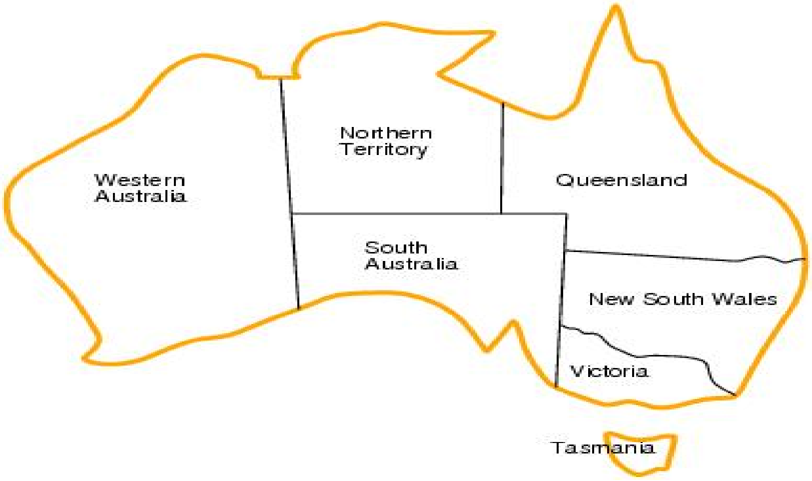
Symmetries



- Map coloring with *n* colors has *n!* permutations for every solution.
- Value symmetry.
- Add symmetry-breaking constraint e.g., $NT < SA < WA$.
- In general breaking all symetries is NP-hard.

Lecture 9: Local Search (Meta-Heuristics)

1. Hill-climbing
2. Simulated Annealing
3. Tabu search
4. Genetic algorithms
5. Constraint-Based Local Search

Local Search: Overview & Motivation

- Searches in the space of complete solutions
- Has low memory consumption
- Effective at solving large optimization problems
- Easy to implement real-world constraints
- Therefore, local search is widely used in industry and academia
- However:
  - Incomplete method, i.e. no guarantees of optimality

Outline

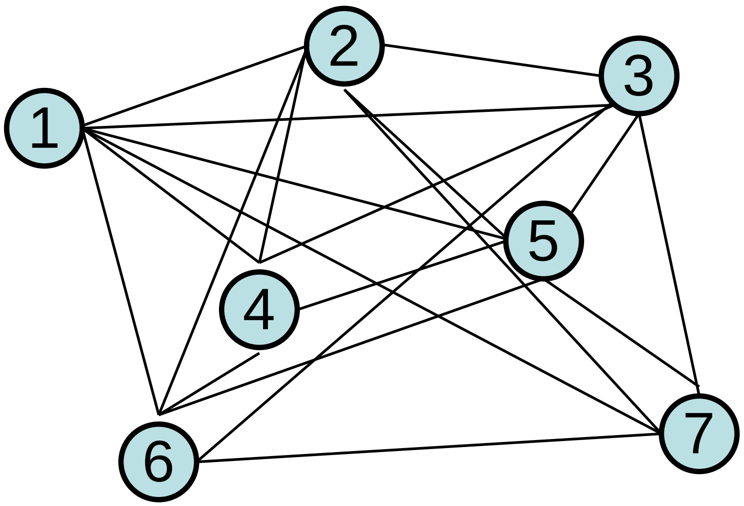
1. Local Search
2. Hill Climbing
3. Simulated Annealing
4. (Break)
5. Tabu Search
6. Genetic Algorithms
7. Constraint Based Local Search

Local Search

1. Begin with a complete assignment to variables.
   - (A solution to the problem)
2. Search by moving to other complete assignments.
   - (Explore the “neighborhood”)
3. Repeat the previous step until the assignment is “Good enough”
   - (Termination condition)

Traveling Salesman Problem (TSP)

- Given: A fully connected undirected graph.
  - Edge costs: distance between nodes

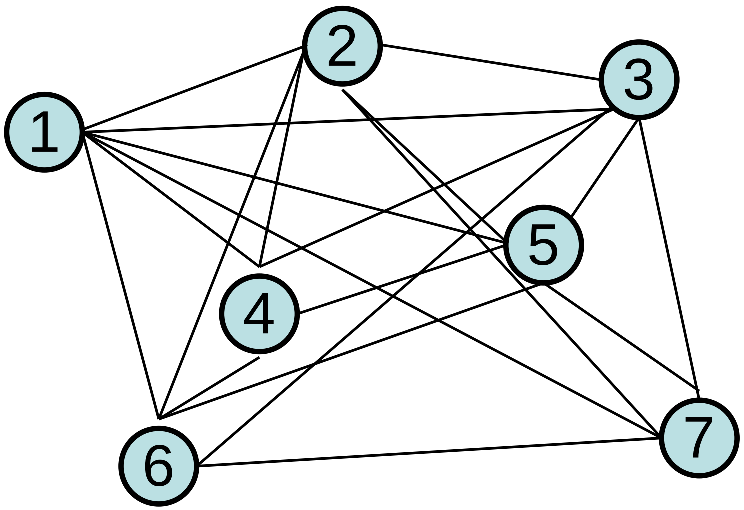


	1	2	3	4	5	6	7
1	0	2.2	5	2.8	4.1	5	8.5
2	2.2	0	3	3	2.8	6	8
3	5	3	0	4.2	2.2	7.2	7
4	2.8	3	4.2	0	2.2	3.1	5.7
5	4.1	2.8	2.2	2.2	0	5	5.4
6	5	6	7.2	3.1	5	0	5.1
7	8.5	8	7	5.7	5.4	5.1	0

- Task: Find the minimum cost path that visits all nodes and returns to the start.

TSP: Initial Solution

- Local searches need an initial solution.
- We can store a TSP solution as a permutation.

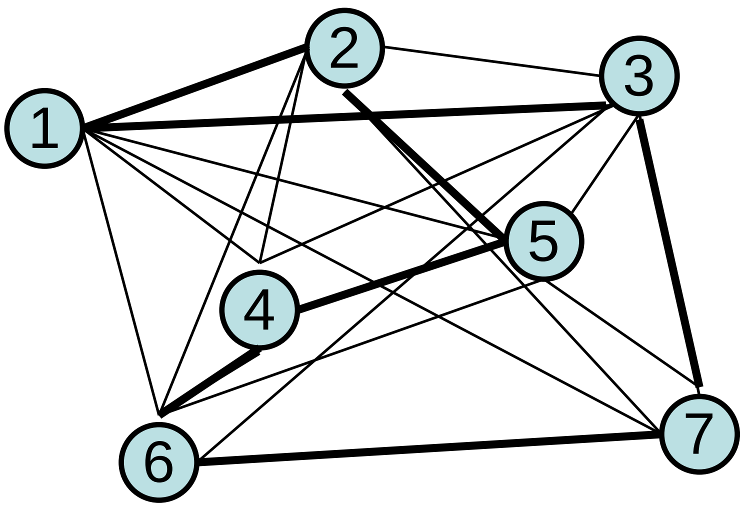


	1	2	3	4	5	6	7
1	0	2.2	5	2.8	4.1	5	8.5
2	2.2	0	3	3	2.8	6	8
3	5	3	0	4.2	2.2	7.2	7
4	2.8	3	4.2	0	2.2	3.1	5.7
5	4.1	2.8	2.2	2.2	0	5	5.4
6	5	6	7.2	3.1	5	0	5.1
7	8.5	8	7	5.7	5.4	5.1	0

- How would you construct an initial solution?

TSP: Initial Solution

- Nearest neighbor heuristic

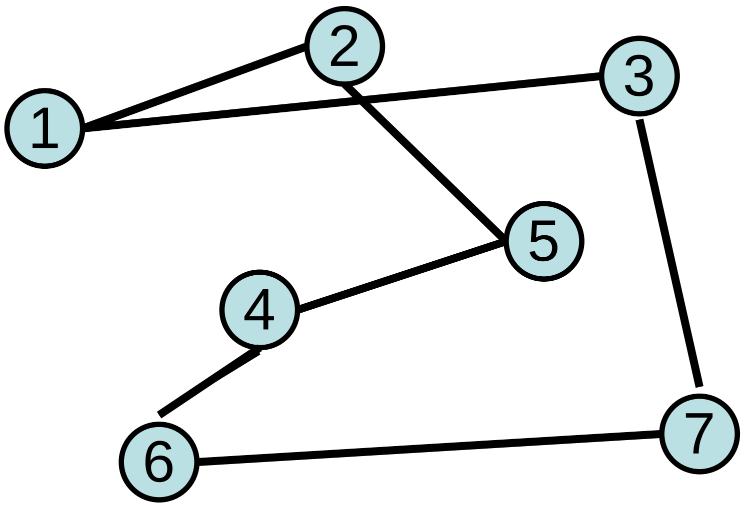


	1	2	3	4	5	6	7
1	0	2.2	5	2.8	4.1	5	8.5
2	2.2	0	3	3	2.8	6	8
3	5	3	0	4.2	2.2	7.2	7
4	2.8	3	4.2	0	2.2	3.1	5.7
5	4.1	2.8	2.2	2.2	0	5	5.4
6	5	6	7.2	3.1	5	0	5.1
7	8.5	8	7	5.7	5.4	5.1	0

- Initial solution: 1 2 5 4 6 7 3
- Cost: 27.4

TSP: Neighborhoods

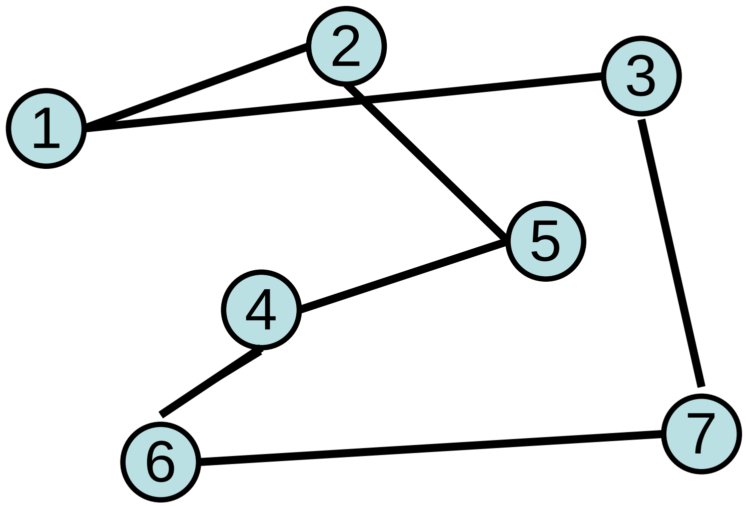
- How can we improve our initial solution?



	1	2	3	4	5	6	7
1	0	2.2	5	2.8	4.1	5	8.5
2	2.2	0	3	3	2.8	6	8
3	5	3	0	4.2	2.2	7.2	7
4	2.8	3	4.2	0	2.2	3.1	5.7
5	4.1	2.8	2.2	2.2	0	5	5.4
6	5	6	7.2	3.1	5	0	5.1
7	8.5	8	7	5.7	5.4	5.1	0

TSP: 2-Opt Neighborhood

- 2-Opt removes 2 edges that cross each other and replaces them with non-crossing edges.

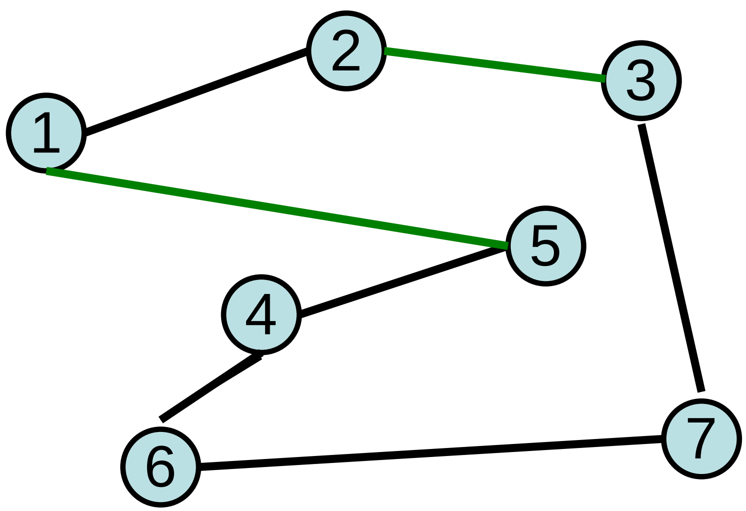


	1	2	3	4	5	6	7
1	0	2.2	5	2.8	4.1	5	8.5
2	2.2	0	3	3	2.8	6	8
3	5	3	0	4.2	2.2	7.2	7
4	2.8	3	4.2	0	2.2	3.1	5.7
5	4.1	2.8	2.2	2.2	0	5	5.4
6	5	6	7.2	3.1	5	0	5.1
7	8.5	8	7	5.7	5.4	5.1	0

- Which edges should we pick to remove?

TSP: 2-Opt Neighborhood

- 2-Opt removes 2 edges that cross each other and replaces them with non-crossing edges.

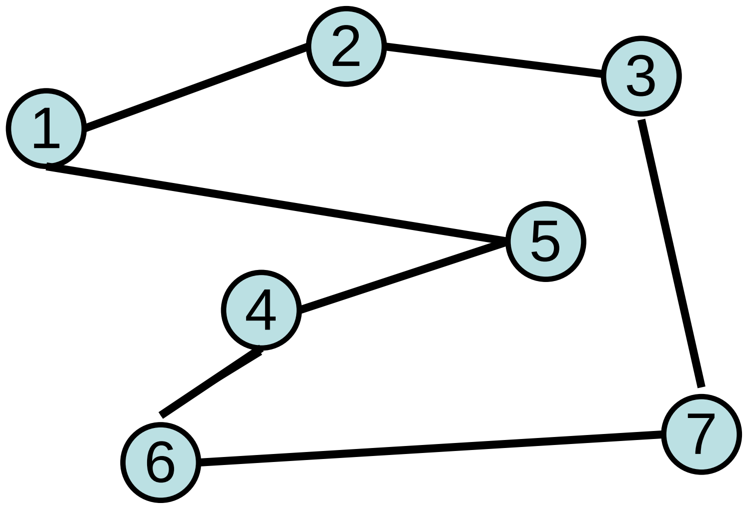


	1	2	3	4	5	6	7
1	0	2.2	5	2.8	4.1	5	8.5
2	2.2	0	3	3	2.8	6	8
3	5	3	0	4.2	2.2	7.2	7
4	2.8	3	4.2	0	2.2	3.1	5.7
5	4.1	2.8	2.2	2.2	0	5	5.4
6	5	6	7.2	3.1	5	0	5.1
7	8.5	8	7	5.7	5.4	5.1	0

- New solution: 1 2 3 7 6 4 5
- New cost: 26.7 (previous cost: 27.1)

TSP: *k*-Opt Neighborhood

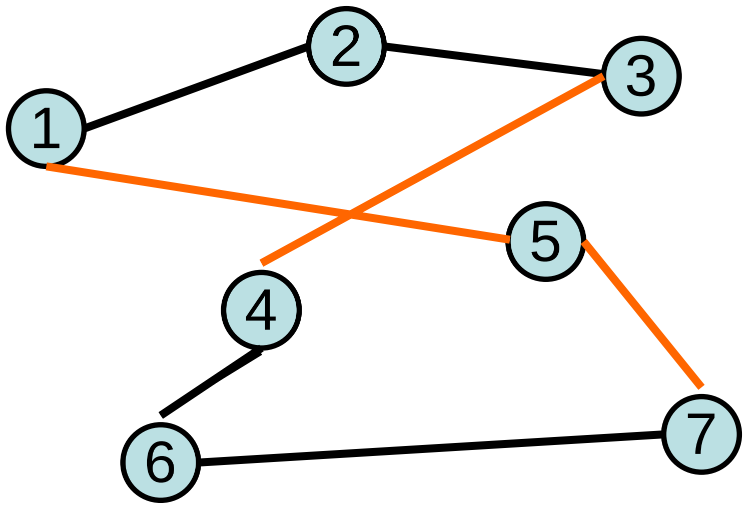
- Remove *k* edges and repair the path.
- Lets try *k*=3



	1	2	3	4	5	6	7
1	0	2.2	5	2.8	4.1	5	8.5
2	2.2	0	3	3	2.8	6	8
3	5	3	0	4.2	2.2	7.2	7
4	2.8	3	4.2	0	2.2	3.1	5.7
5	4.1	2.8	2.2	2.2	0	5	5.4
6	5	6	7.2	3.1	5	0	5.1
7	8.5	8	7	5.7	5.4	5.1	0

TSP: *k*-Opt Neighborhood

- Remove *k* edges and repair the path.
- Lets try *k*=3

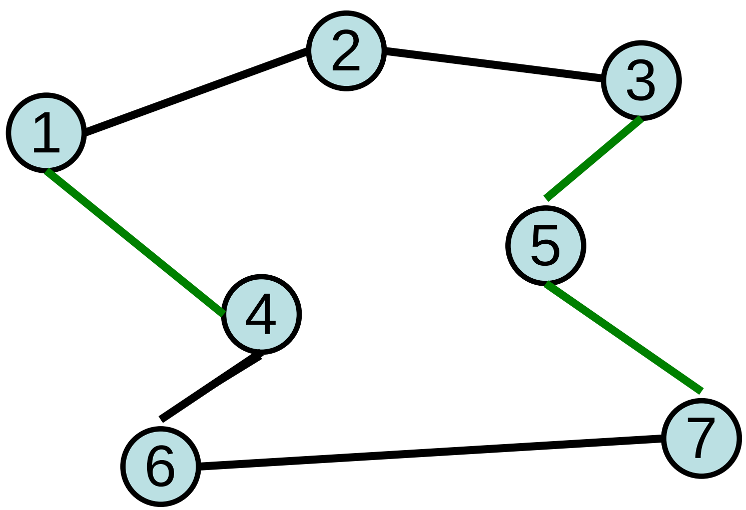


	1	2	3	4	5	6	7
1	0	2.2	5	2.8	4.1	5	8.5
2	2.2	0	3	3	2.8	6	8
3	5	3	0	4.2	2.2	7.2	7
4	2.8	3	4.2	0	2.2	3.1	5.7
5	4.1	2.8	2.2	2.2	0	5	5.4
6	5	6	7.2	3.1	5	0	5.1
7	8.5	8	7	5.7	5.4	5.1	0

- Possible solution: 1 2 3 4 6 7 5
- Cost: 27.1

TSP: *k*-Opt Neighborhood

- Remove *k* edges and repair the path.
- Lets try *k*=3

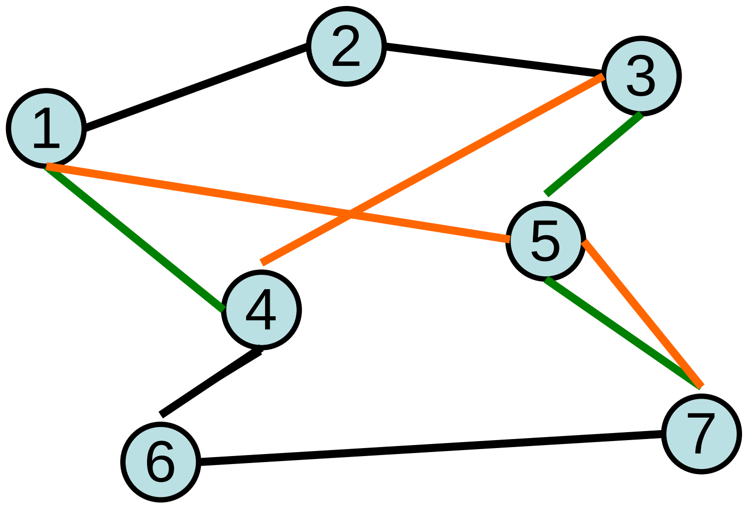


	1	2	3	4	5	6	7
1	0	2.2	5	2.8	4.1	5	8.5
2	2.2	0	3	3	2.8	6	8
3	5	3	0	4.2	2.2	7.2	7
4	2.8	3	4.2	0	2.2	3.1	5.7
5	4.1	2.8	2.2	2.2	0	5	5.4
6	5	6	7.2	3.1	5	0	5.1
7	8.5	8	7	5.7	5.4	5.1	0

- Possible solution: 1 2 3 5 7 6 4
- Cost: 23.8

TSP: Neighbor Selection

- Which neighbor should we choose?

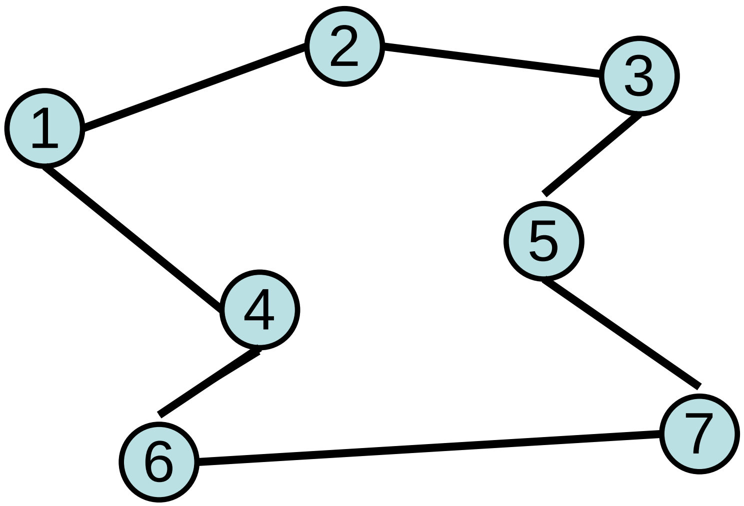


	1	2	3	4	5	6	7
1	0	2.2	5	2.8	4.1	5	8.5
2	2.2	0	3	3	2.8	6	8
3	5	3	0	4.2	2.2	7.2	7
4	2.8	3	4.2	0	2.2	3.1	5.7
5	4.1	2.8	2.2	2.2	0	5	5.4
6	5	6	7.2	3.1	5	0	5.1
7	8.5	8	7	5.7	5.4	5.1	0

- 23.8 vs. 27.1

TSP: Termination

- When should we stop performing improvements?



	1	2	3	4	5	6	7
1	0	2.2	5	2.8	4.1	5	8.5
2	2.2	0	3	3	2.8	6	8
3	5	3	0	4.2	2.2	7.2	7
4	2.8	3	4.2	0	2.2	3.1	5.7
5	4.1	2.8	2.2	2.2	0	5	5.4
6	5	6	7.2	3.1	5	0	5.1
7	8.5	8	7	5.7	5.4	5.1	0

TSP: Termination

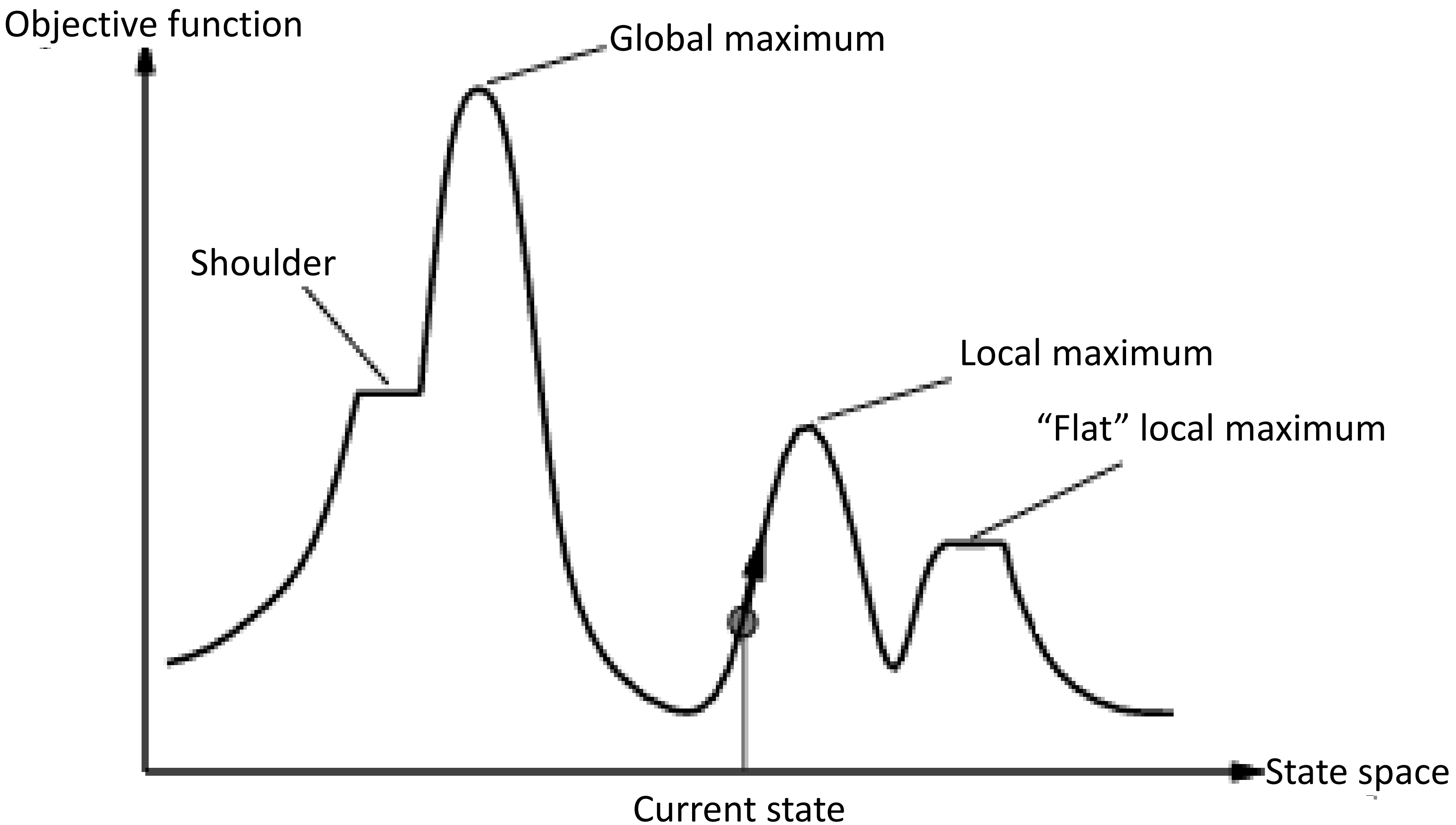
- When should we stop performing improvements?
- Budget based:
  - Max. cost evaluations
  - CPU time
- Solution quality
  - Within X% of a lower bound
  - Business requirements satisfied
- Convergence criteria:
  - No improvement in last Y iterations.
  - Average improvement below threshold ε

Hill Climbing

Hill Climbing Algorithm

```
function HILL-CLIMBING( problem ) return a state that is a local maximum
    current ← MAKE-NODE( problem.INITIAL-STATE)
    loop do
        neighbor ←  a highest-value successor of current
        if neighbor.VALUE ≤ current.VALUE then return current.STATE
        current ← neighbor
```

State-Space Landscape



Hill Climbing: Pro & Con

- Advantages
  - Fast convergence to a local maximum
  - Often results in good (but not optimal) solutions
- Disadvantages
  - Gets stuck in local maxima
  - Gets stuck on shoulders and plateaus

Exploitation vs. Exploration

- Exploitation
  - Greedy; always select most improving neighbor
- Exploration
  - Also select less improving and non-improving neighbors

Hill Climbing ⟷ Random Walk

Exploitation (a.k.a. intensification)

Exploration (a.k.a. diversification)

Escaping Local Maxima

- Variations of Hill-Climbing
  - Sideways move: allow non-improving moves to traverse plateaus.
  - Stochastic Hill-Climbing: random choice of uphill moves.
  - First-Choice Hill-Climbing: random generation of neighbors.
  - Random-restart Hill-Climbing: restart the search from a different initial state

Simulated Annealing

Simulated Annealing

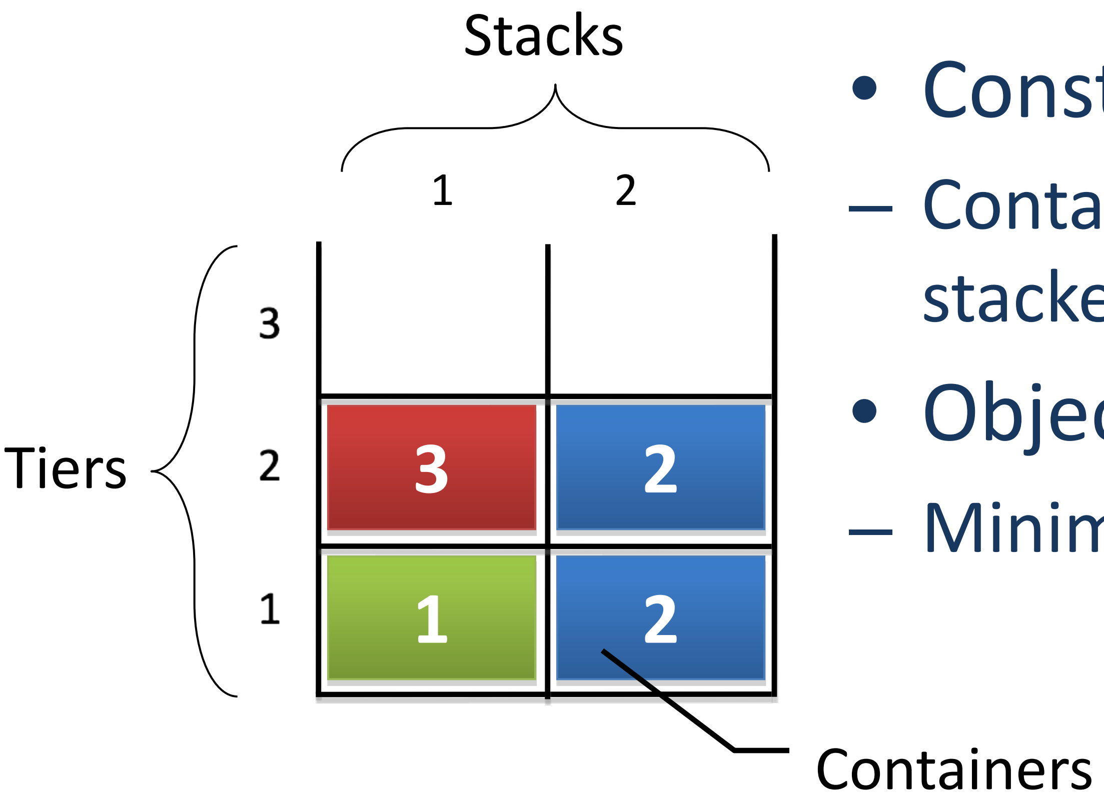
- Inspired by annealing in metallurgy
  - Used to harden metals by gradually cooling them, allowing atoms to find a low-energy crystalline state
- Idea:
  - Escape local maxima by allowing some "bad" moves but gradually decrease their frequency

Simulated Annealing Implementation

```
function SIMULATED-ANNEALING( problem, schedule ) returns a solution state
  input:  problem,  a problem
          schedule, a mapping from time to "temperature"

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

Ex. Container Stowage Problem

- Given a feasible container configuration, find the one which minimizes overstowage.



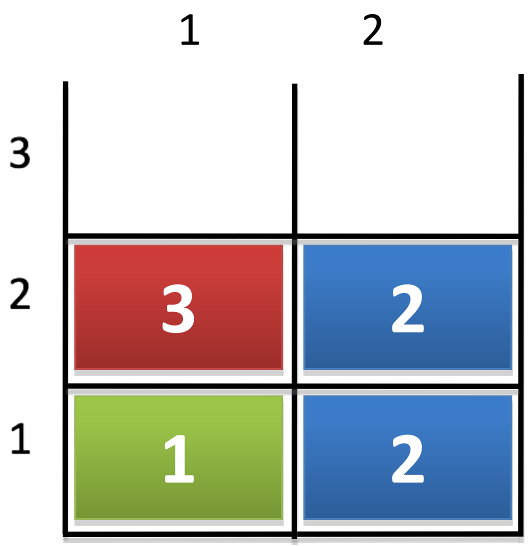
- Constraints
  – Containers must be stacked
- Objective
  – Minimize overstowage

Ex. Container Stowage Problem

- State: A container configuration
- Neighborhood: Container swaps (*complete*)
- Objective function (*Value*): Number of overstowed containers.
- Termination criteria: *Value* = 0
- ΔE := *current*.VALUE – *next*.VALUE

Obs: Minimization problem!

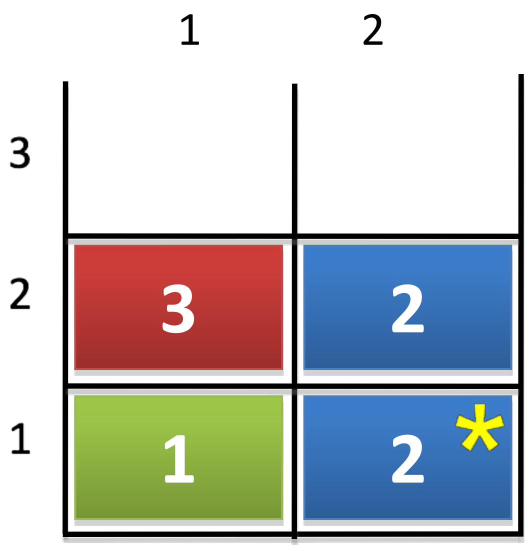
Ex. Container Stowage Problem



Objective: 1

Temperature: 10

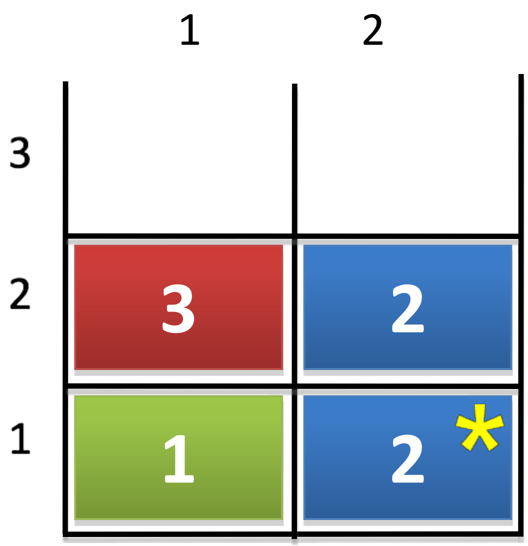
Ex. Container Stowage Problem



Objective: 1

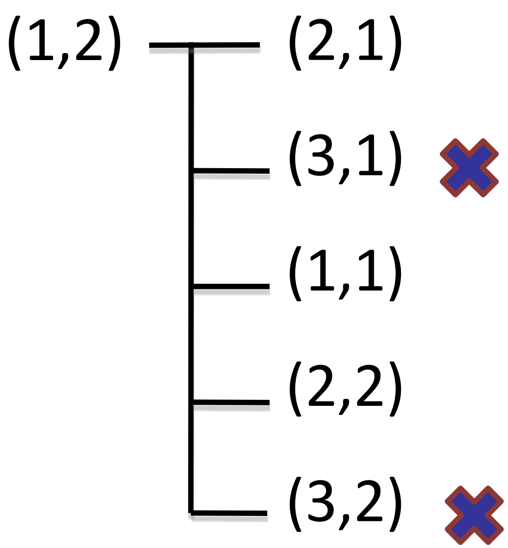
Temperature: 10

Ex. Container Stowage Problem

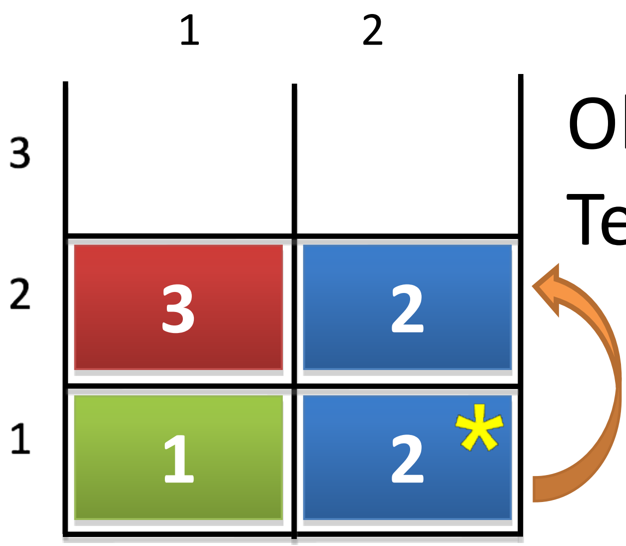


Objective: 1

Temperature: 10



Ex. Container Stowage Problem

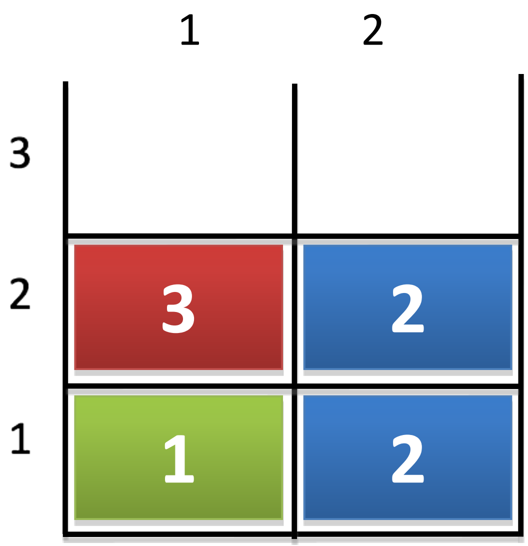


Objective: 1

Temperature: 10

(1,2)
- (2,1)
- (3,1) ✗
- (1,1)
- (2,2) $\Delta E = 0 : e^{\frac{0}{10}} = 1$ ✓
- (3,2) ✗

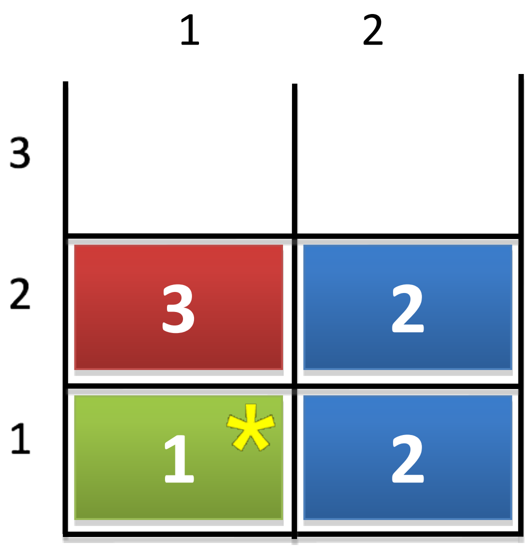
Ex. Container Stowage Problem



Objective: 1

Temperature: 2

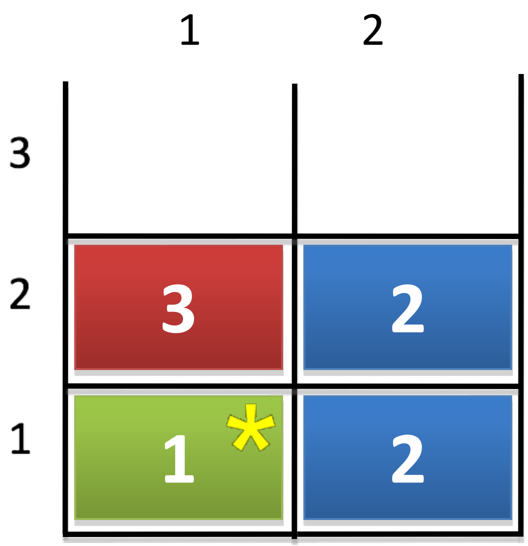
Ex. Container Stowage Problem



Objective: 1

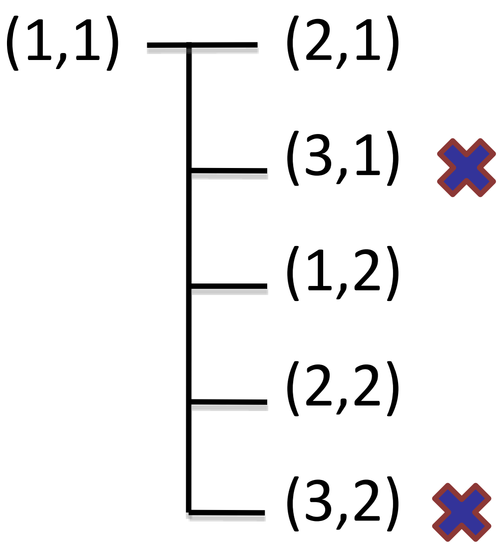
Temperature: 2

Ex. Container Stowage Problem

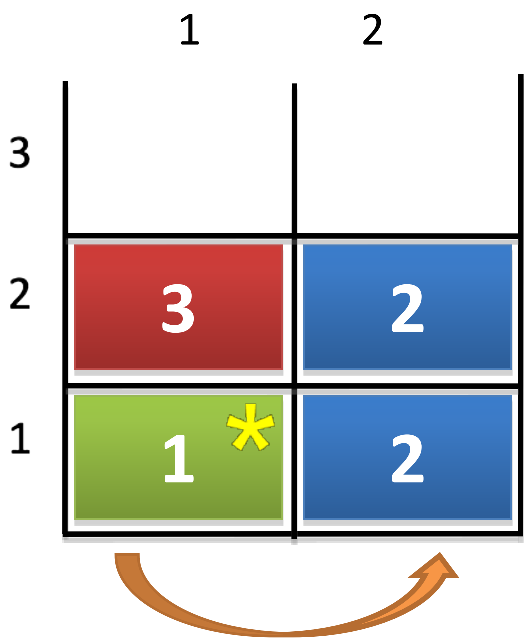


Objective: 1

Temperature: 2



Ex. Container Stowage Problem

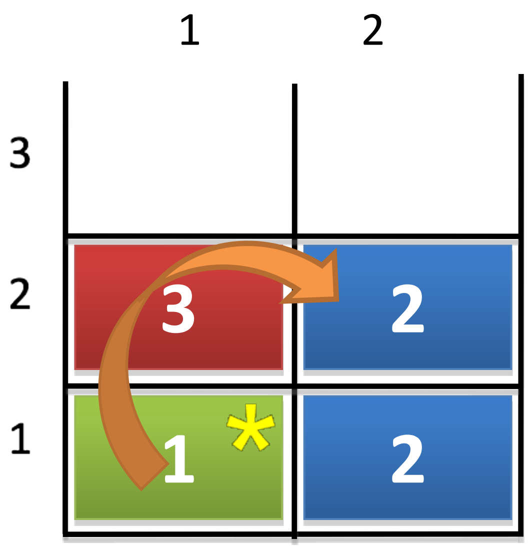


Objective: 1

Temperature: 2

(1,1)
- (2,1)
- (3,1) ✖
- (1,2) $\Delta E = -1 : e^{-\frac{1}{2}} \approx 0{,}60$ 🛇
- (2,2)
- (3,2) ✖

Ex. Container Stowage Problem

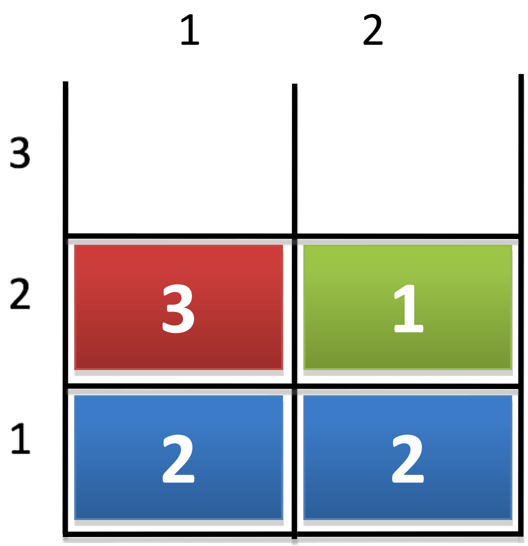


Objective: 1

Temperature: 2

(1,1)
- (2,1)
- (3,1)
- (1,2) $\Delta E = -1 : e^{-\frac{1}{2}} \approx 0{,}60$
- (2,2) $\Delta E = 0 : e^{\frac{0}{2}} = 1$
- (3,2)

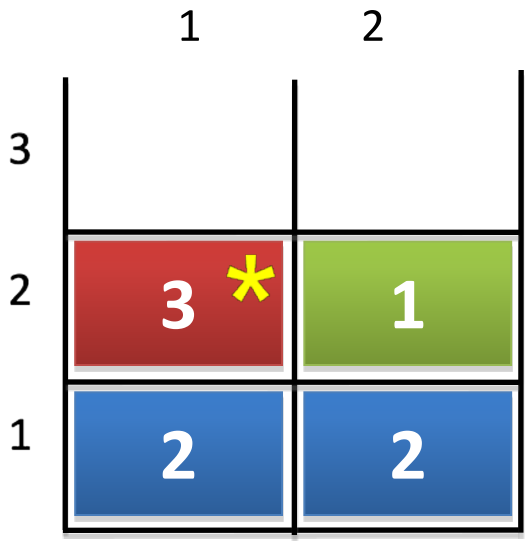
Ex. Container Stowage Problem



Objective: 1

Temperature: 0,5

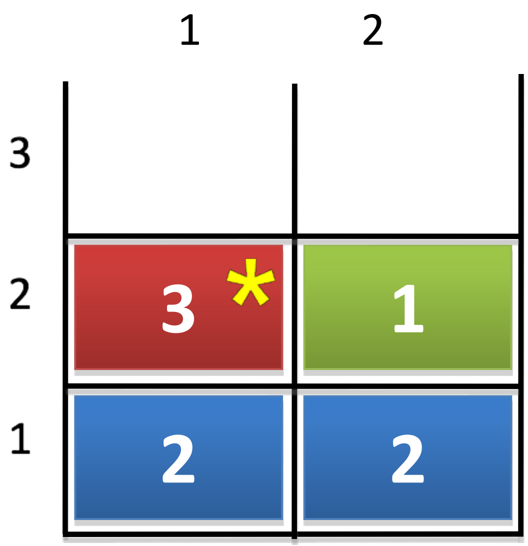
Ex. Container Stowage Problem



Objective: 1

Temperature: 0,5

Ex. Container Stowage Problem

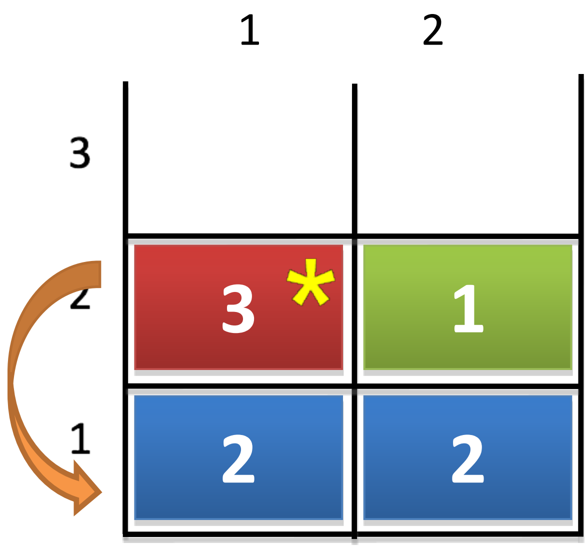


Objective: 1

Temperature: 0,5

(2,1)
- (3,1)
- (1,1)
- (1,2)
- (2,2)
- (3,2)

Ex. Container Stowage Problem

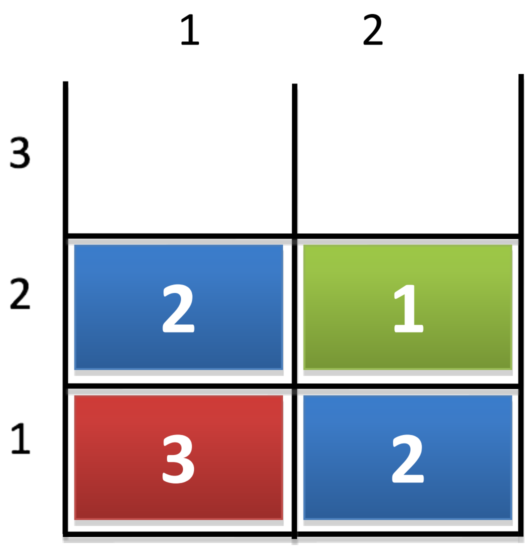


Objective: 1

Temperature: 0,5

(2,1)
- (3,1)
- (1,1) $\Delta E = 1$ Improving !
- (1,2)
- (2,2)
- (3,2)

Ex. Container Stowage Problem



Objective: 0

Temperature: 0,2

Lower bound ! Terminate!

Tabu Search (TS)

Tabu Search

- A tabu (also spelled taboo) is a strong social prohibition (or ban) against words, objects, actions, or discussions that are considered undesirable or offensive by a group, culture, society, or community.

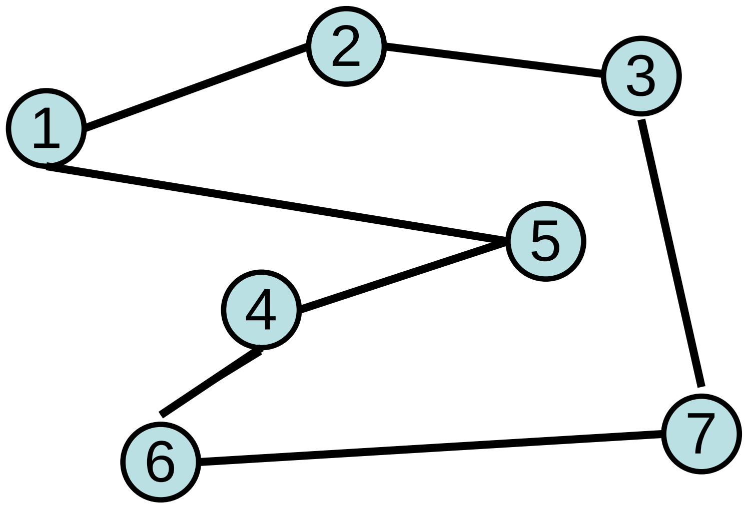
  - "Taboo" Wikipedia

Tabu Search

- Idea:
  - Accept the best neighbor at each iteration
  - Avoid previously seen solutions by keeping a memory (tabu list) of previous states

Tabu Search: TSP Example

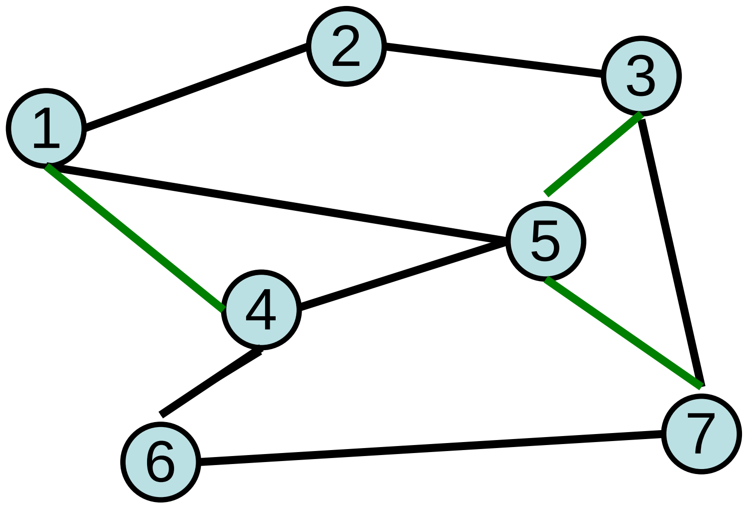
- What could we store in our tabu list?



	1	2	3	4	5	6	7
1	0	2.2	5	2.8	4.1	5	8.5
2	2.2	0	3	3	2.8	6	8
3	5	3	0	4.2	2.2	7.2	7
4	2.8	3	4.2	0	2.2	3.1	5.7
5	4.1	2.8	2.2	2.2	0	5	5.4
6	5	6	7.2	3.1	5	0	5.1
7	8.5	8	7	5.7	5.4	5.1	0

Tabu Search: TSP Example

- We could store an entire solution
- Or, just store the changes we made



	1	2	3	4	5	6	7
1	0	2.2	5	2.8	4.1	5	8.5
2	2.2	0	3	3	2.8	6	8
3	5	3	0	4.2	2.2	7.2	7
4	2.8	3	4.2	0	2.2	3.1	5.7
5	4.1	2.8	2.2	2.2	0	5	5.4
6	5	6	7.2	3.1	5	0	5.1
7	8.5	8	7	5.7	5.4	5.1	0

- Tabu list:
  1. -(1,5), -(4,5), -(3,7)
  2. +(1,4), +(5,7), +(3,5)

Tabu Search Implementation

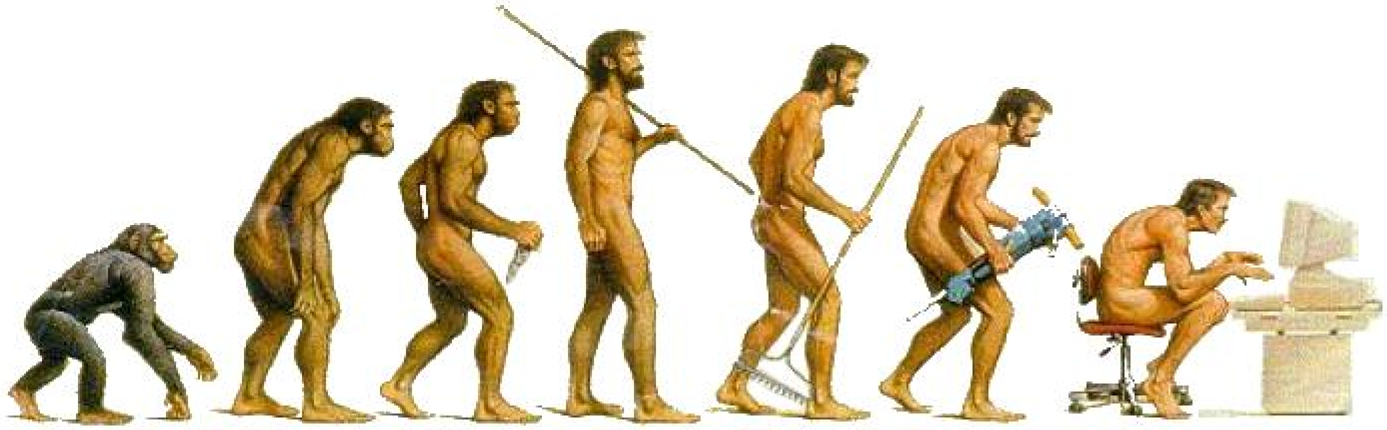
```
function TABU-SEARCH( problem ) returns a solution state
  inputs: problem, a problem

  current ← MAKE-NODE( problem.INITIAL-STATE)
  best ← current
  T ← Tabu list
  while ( termination criterion not satisfied ) do
     current ← a highest-valued successor of current legal wrt. T
     if  current.VALUE > best.VALUE then
        best ← current
        ADD( T , ACTION-TO( current ) )
  return best
```

Tabu Search Variations

- Tabu list
  - Variable length
  - Aspiration criteria (override tabu, e.g. if improving move)
- Probabilistic tabu search
  - Only consider a random sample of the neighborhood

Genetic Algorithms

Evolution



(or is it?)

Genetic Algorithms

- Individual: A variable assignment ("Genome")
  - Often represented by a bit string
- Population: *n* individuals
  - Initialize to randomly generated states
- Fitness function: Evaluates the "fitness" of an individual
- Selection: Identify the most fit members of a population
- Crossover: Form new individuals out of multiple individuals
- Mutation: Randomly change a value in an individual's genome

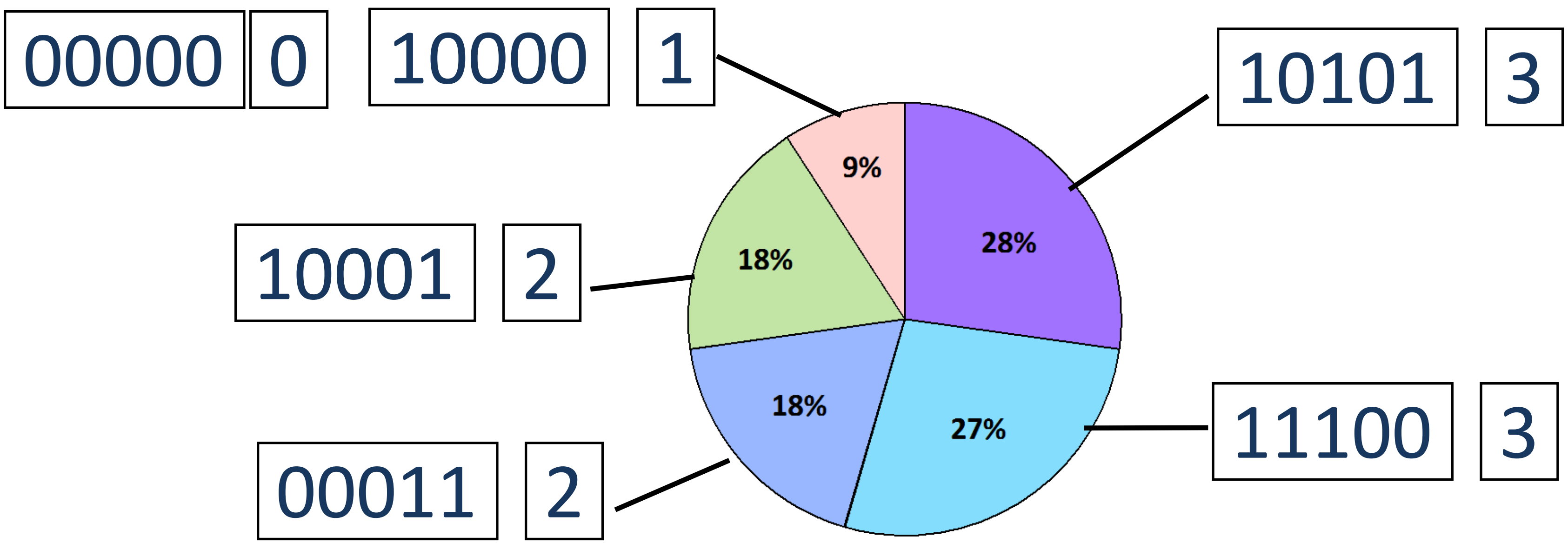
Example - 1 max

- Genome: bit string of length 5
- Fitness function: $f(x)$ = # of 1s in the bit string
  - e.g. $f(00110) = 2$, $f(111111) = 5$
- Goal: Maximize $f(x)$
- Step 1: Initialize population

10000	10101	10001	00011	11100	00000

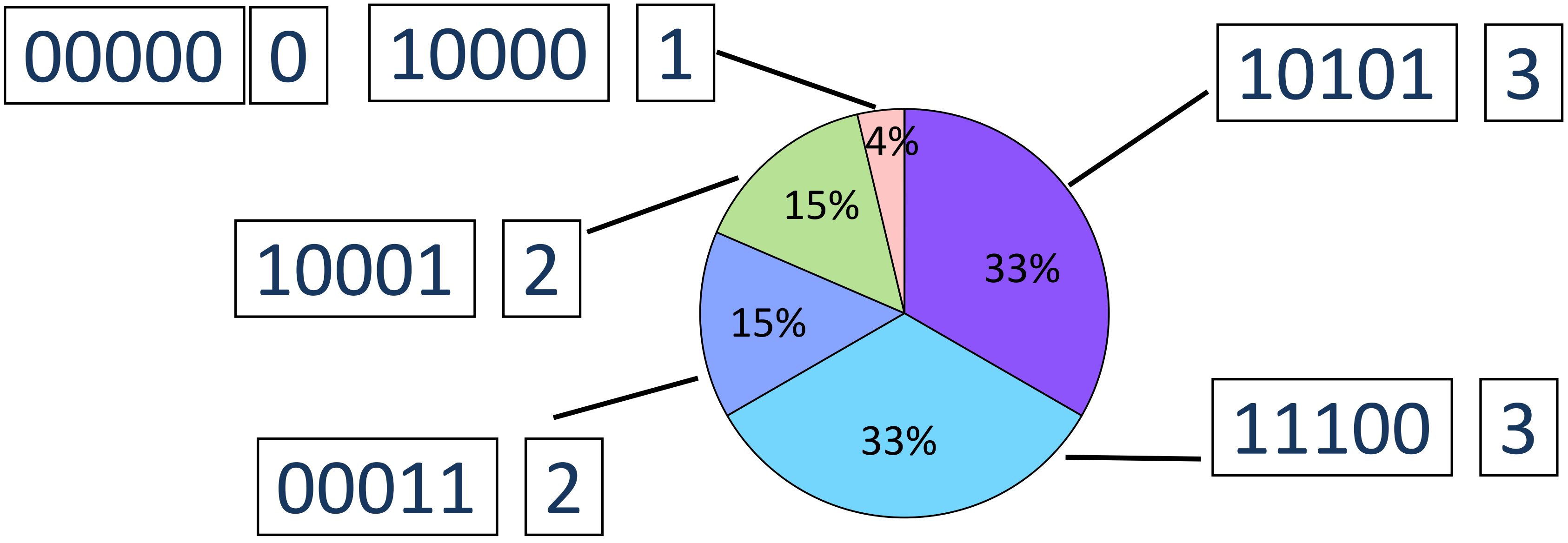
Example - 1 max

- Step 2: Evaluate the population's fitness
- Step 3: Selection. (Roulette wheel selection)
  - Crossover genomes with probability proportional to their fitness



Example - 1 max

- Fitness scaling can improve performance.
  - Square (or cube) the fitness of each individual before performing selection



Example - 1 max

- Select genomes and perform crossover.

Genome	Fitness
10101	3
11100	3
10001	2
00011	2
10000	1
00000	0

10101 → 101|01 → 10111 (Fitness: 4)

00011 → 000|11 → 00001 (Fitness: 1)

Pick random crossover point

Generate children

Example - 1 max

- Continue selection until new population is formed
  – Individuals are often allowed to be part of multiple crossovers.
- Step 4: Mutation
  – With some probability, usually < 0.1 make a small change in the genome

$\boxed{00001} \longrightarrow \boxed{10001}$

Flip random bit

Example - 1 max

- Population at the end of the generation:

10111	4
11100	3
10001	2
10001	2
10000	1
00000	0

- Unless a termination criteria has been reached, continue with the next generation.

Genetic Algorithms in Practice

- Necessary that "Genome" forms meaningful components of the problem
- Number of crossovers: 1/2 * population
  - Could mean some individuals crossover more than once
- Population size difficult to determine
  - Between 25 and 100, depending on the problem
- Terminate criteria vary
  - Little change in average fitness of the population over last n generations, where $n \approxeq 5$
- Choice of crossover operator extremely important
  - Single point vs. multiple point, etc.

Constraint Based Local Search

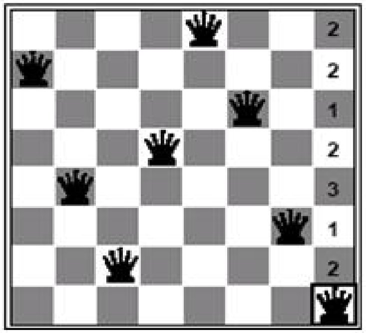
Constraint Based Local Search

- Initial state: random or greedy assignment process
- In this case constraint satisfaction problems
  - allow states with unsatisfied constraints
  - operators **reassign** variable values
- Variable selection: randomly select any conflicted variable
- Value selection: *min-conflicts heuristic*
  - Select new value that results in a minimum number of conflicts with the other variables

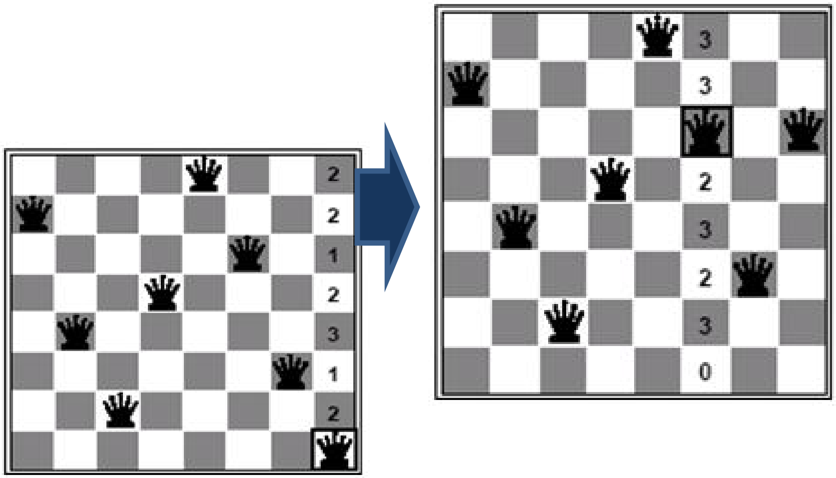
Min-conflict algorithm

```
function MIN-CONFLICT(csp, max_steps) returns a solution or failure
    inputs: csp, a constraint satisfaction problem
            max_steps, the number of steps allowed before giving up

    current ← an initial complete assignment for csp
    for i = 1 to max_steps do
        if current is a solution for csp then return current
        var ← a randomly chosen conflicted variable from csp.VARIABLES
        value ← the value v for var that minimizes CONFLICTS(var,v,current,csp)
        set var = value in current
    return failure
```

Min-conflict algorithm



Min-conflict algorithm



Min-conflict algorithm