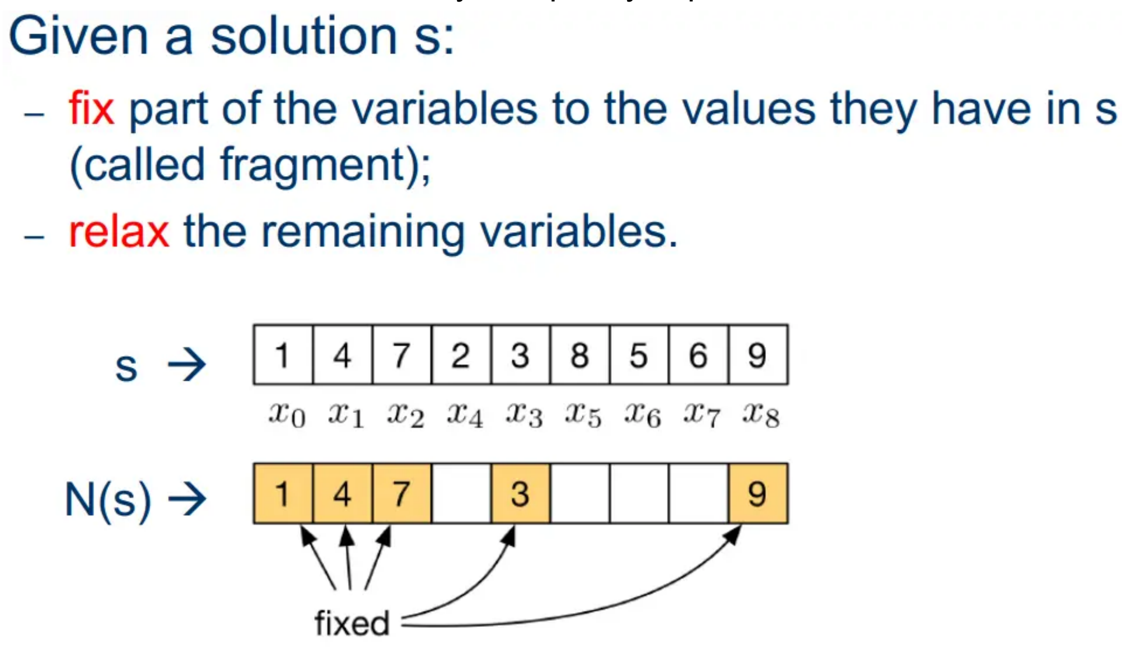
- defining an appropriate neighbourhood structure. (a generic one wouldn't rely on the problem)
- choosing a way to examine the neighbourhood of a solution.
- choosing the size of the neighbourhood:
    - Small neighbourhoods
        - PRO: it is fast to find an improving neighbour (if any).
        - CONS: the average quality of the local minima is low.
    - Large neighbourhoods
        - PRO: the average quality of the local minima is high.
        - CONS: finding an improving neighbour might be difficult.

Large Neighbourhood Search

Use a generic (applicable to any problem) and large (contains a lot of solutions) neighbourhood, and explore it with a complete method like CP (very suitable)!
Core idea:

- view the exploration of a neighbourhood as the solution of a sub-problem;

- use tree search to exhaustively but quickly explore it.



Given a solution s:

- fix part of the variables to the values they have in s (called fragment);
- relax the remaining variables.

Advantages over LS and CP

- Efficient neighbourhood exploration.
    - Thanks to propagation and advanced search techniques of CP.
- LNS is easier to develop than LS.
    - Easy and problem-independent neighbourhood definition.
    - No need to ensure that complicated constraints are satisfied.
- More scalable than using only CP on the problem.
    - Subproblems are typically much smaller.
    - We can control the subproblem size.
    - The fixed-variables reduce the domain sizes.
    - Propagation works best when domains are small

ACO + CP

- Constructive techniques with complementary strengths:
    - ACO is characterized by a learning capability;
    - CP is efficient in handling constraints. I
- Typical ACO + CP approaches:
    - Use CP as solution construction for artificial ants.
    - Use ACO for variable and value ordering heuristics in CP.