Knowledge Representation and Reasoning: Constraint Satisfaction and Local Search

Chapters 4.1, 6.4

Constraint Satisfaction Problems

◇ Binary constraint network $\gamma = \langle V, D, C \rangle$

— V a finite set of variables $v_1, \ldots, v_n$

— D a set of [finite] sets $D_{v_1}, \ldots, D_{v_n}$

— C a set of binary relations $\{C_{u,v} \mid u, v \in V, u \neq v\}$

$C_{u,v} \subseteq D_u \times D_v$

Outline of the lecture

- ♢ Recall [optimal] constraint solving
- ♢ Local Search in general
- ♢ Large Neighbourhood Search in particular
- ♢ Constraint modelling
- ♢ Summary

Recall

- ◊ CSP may be solved for satisfiability (any solution will do)
- ◊ May instead require optimality (best solution)
- ◊ Objective function (i.e. cost) measures badness of solutions
- ◊ FD solvers commonly use Depth-First Branch and Bound (DFBB)
  — Use a lower bound L on the objective and an upper bound B
  — Backtrack whenever $L \geq B$
  — Revise B whenever a solution is found
- ◊ DFBB fits well with backtracking CSP solution methods
- ◊ Gives a sequence of monotonically improving solutions

There is a problem

- Absolute optimality is often too hard to compute
- Need methods that scale up better and yield (probably) good solutions
- Enter **Local Search**
  (actually a generic name for many search techniques)
- Widely used in industrial applications

Local Search

- ♢ The general idea: search in the space of total assignments
- ♢ An alternative to backtracking: not so rigidly defined
  — Usually involves randomness at some point
- ♢ Example (for satisfiability):
  - Start with a random assignment of values to all variables
  - (Of course, this doesn't satisfy all constraints)
  - Repeatedly:
    - Choose a variable (random choice is good)
    - Revise its value to minimise its constraint violations
  - Stop when all constraints are satisfied
- ♢ Optimisation version might remember the best assignment so far, and stop when the objective function hasn't improved for a while.

Iterative improvement algorithms

◊ Local search of this kind iteratively improves total assignments

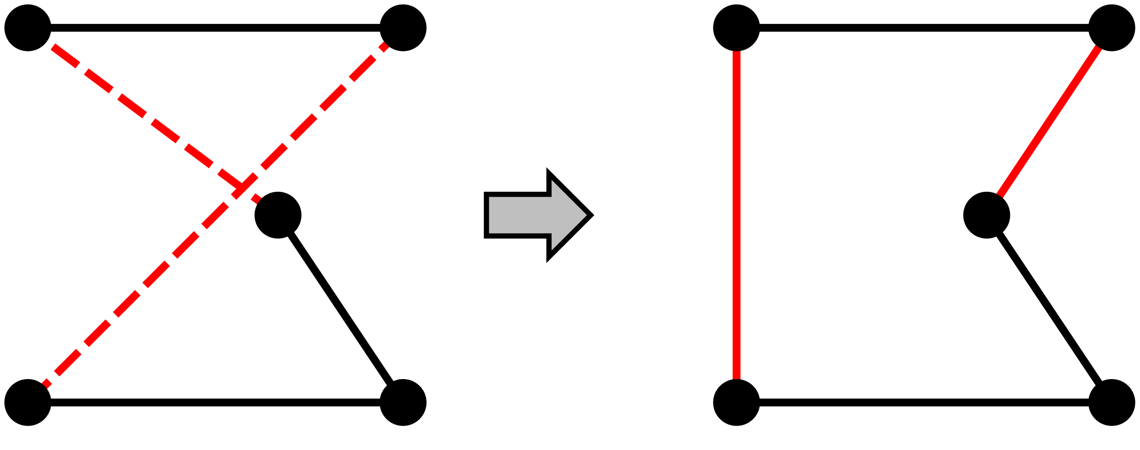
◊ General idea: keep a single "current" state, try to improve it
— perform local moves in the neighbourhood of the current state
— no guarantee of completeness (may fail to find any solution)
— no guarantee of optimality
— no possibility of showing unsatisfiability

◊ However, method scales up better than complete search in many domains

◊ Small memory requirements, suitable for online as well as offline search

Example: Travelling Salesperson Problem

Start with any complete tour, perform pairwise exchanges to reduce tour duration

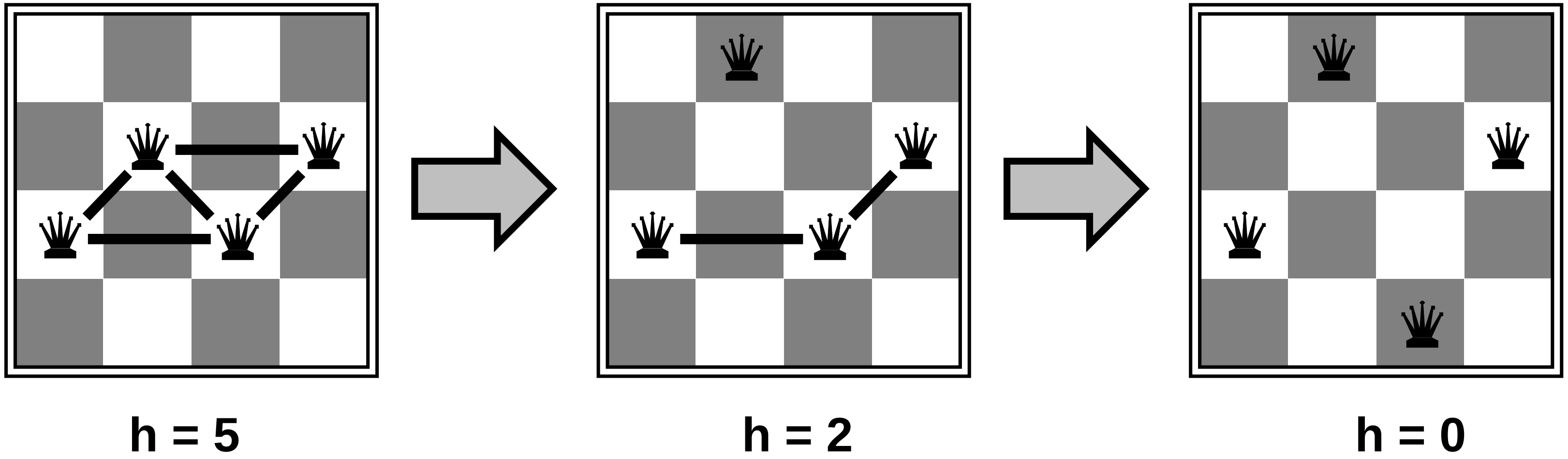


Variants of this approach get within 1% of optimal very quickly with thousands of cities

Example: n-queens

Put n queens on an $n \times n$ board with no two queens on the same row, column, or diagonal

Move a queen in its column to reduce number of conflicts



Variants of this approach almost always solve n-queens problems almost instantaneously for very large n, e.g., $n = 1$ **million**

Hill-climbing (or gradient ascent/descent)

Moves to the best neighbour

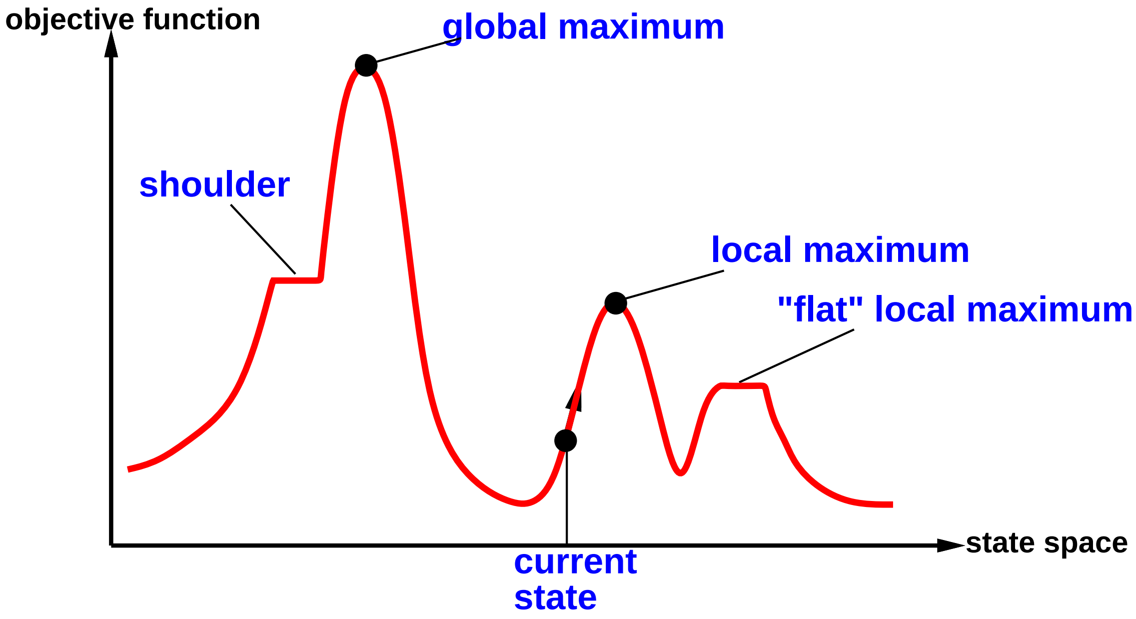
Climbing a mountain in the fog without a map: just go up!

function HILL-CLIMBING(*problem*) **returns** a state that is a local maximum
  inputs: *problem*, a problem
  local variables: *current*, a node
                neighbour, a node

  current ← MAKE-NODE(INITIAL-STATE[*problem*])
  loop do
    neighbour ← a highest-valued successor of *current*
    if VALUE[neighbour] ≤ VALUE[current] **then return** STATE[*current*]
    current ← *neighbour*
  end

Hill-climbing contd.

Useful to consider state space landscape



Random-restart hill climbing overcomes local maxima—trivially complete

Random sideways moves 😊 escape from shoulders ☹ loop on flat maxima

8 queens: simple HC has 14% success rate, HC + sideway moves 94%
3 million queens: HC + random restart + sideway moves needs < 1 mn

Simulated annealing

Idea: escape local maxima by allowing some "bad" moves
but gradually decrease their badness and frequency

```
function SIMULATED-ANNEALING( problem, schedule) returns a solution state
    inputs: problem, a problem
            schedule, a mapping from time to "temperature"
    local variables: current, a node
                     neighbour, a node
                     T, a "temperature" controlling prob. of downward steps

    current ← MAKE-NODE(INITIAL-STATE[problem])
    for t ← 1 to ∞ do
        T ← schedule[t]
        if T = 0 then return current
        neighbour ← a randomly selected successor of current
        ΔE ← VALUE[neighbour] – VALUE[current]
        if ΔE > 0 then current ← neighbour
        else current ← neighbour only with probability e^(ΔE/T)
```

Population-based methods

♢ Local beam search: maintain a population of k states; add some neighbours at each step; delete the worst ones to keep the population stable

— As usual, many variants exist

♢ Genetic (evolutionary) algorithms: define "crossover" between pairs of states in the population (compare genetics); offspring have some features of each parent; cull according to a "fitness" measure

— Much research over several decades

— Again, many variations on the general method

Population-based search and simulated annealing will not be covered in this course: COMP4660 / COMP8420 covers evolutionary algorithms in some detail

A hybrid: Large Neighbourhood Search

- Search in the space of solutions rather than states
- Given a current solution:
  - Destroy part of it by forgetting the values of some variables
  - See the problem of assigning values to those variables as a CSP
  - Solve [optimally] using a complete search method such as DFBB
- Alternates destroy and repair phases
  — Repair phase searches a neighbourhood of the current solution
  — Neighbourhood size remains tolerable, even if the problem is huge
- The old solution is still in the neighbourhood, so there is always a next solution available, given enough search
- May search for the optimal solution in the neighbourhood, or for an improvement on the previous solution, or just for any solution in the neighbourhood.

LNS: Example (1)

- ♢ Energy-aware scheduling of meetings in a large building or buildings

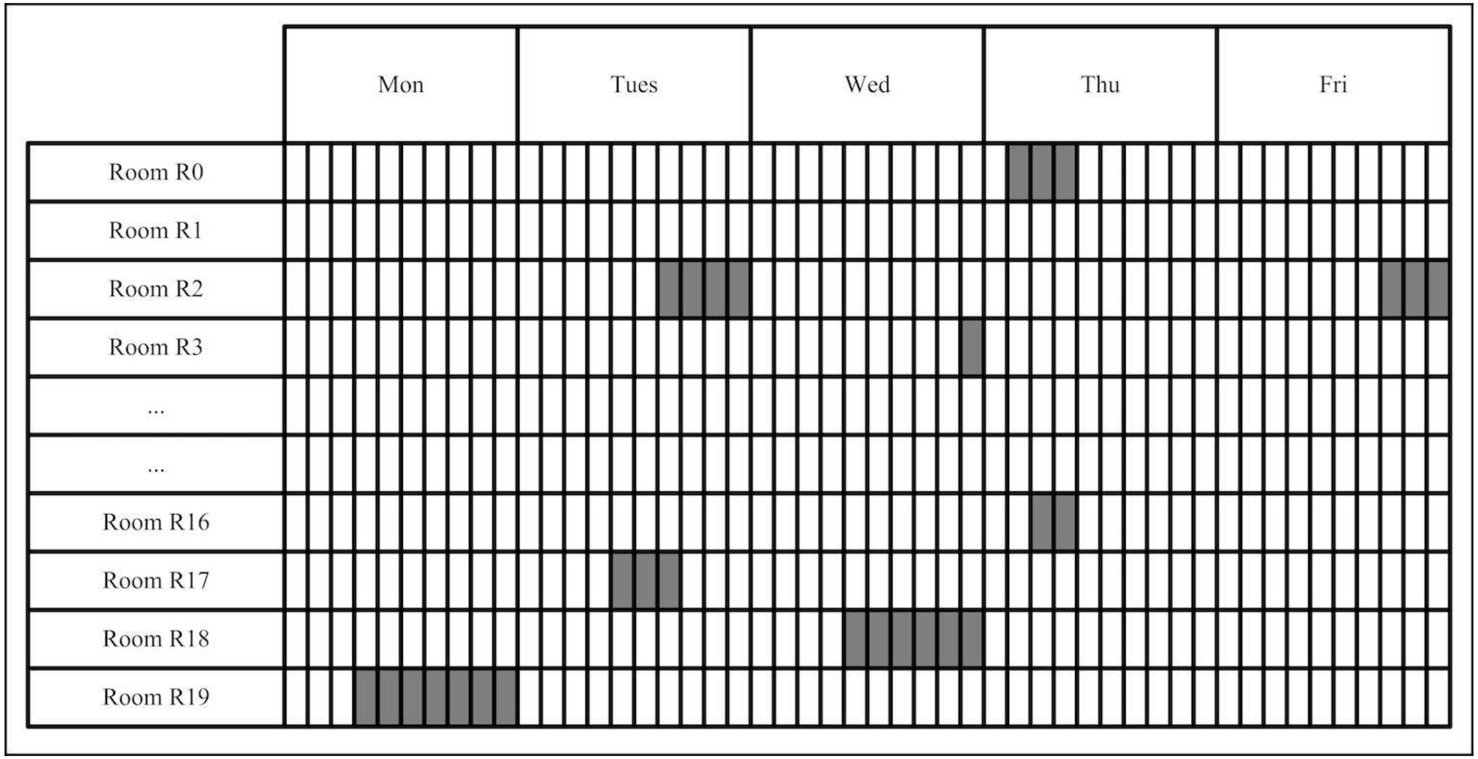
- ♢ Each meeting has:
  a set of participants
  a set of allowable locations (rooms)
  a set of allowable times
  a duration (30 minutes, 1 hour, 2 hours, ...)

- ♢ Constraints:
  — All classes are scheduled exactly once
  — All particular constraints on meetings are met
  — No two meetings in the same room at the same time
  — No conflict of meetings for participants

- ♢ Objective function: energy consumption (separately calculated)

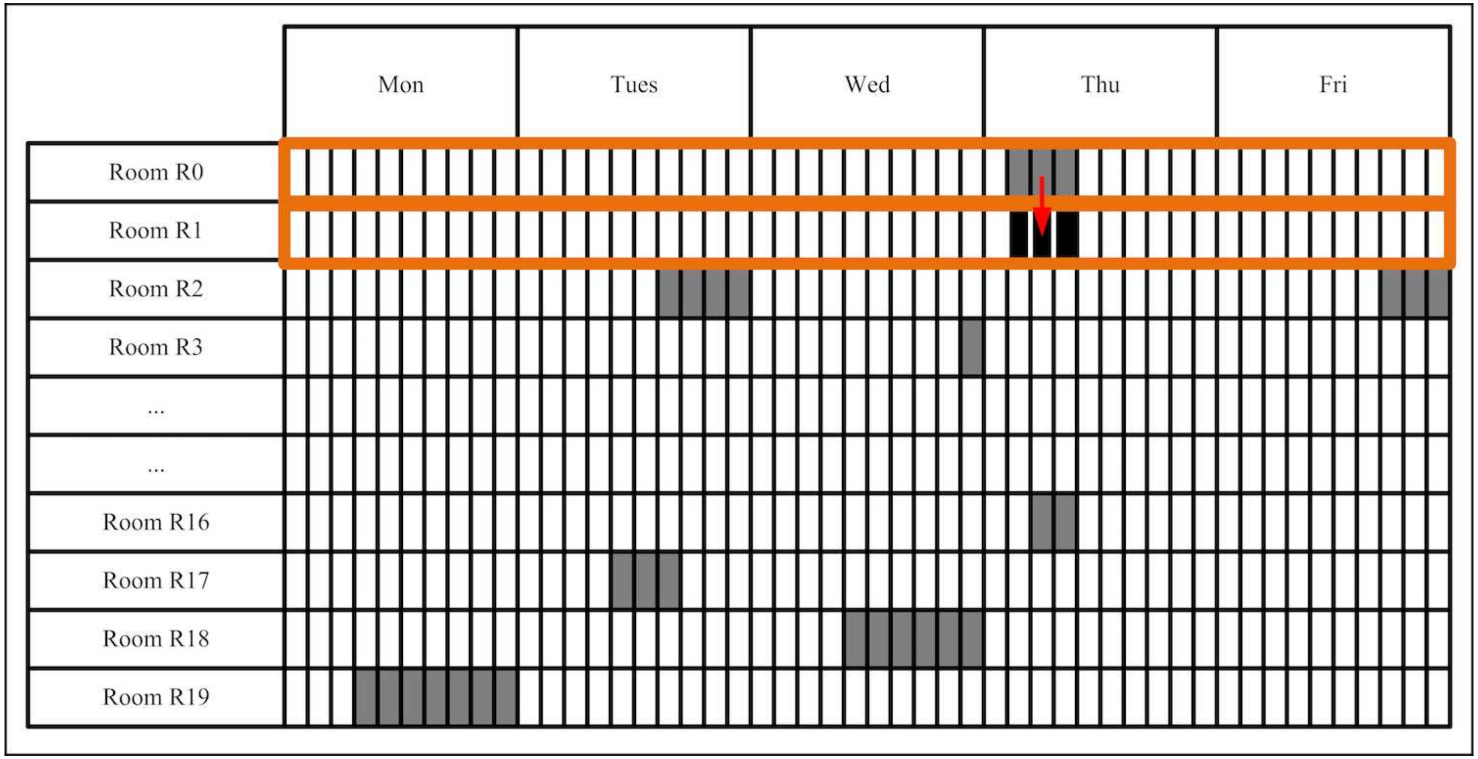
- ♢ With 200 meetings in 20 rooms, too hard for DFBB

LNS: Example (2)



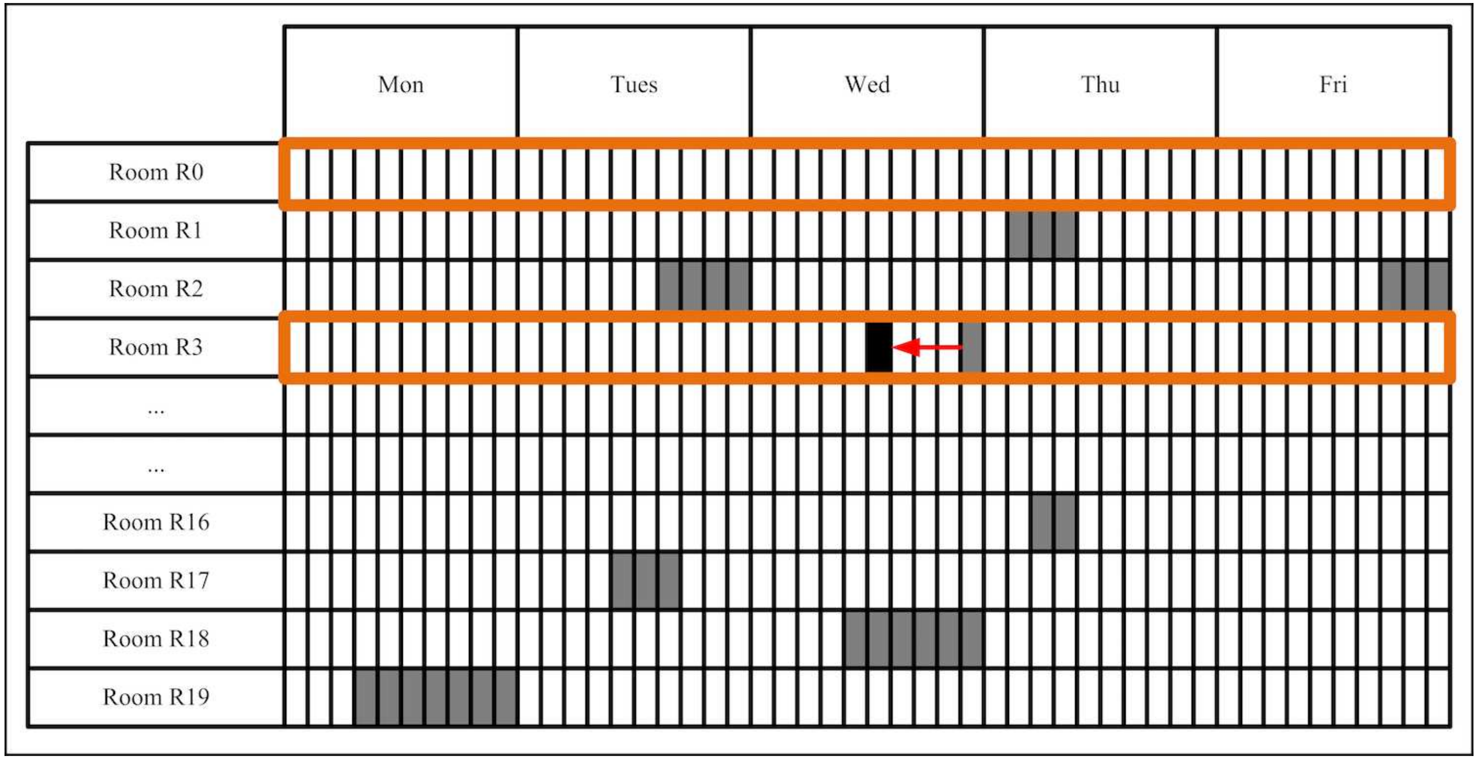
◊ Initial schedule generated by the MIP solver

LNS: Example (2)



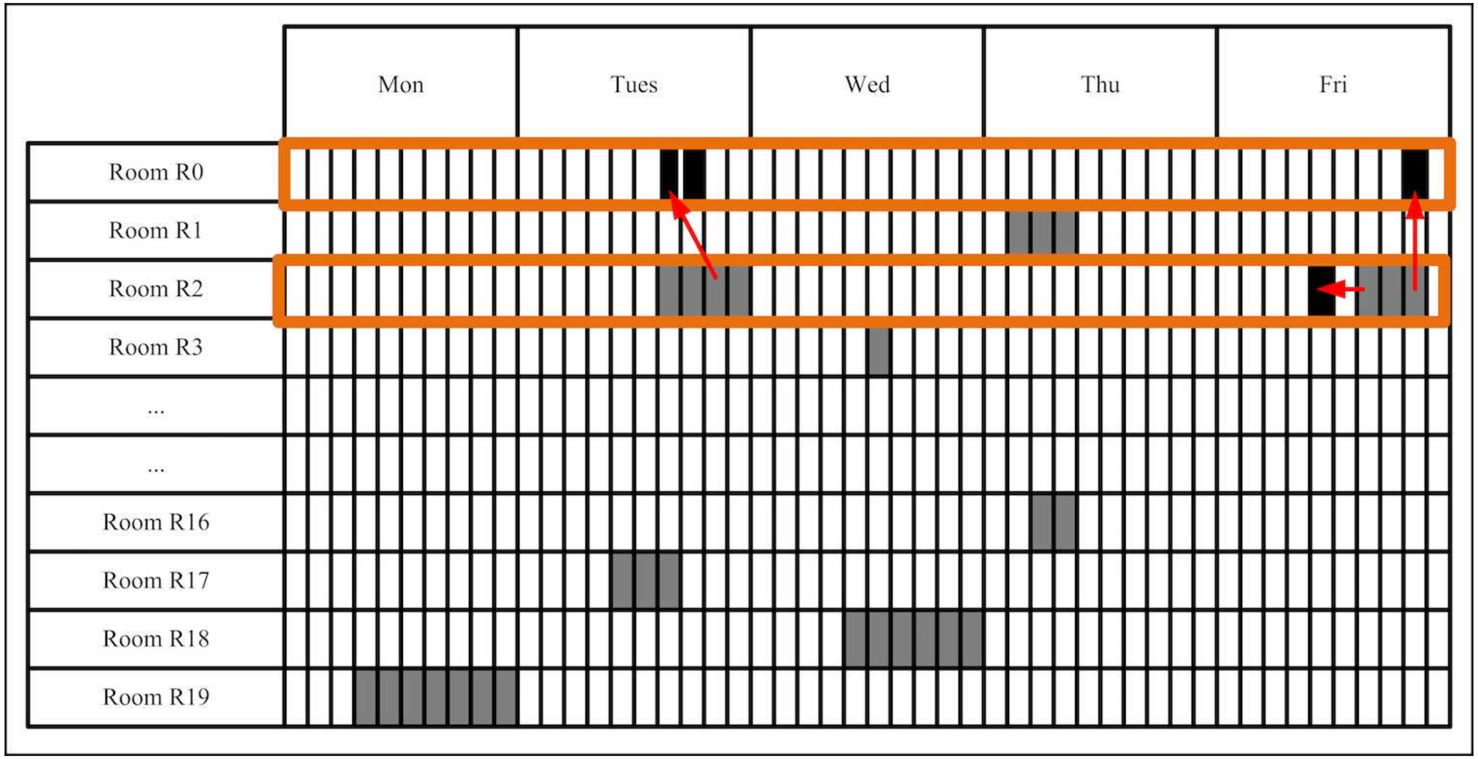
- Schedule for rooms R0 and R1 destroyed and re-optimised

LNS: Example (2)



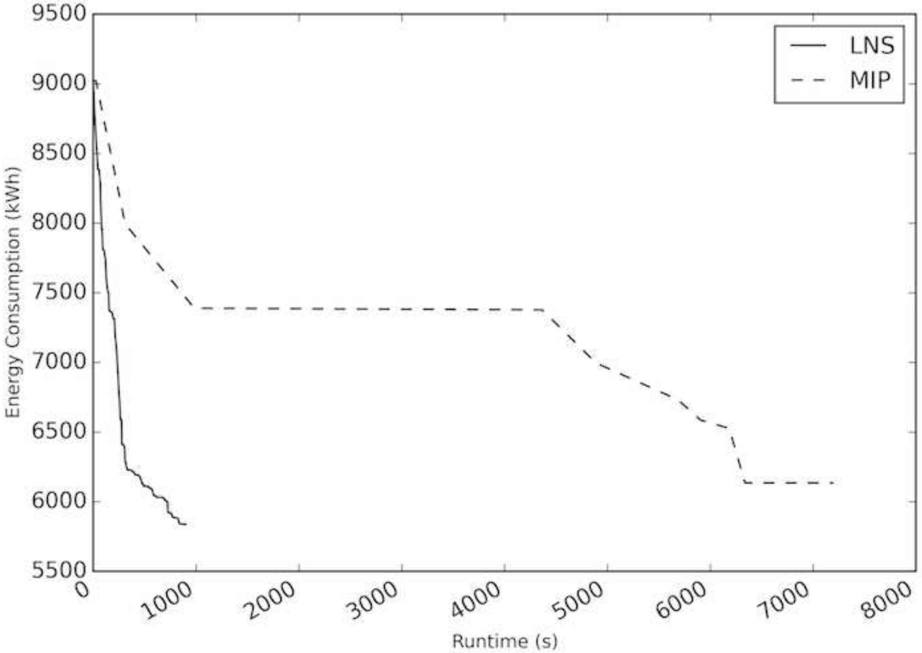
◇ Schedule for rooms R0 and R3 destroyed and re-optimised

LNS: Example (2)



◇ Schedule for rooms R0 and R2 destroyed and re-optimised

LNS: Example (3)



◇ MIP (complete) solver found fairly good solutions in 2 hours

◇ LNS using the same solver found better ones in 15 minutes

LNS: Notes

◇ The initial solution must come from somewhere.

◇ Performance is sensitive to the choice of what to destroy

◇ We may choose to abstract from the current solution
— use only some decision variables, for a partial description
— designed so the rest can be recovered by easy search
— destroy part of the abstract solution
— gives the complete search freedom to optimise minor aspects

◇ Local optima are still a problem, as with all local search
— random restart is commonly used to escape

◇ There is always a tradeoff between neighbourhood size and speed
— Large neighbourhoods increase the chance of improvement
— but they may create hard problems for the complete search

Summary

◇ Local search explores the space of total assignments

◇ Usually step to a neighbour, looking for improvement
— but sometimes jump further, or even re-start entirely

◇ No guarantees (incomplete, sub-optimal, ...)
But in many cases scales up better than systematic search

◇ Many varieties
— hill-climbing, random walks, simulated annealing, population methods

◇ Large neighbourhood search the best of both worlds (?)
— DFBB or similar to solve small problems quickly and completely
— local search for scaling up
— tradeoff between the two is fundamental
— requires experiment to tune parameters and define neighbourhoods