Optimization with Metaheuristics

CSE480 & CSE591- Week 6 - Tabu Search & Example Problem Models & Iterated Local Search

Asst. Prof. Gizem Süngü Terci

November 7, 2025

What is Tabu Search?

- **Tabu Search (TS)** is a *meta-heuristic* that guides a local search to find near-optimal solutions.
- Key idea: avoid revisiting recently explored, poor moves/solutions by maintaining a **tabu list** (short-term memory).
- It will be used to guide the movement from one solution to the next one avoiding cycling.
- Deterministically allows non-improving ("tabu") moves to escape local optima when beneficial.
- Heuristic intuition: **Hill climbing + short-term memory = Tabu Search**.
- Origins: **Fred Glover (1986)**; popularity surged in the 1990s.

CLOSED, OPEN, and TABU (Solutions & Moves)

- Treat **nodes** as candidate **solutions/states**; neighbors come from applying **moves**.
- Classical search: **CLOSED** (visited solutions) and **OPEN** (frontier); newly generated solutions found in **CLOSED** are removed from **OPEN**.
- **Tabu Search:** recently used **moves or solution attributes** are marked **tabu** (short-term memory) to avoid cycles.
- **Aspiration:** a tabu move/solution can be accepted if it improves the **best-so-far**.
- Practically, tabu lists track **move attributes** (e.g., swaps, insertions) rather than full solutions for efficiency.

Terminology

- **Tabu list:** Tracks recent moves/solutions (short-term memory).
- **Tabu tenure:** Number of iterations an item remains tabu.
- **Frequency memory (diversification):** Moves chosen too frequently without success become less likely later to push the search into new regions.

Three main strategies

1. **Neighborhood search:** Generate candidate solutions via allowable moves; pick the best next.
2. **Aspiration criteria:** Override tabu if a candidate is *better than the best-so-far*.
3. **Memory mechanism:** Maintain and update tabu list (and possibly frequency metrics) to balance *exploration* vs *exploitation*.

Main concepts — I

1. TS uses **memory** to record search information.
2. Generate a neighborhood from the current solution; accept the best candidate (even when not strictly improving).
3. This flexibility can induce **cycles**.
4. Previously visited solutions *could* be revisited.
5. To avoid cycling, use a **tabu list** to forbid recently visited moves/solutions.

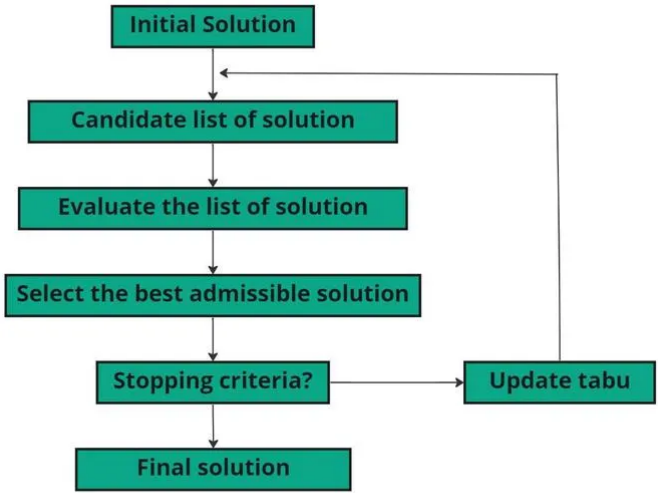
Main concepts — II

6. **Tabu list length** controls the search.
7. Larger tenure $\Rightarrow$ wider exploration, more forbiddances.
8. Smaller tenure $\Rightarrow$ focused search.
9. Tabu list updated each iteration (FIFO behavior).
10. **Tabu tenure** = number of iterations a move is forbidden.
11. **Aspiration:** allow tabu moves that improve the *global best*.

Basic Tabu Search algorithm

- Initialize N (current/best), `CLOSED` (tabu list), parameters: `TabuListSize`, `TT` (tenure).
- Loop until termination:
  1. Generate children `CHILD` $\leftarrow$ `MOVEGEN`(N).
  2. Filter out tabu; append aspirational candidates.
  3. Select next N; scan neighbors to update N by best fitness.
  4. Update best solution if improved.
  5. Append N to `CLOSED`; truncate if exceeds size.

Design Flow


Initial Solution
Candidate list of solution
Evaluate the list of solution
Select the best admissible solution
Stopping criteria?
Update tabu
Final solution

Termination criteria

- Maximum iterations reached.
- Target solution quality reached.
- Time limit reached.
- No improvement for a given window.
- Combined criteria (e.g., time limit + quality threshold).

Knapsack Problem – Definition

- We are given a set of n items.
- Each item i has:
  - a value (profit) v_i
  - a weight (cost) w_i
- There is one knapsack with capacity W.
- **Goal:** Select a subset of items to put into the knapsack so that
  - the total value is **maximized**, and
  - the total weight does **not exceed** the capacity W.
- This version is called the **0–1 Knapsack Problem** because each item is either taken (1) or not taken (0).

Parameters and Decision Variables

Given:

- n: number of items
- v_i: value (profit) of item i
- w_i: weight (cost) of item i
- W: knapsack capacity

Decision variable:

$$x_i = \begin{cases} 1, & \text{if item } i \text{ is selected} \\ 0, & \text{otherwise} \end{cases} \qquad i = 1, \ldots, n$$

Mathematical Model

Objective function: maximize total value

$$\max \ Z = \sum_{i=1}^{n} v_i x_i$$

Capacity constraint: total weight cannot exceed W

$$\sum_{i=1}^{n} w_i x_i \leq W$$

Integrality (binary) constraint:

$$x_i \in \{0, 1\} \quad \text{for } i = 1, \ldots, n$$

Individual Representation

- In the **0–1 Knapsack Problem**, each solution indicates which items are selected.
- We represent a candidate solution as a **binary vector**:

$$X = [x_1, x_2, \ldots, x_n]$$

where

$$x_i = \begin{cases} 1, & \text{if item } i \text{ is included in the knapsack} \\ 0, & \text{otherwise} \end{cases}$$

- Each binary vector corresponds to a different subset of items.
- **Example:** For $n = 5$, a candidate $X = [1, 0, 1, 1, 0]$ means items 1, 3, and 4 are selected.

Fitness Function for the Knapsack Problem

- The **fitness (cost) function** evaluates how good a candidate solution $X = [x_1, x_2, \ldots, x_n]$ is.
- It depends on:
  - total profit of selected items, and
  - whether the total weight satisfies the capacity constraint.

If the solution is feasible:

$$f(X) = \sum_{i=1}^{n} v_i x_i \quad \text{subject to} \quad \sum_{i=1}^{n} w_i x_i \leq W$$

If the solution is infeasible (overweight):

$$f(X) = \sum_{i=1}^{n} v_i x_i - \lambda \left(\sum_{i=1}^{n} w_i x_i - W \right)$$

where λ is a penalty factor to discourage infeasible solutions.

Tabu Search for 0–1 Knapsack

1. Generate an initial solution x and set $x^{best} = x$.
2. Initialize Tabu List $T = \emptyset$.
3. **for** $k = 1$ to MaxIter **do**
   - 3.1 Generate neighborhood $N(x)$ by 1-bit flips.
   - 3.2 Select $x' \in N(x)$ with the best objective value such that it is not tabu, **or** it satisfies aspiration.
   - 3.3 Set $x = x'$.
   - 3.4 If $f(x) > f(x^{best})$ then $x^{best} = x$.
   - 3.5 Add the performed move to the tabu list with a tenure.
   - 3.6 Decrease the tenure of existing tabu entries.
4. **end for**
5. Return x^{best}.

Example Setting for Tabu Search

We illustrate Tabu Search on a small 0–1 Knapsack instance.

- Knapsack capacity: $W = 5$
- Items:
  - Item 1: $w_1 = 2$, $v_1 = 6$
  - Item 2: $w_2 = 3$, $v_2 = 10$
  - Item 3: $w_3 = 4$, $v_3 = 12$
  - Item 4: $w_4 = 1$, $v_4 = 7$
- Individual representation: $x = (x_1, x_2, x_3, x_4)$, $x_i \in \{0, 1\}$
- Neighborhood (search) operator: **1-bit flip** (add/remove one item)
- We keep a **tabu list** of recent flips.

Initial Solution

- Start with solution: $x = (1, 0, 0, 1)$
- Weight $= w_1 + w_4 = 2 + 1 = 3 \leq 5$
- Value $= v_1 + v_4 = 6 + 7 = 13$
- Set:
$$x^{best} = (1, 0, 0, 1), \quad f(x^{best}) = 13$$
- Tabu list is initially empty.
- Tabu tenure (example): 2 iterations.

Iteration 1 of Tabu Search

Current solution: $x = (1, 0, 0, 1)$, value $= 13$.
Generate neighbors by flipping one bit:

- Flip $x_1 \to (0, 0, 0, 1)$: weight $= 1$, value $= 7$ (feasible)
- Flip $x_2 \to (1, 1, 0, 1)$: weight $= 6 > 5$ (infeasible)
- Flip $x_3 \to (1, 0, 1, 1)$: weight $= 7 > 5$ (infeasible)
- Flip $x_4 \to (1, 0, 0, 0)$: weight $= 2$, value $= 6$ (feasible)

Best feasible neighbor: $(0, 0, 0, 1)$ with value 7.

- Move to $x = (0, 0, 0, 1)$
- Keep $x^{best} = (1, 0, 0, 1)$ (since $13 > 7$)
- Add move "flip item 1" to tabu list (tenure $= 2$)

Iteration 2 of Tabu Search

Current solution: $x = (0, 0, 0, 1)$, value $= 7$.
Generate neighbors:

- Flip $x_1 \to (1, 0, 0, 1)$: value $= 13$ **(tabu move)**
- Flip $x_2 \to (0, 1, 0, 1)$: weight $= 4$, value $= 17$
- Flip $x_3 \to (0, 0, 1, 1)$: weight $= 5$, value $= 19$
- Flip $x_4 \to (0, 0, 0, 0)$: value $= 0$

Select best admissible neighbor: $(0, 0, 1, 1)$ with value 19.

- Update current solution: $x = (0, 0, 1, 1)$
- Since $19 > 13$, update $x^{best} = (0, 0, 1, 1)$
- Add move "flip item 3" to tabu list

Before next iteration

After selecting $x = (0, 0, 1, 1)$:

- New current solution: $x = (0, 0, 1, 1)$
- Objective value: $f(x) = v_3 + v_4 = 12 + 7 = 19$
- This is **better than** the previous best $f(x^{best}) = 13$
- Therefore, update:
$$x^{best} \leftarrow (0, 0, 1, 1), \quad f(x^{best}) = 19$$
- Add the performed move "flip item 3" to the tabu list.

Tabu list status:

- flip item 1: still tabu (1 more iteration)
- flip item 3: tabu (just added, full tenure)

Remark: Even if a move is tabu, it can be accepted later by *aspiration* if it produces a solution better than x^{best}.

Neighborhood Operators in Local Search

- **Definition:** A *neighborhood (search) operator* defines how to generate **neighboring solutions** from a current solution. The set of all neighbors is called the **neighborhood** $N(x)$.
- **Purpose:**
  - To explore nearby regions of the search space.
  - To find improved solutions iteratively.
  - To balance **exploration** (diversity) and **exploitation** (improvement).
- **Common Types:**
  - **Bit-flip:** change one variable (e.g., toggle a 0/1 in knapsack).
  - **Swap:** exchange positions of two elements (e.g., in scheduling or TSP).
  - **Insert/Remove:** add or remove one element from a set.
  - **Shift or Move:** relocate an element to a new position.
- **Neighborhood Size:** Larger neighborhoods allow deeper exploration but increase computational cost.

Bin Packing Problem (BPP) – Definition

- The **Bin Packing Problem (BPP)** aims to pack a set of items into the minimum number of fixed-capacity bins.
- Each item i has a size s_i ($0 < s_i \leq 1$).
- Each bin j has a capacity C (usually normalized to 1).
- **Goal:** minimize the number of bins used so that the sum of item sizes in each bin does not exceed its capacity.

Real-life examples:

- Packing goods into containers or trucks.
- Memory or task allocation in computer systems.
- Resource usage optimization in production planning.

Mathematical Model of the BPP

Parameters:

- n: number of items and m: maximum number of bins (upper bound)
- s_i: size of item i
- C: capacity of each bin

Decision variables:

$$x_{ij} = \begin{cases} 1, & \text{if item } i \text{ is packed in bin } j, \\ 0, & \text{otherwise} \end{cases} \qquad y_j = \begin{cases} 1, & \text{if bin } j \text{ is used,} \\ 0, & \text{otherwise} \end{cases}$$

Fitness (Cost) function:

$$\min \sum_{j=1}^{m} y_j$$

Constraints

$$\text{s.t.} \quad \sum_{j=1}^{m} x_{ij} = 1, \quad i = 1, \ldots, n \text{ and } \sum_{i=1}^{n} s_i x_{ij} \leq C y_j, \quad j = 1, \ldots, m \text{ with } x_{ij}, y_j \in \{0, 1\}$$

Individual (Solution) Representation for BPP

- Each candidate solution defines **which bin each item is assigned to.**
- Represented as an integer vector:

$$X = [b_1, b_2, \ldots, b_n]$$

  where b_i indicates the bin number of item i.
- Example (for $n = 6$ items):

$$X = [1, 1, 2, 3, 3, 2] \Rightarrow \text{items } 1, 2 \text{ in bin 1; } 3, 6 \text{ in bin 2; } 4, 5 \text{ in bin 3.}$$

- Fitness (objective): the number of bins used, i.e.

$$f(X) = \text{count of unique } b_i \text{ values.}$$

- Lower $f(X)$ is better.

Neighborhood Operators for BPP

- A **neighborhood** defines how to modify a packing configuration to create a new one.
- Common operators:
  - **Move:** take one item from a bin and place it into another bin.
  - **Swap:** exchange two items from different bins.
  - **Repack:** empty one bin and reassign its items elsewhere.
- Each neighbor must still satisfy:

$$\sum_{i \in \text{bin } j} s_i \leq C$$

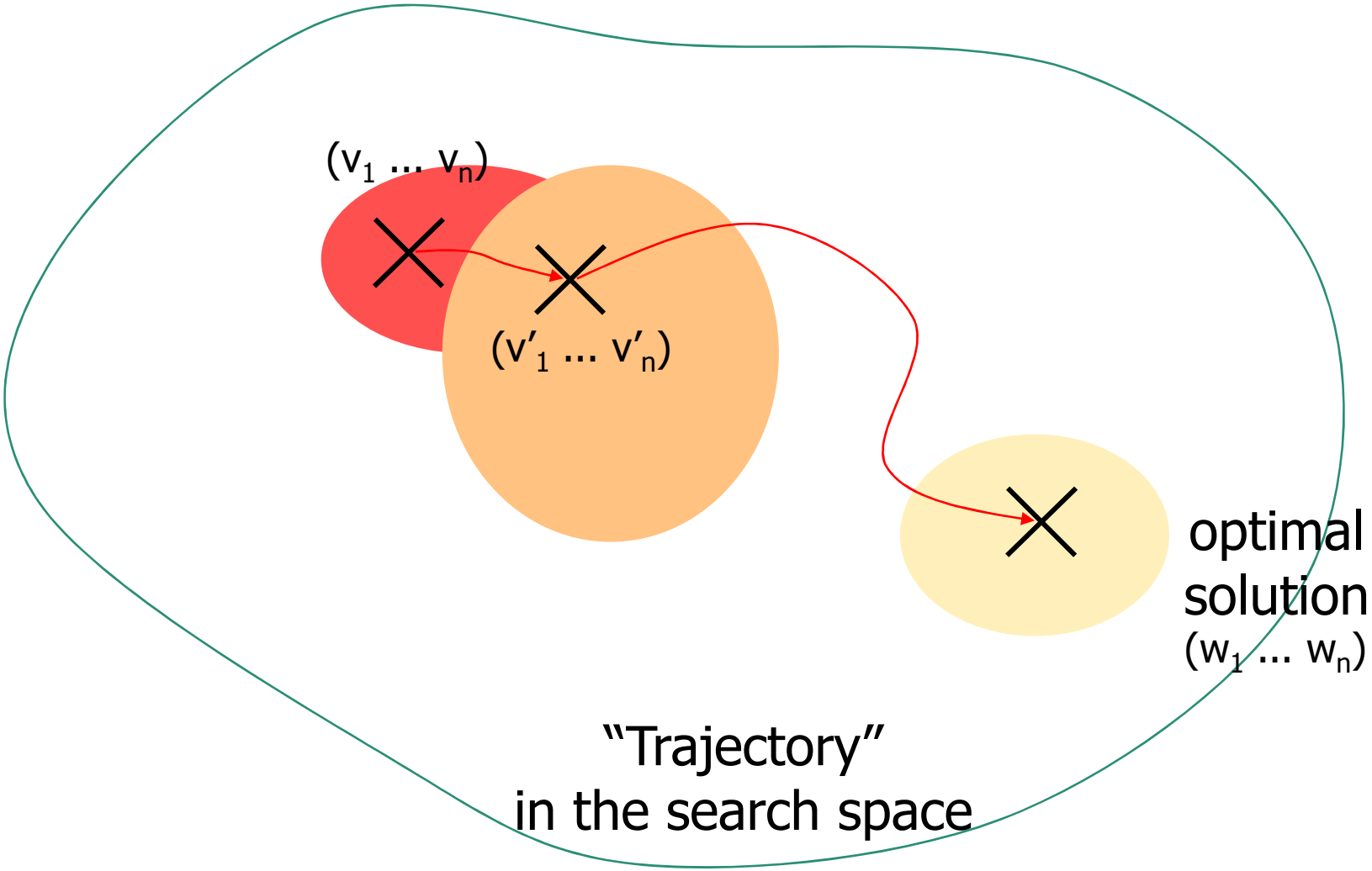
- **Purpose:** to explore nearby packings while reducing the number of bins used.

Simulated Annealing for the Bin Packing Problem

Algorithm Steps:

1. Generate an initial feasible solution X (e.g., using First Fit Decreasing).
2. Set $X_{best} = X$, and initialize temperature $T = T_0$.
3. Repeat until the stopping condition is met:
   - 3.1 Generate a neighbor X' using a **move** or **swap** operator.
   - 3.2 Compute change in fitness:
   $$\Delta = f(X') - f(X)$$
   - 3.3 If $\Delta < 0$, accept X' (improvement).
   - 3.4 Else, accept X' with probability:
   $$P = e^{-\Delta/T}$$
   - 3.5 Update temperature:
   $$T = \alpha \times T \quad (0 < \alpha < 1)$$
   - 3.6 Update X_{best} if $f(X') < f(X_{best})$.
4. Return X_{best} as the best packing.

Iterative Improvement



Basic Local Search : Iterative Improvement

```
s ← GenerateInitialSolution()
repeat
    s ← Improve(N(s))
until no improvement is possible
```

- Improve($\mathcal{N}$(S)) can be :
  ① First improvement
  ② Best improvement
  ③ Intermediate option, e.g. “Best among *n*”

- observation: stops in local optimum

Local Search 1 : Best Accept

```
1: input: starting solution, s_0
2: input: neighborhood operator, N
3: input: evaluation function, f
4: current ⇐ s_0
5: done ⇐ false
6: while done = false do
7:    best_neighbor ⇐ current
8:    for each s ∈ N(current) do
9:       if f(s) < f(best_neighbor) then
10:         best_neighbor ⇐ s
11:      end if
12:   end for
13:   if current = best_neighbor then
14:      done ⇐ true
15:   else
16:      current ⇐ best_neighbor
17:   end if
18: end while
```

Local Search 2 : First Accept

```
1: input: starting solution, s_0
2: input: neighborhood operator, N
3: input: evaluation function, f
4: current ⇐ s_0
5: done ⇐ false
6: while done = false do
7:    best_neighbor ⇐ current
8:    for each s ∈ N(current) do
9:       if f(s) < f(best_neighbor) then
10:         best_neighbor ⇐ s
11:         exit the for-loop
12:      end if
13:   end for
14:   if current = best_neighbor then
15:      done ⇐ true
16:   else
17:      current ⇐ best_neighbor
18:   end if
19: end while
```

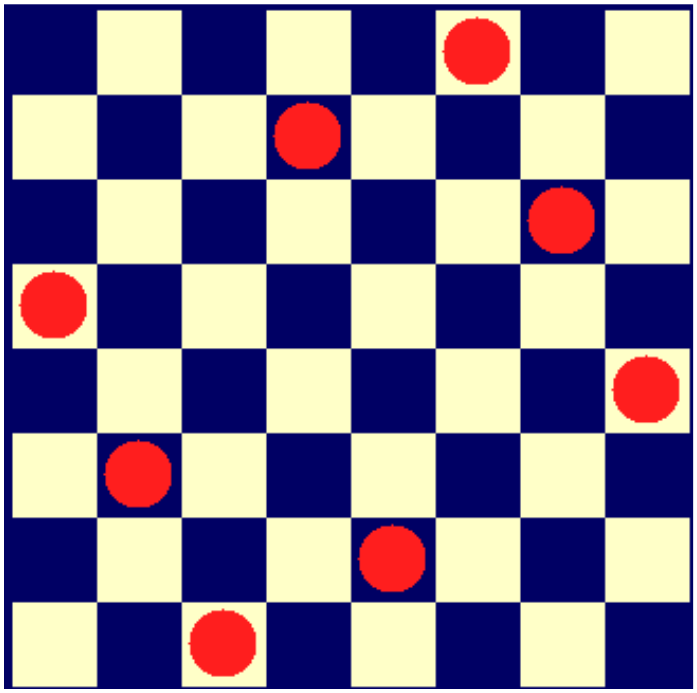
Remark: for a minimization problem

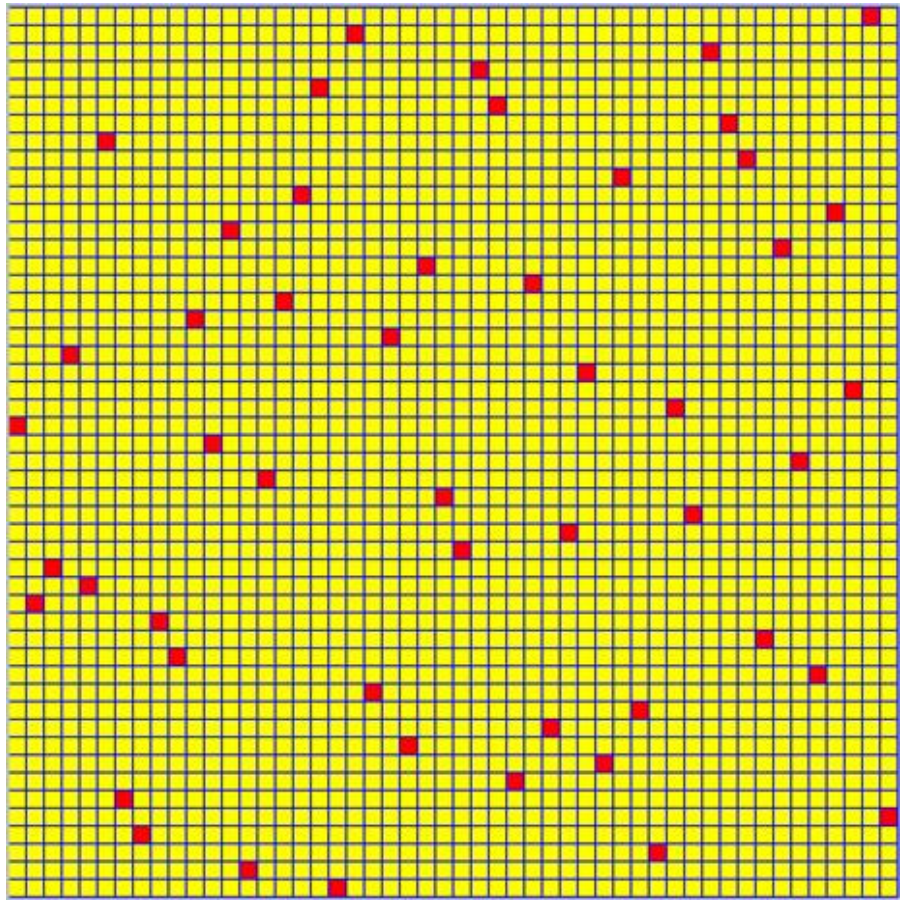
[From A. Løkketangen]

God save the Queens

- Place 8 queens on a chessboard

  s.t. no two queens attack each other
- Generalized to NxN chessboard

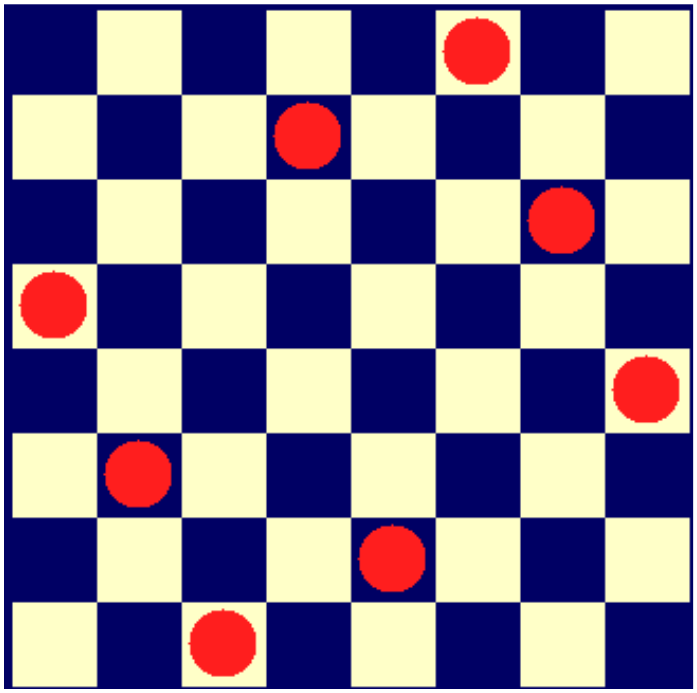


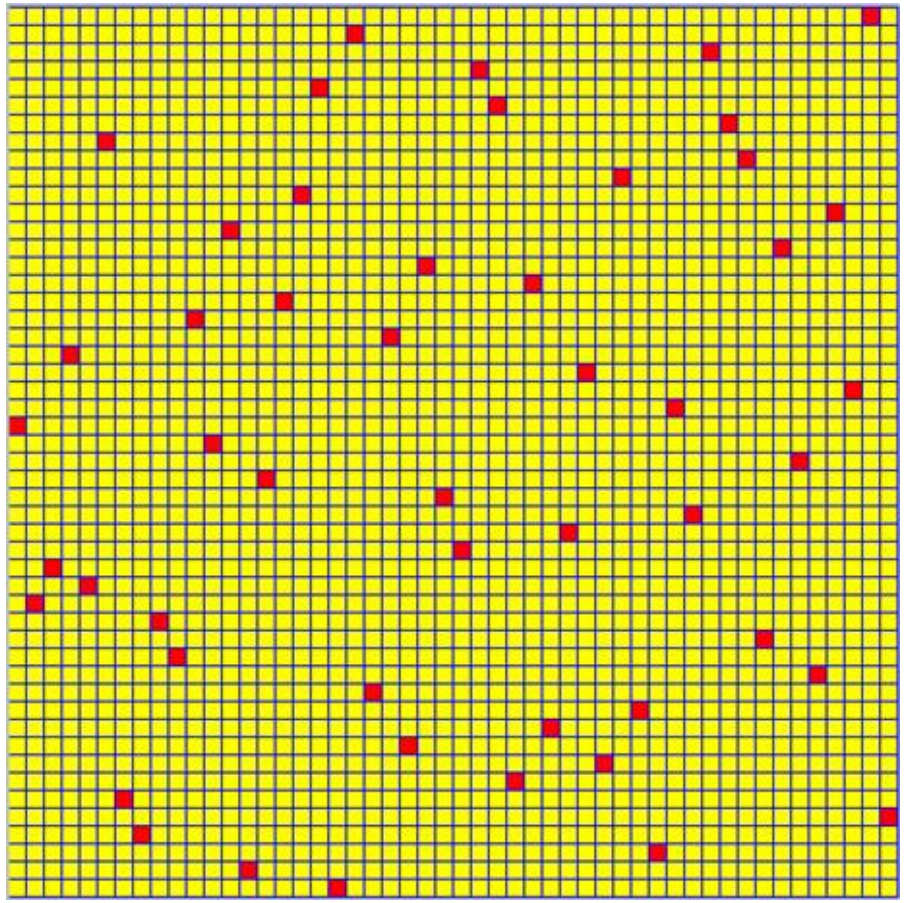


one solution for 50 x 50 chessboard

God save the Queens

- Place 8 queens on a chessboard
  s.t. no two queens attack each other
- Generalized to NxN chessboard





one solution for 50 x 50 chessboard

Can we solve this problem
by Local Search ?

Local Search for N-Queens

- Configuration: (Q1,…,Qn)

  Qi = j means queen on row i and column j
- Objective function :

  minimize the number of queens attacked

  (= 2 x number of violated disequation constraints)
- Neighborhood operator 1: change the position of one queen (i.e. change value of one Qi)
- Size of neighborhood: *n*(n-1)*
- At each step there is a quadratic number of neighbors to evaluate…

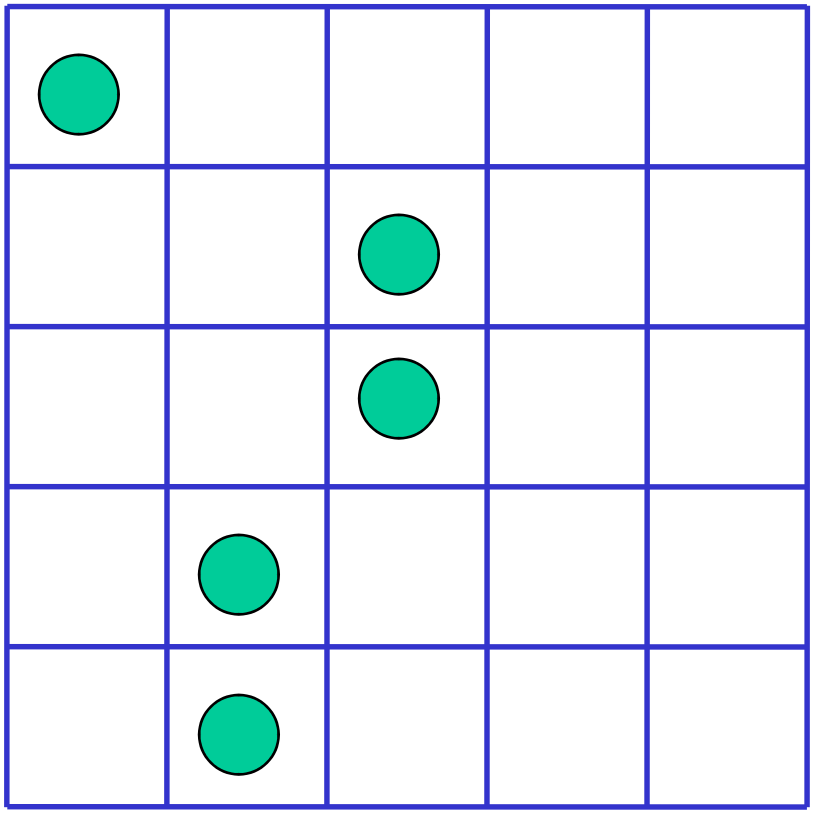
Local Search for N-Queens (2)

- Neighborhood operator 2:
  - compute for each queen the number of other queens attacking it
  - select the most conflicting queen
  - Consider only its alternative positions as neighbors

- Neighborhood of size *n-1*

- Each step of local search is thus faster
- But is it a good heuristics ?

Example



1

1

3

2

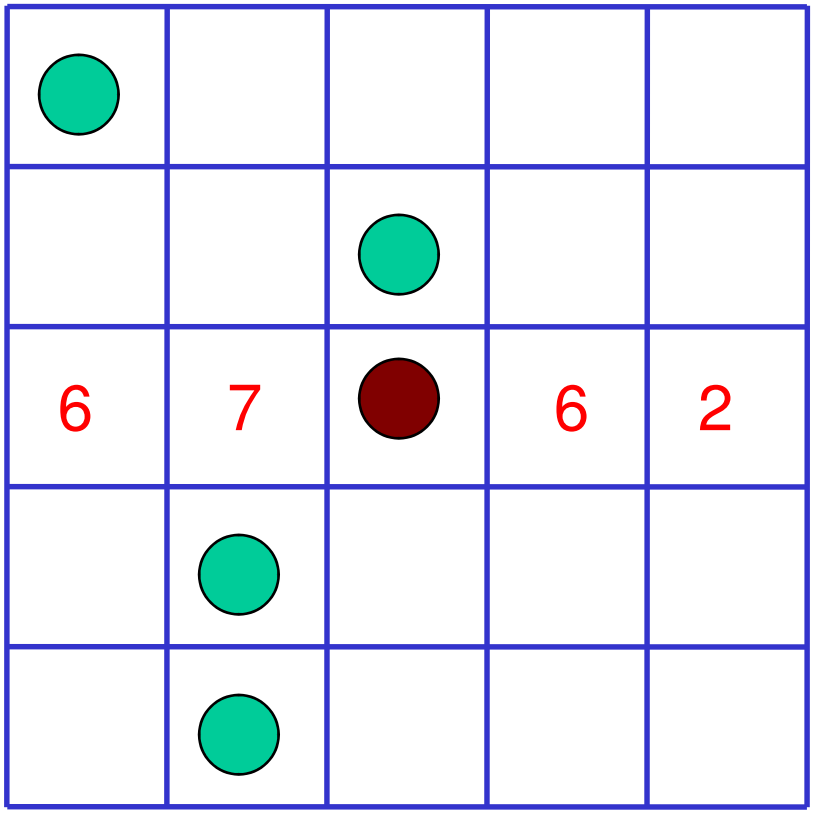
1

8

← cost for each queen

← Global cost

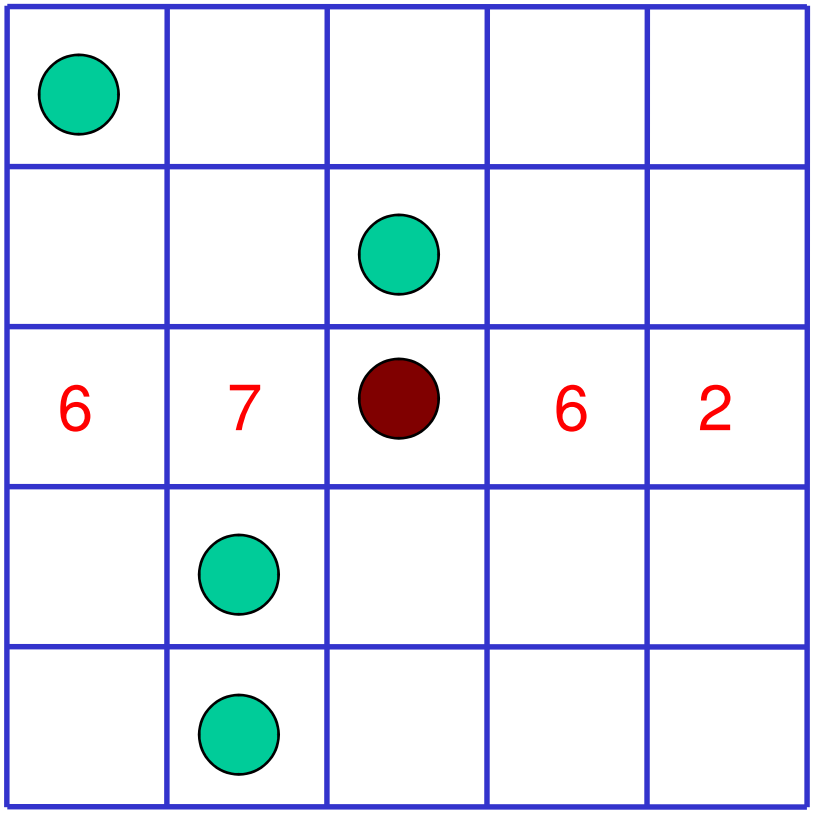
Example



Queen 3 will be selected

Alternative values gives
Other global costs

Example

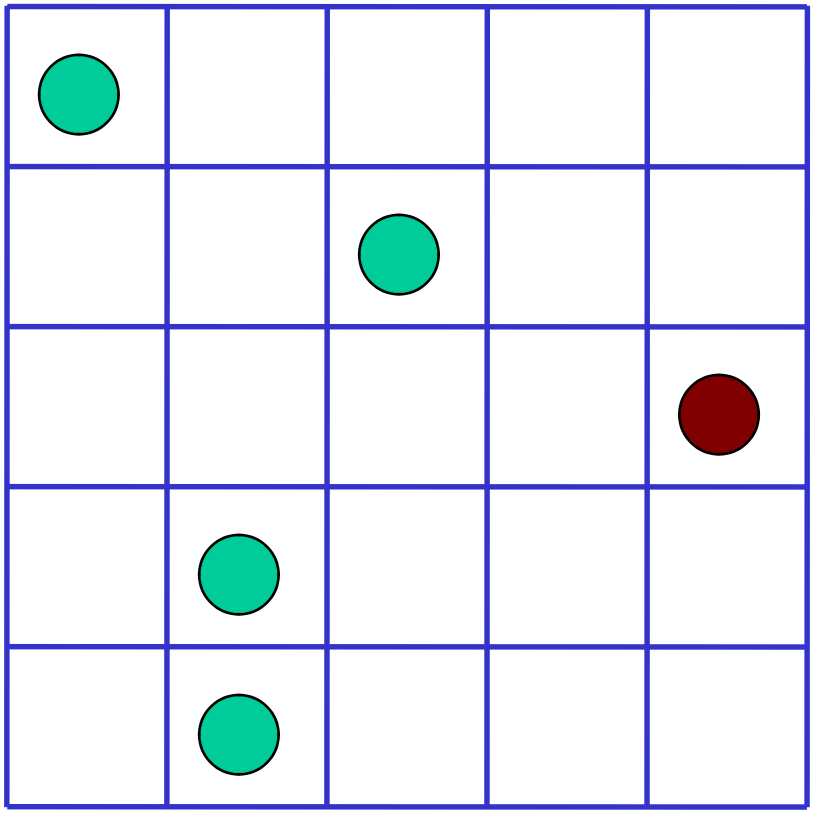


Queen 3 will be selected

Alternative values gives
Other global costs

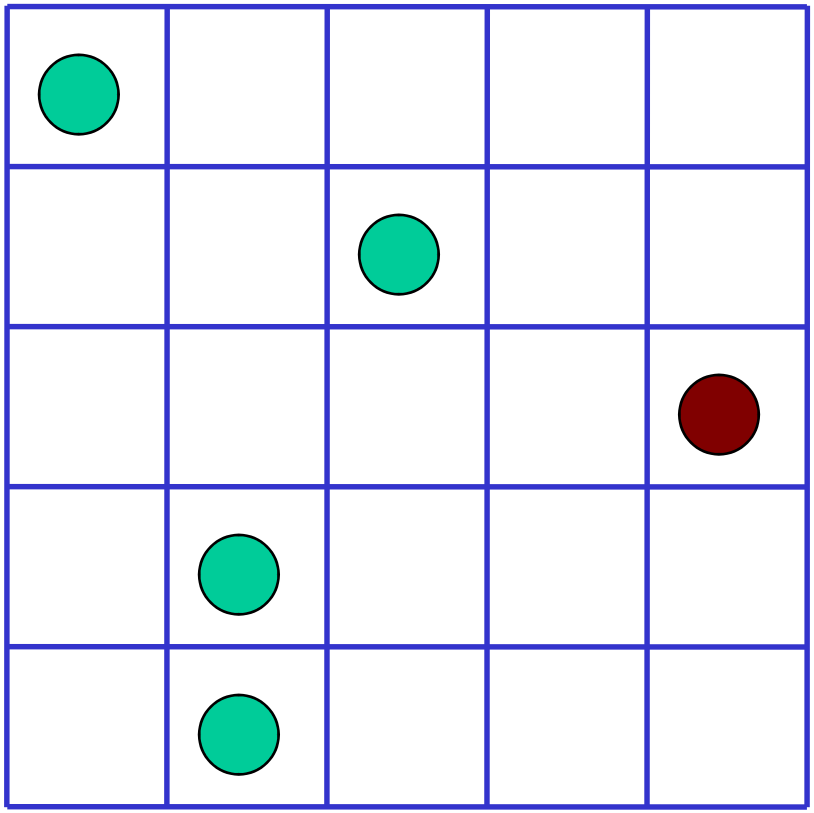
Move to row 5 is the best

Example



… and continue !

Example



0

0

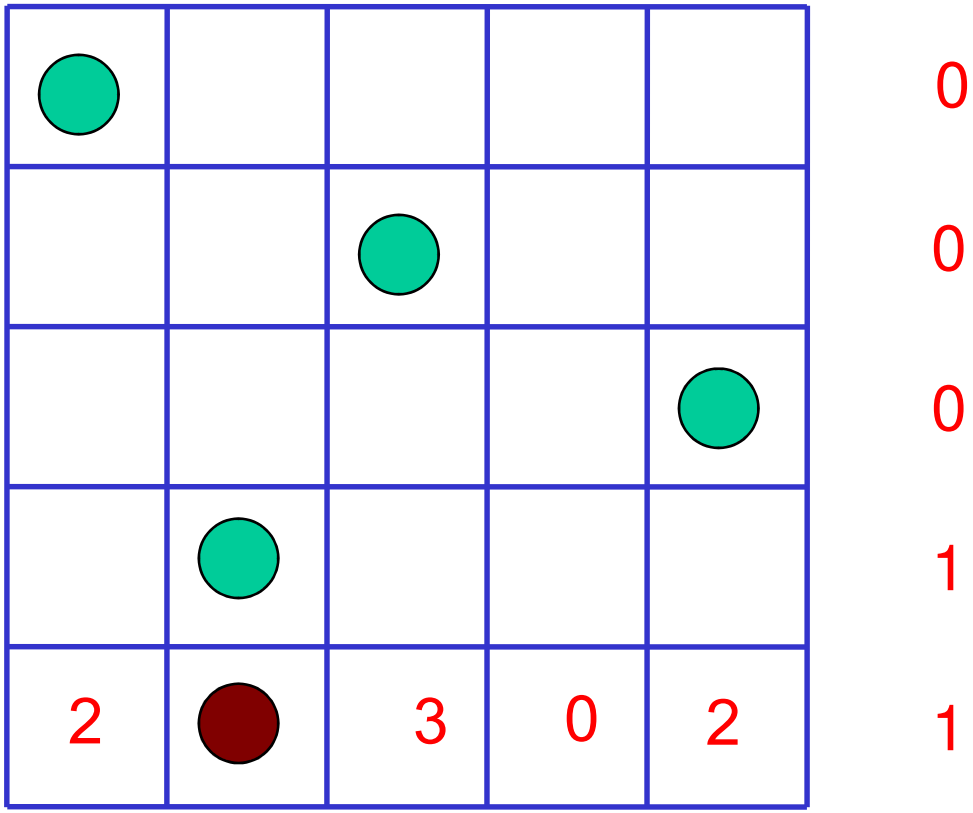
0

1

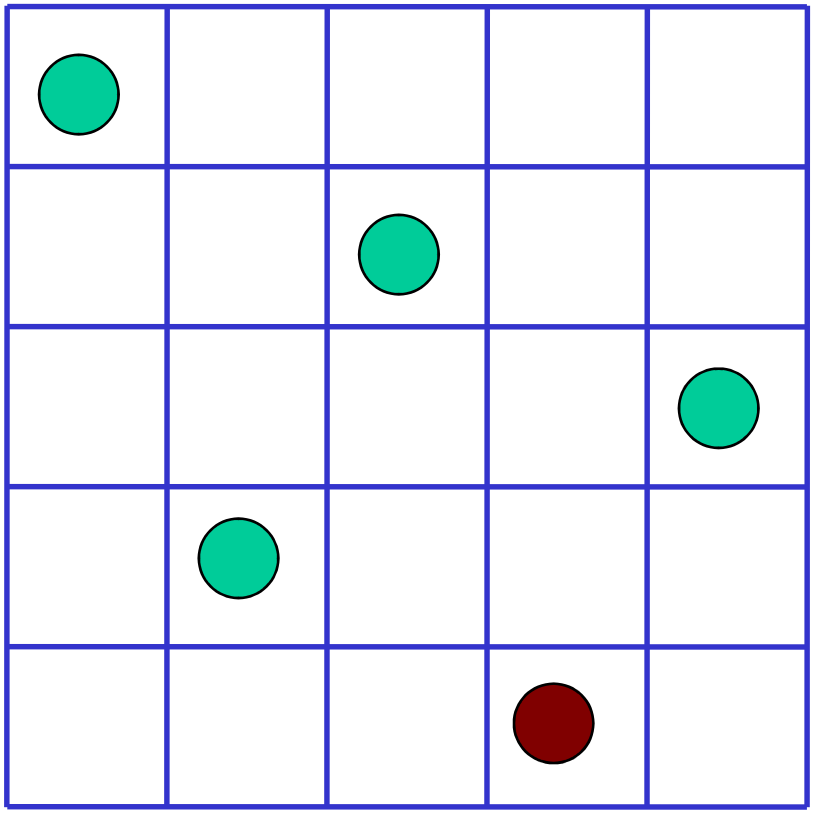
1

2

Example



Example



0

0

0

0

0

SOLVED !