Algorithms and Data Structures
Module 4. NP-hard problems

Lecture 21

Inexact algorithms. Part 2.

TSP: solving (lecture 18)

Possible options for solving any NP-hard problem (e.g. TSP):

- Exactly but inefficiently:
  - ✓exhaustive search (brute-force, backtracking)
  - ✓smart search (branch-and-bound)
- Exactly, efficiently, but not universally:
  - ✓efficiently solvable special cases.
- Efficiently but inexactly:
  - ✓approximate algorithms,
  - ✓heuristics

Heuristics for TSP

There are many ideas/principles that can be used for designing a heuristics algorithm.

We will study just two of them:

- Greedy heuristics.
- Local search.

Local search

Principle scheme of a local search algorithm:

- Build the initial feasible solution.
- Improve the current solution iteratively, applying transformations from a predefined set.
- Stop when no improvement is possible.

Thus, a local search algorithm builds a *local minimum*.

Local search

Local search has some valuable advantages:

- *Universality*, generality. The general scheme can be easily adapted for most optimization problems.
- *Anytime* mode. We can stop the algorithm anytime, yielding the current solution. Thus, the algorithm utilizes as much time as we have, to build as good solution as it can within the given time.

Local search

Let X be a set of all representations that include both feasible and infeasible 'solutions'.

Let $S \subseteq X$ be the set of feasible solutions.

We should select a set of transformations $F = \{f_i: S \longrightarrow S\}$.

For a current solution $s \in S$, the set $N(s) = \{f_i(s): f_i \in F\}$ is called the ***neighborhood*** of s.

To design a good local search algorithm we should find a reasonable trade-off between the quality of the final solution (the larger F, the better) and the velocity of the algorithm.

Local search

The general scheme of LS for minimization problem.

1. Build the initial solution $s \in S$. We can use a greedy algorithm or any other heuristics.
2. For the current solution s, analyze its neighborhood.
3. If there is $s' \in N(s)$ such that $w(s') < w(s)$, then make s' the current solution $(s = s')$ and go to 2.

   Otherwise, stop and return s as the final solution.

Different LS algorithms for a certain problem differ in the set of transformations and the rule of selecting neighbor solution.

Local search

Let us study several LS heuristics for symmetric TSP.

1. Transpositions.

Feasible solutions are represented as vertex permutations. The set of transformations consists of *transpositions*, i.e. permutations that swap two items and keep other items fixed.

E.g., for feasible solution s = (1,2,3,4,5,6,7), its neighborhood contains cycles (1,2,4,3,5,6,7), (5,2,3,4,1,6,7), (1,7,3,4,5,6,2) and so on.

The neighborhood cardinality is $\frac{n(n-1)}{2} = O(n^2)$.

Local search

2. *k*-opt.

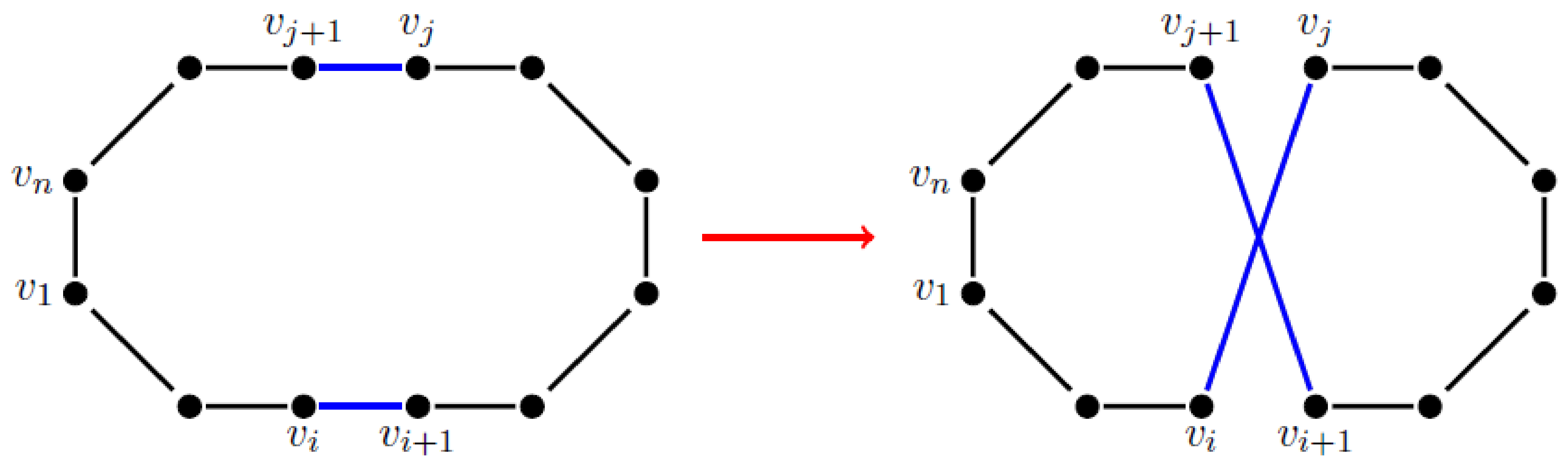
For integer *k*, a transformation is a procedure of the following kind:

1. Select *k* non-adjacent edges in the current solution.
2. Delete the selected edges from the current cycle.
3. Add *k* new edges that connect the endpoins of the deleted edges in a different way.

Local search

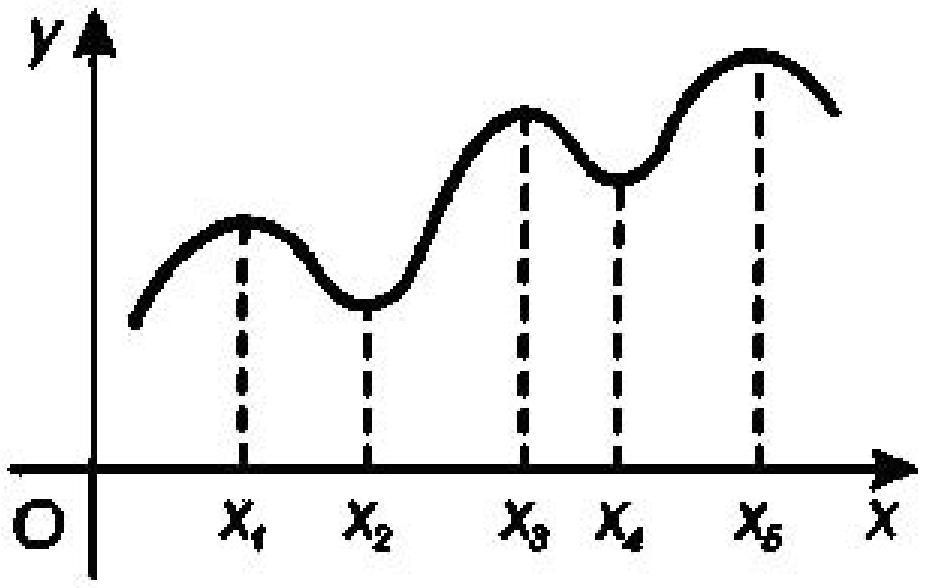
2-opt or 3-opt transformations are usually used.

An example of a 2-opt transformation:



Local search

The standard local search scheme (‘gradient descent’) has an important disadvantage: it is subject to stucking in local optima.



Local search

Possible ways of overcoming this disadvantage allow the algorithm to move to a solution which is worse than the current solution.

a) Deterministic choice: let the algorithm move to the best neighbor even if it is worse than the current solution.

b) Randomized choice.

Local search

Deterministic choice: let the algorithm move to the best neighbor even if it is worse than the current solution.

Issue: the algorithm can (and usually does) loop infinitely.

Solution to this issue: ***tabu search***.

Tabu search

In order to overcome looping, we prohibit moving to recently visited solutions or applying recently applied transformations.

The *tabu list* stores the given number (a predefined parameter) of recent solutions or transformations.

The size of tabu list should be defined empirically.

Tabu search

```
TabuSearch:
T := []
s := InitialSolution()
while (condition):
      s' := the best of N(s) \ T
      Add s' to T
      s := s'
```

Tabu search

In order to overcome looping, we prohibit moving to recently visited solutions or applying recently applied transformations.

The *tabu list* stores the given number (a predefined parameter) of recent solutions or transformations.

The size of tabu list should be defined empirically.

Local search

Randomized version of LS: use random choices to let the algorithm get out from the local minimum vicinity.

a) Random choice of the initial solution.

b) Randomized choice of the next solution within the current neighborhood.

Local search

a) Random choice of the initial solution.

```
RandomizedLocalSearch:
Record = NULL
for i=1 to IterCount:
    s := RandomSolution(x)
    s' := LocalSearch(s)
    if c(s') < c(Record)
        Record := s'
```

Local search

b) Randomized choice of the next solution within the current neighborhood.

For each $s' \in N(s)$ we assign the probability of moving to s'. This probability depends on $w(s')$, but is non-zero for any feasible solution.

Examples of algorithms of such type: Metropolis algorithm, Simulated Annealing.