HUST

ĐẠI HỌC BÁCH KHOA HÀ NỘI

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

ONE LOVE. ONE FUTURE.

ĐẠI HỌC
BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY
OF SCIENCE AND TECHNOLOGY

PLANNING OPTIMIZATION

Heuristic methods

ONE LOVE. ONE FUTURE.

CONTENT

- **Greedy algorithm**
- Local search

Overview

- Exact methods (Constraint Programming, Branch and Cut, ..) cannot handle large-scale combinatorial optimization problems
- In practice, high quality solutions found in a reasonable computation time are required

Overview of greedy methods

- **S**: a solution is represented by a set of components
- **C**: set of candidates of components to be added to the solution
- **select(C)**: select the most promising component among candidates
- **solution(S)**: return true if S is a solution to the original problem
- **feasible(S)**: return true if S does not violate any constraints

```
Greedy() {
  S = {};
  while C ≠ ∅ and
        not solution(S){
    x = select(C);
    C = C \ {x};
    if feasible(S ∪ {x}) {
      S = S ∪ {x};
    }
  }
  return S;
}
```

TSP

- Given n points 1, 2, …, n in which d(i,j) is the distance from point i to point j. Find the shortest closed tour starting from 1 visiting other points and terminating at 1 such that the total travel distance is minimal

TSP

- Greedy idea
  - The tour is initialized by point 1
  - At each step
    - Select the nearest point to the last point of the tour under construction and add this point to the end of the tour

```
greedyTSP( distance matrix d(1..n,1..n)){
    S = [1]; last = 1;
    C = {2,3,…, n};
    while(C not empty){
        j = argMin_{i∈C}{ d(last, i)};
        S = S ::j;
        last = j;
        C = C \ {j};
    }
    S = S::1
    return S;
}
```

Multi Knapsack Problem

- Given unlimited number of bins having capacity Q and n items 1, 2, …, n in which the weight of item i is $w(i)$. How to put these n items into bins such that the total weight of items put into each bin cannot exceed Q and the number of bins used is minimal
- Possible strategies
  - Multi-Stage selection
    - Sort items in some order
    - Consider each item in the sorted list
    - For each item, select an appropriate bin for the current item
  - Single-Stage selection: consider all pairs of (bin, item) for the selection
    - Each step, consider all pairs of bin, item for the selection

Multi Knapsack Problem (Multi-Stage selection)

```
greedyMNS(){
  bins = []; // list of bins being used
  items = sort items in some order;
  for i in items do {
     b = select(bins,load,w(i));
     if b = NULL then {
        b = length(bins) + 1;
        bins = bins::b;
        load[b] = 0;
      }
      binOfItem[i] = b;
      load[b] = load[b] + w[i];
  }
  return binOfItem;
}
```

```
selectFirstFitBin(bins,load,w){
  for b in bins do {
    if load[b] + w <= Q then
      return b;
  }
  return NULL;
}
```

```
selectBestFitBin(bins,load,w){
  R = ∞; sel_b = NULL
  for b in bins do {
    if load[b] + w <= Q and (Q-load[b]-w) then
      R = Q – load[b] - w; sel_b = b;
  }
  return sel_b;
}
```

Multi Knapsack Problem (Single-Stage selection)

```
selectBinAndItem(bins, cand){
  R = ∞; sel_bin = NULL; sel_item = 0;
  for i in cand do{
    for b in bins do{
      if load[b] + w(i) ≤ Q
        and (Q – load[b] – w(i) < R then {
        R = Q – load[b] – w(i);
        sel_bin = b; sel_item = i;
      }
    }
  }
  return (sel_bin, sel_item);
}
```

```
greedyMNS(){
  bins = []; cand = {1, 2, . . ., n};
  while (cand not empty){
    (sel_bin, sel_item) = selectBinAndItem(bins, cand);
    if (sel_bin = NULL) then {
        b = bins.length; bins = bins::<b>;
        sel_bin = b; sel_item = argmax_{i∈cand}(w(i));
     }
     cand.remove(sel_item);
     binOfItem[sel_item] = sel_bin;
     load[sel_bin] = load[sel_bin] + w(sel_item);
  }
  return binOfItem;
}
```

Exercise - BCA

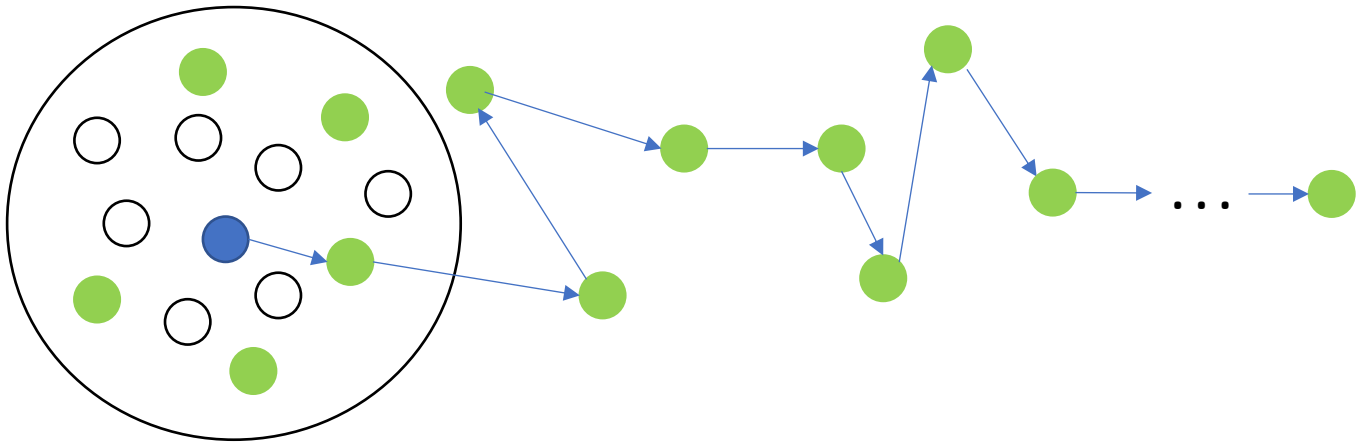
- At the beginning of the semester, the head of a computer science department have to assign courses to teachers in a balanced way.
- The department has *m* teachers $T=\{1, 2, ..., m\}$ and *n* courses $C=\{1, 2,..., n\}$.
- Each course $c \in C$ has a duration h_c.
- Each teacher $t \in T$ has a preference list which is a list of courses he/she can teach depending on his/her specialization.
- We know a list of pairs of conflicting two courses that cannot be assigned to the same teacher as these courses have been already scheduled in the same slot of the timetable. This conflict information is represented by a conflict matrix *A* in which $A(i,j)=1$ indicates that course *i* and *j* are conflict.
- The load of a teacher is the total duration of courses assigned to her/him.
- How to assign *n* courses to *m* teachers such that each course assigned to a teacher is in his/her preference list, no two conflicting courses are assigned to the same teacher, and the maximal load for all teachers is minimal

CONTENT

- Greedy algorithm
- **Local search**

Local search

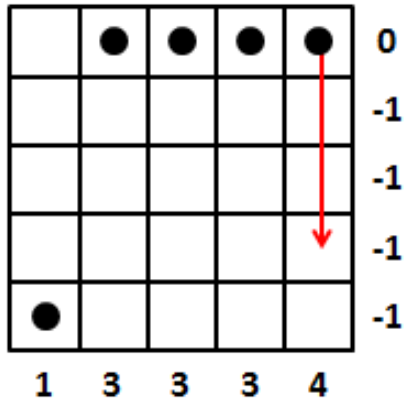
- Method for finding high-quality solutions to optimization problems
- Key idea
  - Generate an initial solution
  - Iteratively move from a current solution to one of its neighbors

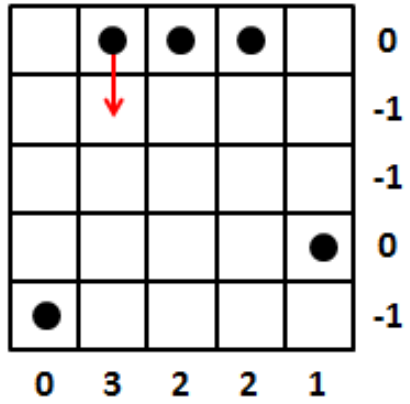


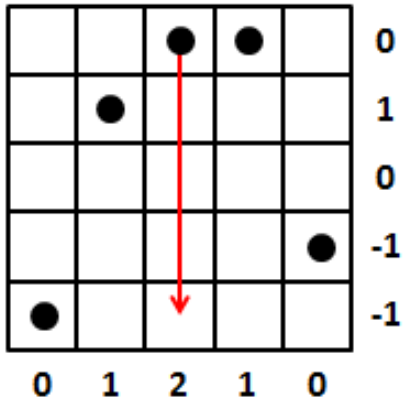

```
function LocalSearch(N, f){
  // N: neighborhood
  // f: quality function
  s = Generate an initial solution;
  s* = s;
  while termination not reach do {
    s = Select(N(s), s); //neighbor selection
    if s = NULL then break;
    if s is better than s* then
      s* = s;
  }
  return s*;
}
```

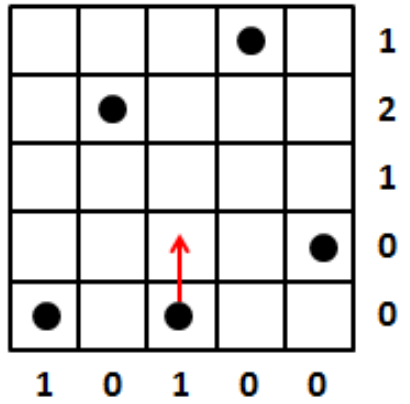
Local search

- Example of n-queen: How to place n queens on a chess board such that no two queens attack each other



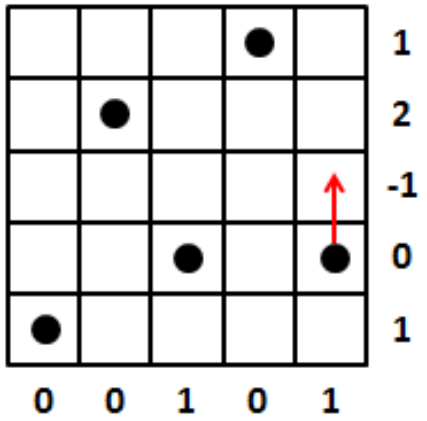


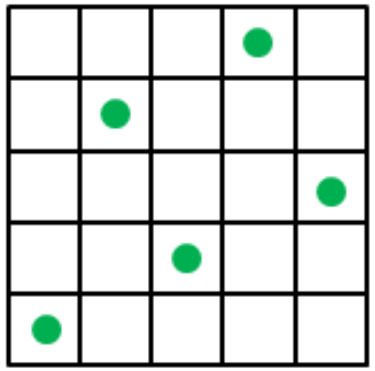




Local search

- Example of n-queen: How to place n queens on a chess board such that no two queens attack each other

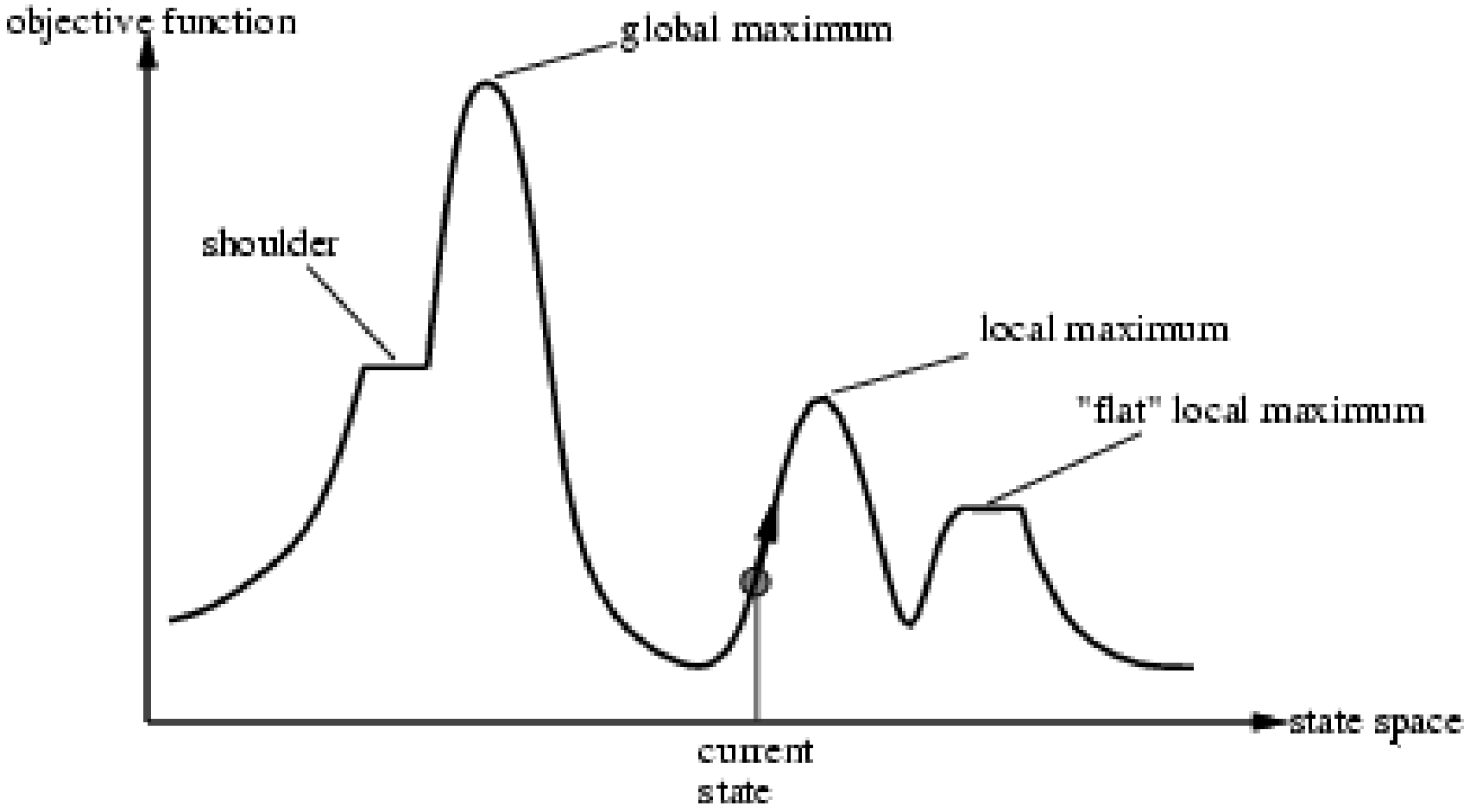




Local search

- Quality function ***f*(*s*)**
  - Measure the quality of a solution ***s***
  - Depend on specific problems
- For pure optimization problems
  - Quality function can be the objective function of the problem
- For satisfaction problems (n-queen, sudoku, etc.)
  - Quality function can be the number of constraint violations
- For constrained optimization problems
  - Quality function can be the combination of objective function (optimality) and the constraint violations (satisfiability)
- Without loss of generality, suppose that *f*(*s*) is need to be maximized

“Landscape” of search



Hill Climbing gets stuck in local minima depending on?

Neighbor selections

```
function LocalSearch(N, f){
  // N: neighborhood
  // f: quality function
  s = Generate an initial solution;
  s* = s;
  while termination not reach do {
    s = Select(N(s), f, s); //neighbor selection
    if s = NULL then break;
    if f(s) > f(s*) then
      s* = s;
  }
  return s*;
}
```

```
function S-Improvement(N, f, s){
    s' = argMin_{x∈N} f(x);
    if f(s') > f(s) then return s';
    else return NULL;
}
```

```
function S-RandomImprovement(N, f, s) {
    select n∈ N with probability 1/|N|;
    if f(n) > f(s) then return n;
    else return s;
}
```

```
function S-Metropolis(N, f, s, t){
    select n∈ N with probability 1/|N|;
    if f(n) > f(s) then return n;
    else with probability e^{(f(s)−f(n))/t};
    else return s;
}
```

HUST

hust.edu.vn fb.com/dhbkhn

THANK YOU !