AI505

Optimization

Discrete Optimization Metaheuristics

Marco Chiarandini

Department of Mathematics & Computer Science
University of Southern Denmark

Outline

Remarks

See solutions to Sheet 11 on SMTWTP

- Logging
- Testing
- Multiple neighborhoods
- Metaheuristics

Outline

Escaping Local Optima

Possibilities:

- **Restart:** re-initialize search whenever a local optimum is encountered.
  (Often rather ineffective due to cost of initialization.)
- **Non-improving steps**: in local optima, allow selection of candidate solutions with equal or worse evaluation function value, *e.g.*, using minimally worsening steps.
  (Can lead to long walks in **plateaus**, *i.e.*, regions of search positions with identical evaluation function.)
- **Diversify the neighborhood**

Note: None of these mechanisms is guaranteed to always escape effectively from local optima.

Diversification **vs** Intensification

- Goal-directed and randomized components of LS strategy need to be balanced carefully.
- **Intensification**: aims at greedily increasing solution quality, *e.g.*, by exploiting the evaluation function.
- **Diversification**: aims at preventing search stagnation, that is, the search process getting trapped in confined regions.

Examples:

- Iterative Improvement (II, First improvement or Best improvement): **intensification** strategy.
- Uninformed Random Walk/Picking (URW/P): **diversification** strategy.

Balanced combination of intensification and diversification mechanisms forms the basis for advanced LS methods.

Outline

Greedy Local Search

Key idea: Best improvement (Hill Climber or Steepest Descent) + Sideways Moves + seldom worsening moves

```
Algorithm 6.1: GSAT (F)
Input       : A CNF formula $F$
Parameters  : Integers MAX-FLIPS, MAX-TRIES
Output      : A satisfying assignment for $F$, or FAIL
begin
    for $i \leftarrow 1$ to MAX-TRIES do
        $\sigma \leftarrow$ a randomly generated truth assignment for $F$
        for $j \leftarrow 1$ to MAX-FLIPS do
            if $\sigma$ satisfies $F$ then return $\sigma$            // success
            $v \leftarrow$ a variable flipping which results in the greatest decrease
                  (possibly negative) in the number of unsatisfied clauses
            Flip $v$ in $\sigma$
    return FAIL                     // no satisfying assignment found
end
```

- GSAT begins with a rapid greedy descent towards a better truth assignment
- then long sequences of **sideways** moves take place. Sideways moves are moves that do not increase or decrease the total number of unsatisfied clauses. They navigate through **plateaux**, which is SAT are many and large
- GSAT [Selman et al. 1992] at its times was able to beat complete search algorithms (they were not using CDC)

Randomized Iterative Improvement

aka, Stochastic Hill Climbing

Key idea: Allow worsening moves: In each search step, with a fixed probability perform an uninformed random walk step instead of an iterative improvement step.
greedy + uniform random walk

Randomized Iterative Improvement (RII):

```
determine initial candidate solution $s$
while termination condition is not satisfied do
    With probability wp:
        choose a neighbor $s'$ of $s$ uniformly at random
    Otherwise:
        choose a neighbor $s'$ of $s$ such that $f(s') < f(s)$ or,
          if no such $s'$ exists, choose $s'$ such that $f(s')$ is minimal
    $s := s'$
```

With infite time it reaches optimum with $p > 0$ [Hoos and Tsang].

Example: Randomized Iterative Improvement for SAT

```
procedure RIISAT($F$, wp, maxSteps)
    input: a formula $F$, probability wp, integer maxSteps
    output: a model $\varphi$ for $F$ or $\emptyset$

    choose assignment $\varphi$ for $F$ uniformly at random;
    steps := 0;
    while not($\varphi$ is not proper) and (steps $<$ maxSteps) do
        with probability wp do
            select $x$ in $X$ uniformly at random and flip;
        otherwise
            select $x$ in $X^c$ uniformly at random from those that
                maximally decrease number of clauses violated;
        change $\varphi$;
        steps := steps$+1$;
    end
    if $\varphi$ is a model for $F$ then return $\varphi$
    else return $\emptyset$
    end
end RIISAT
```

X^c set of variables in violated clauses

Note:

- No need to terminate search when local minimum is encountered

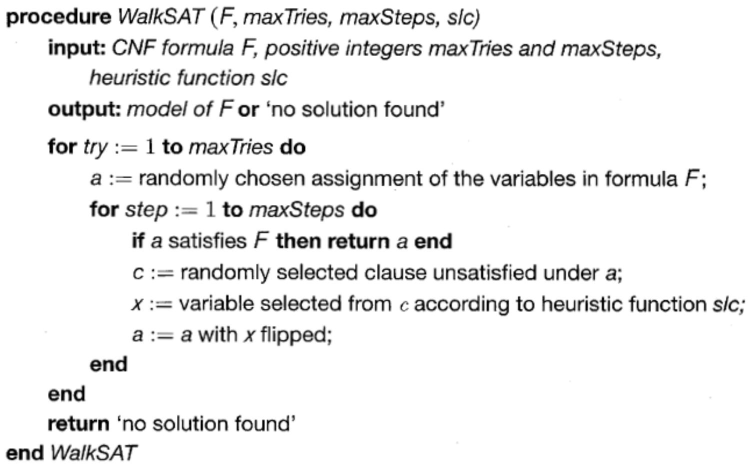
  Instead: Impose limit on number of search steps or CPU time, from beginning of search or after last improvement.

- Probabilistic mechanism permits arbitrary long sequences of random walk steps

  Therefore: When run sufficiently long, RII is guaranteed to find (optimal) solution to any problem instance with arbitrarily high probability.

- GWSAT [Selman et al., 1994], was at some point state-of-the-art for SAT.

Focused Local Search: WalkSAT



```
procedure WalkSAT (F, maxTries, maxSteps, slc)
    input: CNF formula F, positive integers maxTries and maxSteps,
        heuristic function slc
    output: model of F or 'no solution found'
    for try := 1 to maxTries do
        a := randomly chosen assignment of the variables in formula F;
        for step := 1 to maxSteps do
            if a satisfies F then return a end
            c := randomly selected clause unsatisfied under a;
            x := variable selected from c according to heuristic function slc;
            a := a with x flipped;
        end
    end
    return 'no solution found'
end WalkSAT
```

Example of **slc** heuristic: with prob. *wp* select a random move, with prob. $1 - wp$ select the best

Extension to CSP. Recall the Definitions

Constraint Satisfaction Problem (CSP)

A CSP is a finite set of variables X, together with a finite set of constraints C, each on a subset of X. A **solution** to a CSP is an assignment of a value $d \in D(x)$ to each $x \in X$, such that all constraints are satisfied simultaneously.

Constraint Optimization Problem (COP)

A COP is a CSP P defined on the variables $x_1, \ldots, x_n$, together with an objective function $f : D(x_1) \times \cdots \times D(x_n) \to Q$ that assigns a value to each assignment of values to the variables. An **optimal solution** to a minimization (maximization) COP is a solution d to P that minimizes (maximizes) the value of $f(d)$.

$\leadsto$ Constraints in a CSP can be relaxed and their violations determine the objective function. This is the most common approach in LS

Min-Conflict Heuristic

```
procedure MCH ($P$, maxSteps)
    input: CSP instance $P$, positive integer maxSteps
    output: solution of $P$ or "no solution found"
    $a$ := randomly chosen assignment of the variables in $P$;
    for step := 1 to maxSteps do
        if $a$ satisfies all constraints of $P$ then return $a$ end
        $x$ := randomly selected variable from conflict set $K(a)$;
        $v$ := randomly selected value from the domain of $x$ such that
              setting $x$ to $v$ minimises the number of unsatisfied constraints;
        $a$ := $a$ with $x$ set to $v$;
    end
    return "no solution found"
end MCH
```

Outline

Guided Local Search

- **Key Idea:** Modify the evaluation function whenever a local optimum is encountered.
- Associate **weights** (**penalties**) with solution components; these determine impact of components on evaluation function value.
- Perform Iterative Improvement; when in local minimum, increase penalties of some solution components until improving steps become available.

Guided Local Search (GLS):

```
determine initial candidate solution $s$
initialize penalties
while termination criterion is not satisfied do
    compute modified evaluation function $g'$ from $g$
        based on penalties
    perform subsidiary local search on $s$
        using evaluation function $g'$
    update penalties based on $s$
```

Guided Local Search

- **Modified evaluation function:**

$$g'(s) := f(s) + \sum_{i \in SC(s)} \texttt{penalty}(i),$$

  where $SC(s)$ is the set of solution components used in candidate solution s.

- **Penalty initialization:** For all i: $\texttt{penalty}(i) := 0$.

- **Penalty update** in local minimum s: Typically involves **penalty increase** of some or all solution components of s; often also occasional **penalty decrease** or **penalty smoothing**.

- **Subsidiary local search:** Often **Iterative Improvement**.

Potential problem:
Solution components required for (optimal) solution may also be present in many local minima.

Possible solutions:

A: Occasional decreases/smoothing of penalties.

B: Only increase penalties of solution components that are least likely to occur in (optimal) solutions.

Implementation of **B**: Only increase penalties of solution components i with maximal utility [Voudouris and Tsang, 1995]:

$$\mathtt{util}(s,i) := \frac{f_i(s)}{1 + \mathtt{penalty}(i)}$$

where $f_i(s)$ is the solution quality contribution of i in s.

Example: Guided Local Search (GLS) for the TSP

[Voudouris and Tsang 1995; 1999]

- **Given:** TSP instance π
- **Search space:** Hamiltonian cycles in π with n vertices;
- **Neighborhood:** 2-edge-exchange;
- **Solution components** edges of π;
  $f_e(G, p) := w(e)$;
- **Penalty initialization:** Set all edge penalties to zero.
- **Subsidiary local search:** Iterative First Improvement.
- **Penalty update:** Increment penalties of all edges with maximal utility by

$$\lambda := 0.3 \cdot \frac{w(s_{2\text{-opt}})}{n}$$

where $s_{2\text{-opt}} =$ 2-optimal tour.

Guided Local Search for SAT

- Assign a positive weight to each clause
- attempt to minimize the sum of the weights of the unsatisfied clauses.
- The clause weights are dynamically modified (additively or multiplicatively) as the search progresses, increasing the weight of the clauses that are currently unsatisfied.
- Depends on:
  how often and by how much the weights of unsatisfied clauses are increased, and
  how are all weights periodically decreased in order to prevent certain weights from becoming dis-proportionately high.

Discrete Lagrangian Method

- Change the objective function bringing constraints g_i into it

$$L(\boldsymbol{s}, \boldsymbol{\lambda}) = f(\boldsymbol{s}) + \sum_i \lambda_i g_i(\boldsymbol{s})$$

- λ_i are continous variables called Lagrangian Multipliers
- $L(\boldsymbol{s}^*, \lambda) \leq L(\boldsymbol{s}^*, \boldsymbol{\lambda}^*) \leq L(\boldsymbol{s}, \boldsymbol{\lambda}^*)$
- Alternate optimizations in $\boldsymbol{s}$ and in $\boldsymbol{\lambda}$

Discrete Lagrangian Method for SAT

let $U_i(x)$ be a function that is 0 if C_i is satisfied by a solution x, and 1 otherwise.

$$\text{minimize } N(x) = \sum_{i=1}^{m} U_i(x)$$
$$\text{s.t.} U_i(x) = 0 \qquad \forall i \in \{1, 2, \ldots, m\}$$

Discrete Lagrangian Function:

$$L_d(x, \lambda) = N(x) + \sum_{i=1}^{m} \lambda_i U_i(x)$$

A point $(x^*, \lambda^*) \in \{0, 1\}^n \times \mathbb{R}^m$ is called a **saddle point** of the Lagrange function $L_d(x, \lambda)$ if it is a local minimum w.r.t. x^* and a local maximum w.r.t. λ^*. Formally, (x^*, λ^*) is a saddle point for L_d if $L_d(x^*, \lambda) \leq L_d(x^*, \lambda^*) \leq L_d(x, \lambda^*)$

Probabilistic Iterative Improv.

Key idea: Accept worsening steps with probability that depends on respective deterioration in evaluation function value:
bigger deterioration $\cong$ smaller probability

Realization:

- Function $p(f, s)$: determines probability distribution over neighbors of s based on their values under evaluation function f.
- Accept s' neighbor of s with probability $p(f, s, s')$.

Note:

- Behavior of PII crucially depends on choice of p.
- II and RII are special cases of PII.

Example: Metropolis PII for the TSP

- **Search space S:** set of all Hamiltonian cycles in given graph G.
- **Solution set:** same as S
- **Neighborhood relation $\mathcal{N}(s)$:** 2-edge-exchange
- **Initialization:** an Hamiltonian cycle uniformly at random.
- **Step function:** implemented as 2-stage process:

  1. select neighbor $s' \in N(s)$ uniformly at random;
  2. accept as new search position with probability:

$$p(T, s, s') := \begin{cases} 1 & \text{if } f(s') \leq f(s) \\ \exp \frac{-(f(s')-f(s))}{T} & \text{otherwise} \end{cases}$$

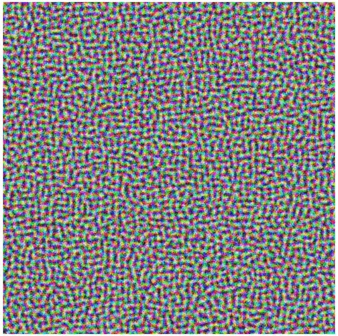
  (Metropolis condition), where **temperature** parameter T controls likelihood of accepting worsening steps.
- **Termination:** upon exceeding given bound on run-time.

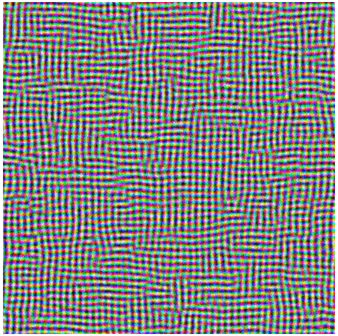
Outline

Inspired by statistical mechanics in matter physics:

- candidate solutions $\cong$ states of physical system
- evaluation function $\cong$ thermodynamic energy
- globally optimal solutions $\cong$ ground states
- parameter T $\cong$ physical temperature

Note: In physical process (*e.g.*, annealing of metals), perfect ground states are achieved by very slow lowering of temperature.





Simulated Annealing

Key idea: Vary temperature parameter, *i.e.*, probability of accepting worsening moves, in Probabilistic Iterative Improvement according to annealing schedule (aka **cooling schedule**).

Simulated Annealing (SA):

```
determine initial candidate solution $s$
set initial temperature $T$ according to annealing schedule
while termination condition is not satisfied: do
    while maintain same temperature T according to annealing schedule do
        probabilistically choose a neighbor $s'$ of $s$ using proposal mechanism
        if $s'$ satisfies probabilistic acceptance criterion (depending on $T$) then
            $s := s'$
    update $T$ according to annealing schedule
```

- 2-stage step function based on
  - proposal mechanism (often uniform random choice from $N(s)$)
  - acceptance criterion (often **Metropolis condition**)

- Annealing schedule
  (function mapping run-time t onto temperature $T(t)$):
  - initial temperature T_0
    (may depend on properties of given problem instance)
  - temperature update scheme
    (*e.g.*, linear cooling: $T_{i+1} = T_0(1 - i/I_{max})$,
    geometric cooling: $T_{i+1} = \alpha \cdot T_i$)
  - number of search steps to be performed at each temperature
    (often multiple of neighborhood size)
  - may be **static** or **dynamic**
  - seek to balance moderate execution time with asymptotic behavior properties

- Termination predicate: often based on **acceptance ratio**,
  i.e., ratio accepted / proposed steps **or** number of idle iterations

Example: Simulated Annealing for TSP

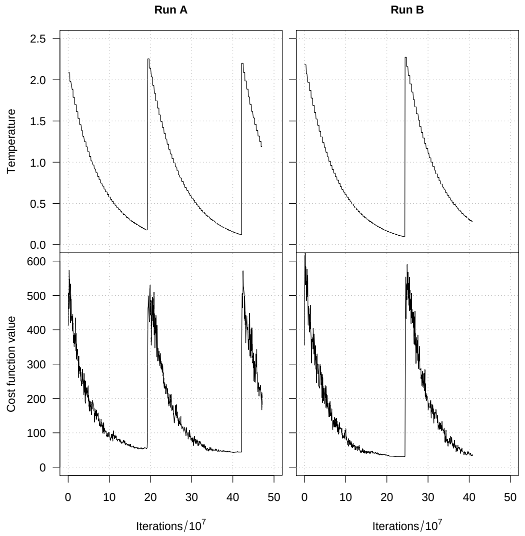
Extension of previous PII algorithm for the TSP, with

- proposal mechanism: uniform random choice from 2-exchange neighborhood;
- acceptance criterion: Metropolis condition (always accept improving steps, accept worsening steps with probability $\exp\left[-(f(s') - f(s))/T\right]$);
- annealing schedule: geometric cooling $T := 0.95 \cdot T$ with $n \cdot (n-1)$ steps at each temperature (n = number of vertices in given graph), T_0 chosen such that 97% of proposed steps are accepted;
- termination: when for five successive temperature values no improvement in solution quality and acceptance ratio $< 2\%$.

Improvements:

- neighborhood pruning (*e.g.*, candidate lists for TSP)
- greedy initialization (*e.g.*, by using NNH for the TSP)
- **low temperature starts** (to prevent good initial candidate solutions from being too easily destroyed by worsening steps)

Profiling



Run A
Run B
Temperature
Cost function value
Iterations/10^7
Iterations/10^7

Outline

Iterated Local Search

Key Idea: Use two types of LS steps:

- **subsidiary local search** steps for reaching local optima as efficiently as possible (intensification)
- **perturbation steps** for effectively escaping from local optima (diversification).

Also: Use **acceptance criterion** to control diversification **vs** intensification behavior.

Iterated Local Search (ILS):

```
determine initial candidate solution $s$
perform subsidiary local search on $s$
while termination criterion is not satisfied do
    $r := s$
    perform perturbation on $s$
    perform subsidiary local search on $s$
    based on acceptance criterion,
        keep $s$ or revert to $s := r$
```

Note:

- **Subsidiary local search** results in a local minimum.
- ILS trajectories can be seen as walks in the space of local minima of the given evaluation function.
- **Perturbation phase** and **acceptance criterion** may use aspects of **search history** (*i.e.*, limited memory).
- In a high-performance ILS algorithm, **subsidiary local search**, **perturbation mechanism** and **acceptance criterion** need to complement each other well.

Components

Subsidiary local search:

- More effective subsidiary local search procedures lead to better ILS performance.
  Example: 2-opt **vs** 3-opt **vs** LK for TSP.

- Often, subsidiary local search = iterative improvement,
  but more sophisticated LS methods can be used.
  (*e.g.*, Tabu Search).

Components

Perturbation mechanism:

- Needs to be chosen such that its effect **cannot** be easily undone by subsequent local search phase.
  (Often achieved by search steps larger neighborhood.)
  Example: local search = 3-opt, perturbation = 4-exchange steps in ILS for TSP.
- A perturbation phase may consist of one or more perturbation steps.
- Weak perturbation $\Rightarrow$ short subsequent local search phase;
  but: risk of revisiting current local minimum.
- Strong perturbation $\Rightarrow$ more effective escape from local minima;
  but: may have similar drawbacks as random restart.
- Advanced ILS algorithms may change nature and/or strength of perturbation adaptively during search.

Components

Acceptance criteria:

- Always accept the **best** of the two candidate solutions

  $\Rightarrow$ ILS performs Iterative Improvement in the space of local optima reached by subsidiary local search.

- Always accept the **most recent** of the two candidate solutions

  $\Rightarrow$ ILS performs random walk in the space of local optima reached by subsidiary local search.

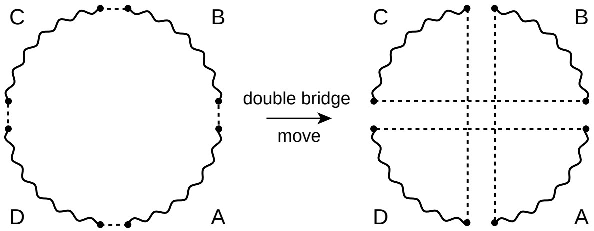
- Intermediate behavior: select between the two candidate solutions based on the **Metropolis criterion** (*e.g.*, used in **Large Step Markov Chains** [Martin et al., 1991].

- Advanced acceptance criteria take into account search history,
  e.g., by occasionally reverting to **incumbent solution**.

Examples

Example: Iterated Local Search for the TSP (1)

- **Given:** TSP instance π.
- **Search space:** Hamiltonian cycles in π.
- **Subsidiary local search:** Lin-Kernighan variable depth search algorithm
- **Perturbation mechanism:**
  'double-bridge move' = particular 4-exchange step:



- **Acceptance criterion:** Always return the best of the two given candidate round trips.

Outline

Tabu Search

Key idea: Avoid repeating history (memory)
How can we remember the history without

- memorizing full solutions (space)
- computing hash functions (time)

$\rightsquigarrow$ use attributes

Tabu Search

Key idea: Use aspects of search history (memory) to escape from local minima.

- Associate **tabu attributes** with candidate solutions or solution components.
- Forbid steps to search positions recently visited by underlying iterative best improvement procedure based on tabu attributes.

Tabu Search (TS):

```
determine initial candidate solution $s$
While termination criterion is not satisfied:
    determine set $N'$ of non-tabu neighbors of $s$
    choose a best candidate solution $s'$ in $N'$

    update tabu attributes based on $s'$
    $s := s'$
```

Example: Tabu Search for CSP

- **Search space:** set of all complete assignments of X.
- **Neighborhood structure:** one exchange
- **Memory:** Associate tabu status (Boolean value) with each pair (variable,value) (x, val).
- **Initialization:** a random assignment
- **Search steps:**
  - pairs (x, v) are tabu if they have been changed in the last `tt` steps;
  - neighboring assignments are admissible if they can be reached by changing a non-tabu pair or have fewer unsatisfied constraints than the best assignments seen so far (aspiration criterion);
  - choose uniformly at random admissible neighbors with minimal number of unsatisfied constraints.
- **Termination:** upon finding a feasible assignment **or**
  after given bound on number of search steps has been reached **or**
  after a number of idle iterations

Note:

- **Admissible neighbors of s**: Non-tabu search positions in $N(s)$
- **Tabu tenure**: a fixed number of subsequent search steps for which the last search position or the solution components just added/removed from it are declared **tabu**
- **Aspiration criterion** (often used): specifies conditions under which tabu status may be overridden (*e.g.*, if considered step leads to improvement in incumbent solution).
- Crucial for efficient implementation:
  - efficient **best improvement** local search
    $\leadsto$ pruning, delta updates, (auxiliary) data structures
  - efficient determination of tabu status:
    store for each variable x the number of the search step when its value was last changed it_x; x is tabu if $it - it_x < tt$, where it = current search step number.

Design Choices

Design choices:

- Neighborhood exploration:
  - no reduction
  - min-conflict heuristic
- Prohibition power for `move = <x,new_v,old_v>`
  - `<x,-,->`
  - `<x,-,old_v>`
  - `<x,new_v,old_v>, <x,old_v,new_v>`
- Tabu list dynamics:
  - Interval: $\texttt{tt} \in [t_b, t_b + w]$
  - Adaptive: $\texttt{tt} = \lfloor \alpha \cdot c \rfloor + \texttt{RandU}(0, t_b)$

Outline

Variable Neighborhood Search

Variable Neighborhood Search is a method based on the systematic change of the neighborhood during the search.

Central observations

- a local minimum w.r.t. one neighborhood function is not necessarily locally minimal w.r.t. another neighborhood function
- a global optimum is locally optimal w.r.t. **all** neighborhood functions

Key principle: change the neighborhood during the search

- Several adaptations of this central principle
  - (Basic) Variable Neighborhood Descent (VND)
  - Variable Neighborhood Search (VNS)
  - Reduced Variable Neighborhood Search (RVNS)
  - Variable Neighborhood Decomposition Search (VNDS)
  - Skewed Variable Neighborhood Search (SVNS)
- Notation
  - N_k, $k = 1, 2, \ldots, k_m$ is a set of neighborhood functions
  - $N_k(s)$ is the set of solutions in the k-th neighborhood of s

How to generate the various neighborhood functions?

- for many problems different neighborhood functions (local searches) exist / are in use
- change parameters of existing local search algorithms
- use k-exchange neighborhoods; these can be naturally extended
- many neighborhood functions are associated with distance measures; in this case increase the distance

Basic Variable Neighborhood Descent

```
**Procedure** BVND
**input**  : $N_k$, $k = 1, 2, \ldots, k_{max}$, and an initial solution $s$
**output:** a local optimum $s$ for $N_k$, $k = 1, 2, \ldots, k_{max}$
$k \leftarrow 1$
**repeat**
    $s' \leftarrow$ FindBestNeighbor($s$,$N_k$)
    **if** $f(s') < f(s)$ **then**
        $s \leftarrow s'$
        $(k \leftarrow 1)$
    **else**
        $k \leftarrow k + 1$
**until** $k = k_{max}$;
```

Variable Neighborhood Descent

```
Procedure VND
input  : $N_k$, $k = 1, 2, \ldots, k_{max}$, and an initial solution $s$
output: a local optimum $s$ for $N_k$, $k = 1, 2, \ldots, k_{max}$
$k \leftarrow 1$
repeat
    $s' \leftarrow$ IterativeImprovement($s$,$N_k$)
    if $f(s') < f(s)$ then
        $s \leftarrow s'$
        $k \leftarrow 1$
    else
        $k \leftarrow k + 1$
until $k = k_{max}$;
```

- Final solution is locally optimal w.r.t. all neighborhoods
- First improvement may be applied instead of best improvement
- Typically, order neighborhoods from smallest to largest
- If iterative improvement algorithms II_k, $k = 1, \ldots, k_{max}$ are available as black-box procedures:
  - order black-boxes
  - apply them in the given order
  - possibly iterate starting from the first one
  - order chosen by: **solution quality** and **speed**

Basic Variable Neighborhood Search

```
Procedure BVNS
input  : $N_k$, $k = 1, 2, \ldots, k_{max}$, and an initial solution $s$
output: a local optimum $s$ for $N_k$, $k = 1, 2, \ldots, k_{max}$
repeat
    $k \leftarrow 1$
    repeat
        $s' \leftarrow$ RandomPicking($s$,$N_k$)
        $s'' \leftarrow$ IterativeImprovement($s'$,$N_k$)
        if $f(s'') < f(s)$ then
            $s \leftarrow s''$
            $k \leftarrow 1$
        else
            $k \leftarrow k + 1$
    until $k = k_{max}$;
until Termination Condition;
```

To decide:

- which neighborhoods
- how many
- which order
- which change strategy

- Extended version: parameters k_{min} and k_{step}; set $k \leftarrow k_{min}$ and increase by k_{step} if no better solution is found (achieves diversification)

Outline

Constructive search

What is a **partial** solution (as opposed to a **complete** solution)?

- Solutions as subsets of a larger **ground set of solution components**
- Partial solutions as a representation of all candidate solutions that contain them
- Not all subsets of components are valid partial solutions
- Construction rule
- Assessment of partial solutions:
    - inferred from the sets of solutions that they represent
    - Lower bound (minimization) or upper bound (maximization)

Outline

Complete Search Methods

Tree (or graph) search in

Uninformed settings (satisfaction probs)

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search
- Bidirectional Search

Informed settings (optimization probs)

- best-first search, aka, greedy search
- A* search
- Iterative Deepening A*
- Memory bounded A*
- Recursive best first

In construction heuristics for this course, we can assume tree search of fixed known depth.

Complete Tree Search

Uninformed

Search Space

tree with branching factor at the top level nd
at the next level $(n-1)d$.
The tree has $n! \cdot d^n$ even if only d^n possible complete assignments.

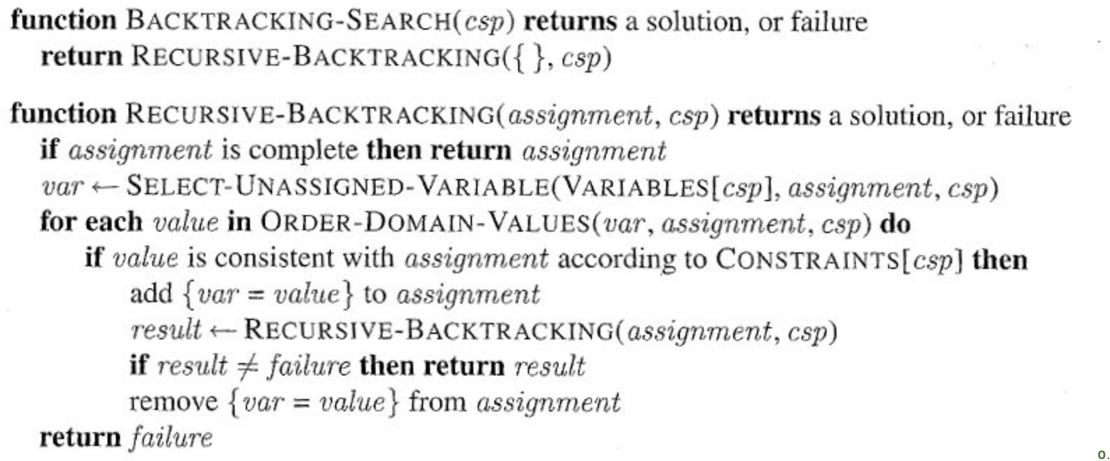
Informed

- CSP is **commutative** in the order of application of any given set of action. (we reach same partial solution regardless of the order)
- Hence generate successors by considering possible assignments for only a single variable at each node in the search tree.
- look-ahead, best first, etc.

Dealing with Constraints

Backtracking search

depth-first search that chooses one variable at a time and backtracks when a variable has no legal values left to assign.



```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
   return RECURSIVE-BACKTRACKING({ }, csp)

function RECURSIVE-BACKTRACKING(assignment, csp) returns a solution, or failure
   if assignment is complete then return assignment
   var ← SELECT-UNASSIGNED-VARIABLE(VARIABLES[csp], assignment, csp)
   for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
      if value is consistent with assignment according to CONSTRAINTS[csp] then
         add {var = value} to assignment
         result ← RECURSIVE-BACKTRACKING(assignment, csp)
         if result ≠ failure then return result
         remove {var = value} from assignment
   return failure
```

Backtrack Search

- No need to copy solutions all the times but rather extensions and undo extensions
- Since CSP is standard then the alg is also standard and can use general purpose algorithms for initial state, successor function and goal test.
- Backtracking is **uninformed and complete**. Other search algorithms may use **information** in form of heuristics.

Backtracking
General Concepts

Decisions in general purpose methods:

1) Which variable should we assign next, and in what order should its values be tried?

2) What are the implications of the current variable assignments for the other unassigned variables?

3) When a path fails – that is, a state is reached in which a variable has no legal values can the search avoid repeating this failure in subsequent paths?

Search (1) + Inference (2) + Backtracking (3) = Constraint Programming

In the general case, at point 1) we use heuristic rules.

If we do not backtrack (point 3) then we have a **construction heuristic**.

1) Which variable should we assign next, and in what order should its values be tried?

- Select-Initial-Unassigned-Variable
- Select-Unassigned-Variable
  - most constrained first = fail-first heuristic
    = Minimum remaining values (MRV) heuristic
    (tend to reduce the branching factor and to speed up pruning)
  - least constrained last

  Eg.: max degree, farthest, earliest due date, etc.
- Order-Domain-Values
  - greedy
  - least constraining value heuristic
    (leaves maximum flexibility for subsequent variable assignments)
  - maximal regret
    implements a kind of look ahead

2) **What are the implications of the current variable assignments for the other unassigned variables?**

Propagating information through constraints:

- Implicit in Select-Unassigned-Variable
- Forward checking (coupled with Minimum Remaining Values)
- Constraint propagation in CSP
  - arc consistency: force all (directed) arcs uv to be consistent:
    $\exists$ a value in $D(v)$: $\forall$ values in $D(u)$, otherwise detects inconsistency

    can be applied as preprocessing or as propagation step after each assignment (Maintaining Arc Consistency)

    Applied repeatedly

3) **When a path fails – that is, a state is reached in which a variable has no legal values can the search avoid repeating this failure in subsequent paths?**

Backtracking-Search

- chronological backtracking, the most recent decision point is revisited
- backjumping, backtracks to the most recent variable in the conflict set (set of previously assigned variables connected to X by constraints).

Outline

Dealing with Objectives

A* search

- The priority assigned to a node x is determined by the function

$$f(x) = g(x) + h(x)$$

  $g(x)$: cost of the path so far
  $h(x)$: heuristic estimate of the minimal cost to reach the goal from x.
- It is optimal if $h(x)$ is an
  - admissible heuristic: **never overestimates** the cost to reach the goal
  - consistent: $h(n) \leq c(n, a, n') + h(n')$
    (consistent $\implies$ admissible, only necessary in graph search)

A* best-first search

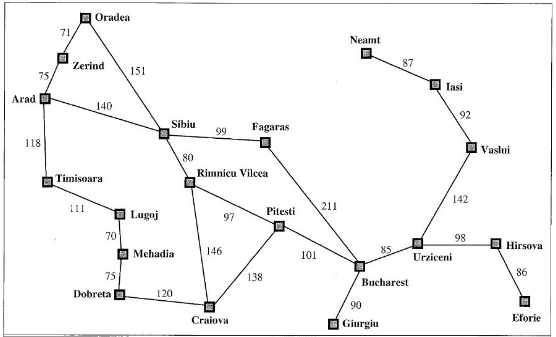


Figure 3.2 A simplified road map of part of Romania.

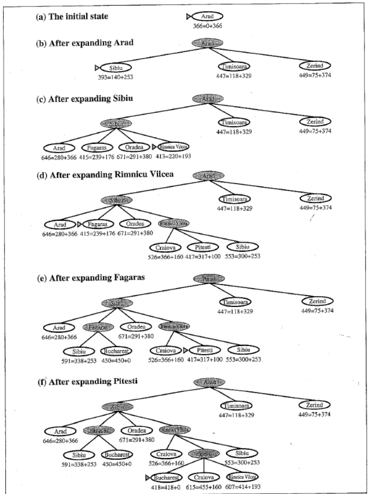


Figure 4.3 Stages in an A* search for Bucharest. Nodes are labeled with $f = g + h$. The h values are the straight-line distances to Bucharest taken from Figure 4.1.

A* search

Possible choices for admissible heuristic functions

- optimal solution to an easily solvable **relaxed problem**
- optimal solution to an easily solvable **subproblem**
- learning from experience by gathering statistics on state features
- preferred heuristics functions with higher values (provided they do not overestimate)
- if several heuristics available $h_1, h_2, \ldots, h_m$ and not clear which is the best then:

$$h(x) = \max\{h_1(x), \ldots, h_m(x)\}$$

A* search

Drawbacks

- Time complexity: In the worst case, the number of nodes expanded is exponential, (but it is polynomial when the heuristic function h meets the following condition:

$$|h(x) - h^*(x)| \leq O(\log h^*(x))$$

  h^* is the optimal heuristic, the exact cost of getting from x to the goal.)

- Memory usage: In the worst case, it must remember an exponential number of nodes. Several variants: including iterative deepening A* (IDA*), memory-bounded A* (MA*) and simplified memory bounded A* (SMA*) and recursive best-first search (RBFS)

Incomplete Search

Complete search is often better suited when ...

- proofs of insolubility or optimality are required;
- time constraints are not critical;
- problem-specific knowledge can be exploited.

Incomplete search is the necessary choice when ...

- non linear constraints and non linear objective function;
- reasonably good solutions are required within a short time;
- problem-specific knowledge is rather limited.

Outline

Greedy algorithms

Greedy algorithms

- Strategy: always make the choice that is best at the moment
- They are not generally guaranteed to find globally optimal solutions
  (but sometimes they do: Minimum Spanning Tree, Single Source Shortest Path, etc.)

We will see problem sepcific examples

Best-first search

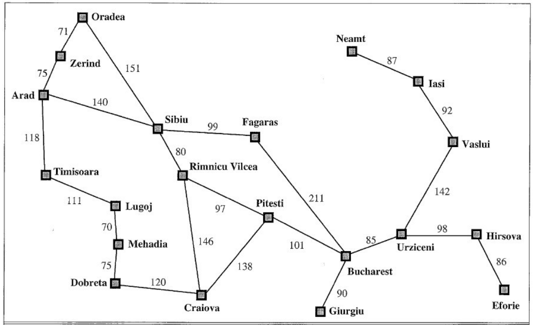


Figure 3.2 A simplified road map of part of Romania.

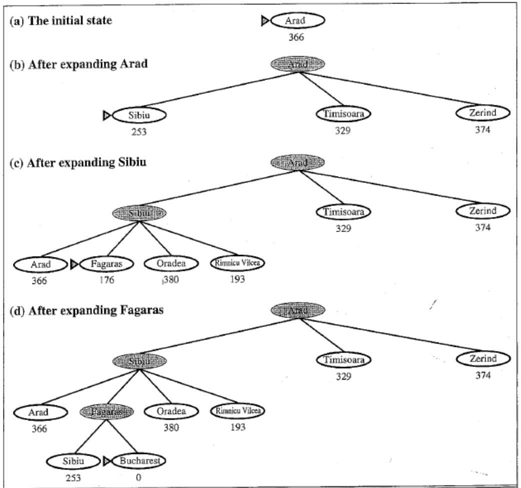


Figure 4.2 Stages in a greedy best-first search for Bucharest using the straight-line distance heuristic h_{SLD}. Nodes are labeled with their h-values.

Incomplete Search

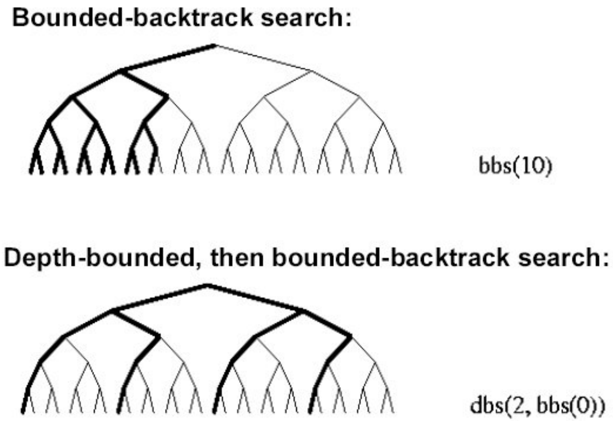
On backtracking framework
(beyond depth-first search)

- Bounded backtrack
- Credit-based search
- Limited Discrepancy Search
- Barrier Search
- Randomization in Tree Search
- Random Restart

Outside the exact framework
(beyond greedy search)

- Random Restart
- Rollout/Pilot Method
- Beam Search
- Iterated Greedy
- GRASP
- (Adaptive Iterated Construction Search)
- (Multilevel Refinement)

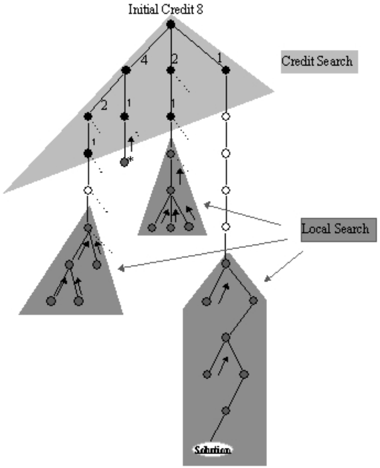
Bounded backtrack


Bounded-backtrack search:
bbs(10)
Depth-bounded, then bounded-backtrack search:
dbs(2, bbs(0))

http://4c.ucc.ie/~hsimonis/visualization/techniques/partial_search/main.htm

Credit-based search

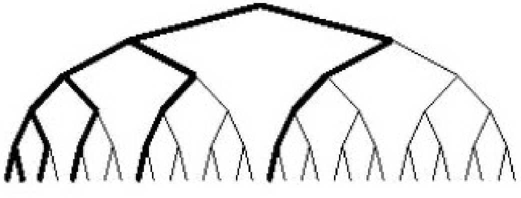
- Key idea: important decisions are at the top of the tree
- **Credit** = backtracking steps
- Credit distribution: one half at the best child the other divided among the other children.
- When credits run out follow deterministic best-search
- In addition: allow limited backtracking steps (eg, 5) at the bottom
- **Control parameters**: initial credit, distribution of credit among the children, amount of local backtracking at bottom.



Limited Discrepancy Search

Limited Discrepancy Search (LDS)

- Key observation that often the heuristic used in the search is nearly always correct with just a few exceptions.
- Explore the tree in increasing number of **discrepancies**, modifications from the heuristic choice.
- Eg: count one discrepancy if second best is chosen
  count two discrepancies either if third best is chosen or twice the second best is chosen
- **Control parameter**: the number of discrepancies



Randomization in Tree Search

The idea comes from complete search: the important decisions are made up in the search tree (backdoors - set of variables such that once they are instantiated the remaining problem simplifies to a tractable form)
$\rightsquigarrow$ random selections + restart strategy

Random selections

- randomization in variable ordering:
  - breaking ties at random
  - use heuristic to rank and randomly pick from small factor from the best
  - random pick among heuristics
  - random pick variable with probability depending on heuristic value
- randomization in value ordering:
  - just select random from the domain

Restart strategy in backtracking

- Example: $S_u = (1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 4, 8, 1, \ldots)$

Outline

Rollout/Pilot Method

Derived from A*

- Each candidate solution is a collection of m components $S = (s_1, s_2, \dots, s_m)$.
- Master process adds components sequentially to a partial solution $S_k = (s_1, s_2, \dots s_k)$
- At the k-th iteration the master process evaluates feasible components to add based on an **heuristic look-ahead strategy**.
- The evaluation function $H(S_{k+1})$ is determined by sub-heuristics that complete the solution starting from S_k
- Sub-heuristics are combined in $H(S_{k+1})$ by
  - weighted sum
  - minimal value

Note: this evaluation is not a lower bound!

Speed-ups:

- halt whenever cost of current partial solution exceeds current upper bound
- evaluate only a fraction of possible components

Outline

Beam Search

Based on the tree search framework:

- maintain a set B of bw (beam width) partial candidate solutions
- at each iteration extend each solution from B in fw (filter width) possible ways
- rank each $bw \times fw$ candidate solutions and take the best bw partial solutions
- complete candidate solutions obtained by B are maintained in B_f
- Stop when no partial solution in B is to be extended

Outline

GRASP

Greedy Randomized Adaptive Search Procedure

Key Idea: Combine randomized constructive search with subsequent local search.

Motivation:

- Candidate solutions obtained from construction heuristics can often be substantially improved by local search.
- Local search methods often require substantially fewer steps to reach high-quality solutions when initialized using greedy constructive search rather than random picking.
- By iterating cycles of constructive + local search, further performance improvements can be achieved.

Greedy Randomized "Adaptive" Search Procedure (GRASP):

```
while termination criterion is not satisfied do
    generate candidate solution $s$ using
        subsidiary greedy randomized constructive search
    perform subsidiary local search on $s$
```

- Randomization in **constructive search** ensures that a large number of good starting points for **subsidiary local search** is obtained.
- Constructive search in GRASP is 'adaptive' (or dynamic): Heuristic value of solution component to be added to a given partial candidate solution may depend on solution components present in it.
- Variants of GRASP without local search phase (aka **semi-greedy heuristics**) typically do not reach the performance of GRASP with local search.

Restricted candidate lists (RCLs)

- Each step of **constructive search** adds a solution component selected uniformly at random from a **restricted candidate list (RCL)**.
- RCLs are constructed in each step using a **heuristic function** h.
  - RCLs based on **cardinality restriction** comprises the k best-ranked solution components. (k is a parameter of the algorithm.)
  - RCLs based on **value restriction** comprise all solution components l for which $h(l) \leq h_{min} + \alpha \cdot (h_{max} - h_{min})$, where h_{min} = minimal value of h and h_{max} = maximal value of h for any l. (α is a parameter of the algorithm.)
  - Possible extension: **reactive GRASP** (*e.g.*, dynamic adaptation of α during search)

Example: Squeaky Wheel

Key idea: solutions can reveal problem structure which maybe worth to exploit.

Use a greedy heuristic repeatedly by prioritizing the elements that create troubles.

Squeaky Wheel

- **Constructor**: greedy algorithm on a sequence of problem elements.
- **Analyzer**: assign a penalty to problem elements that contribute to flaws in the current solution.
- **Prioritizer**: uses the penalties to modify the previous sequence of problem elements. Elements with high penalty are moved toward the front.

Possible to include a local search phase between one iteration and the other

Outline

Iterated Greedy

Key idea: use greedy construction

- alternation of **construction** and **deconstruction** phases
- an acceptance criterion decides whether the search continues from the new or from the old solution.

Iterated Greedy (IG):

```
determine initial candidate solution $s$
while termination criterion is not satisfied do
    $r := s$
    (randomly or heuristically) destruct part of $s$
    greedily reconstruct the missing part of $s$
    based on acceptance criterion,
        keep $s$ or revert to $s := r$
```

Adaptive Large Neighborhood Search

https://imada.sdu.dk/u/marco/DM841/slides/dm841-hr-alns.pdf

Outline

Ant Colony Optimization

https://imada.sdu.dk/u/marco/DM841/slides/dm841-hr-aco.pdf

Evolutionary Algorithms

https://imada.sdu.dk/u/marco/DM841/slides/dm841-hr-evolutionary.pdf