Trajectory based Search Algorithms (II)

- Simulated Annealing (SA)
- Tabu Search (TS)
- Variable Neighborhood Search (VNS)
- Guided Local Search
- Greedy Randomized Adaptive Search Procedure (GRASP)

Simulated Annealing

Idea:

- accept, with some probability, also perturbations which lead to an increase of the objective function (in the case of minimization problems)

Inspiration:

- SA algorithms are inspired by the process of restructuring the internal configuration in a solid which is annealed (e.g. crystallization process):
  - The solid is heated (up to the melting point): its particles are randomly distributed.
  - The material is the slowly cooled down: its particles are reorganized in order to reach a low energy state

Contributors: Metropolis(1953), Kirkpatrick, Gelatt, Vecchi (1983), Cerny (1985)

Simulated Annealing

Analogy:

Physical process:		Minimization problem:
• System energy	→	Objective function
• System state	→	Configuration (candidate solution)
• Change of the system state	→	Perturbation of the current configuration
• Temperature	→	Parameter which controls the optimization process

Simulated Annealing

Some physics:

- Each state of the system has a corresponding probability

- The probability corresponding to a given state depends on the energy of the state and on the system temperature (Boltzmann distribution)

$$P_T(s) = \frac{1}{Z(T)} \exp\left(-\frac{E(s)}{k_B T}\right)$$

$$Z(T) = \sum_{s \in S} \exp\left(-\frac{E(s)}{k_B T}\right)$$

E(s) = energy of state s

T = temperature

Z(T)=partition function (normalization factor)

k_B = Boltzmann constant

Simulated Annealing

Some physics:

- Large values of T (T goes to infinity): the argument of exp is almost 0 => the states have all the same probability to be reached

- Small values of T (T goes to 0): only the states with zero energy will have non-zero probabilities

$$P_T(s) = \frac{1}{Z(T)} \exp(-\frac{E(s)}{k_B T})$$

$$Z(T) = \sum_{s \in S} \exp(-\frac{E(s)}{k_B T})$$

E(s) = energy of state s

T = temperature

Z(T)=partition function (normalization factor)

k_B = Boltzmann constant

Simulated Annealing

How can be used these results from physics to solve an optimization problem ?

- It would be enough to generate configurations according to the Boltzmann distribution for smaller and smaller values of the temperature.

- **Problem:** it is difficult to compute the partition function Z(T) (it means to compute a sum over all possible configurations in the search space which is practically impossible for real-world problems – it would correspond to an exhaustive search)
- **Solution:** the distribution is approximated by simulating the evolution of a stochastic process (Markov chain) having as stationary distribution the Boltzmann distribution => Metropolis algorithm

Simulated Annealing

Metropolis algorithm (1953)

```
Init S(0)
k=0
REPEAT
 S'=perturb(S(k))
 IF  f(S')<f(S(k)) THEN
  S(k+1)=S'        // (unconditionally)
 ELSE
   IF random(0,1)<exp(-(f(S')-f(S(k)))/T   THEN
     S(k+1)=S'  //with probab. min{1,exp(-(f(S')-f(S(k)))/T)}
   ELSE
     S(k+1)=S(k)  // no change
 k=k+1
UNTIL "a stopping condition is satisfied"
```

Simulated Annealing

Properties of the Metropolis algorithm

- Another acceptance probability (logistic):

```
P(S(k+1)=S') = 1/(1+exp((f(S')-f(S(k))/T))
```

- Implementation issue: assigning a value with a given probability is based on generating a random value in (0,1)

```
u=Random(0,1)
IF u<P(S(k+1)=S') THEN S(k+1)=S'
                  ELSE S(k+1)=S(k) //unchanged
```

- Large values for T -> high acceptance probability for any configuration (pure random search)

  Small values for T -> High acceptance probabilities only for the states with low energy values (greedy search - similar to a gradient descent method)

Simulated Annealing

Properties of the Metropolis algorithm

- The perturbation rules used to generate new configurations depend on the problem to be solved

Optimization in continuous domains

$S' = S + z$

$z = (z_1, \ldots, z_n)$

z_i : generated according to the distribution:

- N(0,T)
- Cauchy(T) (Fast SA)
- etc

Combinatorial optimization

The new configuration is selected deterministically or randomly from the neighborhood of the current configuration

Example:

- TSP: 2-opt transformation
- Knapsack: add/remove an object (0->1, 1->0)

Simulated Annealing

Simulated Annealing = repeated application of the Metropolis algorithm for decreasing values of the temperature

General structure

```
Init S(0), T(0)
i=0
REPEAT
     apply Metropolis  (for kmax iterations)
     compute T(i+1)
     i=i+1
UNTIL T(i)<eps
```

Problem: How to choose the cooling scheme ?

Simulated Annealing

Cooling schemes:

$T(k)=T(0)/(k+1)$

$T(k)=T(0)/\ln(k+c)$

$T(k)=aT(k-1)$ $(a<1$, ex: $a=0.995)$

Remark. T(0) should be chosen such that during the first iterations almost all new configurations are accepted (this ensures a good exploration of the search space)

Simulated Annealing

Convergence properties:

If the following properties are satisfied:

- $Pg(S(k+1)=x'|S(k)=x)>0$ for any x and x' (the transition probability between any two configurations is non-zero)
- $Pa(S(k+1)=x'|S(k)=x)=\min\{1,\exp(-(f(x')-f(x))/T)\}$ (Metropolis acceptance probability)
- $T(k)=C/\lg(k+c)$ (logarithmic cooling schedule)

then $P(|f(S(k))-f(S^*)|<\varepsilon) \to 1$ when k goes to infinity (for any small ε) (x(k) is convergent in probability to the global minimum S^*)

Rmk: the logarithmic cooling schedule leads to a very slow algorithm

Simulated Annealing

Variant: another acceptance probability (Tsallis)

$$P_a(x') = \begin{cases} 1, & \Delta f \le 0 \\ (1-(1-q)\Delta f / T)^{1/(1-q)}, & \Delta f > 0,\ (1\text{-}q)\Delta f \le 1 \\ 0, & \Delta f > 0,\ (1\text{-}q)\Delta f > 1 \end{cases}$$

$$\Delta f = f(x') - f(x)$$

$$q \in (0,1)$$

Simulated Annealing

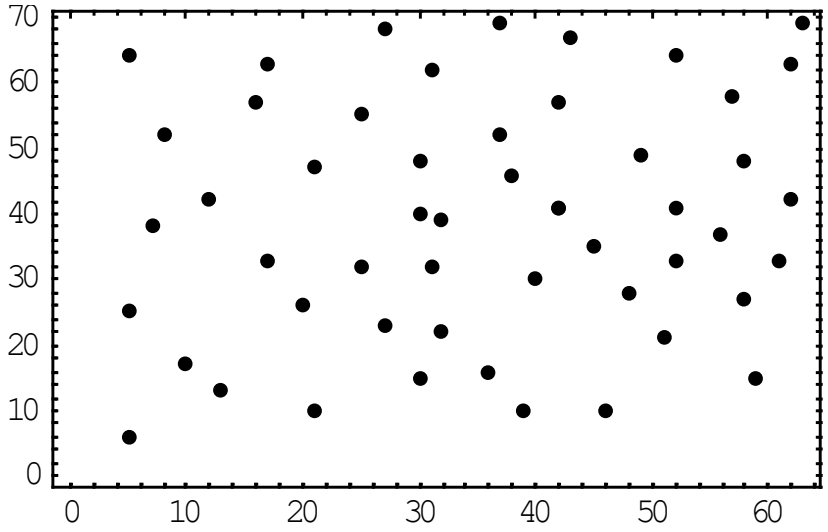
Example: Travelling Salesman Problem (TSP)
(TSPLib: http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95)

Test instance: eil51 – 51 towns

Parameters:

- 5000 iterations; T is changed at each 100 iterations
- T(k)=T(0)/(1+log(k))

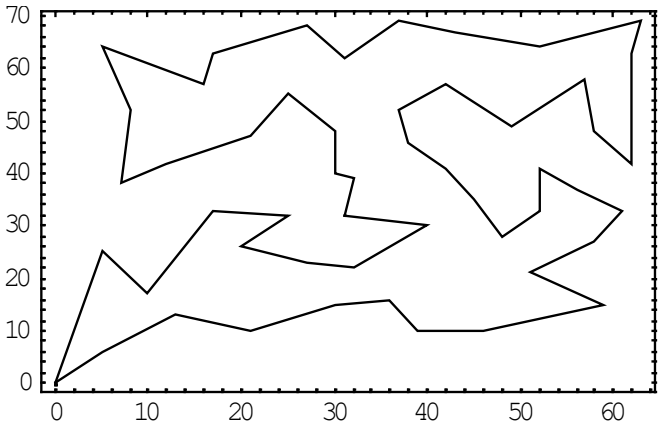
Location of towns



Simulated Annealing

Example: TSP

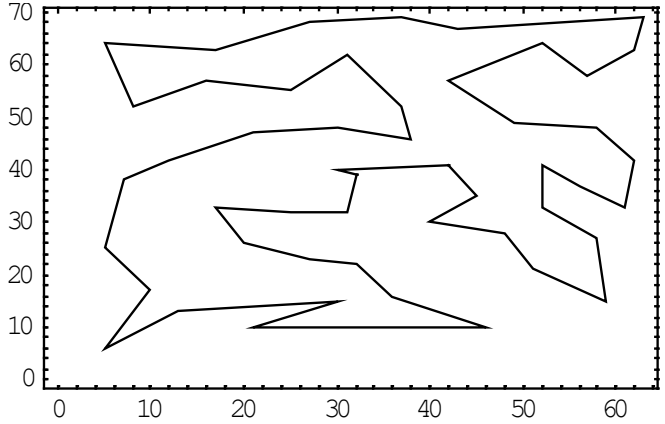
Test instance: eil51 (TSPlib)



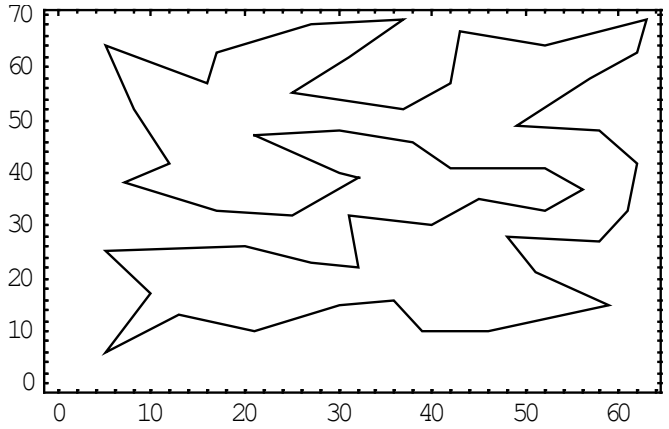
T(0)=10,cost=478.384

Minimal cost: 426

T(0)=5, cost=474.178



T(0)=1, cost=481.32



Tabu Search

Creator: Fred Glover (1986)

Applicability : mainly combinatorial optimization problems

Particularity:

- It is an iterative local search technique based on the exploration of the neighborhood of the current element
  - the neighborhood is defined as the set of all configurations which can be reached from the current configuration by applying once the search operator
  - the search operators are specific to the problem

Tabu Search

- Examples of neighbourhoods of a configuration S:
  - TSP: configurations which can be obtained by applying a 2-opt transformation to S;
  - Knapsack problem: binary vectors which are at a Hamming distance equal to 1 from S
  - Assignment problem (bin pack):
    - move an object from one bin to another one
    - exchange the position of two objects which belong to different bins
- It uses a list of prohibited configurations (called tabu list) which contains the configurations which are not acceptable in the following iterations (usually the tabu list is implemented as a circular list with a given size)

Tabu Search

General Structure:

```
S = initial configuration
S*=S      // S* is the best configuration so far
T = [S]   // initialization of the tabu list
REPEAT
    - S' = <the best element from the neighborhood N(S)
      which does not belong to T>
    - If f(S')<f(S*) then S*= S'  // save if better
    - S=S'  // change accepted even if not better
    - Include S in T  (if T is too large remove its oldest
      element)
UNTIL <stopping condition>
```

Remarks:

1. If the neighbourhood is too large then it is possible to evaluate only a sample from the neighbourhood.
2. TS explicitly uses the search history since the tabu list can be interpreted as a short term memory (to escape from local optima and to explore the search space)

Tabu Search

Remark:

- storing complete candidate solutions in the tabu list could be inefficient
- features of the candidate solutions could be stored instead

Examples of features:

- Permutation-type encodings (e.g. TSP): interchange of two elements (e.g. (i,j) is a feature which denote that s_i was swapped with s_i).
- Binary encodings (e.g. knapsack problem): change of a component (e.g. (i,0) is a feature denoting that s_i is 0 and (i,1) is a feature denoting that s_i is 1).

Remarks:

- using features instead of full configurations could introduce some excessive constraints (some good configurations are excluded)
- to avoid such situations is used a so-called aspiration rule: configurations better than the best discovered up to the current step are accepted even if they have prohibited features

Tabu Search

Intensification:

Aim: exploitation of promising regions (i.e. penalize configurations which are far away from the current one)

Implementation:

- Count the number of iterations when a component or feature remains unchanged – the good components are those with large values of the corresponding counter (this means that they are preserved in the competition between the current and new configurations)
- Restart the search process from the best configuration by keeping the good components/ features fixed (the neighbourhood is constructed only by changing the other components)

Tabu Search

Diversification:

Aim: exploration of the unvisited regions (i.e. penalize configurations which are close to the current one)

Implementation:

- Compute the frequency of values used for each component. The values with low frequencies are considered under explored
- Restart the search process from configurations which contain under explored values or change the fitness function by penalizing the very frequent values of the components in order to stimulate the search in new (or less explored) regions

Variable Neighborhood Search

VNS - Variable Neighborhood Search [Mladenovic, P. Hansen, 1997]

Idea: they use a set of neighborhoods V_1, V_2,...,V_{kmax} which is explored in an incremental way; in each neighborhood the search is done using a local search method

Remark: the neihborhood set for a configuration x is established depending on the problem to be solved but such that if k1<k2 then the elements of $V_{k1}(x)$ can be obtained from x using fewer operations than are necessary to construct the elements of $V_{k2}(x)$ ($V_{k1}(x)$ can be included in $V_{k2}(x)$ but it is not necessary)

Example: for the traveling salesman problem $V_k(x)$ can contain the configurations (routes) obtained from x by applying k swaps of some randomly selected nodes

Variable Neighborhood Search

VNS - Variable Neighborhood Search [Mladenovic, P. Hansen, 1997]

General structure

```
Initialize S (randomly in the search space)
k=1
WHILE k<=kmax DO
     select S' randomly from V_k(S);
     construct S'' from S' by applying a local search method
     IF f(S'')<f(S) THEN S=S''; k=1
                    ELSE k=k+1
```

shaking

Local search

Accept the change or move to the next neighbour

Remark: the local search corresponding to a given value of k is NOT limited to $V_k(S)$

Guided Local Search

Creators: Voudouris&Tsang, 1999

Idea:

- the objective function is dynamically modified in order to escape from the local optima regions (the penalized values become less desirable)
- the change is based on the inclusion of some penalty terms corresponding to features of the candidate solutions which should be avoided (e.g. in the case of TSP such a feature could be an edge linking two locations)

$$f_{\text{mod}}(S) = f(S) + \lambda \sum_{i=1}^{m} p_i I_i(S)$$

$$I_i(S) = \begin{cases} 1 & \text{the } i-th \text{ feature is in } S \\ 0 & \text{the } i-th \text{ feature is not in } S \end{cases}$$

p_i = penalty parameter

λ = regularization factor (control of the penalty impact)

Guided Local Search

General structure:

$$U(s,i) = I_i(s)\frac{c_i}{1+p_i}$$

c_i = cost of feature i

p_i = penalty of feature i

```
S=initial configuration
Repeat
    s=LocalSearch(S,f)
    for <all features i with maximal gain U(s,i)
         (which could be obtained by penalization)>
       pi=pi+1
    endfor
    f=Update(f,p)
until <stopping condition>
```

$$f_{\text{mod}}(S) = f(S) + \lambda\sum_{i=1}^{m} p_i I_i(S)$$

$$I_i(S) = \begin{cases} 1 & \text{the } i-th \text{ feature is in } S \\ 0 & \text{the } i-th \text{ feature is not in } S \end{cases}$$

p_i = penalty parameter

λ = regularization factor

(control of the penalty impact)

GRASP

GRASP = Greedy Randomized Adaptive Search Procedure [Feo & Resende, 1995]

Idea: GRASP focuses on constructing good initial configurations by ensuring a compromise between exploration and exploitation

Particularities:

- It is an heuristic which construct the initial configuration component by component
  - For TSP: a location or and edge is added at each step
  - For knapsack problem: one object is selected and added at each step

- The key element is the choice of the component to be included in the candidate solution: GRASP uses a non-uniform random selection in a ranked list (good elements have higher probability to be selected)

GRASP

Particularities

- The list size can change in time (adaptive character) between two extreme cases:
  - Size = 1 -> greedy choice (at each step the best component is selected)
  - Size = maximal number of possible values (if the selection is uniform this leads to purely random search)

```
procedure GRASP(Max_Iterations,Seed)
1     Read_Input();
2     for k = 1,...,Max_Iterations do
3          Solution ← Greedy_Randomized_Construction(Seed);
4          Solution ← Local_Search(Solution);
5          Update_Solution(Solution,Best_Solution);
6     end;
7     return Best_Solution;
end GRASP.
```

GRASP

Overall structure

[Resende& Ribeiro, Greedy randomized adaptive search procedures, Handbook in Metaheuristics, 2002]

```
procedure GRASP(Max_Iterations,Seed)
1    Read_Input();
2    for k = 1,...,Max_Iterations do
3        Solution ← Greedy_Randomized_Construction(Seed);
4        Solution ← Local_Search(Solution);
5        Update_Solution(Solution,Best_Solution);
6    end;
7    return Best_Solution;
end GRASP.
```

```
procedure Greedy_Randomized_Construction(Seed)
1    Solution ← ∅;
2    Evaluate the incremental costs of the candidate elements;
3    while Solution is not a complete solution do
4        Build the restricted candidate list (RCL);
5        Select an element s from the RCL at random;
6        Solution ← Solution ∪ {s};
7        Reevaluate the incremental costs;
8    end;
9    return Solution;
end Greedy_Randomized_Construction.
```

Summary

Building blocks of trajectory based search methods

- Initialization – a candidate solution is constructed component by component (random vs greedy)
- Generation of a new candidate
  - By construction (typical for the initialization stage)
  - By perturbation – based on selection (random vs elitist) from a neighborhood of the current configuration
    - The neighborhood can be fixed or variable (VNS)
    - The perturbation can be small (local search) or large (ILS)
- Acceptance of a new candidate
  - Only if it is better than the current element (Hill climbing/ descent)
  - Accept worse elements (in order to avoid stagnation or cycling)
    - With a probability depending on the quality loss and on some control parameter (SA)
    - If better elements have been already analyzed and they are prohibited (TS)

Summary

Building blocks of trajectory based search methods

- Stopping condition
  - Quality –based: a configuration of acceptable quality has been identified
  - Behaviour-based: no improvement during the last iterations
  - Resource-based: a given number of iterations or objective function evaluations have been done

Next lecture

- Population-based metaheuristics
  - Main particularities
  - General structure
  - Main components
  - Evolutionary algorithms