Lecture 7

Tabu Search,
Dynamic Local Search,
Iterated Local Search

Marco Chiarandini
Outline

1. Competition
2. Tabu Search
3. Dynamic Local Search
4. Ejection Chains and Dynasearch
5. Iterated Local Search
**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 neighbours of $s$
- choose a best improving candidate solution $s'$ in $N'$
- update tabu attributes based on $s'$
- $s := s'$
Example: Tabu Search for GCP – TabuCol

- **Search space:** set of all complete colourings of $G$.
- **Solution set:** proper colourings of $G$.
- **Neighbourhood relation:** one-exchange.
- **Memory:** Associate tabu status (Boolean value) with each pair $(v, c)$.
- **Initialisation:** a construction heuristic

**Search steps:**
- pairs $(v, c)$ are tabu if they have been changed in the last $tt$ steps;
- neighbouring colourings are admissible if they can be reached by changing a non-tabu pair or have fewer unsatisfied edge constr. than the best colouring seen so far (aspiration criterion);
- choose uniformly at random admissible colouring with minimal number of unsatisfied constraints.

- **Termination:** upon finding a proper colouring for $G$ or after given bound on number of search steps has been reached.
Note:

- Non-tabu search positions in $N(s)$ are called **admissible neighbours of $s$**.

- After a search step, the current search position or the solution components just added/removed from it are declared **tabu** for a fixed number of subsequent search steps (**tabu tenure**).

- Often, an additional **aspiration criterion** is used: this specifies conditions under which tabu status may be overridden (e.g., if considered step leads to improvement in incumbent solution).

- Crucial for efficient implementation:
  - keep time complexity of search steps minimal by using special data structures, incremental updating and caching mechanism for evaluation function values;
  - 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.
Note: Performance of Tabu Search depends crucially on setting of tabu tenure $tt$:

- $tt$ too low $\Rightarrow$ search stagnates due to inability to escape from local minima;
- $tt$ too high $\Rightarrow$ search becomes ineffective due to overly restricted search path (admissible neighbourhoods too small)

Advanced TS methods:

- **Robust Tabu Search** [Taillard, 1991]:
  repeatedly choose $tt$ from given interval;
  also: force specific steps that have not been made for a long time.

- **Reactive Tabu Search** [Battiti and Tecchioi, 1994]:
  dynamically adjust $tt$ during search;
  also: use escape mechanism to overcome stagnation.
Further improvements can be achieved by using *intermediate-term* or *long-term memory* to achieve additional *intensification* or *diversification*.

**Examples:**

- Occasionally backtrack to *elite candidate solutions*, *i.e.*, high-quality search positions encountered earlier in the search; when doing this, all associated tabu attributes are cleared.

- Freeze certain solution components and keep them fixed for long periods of the search.

- Occasionally force rarely used solution components to be introduced into current candidate solution.

- Extend evaluation function to capture frequency of use of candidate solutions or solution components.
Tabu search algorithms are state of the art for solving many combinatorial problems, including:

- SAT and MAX-SAT
- the Constraint Satisfaction Problem (CSP)
- many scheduling problems

**Crucial factors in many applications:**

- choice of neighbourhood relation
- efficient evaluation of candidate solutions (caching and incremental updating mechanisms)
Example: Tabu Search for QAP

- **Solution representation**: permutation $\pi$
- **Initial Solution**: randomly generated
- **Neighbourhood**: interchange
  \[ \Delta_I : \delta (\pi) = \{ \pi' | \pi_k' = \pi_k \text{ for all } k \neq \{r, s\} \text{ and } \pi_i' = \pi_j, \pi_j' = \pi_i \} \]
- **Tabu status**: forbid $\delta$ that place back the items in the positions they have already occupied in the last $tt$ iterations (short term memory)

- Implementation details:
  - compute $f(\pi') - f(\pi)$ in $O(n)$ or $O(1)$ by storing the values all possible previous moves.
  - maintain a matrix $[T_{ij}]$ of size $n \times n$ and write the last time item $i$ was moved in location $k$
  - $\delta$ is tabu if it satisfies both:
    - $T_i \pi(j) + \text{tabu list size} \geq \text{current iteration}$
    - $T_j \pi(i) + \text{tabu list size} \geq \text{current iteration}$
Example: Robust Tabu Search for QAP

- **Aspiration criteria:**
  - allow forbidden $\delta$ if it improves the last $\pi^*$
  - select $\delta$ if never chosen in the last $A$ iterations (long term memory)
- **Parameters:** $tt \in [0.9n, 1.1n + 4]$ and $A = 5n^2$
Example: Reactive Tabu Search for QAP

▶ **Aspiration criteria:**
  
  ▶ allow forbidden $\delta$ if it improves the last $\pi^*$

▶ **Tabu Tenure**
  
  ▶ maintain a hash table (or function) to record previously visited solutions

  ▶ increase $tt$ by a factor $\alpha_{inc}(=1.1)$ if the current solution was previously visited

  ▶ decrease $tt$ by a factor $\alpha_{dec}(=0.9)$ if $tt$ not changed in the last $sttc$ iterations or all moves are tabu

▶ Trigger escape mechanism if a solution is visited more than $nr (=3)$ times

▶ Escape mechanism $= 1 + (1 + r) \cdot ma/2$ random moves
Dynamic Local Search

- **Key Idea:** Modify the evaluation function whenever a local optimum is encountered.
- Associate *penalty 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.

**Dynamic Local Search (DLS):**

- determine *initial candidate solution* $s$
- initialise *penalties*
- While *termination criterion* is not satisfied:
  - compute *modified evaluation function* $g'$ from $g$ based on *penalties*
  - perform *subsidiary perturbative search* on $s$
  - using *evaluation function* $g'$
  - *update penalties* based on $s$
Dynamic Local Search (continued)

- **Modified evaluation function:**

\[
g'(\pi, s) := g(\pi, s) + \sum_{i \in SC(\pi', s)} \text{penalty}(i),
\]

where \( SC(\pi', s) \) = set of solution components of problem instance \( \pi' \) used in candidate solution \( s \).

- **Penalty initialisation:** For all \( i \): \( \text{penalty}(i) := 0 \).

- **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 perturbative 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:

[Voudouris and Tsang, 1995] Only increase penalties of solution components \( i \) with maximal utility:

\[
util(s', i) := \frac{g_i(\pi, s')}{1 + \text{penalty}(i)}
\]

where \( g_i(\pi, s') = \text{solution quality contribution of } i \text{ in } s' \).
Example: Guided Local Search (GLS) for the TSP
[Voudouris and Tsang 1995; 1999]

- **Given**: TSP instance $G$
- **Search space**: Hamiltonian cycles in $G$ with $n$ vertices;
- **Neighbourhood**: 2-edge-exchange;
- **Solution components** edges of $G$;
  \[ f(G, p) := w(p); \quad f_e(G, p) := w(e); \]
- **Penalty initialisation**: Set all edge penalties to zero.
- **Subsidiary perturbative search**: Iterative First Improvement.
- **Penalty update**: Increment penalties for all edges with maximal utility by
  \[ \lambda := 0.3 \cdot \frac{w(s_{2-opt})}{n} \]
  where $s_{2-opt} = 2$-optimal tour.
Ejection Chains

- Attempt to use large neighborhoods without examining them exhaustively
- Sequences of successive steps each influenced by the precedent and determined by myopic choices
- Limited in length
- Local optimality in the large neighborhood is not guaranteed.

Example (on TSP): successive 2-exchanges where each exchange involves one edge of the previous

Example (on GCP): successive 1-exchanges: a vertex $v_1$ changes colour from $\varphi(v_1) = c_1$ to $c_2$, in turn forcing some vertex $v_2$ with color $\varphi(v_2) = c_2$ to change to another color $c_3$ (which may be different or equal to $c_1$) and again forcing a vertex $v_3$ with colour $\varphi(v_3) = c_3$ to change to colour $c_4$. 
Dynasearch

- Iterative improvement method based on building complex search steps from combinations of simple search steps.

- Simple search steps constituting any given complex step are required to be *mutually independent*, i.e., do not interfere with each other w.r.t. effect on evaluation function and feasibility of candidate solutions.

*Example:* Independent 2-exchange steps for the TSP:

```
    u_1  u_i  u_{i+1}  u_j  u_{j+1}  u_k  u_{k+1}  u_l  u_{l+1}  u_n  u_{n+1}
```

*Therefore:* Overall effect of complex search step = sum of effects of constituting simple steps; complex search steps maintain feasibility of candidate solutions.

- **Key idea:** Efficiently find optimal combination of mutually independent simple search steps using *Dynamic Programming*.
Hybrid LS Methods

Combination of ‘simple’ LS methods often yields substantial performance improvements.

Simple examples:

- Commonly used restart mechanisms can be seen as hybridisations with Uninformed Random Picking
- Iterative Improvement + Uninformed Random Walk = Randomised Iterative Improvement
Iterated Local Search

**Key Idea:** Use two types of LS steps:

- subsidiary perturbative (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 behaviour.

**Iterated Local Search (ILS):**

1. determine initial candidate solution \( s \)
2. perform *subsidiary perturbative search* on \( s \)
3. While termination criterion is not satisfied:
   - \( r := s \)
   - perform *perturbation* on \( s \)
   - perform *subsidiary perturbative search* on \( s \)
   - based on *acceptance criterion*,
     - keep \( s \) or revert to \( s := r \)
Note:

- *Subsidiary perturbative 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 perturbative search*, *perturbation mechanism* and *acceptance criterion* need to complement each other well.
Subsidiary perturbative search:

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

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

- Needs to be chosen such that its effect *cannot* be easily undone by subsequent perturbative search phase. (Often achieved by search steps larger neighbourhood.)
  Example: perturbative 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 perturbative 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.
Acceptance criteria:

- Always accept the *better* of the two candidate solutions
  ⇒ ILS performs Iterative Improvement in the space of local optima reached by subsidiary perturbative search.

- Always accept the *more recent* of the two candidate solutions
  ⇒ ILS performs random walk in the space of local optima reached by subsidiary perturbative search.

- Intermediate behaviour: 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*. 
Example: Iterated Local Search for the TSP (1)

- **Given:** TSP instance $G$.

- **Search space:** Hamiltonian cycles in $G$.

- **Subsidiary perturbative search:** Lin-Kernighan variable depth search algorithm

- **Perturbation mechanism:**
  ‘double-bridge move’ = particular 4-exchange step:

- **Acceptance criterion:** Always return the better of the two given candidate round trips.
Example: Iterated Local Search for the TSP (2)

Note:
- Double-bridge move perturbation cannot be directly reversed by a sequence of 2-exchange steps as performed by ”usual” LK implementations.
- This perturbation is empirically shown to be effective independent of instance size.

Note:
- This ILS algorithm for the TSP is known as *Iterated Lin-Kernighan (ILK) Algorithm*.
- Although ILK is structurally rather simple, an efficient implementation was shown to achieve excellent performance [Johnson and McGeoch, 1997].
Iterated local search algorithms . . .

- are typically rather easy to implement (given existing implementation of subsidiary simple LS algorithms);

- achieve state-of-the-art performance on many combinatorial problems, including the TSP.

There are many LS approaches that are closely related to ILS, including:

- Large Step Markov Chains [Martin et al., 1991]

- Chained Local Search [Martin and Otto, 1996]

- Variants of Variable Neighbourhood Search (VNS) [Hansen and Mladenović, 2002]