THE UNIVERSITY OF ADELAIDE
AUSTRALIA

Particle Swarm Optimization, and Other Heuristics

Particle Swarm Optimization

- Particle Swarm Optimization belongs to the field of Swarm Intelligence
- Is another type of populated-based algorithm
- Mimics social behavior of natural organisms such as bird flocking and fish schooling to find a place with enough food.
- Coordinated behavior using local movements emerges without any central control.

Neighbourhood

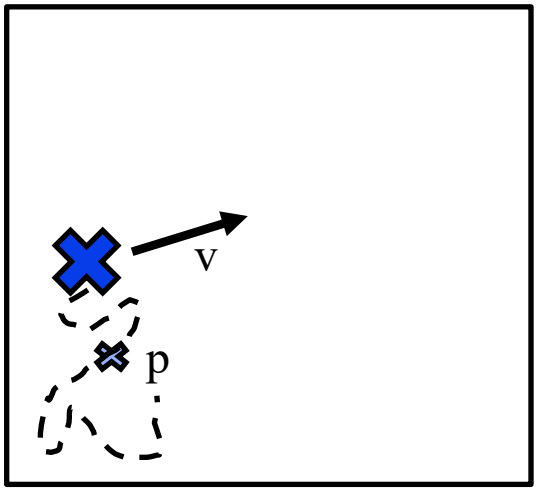
Define a neighborhood for each particle. Neighborhood denotes the social influence between the particles.

Two common methods:

- Gbest Method: Neighborhood is defined as the whole population of particles.
- Lbest Method: A topology is associated with the swarm. Neighborhood is the set of directly connected particles.

Neighbourhood

Depending on the neighborhood, a leader (Lbest, or Gbest) guides the search towards (hopefully) better regions of the search space. We consider Gbest in the following.



Components of a particle:

- Decision vector x
- Vector p which records the best location found so far by the particle.
- Velocity vector v which gives the direction where the particle will travel if not influenced by other particles.

Velocity Update, Option 1

The velocity defines the amount and direction of change that will be applied to the particle.

Formula for velocity update:

$$v_i(t) = v_i(t-1) + \rho_1 C_1 \times (p_i - x_i(t-1)) + \rho_2 C_2 \times (p_g - x_i(t-1))$$

ρ_1, ρ_2: random values in $[0, 1]$

C_1, C_2: constants called learning factors

p_g: position of best neighbour

Velocity Update, Option 2

The velocity defines the amount and direction of change that will be applied to the particle.

Formula for velocity update:

$$v_i(t) = w \times v_i(t-1) + \rho_1 \times (p_i - x_i(t-1)) + \rho_2 \times (p_g - x_i(t-1))$$

w: inertia weight which controls the impact of previous veloctiy

ρ_1, ρ_2: random values in $[0, 1]$

p_g: position of best neighbour

Velocity Update, Option 2

The velocity defines the amount and direction of change that will be applied to the particle.

Formula for velocity update:

$$v_i(t) = w \times v_i(t-1) + \rho_1 \times (p_i - x_i(t-1)) + \rho_2 \times (p_g - x_i(t-1))$$

cognitive component

social component

Updates

Update of position:

$x_i(t)=x_i(t-1)+v_i(t)$

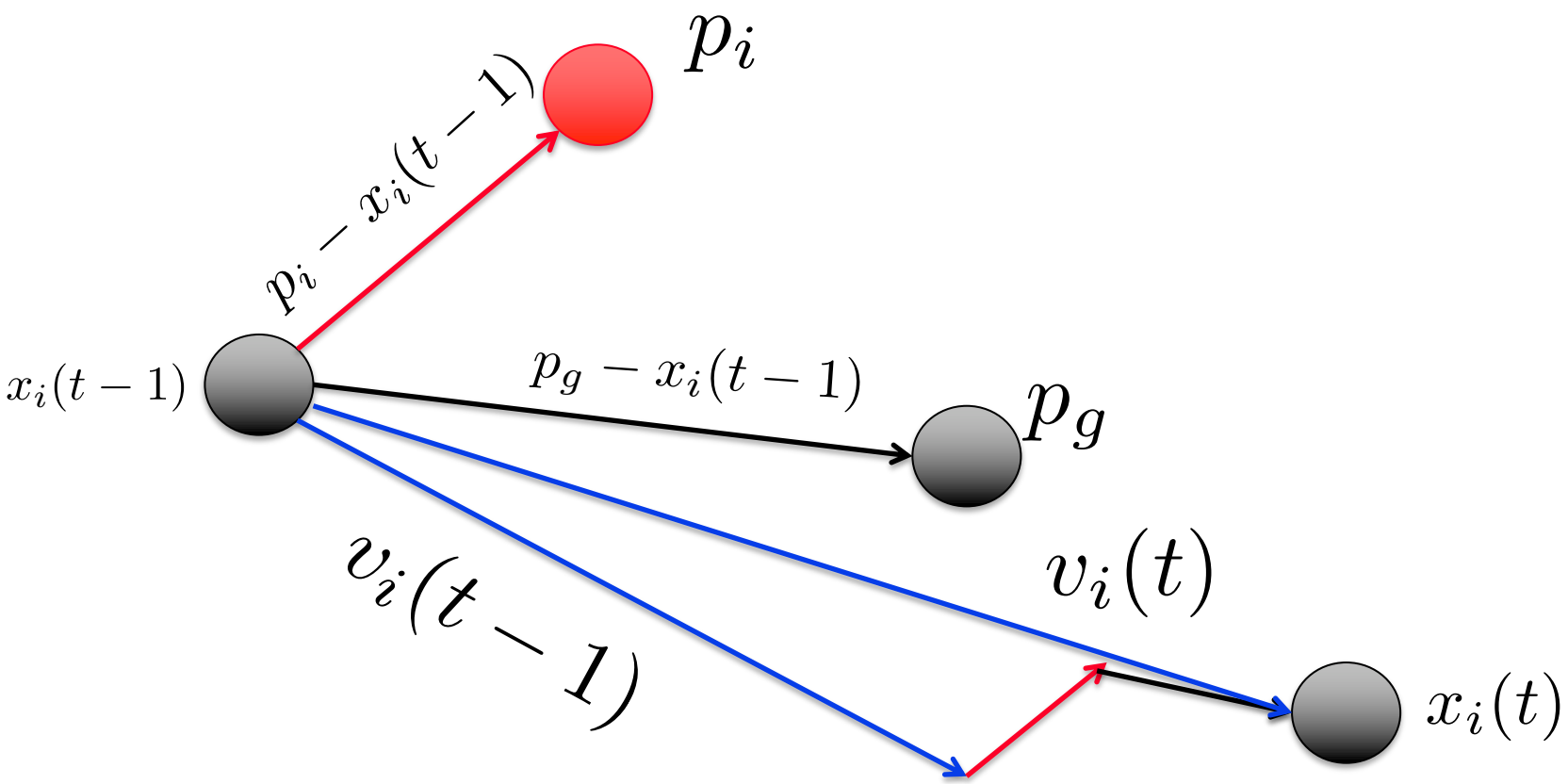
Update of local best solution (in case of minimisation):

If $f(x_i)<f(p_i)$ then $p_i=x_i$

Update of global best solution:

If $f(x_i)<f(p_g)$ then $p_g=x_i$

Movement of particle



Template for PSO Algorithm

Random initialization of the whole swarm $\{x_i(0), v_i(0) \mid 1 \leq i \leq \mu\}$
Compute $f(x_i)$ $1 \leq i \leq \mu$, set $p_i = x_i(0)$, $p_g = \arg\min_{1 \leq i \leq \mu} f(x_i(0))$
$t := 0$;
while termination condition is not fulfilled

Note: this is for the minimisation of f(x)

- $t := t + 1$;
  - for each particle i, $1 \leq i \leq \mu$
    - Update velocity:
      $v_i(t) = w \times v_i(t-1) + \rho_1 \times (p_i - x_i(t-1)) + \rho_2 \times (p_g - x_i(t-1))$
    - Update position: $x_i(t) = x_i(t-1) + v_i(t)$
    - If $f(x_i) < p_i$ then $p_i = x_i$.
    - If $f(x_i) < p_g$ then $p_g = x_i$.

Other Heuristics

- All methods presented so far were motivated by processes observed in nature.
- Often they worked at each time step with a set of solutions (population)
- Consider now some other classical heuristic approaches to solve optimization problems.

Local Search

Neighborhood

- Neighborhood function (in combinatorial optimisation):
  N: S $\rightarrow 2^S$ (2^S is the powerset of S)
  - Specifies for each s $\in$ S a neighborhood N(s)

Local Search

- s :=some initial solution.
- repeat
  - generate an $\tilde{s} \in N(s)$
  - if $f(\tilde{s}) < f(s)$ then $s := \tilde{s}$.
- until $f(\tilde{s}) \geq f(s)$ for all $\tilde{s} \in N(s)$.

Example

Traveling Salesman Problem:

- Given a complete graph G=(V,E) on n vertices and a cost function c: E → R_+.
- Find a tour of minimal cost that visits each vertex exactly once.

Local operations on permutations

Possible:

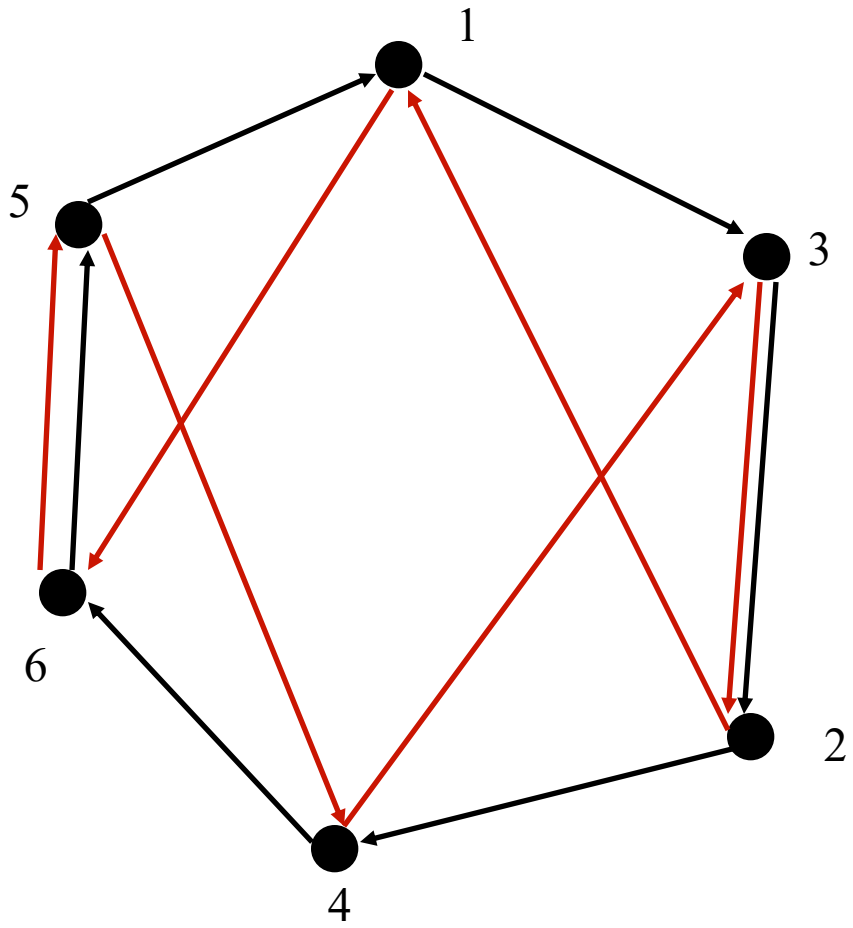
exchange(i,j): exchanges the elements on position i and j in the permutation. Neighborhood contains all possible exchange operations.

jump(i,j): jumps element at position i to position j and shifts elements between them in the appropriate direction.

Neighborhood contains all possible jump operations.

inversion(i,j): inverts the sequence of elements from position i to position j. Neighborhood contains all possible inversion operations.

permutation (5,1,3,2,4,6)

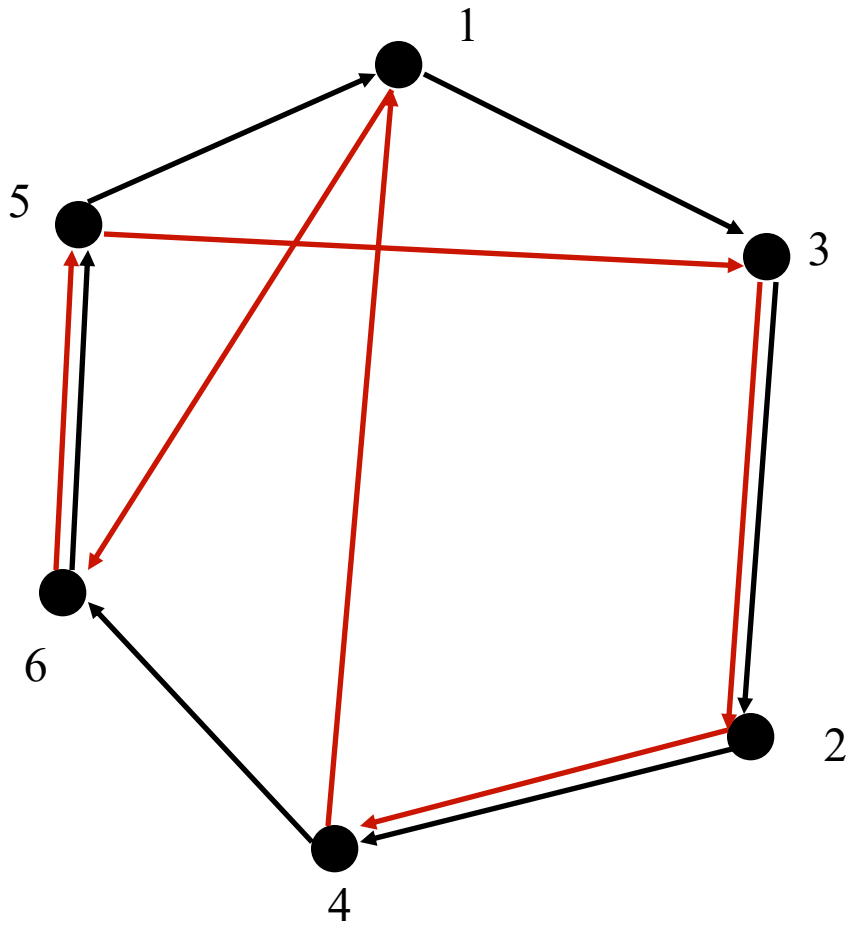


exchange(2,5)

(5,4,3,2,1,6)

Exchanges 4 edges

permutation (5,1,3,2,4,6)

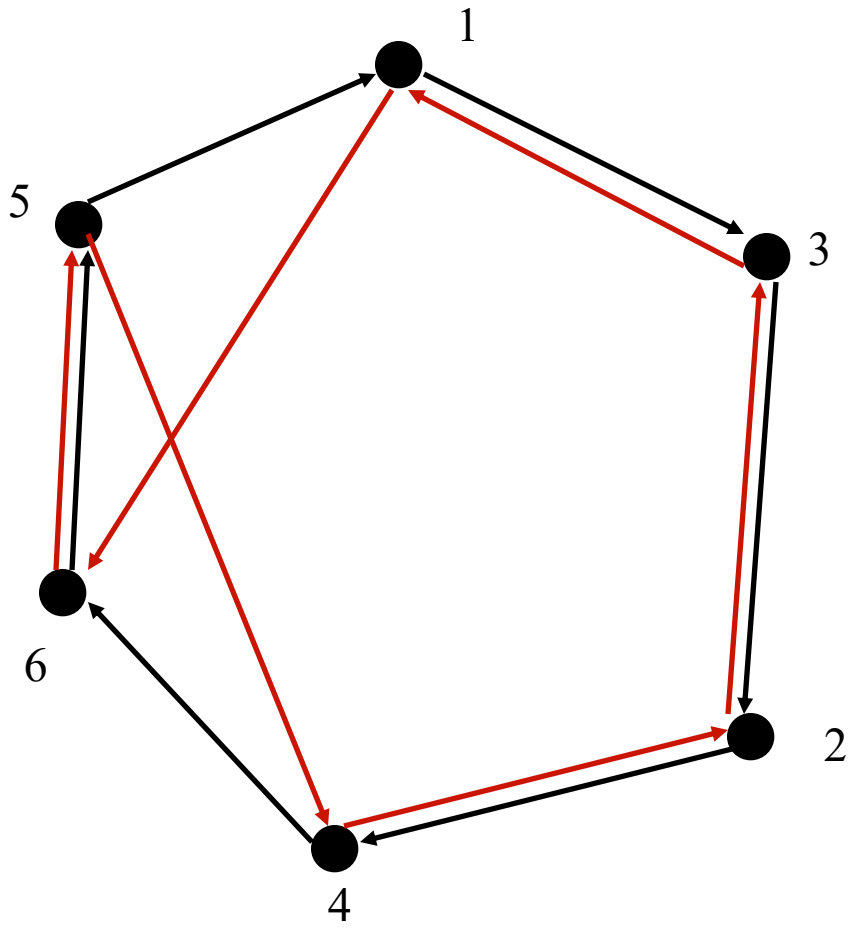


jump(2,5)

(5,3,2,4,1,6)

Exchanges 3 edges

permutation (5,1,3,2,4,6)



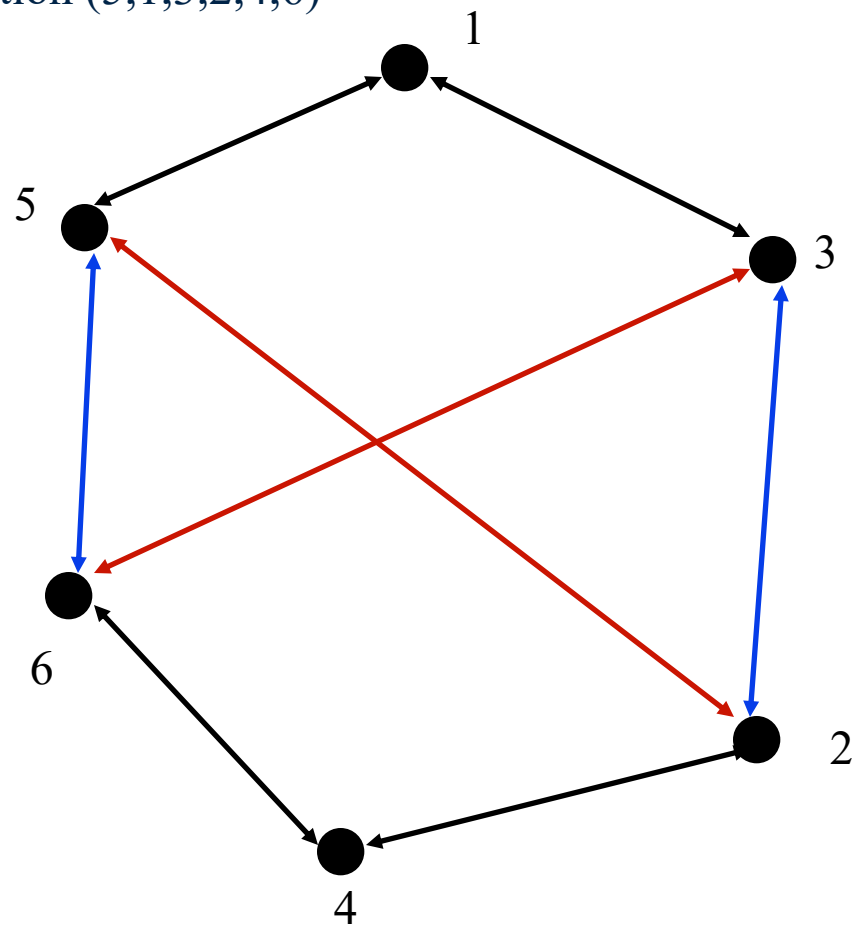
inversion(2,5)

(5,4,2,3,1,6)

Symmetric TSP:
Exchanges just 2 edges

Symmetric TSP

permutation (5,1,3,2,4,6)



2-opt step

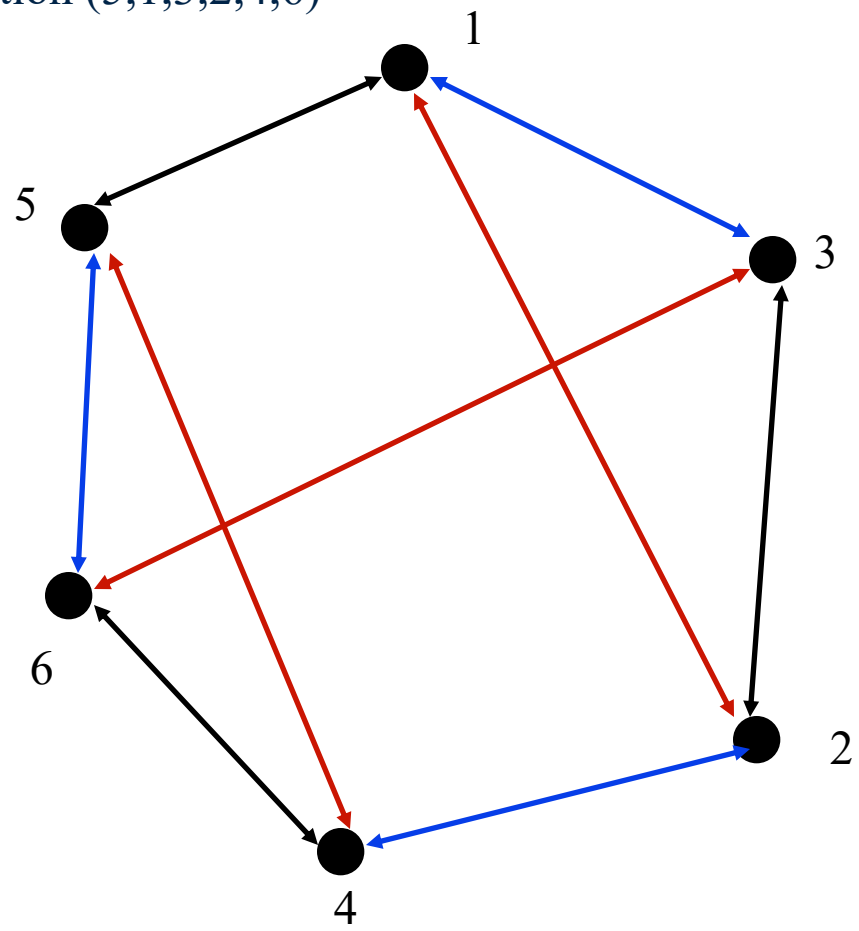
Delete 2 edges

Introduce 2 edges such that a another tour is obtained

Neighborhood defined by all possible operations

Symmetric TSP

permutation (5,1,3,2,4,6)



3-opt step

Delete 3 edges

Introduce 3 edges such that a another tour is obtained

Neighborhood defined by all possible operations

Design of Local Search

What is a good representation of possible solutions

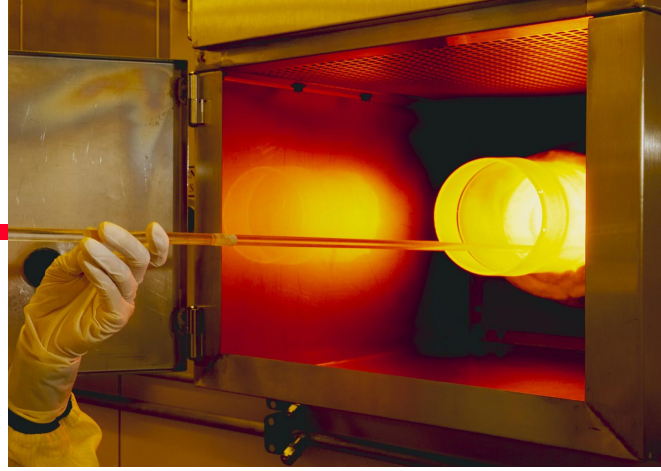
How should the neighborhood look like?

Small neighborhood: small computational costs but more likely to get stuck in local optima

Large neighborhood: fewer local optima but computational costs are higher

Another way to deal with local optima: Accept inferior solutions with some probability (e.g., "Simulated Annealing")

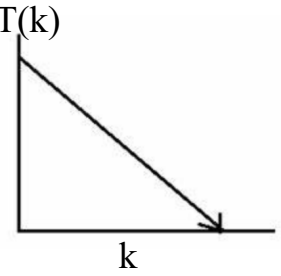
Simulated Annealing



Simulated annealing

- s :=some initial solution.
- k:=1
- repeat
  - generate an $\tilde{s} \in N(s)$
  - if $f(\tilde{s}) < f(s)$ then $s := \tilde{s}$.
  - else
    - with probability $e^{\frac{f(s)-f(\tilde{s})}{T(k)}}$ set $s := \tilde{s}$
  - k:= k+1
- until stopcriterion.

T(k): temperature at time k (decreases overtime)



➔ The probability to replace the current solution with a worse one decreases over time.
For example, assuming a difference of -100 units:
e^(-100/1000) = 90%
e^(-100/100) = 37%
e^(-100/10) = 0%

Summary

- Particle Swarm Optimization
- Local Search
- Simulated Annealing

One word of warning (next slide)

- Tabu Search
- Branch and Bound
- (Integer) Linear Programming

A Word of Warning

There has been a gold-digger phase, where too many (?) concepts have been designed… and new ones are still coming onto the market. Is this realistic?

https://github.com/fcampelo/EC-Bestiary lists about 100 or so different "nature-inspired" algorithms, such as algae, buffaloes, hawks, whales, zombies, …

Markus and many others recommend:

When you come across a concept that was not mentioned in this course, ask yourself: what is the core component that the algorithm has that has not already been covered in the literature? And: is a new name really necessary?

→ We are not talking about terminology (cats vs frogs) but about fundamental concepts.

Note: showing equivalence is a process that is time-consuming, difficult, and unrewarding.

Having said this: huge parts of the community are very healthy and thriving.
→ Stick to the people who publish at the top conferences.

Check this community effort: https://arxiv.org/abs/2011.09821

Summary

- Particle Swarm Optimization
- Local Search
- Simulated Annealing

One word of warning (next slide)

- Tabu Search
- Branch and Bound
- (Integer) Linear Programming

Tabu Search

Similar to local search

Work with one single search point at each time step

Go to the best solution in the neighborhood

May be worse than the current solution.

May create cycles / Shift between solutions.

Use a tabu list for solutions that should not be visited again. (e.g. to prevent cycles)

Tabu Search

Idea:

- Use of short term memory
- Make the best possible move among available choices
- Tabu restrictions should help to escape local optima
- Tabu restriction should avoid cyclic behavior

Tabu Search

Choose the best possible move

- Let N(x) be the set of neighbours of the current solution x
- Choose a solution y $\in$ N*(x) with best function value.
- N*(x) is usually a subset of N(x) (short term memory)
- N*(x) may be an expansion of N(x) (based on long term memory)

Choice of N*(x) is crucial for the success of tabu search

Additional reading material "Principles of Tabu Search" by Glover and Marti
http://www.uv.es/rmarti/paper/docs/ts1.pdf and
http://www.cleveralgorithms.com/nature-inspired/stochastic/tabu_search.html

Tabu Search

Memory:

- Recency (keep track of solutions or solution attributes that have changed in the recent past)
- Frequency (transition memory that counts how often a component changes its value)
- Quality (differentiate the merit of solutions)
- Influence (impact of choices made during the search)

Recency-Based Memory

Keep track of solution components that have changed during the "recent" past.

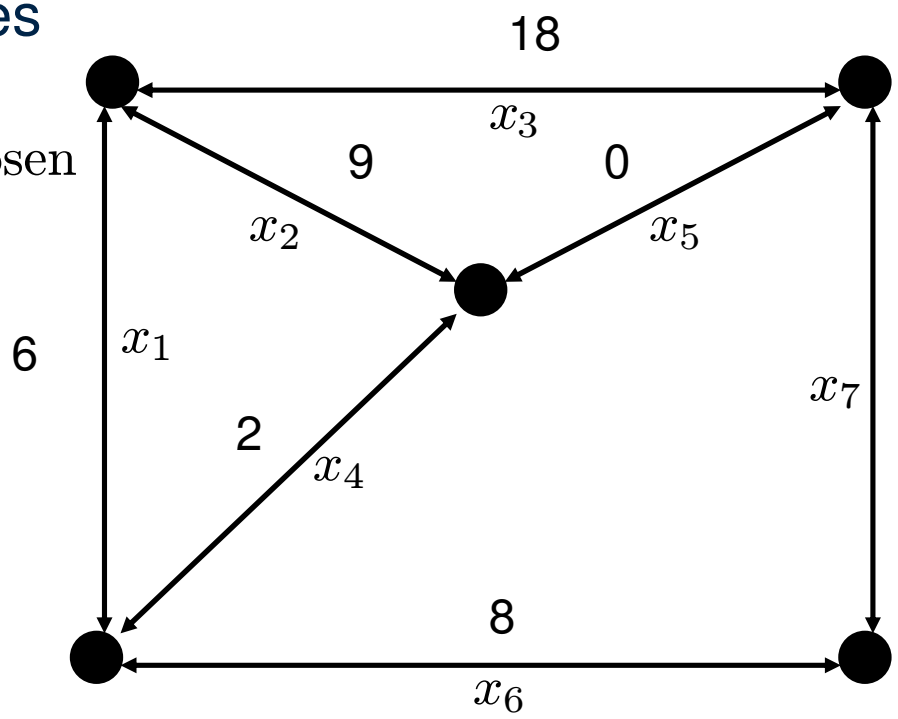
Mark such components as tabu active

Prevents certain solutions from the past to belong to N*(x)

Other solutions that share the tabu active components are excluded as well.

Example: Minimum Spanning Tree with additional restrictions on the edges

x_i iff edge e_i is chosen



Constraints

$x_1 + x_2 + x_6 \leq 1$

$x_1 \leq x_3$

Find spanning tree of minimal cost that obeys given constraints.

Penalty 50 per unit of constraint violation

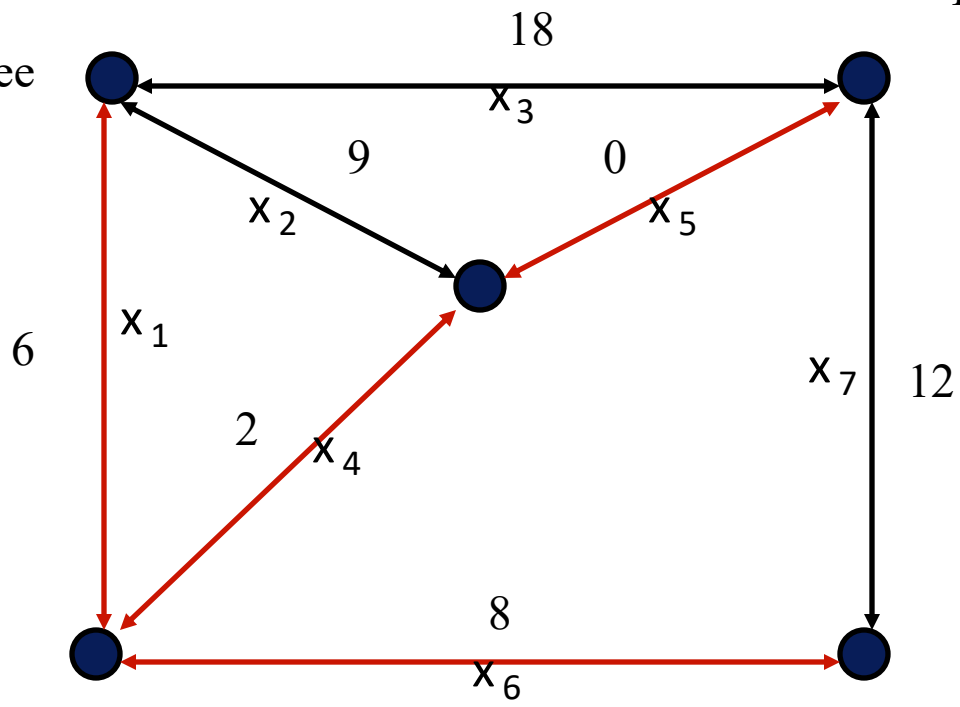
Start with minimum spanning tree

Tabu: Do not remove the last 2 introduced edges unless the best value is improved.

Edge cost 16

Violates 2 constraints

Total cost 16+100=116



Best value 116

$$x_1 + x_2 + x_6 \leq 1$$
$$x_1 \leq x_3$$

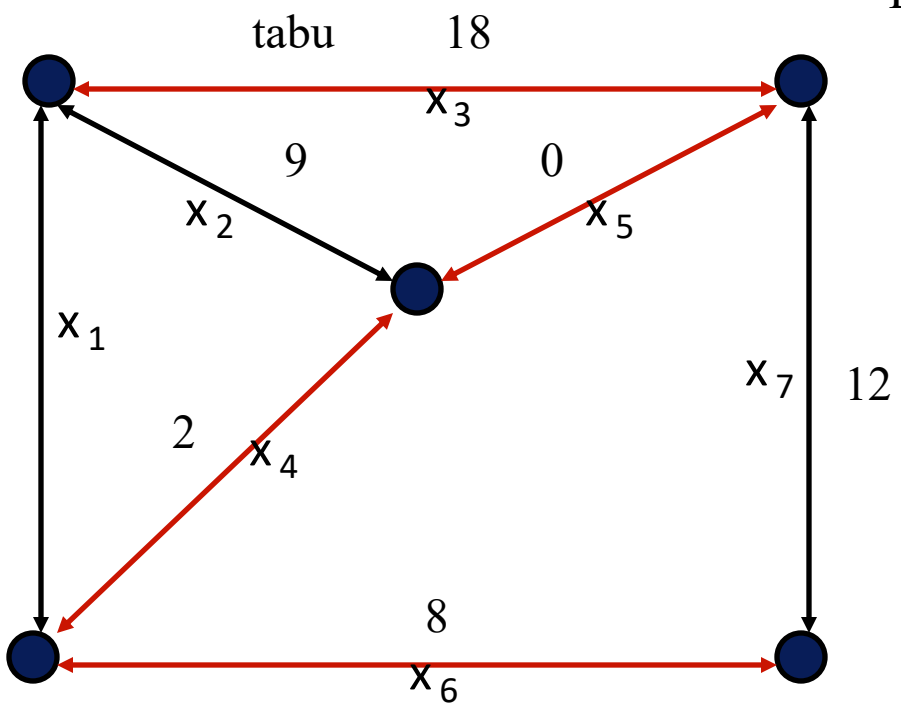
Least cost admissible "edge exchange"

Best value 28

Edge costs 28

Violates no constraint

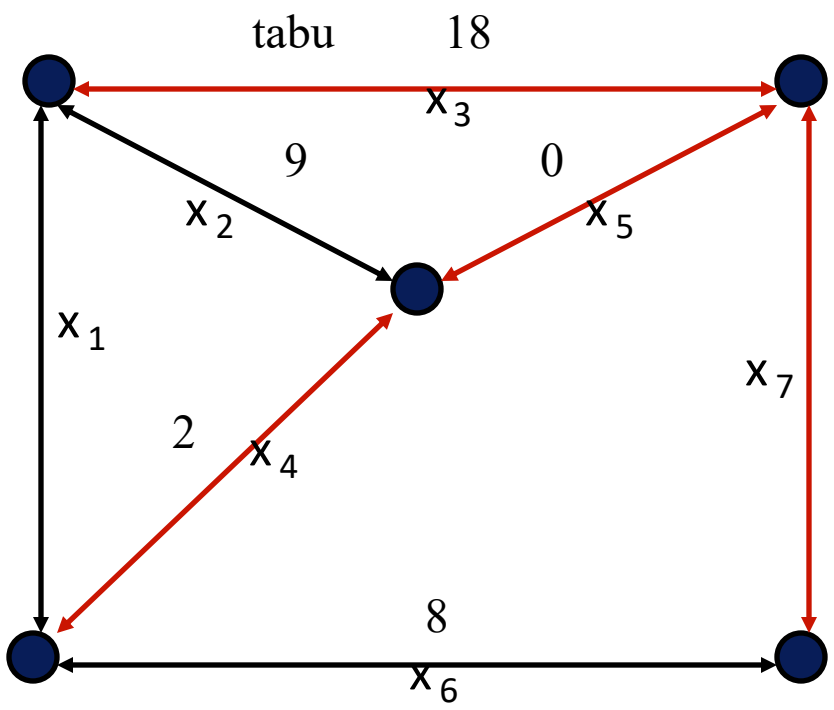
Total cost
28



$$x_1 + x_2 + x_6 \leq 1$$
$$x_1 \leq x_3$$

Least cost admissible "edge exchange"

Best value 28



Edge costs 32

Violates no constraint

Total cost
32

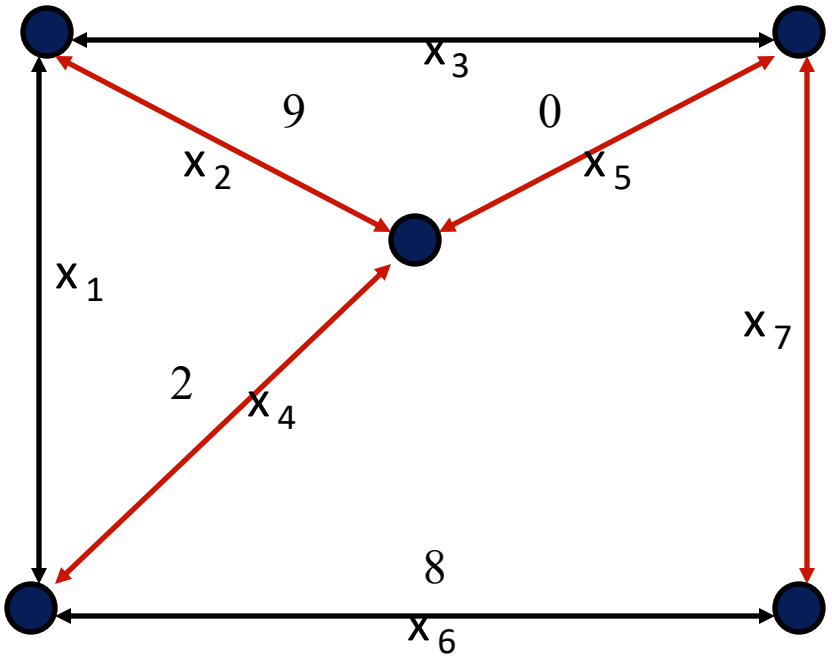
$x_1 + x_2 + x_6 \leq 1$
$x_1 \leq x_3$

Least cost admissible "edge exchange"

Drop, new best solution

Best value 28

tabu 18



Edge costs 23

Violates no constraint

Total cost
23

$$x_1 + x_2 + x_6 \leq 1$$
$$x_1 \leq x_3$$

In fact global optimum

Branch and Bound

- Local Search, Simulated Annealing, Evolutionary algorithms, and Tabu Search do not give guarantees to obtain an optimal/good solution
- Branch and Bound is an exact method
- We consider maximization problems now (method can be easily adapted to minimization)
- Branching divides given problem into subproblems
- Bounding computes lower bounds on the value of an optimal solution (using maximization)

Branch and Bound

Branching:

- Consider problem P
- Let $X(P_0)$ be the set of feasible solutions for P
- Divide it into subproblems $P_1, P_2, \ldots, P_k$ such that

$$X(P_0) = \bigcup_{i=1}^{k} X(P_i)$$

and if possible $X(P_i) \cap X(P_j) = \emptyset,\ i \neq j$

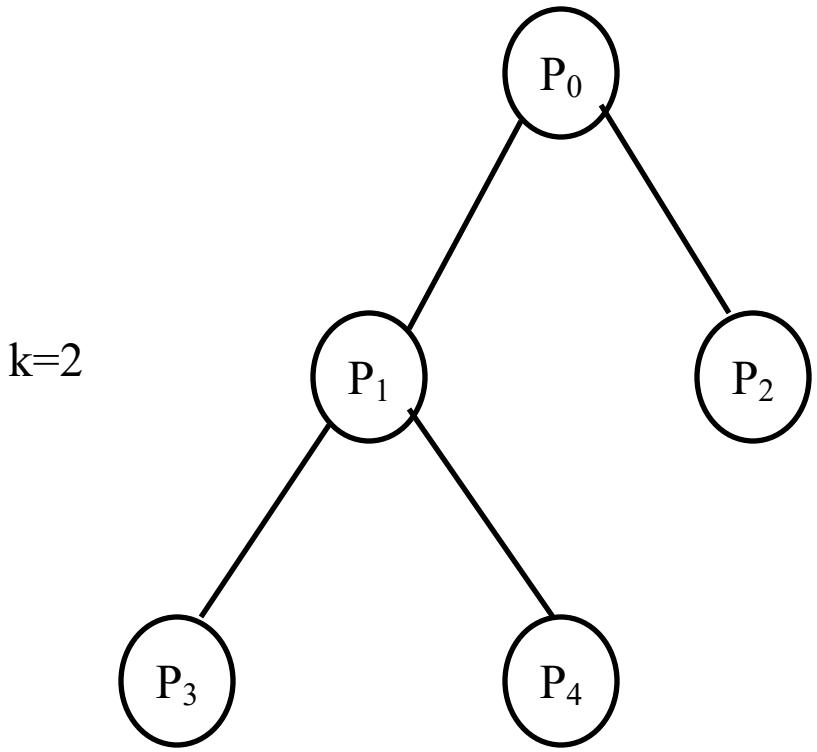
- Problems $P_1, P_2, \ldots, P_k$ can be further divided into subproblems

Branch and Bound

Branch and Bound:

- Start with a solution (i.e. by some heuristic)
- Rules for relaxation
- Rules to determine subproblems
- Rules to select the order for branching

Example for Branching



Bounding

Bounding:

- Compute lower bound L of the value of an optimal solution
- Compute upper bounds for subproblems
- Decide whether further branching is necessary

Lower bound L:

- Initial lower bound may be obtained by a heuristic
- During the run, the lower bound is given by the best solution found so far.

Bounding

Upper bound for subproblems:

- Compute upper bound U_i on the value of an optimal solution for problem P_i by solving a relaxation $P_i^{\sim}$ of P_i.
- Relaxation should be much easier to solve
- It holds:

$$X(P_i) \subseteq X(\tilde{P}_i)$$

Bounding

Consider problem P_i

No further branching:

- If $U_i \leq L$: The optimal solution of the subproblem can not be better than the best so far solution
- $U_i > L$ and solution for $P_i^{\sim}$ is feasible for P_i: new best solution, set $L = U_i$
- $P_i^{\sim}$ has no feasible solution: It follows

$$X(P_i) = \emptyset$$

Linear Programming

Standard form:

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

subject to

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \text{ for } i = 1, 2, \ldots, m$$

$$x_i \geq 0 \text{ for } i = 1, 2, \ldots, n$$

LP can be solved in polynomial time

Integer Linear Programming

Standard form:

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

subject to

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \text{ for } i = 1, 2, \ldots, m$$

$$x_i \geq 0 \text{ and integer for } i = 1, 2, \ldots, n$$

ILP is NP-hard

Example

$$\max x_1 + 2x_2$$

subject to

$$x_1 + 3x_2 \leq 7$$
$$3x_1 + 2x_2 \leq 10$$

$$x_1,\ x_2 \geq 0 \text{ and integers}$$

$(x_1, x_2) = (0, 0)$ is feasible: $L = 0$

Use relaxation of the variables to compute upper bounds

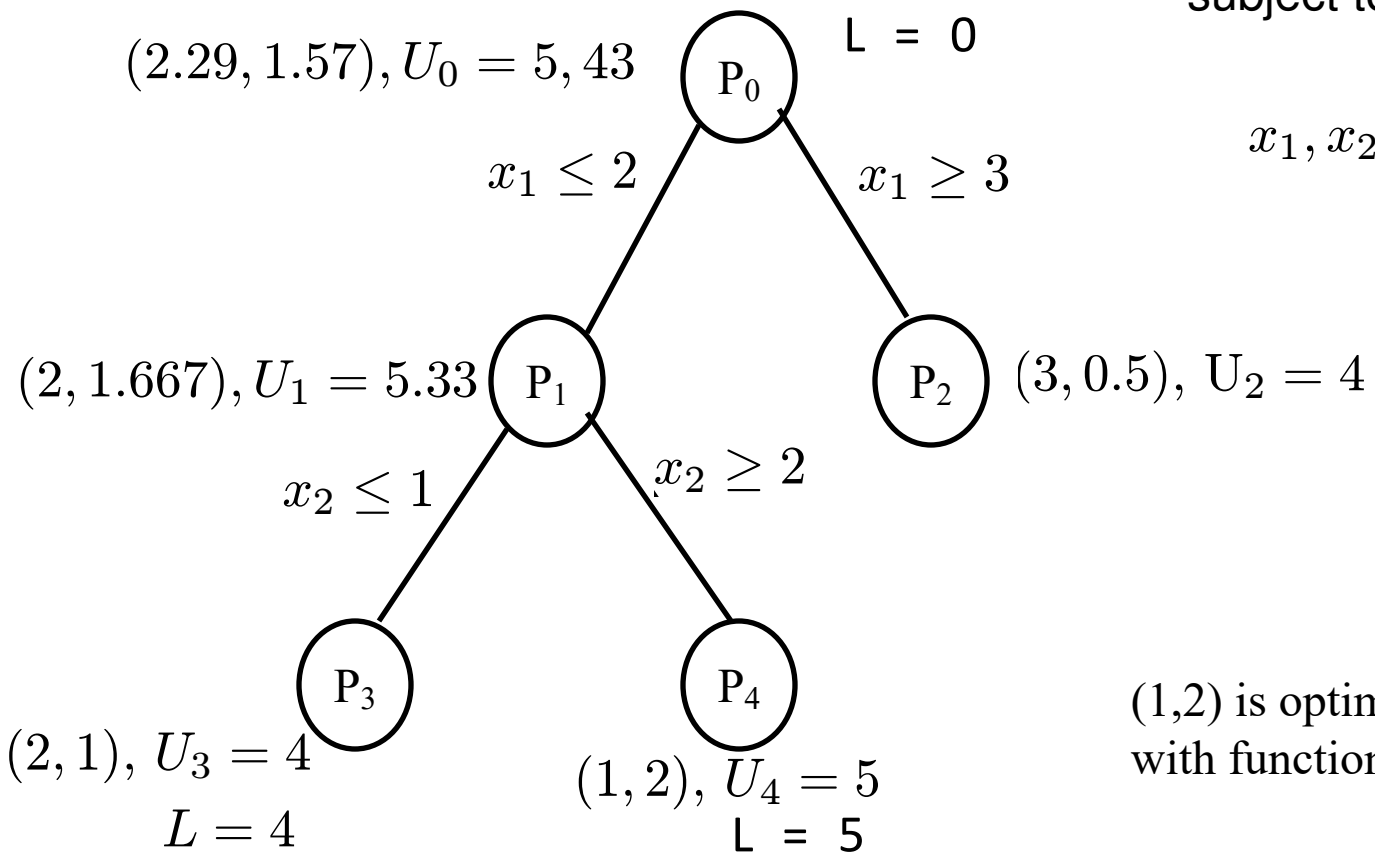
Variables do not have to take on integer values

Example

$$\max x_1 + 2x_2$$

subject to

$$x_1 + 3x_2 \leq 7$$
$$3x_1 + 2x_2 \leq 10$$
$$x_1, x_2 \geq 0 \text{ and integers}$$



(1,2) is optimal solution with function value 5.

Summary

- Particle Swarm Optimization
- Local Search
- Simulated Annealing

One word of warning (next slide)

- Tabu Search
- Branch and Bound
- (Integer) Linear Programming