LOCAL SEARCH DRIVEN BY EVOLUTIONARY ALGORITHM AND ITS APPLICATIONS

Jiří Kubalík
email: kubalik@labe.felk.cvut.cz

Department of Cybernetics
Czech Technical University in Prague
Technicka 2, 166 27, Prague,
Czech Republic
POEMS – Hybrid metaheuristic combining
- Iterated local search
- Evolutionary algorithm

Design of POEMS for single-objective optimization problems
- Solution representation, evolutionary algorithm, niching
- Successful applications

Design of POEMS for multi-objective optimization problems
- Pool of prototypes, multi-objective EA, diversity maintenance
- Successful applications
Optimization Problems of Interest

- **Black-box optimization**
  - No information about what the optimal solution looks like, no information about how to go about finding it in a principled way
  - Very little information about what the optimized function looks like
  - Very little heuristic information to go on
  - Brute-force search is out of the question because of the huge search space
  - We can create or modify inputs to the optimized function, check corresponding candidate solutions, test them and assess their quality

- **Metaheuristics**
  - Single-state
  - Population-based
Single-State Metaheuristics

- **Single-state metaheuristics**
  - operate on a single solution
  - prone to get stuck in a local optimum

- **Components of single-state metaheuristics:**
  - Initialization procedure
  - Assessment procedure
  - Neighborhood structure that defines for every solution \( s \) a set of its neighbors \( N(s) \).
    - Typically, defined by a single variation operator to be applied at given stage of the search process.
  - Selection procedure to decide which candidate solutions to retain and which to reject.
  - Strategy to help escape from a local optimum basin of attraction.

- **Examples:** Hill-climbing with random restarts, Simulated annealing, Tabu search, Iterated local search, Variable neighborhood search

- **Issue:** How to find an optimal balance between exploitation and exploration?
Population-Based Metaheuristics

- Population-based metaheuristics operate on a sample (population) of candidate solutions
  - Information stored in the whole population of candidate solutions is used to direct the search process in subsequent iterations.
  - Less prone to get stuck in a local optimum.

- Evolutionary Algorithms (EAs) - genetic algorithms, genetic programming, ...

- EA components:
  - Initialization procedure
  - Assessment procedure
  - Sampling operators – crossover, mutation, ...
  - Selection procedure to pick candidates from the current population that will take part in the process of creating new candidates for the next generation.
  - Replacement strategy to decide which candidates will survive and which will die.

- Issues
  - How to design a proper crossover operator that would allow effective mixing of partial solutions of two or more parental candidates?
  - Computational demands grow high (often exponentially) with the size of the problem instance.
POEMS Characteristics

- **POEMS** is an iterative optimization framework that operates on a solution **Prototype**.

  Evolutionary algorithm is employed to evolve the best modification of the current Prototype in each iteration.

  Modifications are represented as fixed length sequences of problem specific variation operators (actions).

- POEMS takes the best of the two worlds of the single-state and population-based metaheuristics:
  - Contrary to single-state metaheuristics, that define the neighborhood by a single variation operator, **POEMS searches much bigger neighborhood**. Therefore, the exploration capability of POEMS is better than that of the standard single-state techniques.
  - The prototype’s neighborhood is effectively searched by means of an evolutionary algorithm. The space of prototype’s neighbors is smaller than a space of all candidate solutions to the given problem, thus, moderate resources (i.e. small population size and small number of generations) can be sufficient to effectively search the prototype’s neighborhood.

  It is up to the EA to identify whether **fine-tuning moves or large jumps** are optimal at given stage of the search process.

\begin{algorithm}
\begin{algorithmic}[1]
\STATE $i \leftarrow 0$
\STATE $Prototype^{(i)} \leftarrow \text{generatePrototype}()$
\WHILE {POEMS termination condition not satisfied}
\STATE $i \leftarrow i + 1$
\STATE $BestSequence \leftarrow \text{runEA}(Prototype^{(i-1)})$
\STATE $Cand \leftarrow \text{apply}(BestSequence,Prototype^{(i-1)})$
\IF {$Cand$ is better than or equal to $Prototype^{(i-1)}$}
\STATE $Prototype^{(i)} \leftarrow Cand$
\ELSE
\STATE $Prototype^{(i)} \leftarrow Prototype^{(i-1)}$
\ENDIF
\ENDWHILE
\STATE \textbf{return} $Prototype^{(i)}$
\end{algorithmic}
\end{algorithm}
Actions and Action Sequences

- **Action** is a structure [actionType, actionParameters]
  - actionType – either any problem-specific active action or *nop* (no operation) action, *nop action*
    - void action with no effect on the prototype, regardless of the values of its parameters
    - one or more *nop* actions allowed in a chromosome
    - variable effective length of chromosomes
  - actionParameters

- **Action Sequence** is a linear sequence of actions of maximal length *MaxGenes*
  - *MaxGenes* << *SolutionSize*
  - At least one action must be active
  
  Ex.: *MaxGenes* = 5, Actions = {action1, action2, action3}
  
  AS = (action3, 0.8)(nop, 0.7)(action2, 0.2)(nop, 0.5)(action2, 0.2)

- **Action sequences are assessed based on how they improve/worsen the prototype.**
  - Sequences that do not change the prototype at all are fatally penalized in order to avoid a convergence to useless trivial modifications.
Genetic Operators

- Tournament selection
- Crossover – generalized uniform operator
  Given two parental chromosomes, any combination of their actions forms a valid offspring
  Ex.:
  \[ Par1 = (\text{action3}, 0.8), (\text{nop}, 0.7), (\text{action1}, 0.2), (\text{nop}, 0.5), (\text{action2}, 0.2) \]
  \[ Par2 = (\text{nop}, 0.8), (\text{action1}, 0.8), (\text{action3}, 0.6), (\text{nop}, 0.5), (\text{nop}, 0.5) \]
  \[ Off1 = (\text{action1}, 0.2), (\text{action3}, 0.6), (\text{nop}, 0.7), (\text{action1}, 0.2), (\text{nop}, 0.5) \]
  \[ Off2 = (\text{nop}, 0.8), (\text{action2}, 0.2), (\text{action3}, 0.6), (\text{nop}, 0.5), (\text{action_1}, 0.8) \]

- Mutation – either the `actionType` or its `actionParameters` are changed
- Optionally, multiple duplicates of the same genes are eliminated

original: \[ Off1 = (\text{action1}, 0.2), (\text{action3}, 0.6), (\text{nop}, 0.7), (\text{action1}, 0.2), (\text{nop}, 0.5) \]

after mutation: \[ Off1' = (\text{action1}, 0.5), (\text{action3}, 0.6), (\text{nop}, 0.7), (\text{nop}, 0.2), (\text{nop}, 0.5) \]
Evolutionary Algorithm: Niching

- Steady-state evolution model
  - fast convergence

- Obviously, once the Prototype is already of a good quality then it becomes very likely that any randomly initialized action sequence will worsen it.

- Naturally, EA would tend to converge towards action sequences with smaller number of active actions as they worsen the Prototype less than action sequences with greater number of active actions.

- In order to prevent the evolutionary algorithm from converging towards action sequences with a minimal number of active actions
  - niching tournament selection and
  - niching replacement strategy

are used.
Population divided into $MaxGenes$ equally sized niches, each niche can contain only action sequences with at least $niche_Id$ active actions.

Niching tournament selection
1. Select one niche at random.
2. Run the tournament selection within the niche.

Only action sequences from the same niche compete to each other.

Niching replacement strategy
Given a new action sequence that has $N$ active actions and fitness $F$
1. Look in all niches with
   $niche_Id \leq N$
   in an ascending order for a replacement with fitness worse than or equal to $F$.
2. If such a replacement was found then replace it with the new action sequence.
   Otherwise throw the new action sequence away.

The population is kept diverse through the whole EA run.

<table>
<thead>
<tr>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>$niche_Id = 5$</td>
</tr>
<tr>
<td>action sequences with $N_{active} \geq 5$</td>
</tr>
<tr>
<td>$niche_Id = 4$</td>
</tr>
<tr>
<td>action sequences with $N_{active} \geq 4$</td>
</tr>
<tr>
<td>$niche_Id = 3$</td>
</tr>
<tr>
<td>action sequences with $N_{active} \geq 3$</td>
</tr>
<tr>
<td>$niche_Id = 2$</td>
</tr>
<tr>
<td>action sequences with $N_{active} \geq 2$</td>
</tr>
<tr>
<td>$niche_Id = 1$</td>
</tr>
<tr>
<td>action sequences with $N_{active} \geq 1$</td>
</tr>
</tbody>
</table>
Successful Applications

- **POEMS**
  - Sorting networks design
  - Multiple sequence alignment problem
  - Shortest common supersequence problem
  - Floorplanning
  - Fragment assembly problem
  - Design of hyper-heuristic

- **mPOEMS**
  - SW project portfolio optimization
  - Agent clustering in multi-agent systems
TSP: Proof-of-Concept Experiments

- **Basic Actions**
  - $move(city_1, city_2)$ moves $city_1$ right after $city_2$ in the tour
  - $invert(city_1, city_2)$ inverts a subtour between $city_1$ and $city_2$
  - $swap(city_1, city_2)$ swaps $city_1$ and $city_2$

- EA evolved action sequences of length 10.

- Prototype initialized using the nearest neighbor heuristic.

- Instances of size up to 2000 cities were tested.
TSP: Example of POEMS Run

Initial tour 965.134

after iteration 1

after iteration 2

after iteration 3

Final tour 824.8

<table>
<thead>
<tr>
<th>iteration</th>
<th>prototype fitness</th>
<th>evolved action sequence</th>
<th>final fitness</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>965.134</td>
<td>(move 58 64), (invert 24 40)</td>
<td>952.550</td>
</tr>
<tr>
<td>2</td>
<td>952.550</td>
<td>(move 79 73), (invert 24 18),</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(invert 24 35), (move 79 80),</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(invert 24 40)</td>
<td>927.025</td>
</tr>
<tr>
<td>3</td>
<td>927.025</td>
<td>(invert 97 82), (invert 83 82),</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(move 58 64)</td>
<td>919.573</td>
</tr>
<tr>
<td>4</td>
<td>919.573</td>
<td>(invert 48 47), (invert 69 72),</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(invert 35 47), (swap 7 3)</td>
<td>904.033</td>
</tr>
</tbody>
</table>
Example of Evolved Solution

Initial prototype of length 4403

Final prototype of length 3655
**Problem:**

- Given: A set of primary biological DNA sequences.
- Goal: To find an optimal alignment of the sequences w.r.t. the scoring scheme. Typically used to identify regions of similarity among the sequences.

Gaps ‘-’ are inserted between the residues (amino acids) so that residues with identical or similar characters are aligned in successive columns.

**Ex.:**

<table>
<thead>
<tr>
<th>Orig. seqs</th>
<th>Alignment</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>R T C F G D D H</td>
</tr>
<tr>
<td>S2</td>
<td>T C T F C A D D H</td>
</tr>
<tr>
<td>S3</td>
<td>R T A T F G A D D T</td>
</tr>
</tbody>
</table>

| S1         | R T C - - F G - D D H - |
| S2         | - T C - T F C A D D H - |
| S3         | R T - A T F G A D D - T |
Multiple Sequence Alignment Problem

- **Prototype** – aligned sequences represented by a matrix; each row is one sequence

  
<table>
<thead>
<tr>
<th>S1</th>
<th>R</th>
<th>T</th>
<th>C</th>
<th></th>
<th>F</th>
<th>G</th>
<th></th>
<th>D</th>
<th>D</th>
<th>H</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>S2</td>
<td>–</td>
<td>T</td>
<td>C</td>
<td>–</td>
<td>T</td>
<td>F</td>
<td>C</td>
<td>A</td>
<td>D</td>
<td>D</td>
<td>H</td>
</tr>
<tr>
<td>S3</td>
<td>R</td>
<td>T</td>
<td>–</td>
<td>A</td>
<td>T</td>
<td>F</td>
<td>G</td>
<td>A</td>
<td>D</td>
<td>D</td>
<td>–</td>
</tr>
</tbody>
</table>

- **Basic actions** (variation operators used as building blocks of the action sequences)
  - *PassGaps*(sequenceId, resId) – moves the residue with the resId in the sequence with sequenceId to the other end of the gap (if possible).
  - *MoveBlock*(sequenceId, resId, direction) – moves a contiguous block of residues that contains given residue with the resId one position either to the left or to the right (if possible).
  - *InsertGap*(sequenceId, resId) – a single gap is inserted right in front of the residue with the resId (if possible).
  - *RemoveGap*(sequenceId, resId) – removes a gap that is right in front of the residue with the resId in the sequence sequenceId (if possible).

- **Fitness**

  \[
  \text{AlignmentScore} = \text{SumOfPairs} - \text{GapPenalty}
  \]

  – uses BLOSUM62 substitution matrix – scores alignments of each pair of amino acids based on the similarity of the amino acids' chemical properties.
  – distinguishes between gap opening penalty and gap extension penalty
Compared algorithms

- Clustal X (widely used alignment algorithm)
- $(\mu+\lambda)$-Evolution strategy,
- Best improvement local search algorithm.

Both algorithms use the set of elementary actions proposed for POEMS as the mutation/variation operators.

Main achievements

- POEMS significantly improved solutions generated by Clustal X.
- POEMS clearly outperformed both compared mutation-based optimization approaches.

**Sorting Networks Design**

- **Sorting Network** is a network of \( n \) wires and \( k \) comparators arranged such that for any input sequence of numbers, the output sequence is monotonically sorted. Each **comparator** performs a compare-swap operation; it connects two wires and outputs the two input values in desired order.

- **Given:** A number of inputs.
- **Goal:** To find a perfect sorting network for the given number of inputs that is optimal w.r.t. the given efficiency measure – such as a size (number of comparators) or a number of parallel layers.

**Ex.:**

- Size: 15; #layers: 10
- Size: 12; #layers: 6
Sorting Networks Design

- **Prototype** represented as an ordered list of comparators of a variable length, where each comparator is described by a triple \((Id, Wire_1, Wire_2)\).
  
  Ex.:
  
  \[
  \text{Prototype} = ((1,1,2), (2,3,6), (3,4,5), \ldots, (11,4,5), (12,3,4))
  \]

- **Actions**
  
  - \textbf{SwapComparators}\((Id_1, Id_2)\) – swaps comparators \(Id_1\) and \(Id_2\)
  
  - \textbf{MoveComparator}\((Id_1, Id_2)\) – moves comparator \(Id_1\) after comparator \(Id_2\)
  
  - \textbf{MoveBlock}\((Id_1, Id_2, Id_3)\) – moves a block of comparators between \(Id_1\) and \(Id_2\) after comparator \(Id_3\)
  
  - \textbf{ModifyComparator}\((Id_1, Pin_1, Pin_2)\) – redefines inputs to comparator \(Id_1\)
  
  - \textbf{InsertComparator}\((Id_1, Id_2, Pin_1, Pin_2)\) – inserts new comparator \(Id_1\) after comp. \(Id_3\)
  
  - \textbf{RemoveComparator}\((Id_1)\) – removes/inactivates comparator \(Id_1\)

- **Fitness** – both, the correct as well as incorrect sorting networks are considered.
  
  A sorting score as well as the network size contribute to the SN quality

  \[
  \text{fitness} = \text{NetworkScore} + \left(1 - \frac{\text{UsedComparators}}{\text{MaxComparators}}\right)
  \]
Compared algorithms

- $(\mu+\lambda)$-Evolution strategy,
- $(1+\lambda)$-Evolution strategy.

In both cases, random structured mutations composed of elementary actions designed for the POEMS were used.

Main achievements

- POEMS was able to find sorting networks of the minimal known size up to 16 inputs (the 16-input sorting network is the largest one documented in literature).
- This was achieved without any problem-specific heuristic or local search routines.

1. 60 comparators in 10 parallel layers

2. 60 comparators in 10 parallel layers

3. 60 comparators in 11 parallel layers
**Problem:** Large strands of DNA need to be broken into small fragments that can be read automatically. In the end of the process, the original DNA sequence have to be reconstructed from the fragments. The problem of reconstructing the DNA sequence is known to be NP-hard.

- Given: A set of fragments of unknown orientation.
- Goal: Determine the most likely DNA sequence that can be described by the set of fragments.

**Classical DNA reconstruction process: Overlap-layout-consensus**

1. overlap
2. layout
3. consensus

**Layout phase:** Given the set of overlap relationships between the fragments, determine the optimal order and orientation of all fragments.
Fragment Assembly Problem

- **Prototype** – a sequence of $N$ fragments, where each fragment has its orientation flag (direct vs. reverse-complement orientation).

- **Actions**
  - $MoveFragment(id, k)$ – moves a fragment id by $\text{abs}(k)$ positions to the right or left.
  - $ReverseComplementFragment(id)$ – applies a reverse-complement operation to nucleotides of fragment id.
  - $MoveBlock(id1, id2, k)$ – moves a block of fragments delimited by fragments id1 and id2 by $\text{abs}(k)$ positions to the right or left.
  - $ReverseComplementBlock(id1, id2)$ – reverses a block of fragments (id1 – id2).
  - $MoveReverseComplementBlock(id1, id2, k)$ – combines the previous two actions.

- **Fitness** – reflects two goals
  - to minimize the number of contigs (sequences of fragments in which all adjacent fragments overlap)
  - to maximize the overlap between the adjacent fragments
**Fragment Assembly Problem**

- **Compared algorithm**
  - Problem Aware Local Search (PALS) - a baseline heuristic assembler algorithm. Formally, it is a local search algorithm that explores complete neighborhood of the current solution defined by a single variation move.
  - Phrap and CAP3 - commercially available assembly packages.

- **Main achievements**
  - **POEMS achieves consistently better results than PALS.**
    - POEMS explores more complex neighborhood, defined by a sequence of actions, effectively using an evolutionary algorithm.
  - **POEMS is competitive to commercially available assembly packages Phrap and CAP3.**

---


Problem:
- Given: A set $L$ of strings $s_i$ composed of symbols from a certain alphabet $\Sigma$.
- Goal: To find a minimal-length sequence $S$ of symbols from $\Sigma$ such that all strings in set $L$ can be embedded in $S$, where string $r$ is embedded in $S$ iff all symbols of $r$ are present in $S$ in the very same order (although not necessarily consecutive).

Ex.:
- $s_1 = \{caagccacctacatca\}$
- $s_2 = \{cgagccatccgtaagttg\}$
- $s_3 = \{agaacctgctaatgctaga\}$

\[
\begin{align*}
  s_1 &: \text{ ca ag cca cc ta cat c a} \\
  s_2 &: \text{ c gag ccat ccgtaa g tt g} \\
  s_3 &: \text{ aga acc tgc taaatgc t a ga}
\end{align*}
\]

**Supersequence $S$:** cagagaccatgccgtaaatgcattacga

length=28
Prototype is represented as a variable length ordered list of symbols from alphabet \( \Sigma \).

- does not have to be a valid supersequence, i.e. the prototype is not necessarily a supersequence of each string in \( L \).

Ex.: Given \( \Sigma = \{a, c, t, g\} \), \( L = \{ctg, tca, tag\} \), all of the following strings are valid prototypes

- \( P_1 = {} \)
- \( P_2 = \{t \ c \ a \ t\} \)
- \( P_3 = \{t \ c \ t \ g \ a \ g\} \)

Prototype is initialized to an empty list.

Actions

- \( MoveSymbol(id1, id2) \) - moves a symbol with id1 right after a symbol with id2,
- \( InsertSymbol(id, l) \) - inserts a new symbol \( l \) right after a symbol with id,
- \( DeleteSymbol(id) \) - deletes a symbol with given id,
- \( ChangeSymbol(id, l) \) - changes a symbol with id according to \( l \),
- \( SwapSymbols(id1, id2) \) – swaps the two symbols.

Fitness

\[
F(s) = C(s) + S(s)
\]

Takes into account a coverage achieved by \( s \), \( C(s) \), calculated as the number of symbols of strings in \( L \) correctly covered by \( s \) and the size of the (super)sequence, \( S(s) \).
Moving window optimization strategy specifies a section of interest of a maximal size $W$ within the constructed (super)sequence.
- Only the window of (super)sequence can be optimized in one POEMS iteration.
- Construction of the (super)sequence starts from an empty sequence.

Ex.:

$$
\begin{array}{l}
\text{s}_1: \text{ca ag cc} \quad | \quad \text{a c c ta} \quad \text{catcttagaatgctccttt} \\
\text{s}_2: \text{cagag cc} \quad | \quad \text{at c c ta} \quad \text{aagttagataatgcctccttt} \\
\text{s}_3: \text{aga acc} \quad | \quad \text{t gct} \quad \text{a aatgctaga} \\
\hline
\text{(super)sequence} \quad | \quad \text{s: cagagacc} \quad \text{atgctcgta} \\
\hline
\end{array}
$$

Due to the fitness function of the form $F(s) = C(s) + S(s)$, there is a strong bias towards growing the (super)sequence.

Remedy: Epoch-wise optimization
- The optimized fragment of the (super)sequence of size $W$ remains unchanged for several POEMS iterations.
- This way a pressure towards short (super)sequences is put already along the process of creating a valid supersequence.
Due to the fitness function of the form

\[ F(s) = C(s) + S(s) \]

there is a strong bias towards growing the (super)sequence.

Example:

- \( s_1: \text{ca ag cca} \quad \text{c c tac} \text{atcttaggaatggcctcttt} \)
- \( s_2: \text{cagag ccat} \quad \text{c c ta} \text{aagttagataatgcctcttt} \)
- \( s_3: \text{aga acc t gct} \quad \text{a} \text{aatgctgta} \)

(super)sequence \( s: \text{cagagaccattgctgta} \)

Remedy: Epoch-wise optimization

- The optimized fragment of the (super)sequence of size \( W \) remains unchanged for several POEMS iterations.
  
  A number of iterations in one epoch is control parameter of the algorithm.

- Intensifies the search towards short supersequences already along the process of creating a valid supersequence.
Multi-prototype version
- Pool of unique prototypes is maintained
- Each action sequence has assigned its evaluation prototype
- Crossover operation can be applied to any pair of action sequences
- After the EA has finished a new set of unique prototypes is chosen for next iteration
Compared algorithms for moderate size synthetic and real biological data sets – up to 10 sequences of app. length 1000 [MABS].
- Majority Merge (MM),
- Probabilistic Beam Search (PBS),
- Hybrid Memetic Algorithm – Beam Search (MA-BS) - current state-of-the-art metaheuristic technique for the SCS problem.

Compared algorithm for large scale biological data sets (DNA and protein sequences) – up to 500 sequences of length 1000 and 1000 sequences of length 500.
- Deposition and reduction algorithm [DR],
- Enhanced beam search algorithm [Beam]

Main achievements
- POEMS beats MM and PBS and performs equally to MA-BS on moderate size data.
- POEMS produced significantly better solutions than the baseline DR algorithm on the challenging large data sets.
- POEMS is competitive and sometimes better than the state-of-the-art Enhanced beam search algorithm on the largest biological data sets.

Problem: Floorplanning also known as 2D rectangle packing problem.
- Given: A set of $N$ unoriented blocks (rectangles) with fixed dimensions.
- Goal: To place all blocks on a plane so that there is no overlap between any pair of rectangles and the bounding rectangular area is minimal (or the dead space is minimal).

Ex.: Blocks \{a, b, c, d, e, f\} should be placed in a rectangular area so that the bounding rectangular area is minimal (or the dead space, shown in gray, is minimal).

Prototype solution (floorplan) is encoded by B*-Tree non-slicing representation.

Floorplanning

- **Actions** (variation operators used as building blocks of the action sequences)
  - *Flip* – rotates a block of the specified tree node.
  - *Mirror* – swaps the left and right child of the specified tree node.
  - *Rotate* – recursively flips and mirrors child nodes of the specific node.
  - *ExchangeNodes* – exchanges blocks between two specified tree nodes.
  - *ExchangeSubtrees* – exchanges sub-trees rooted in two specified tree nodes.
  - *MoveSubtree* – places a specified node with its sub-tree to a new position in the tree.
Floorplanning

- **Compared algorithms**
  - CompaSS - commercially available program.
  - B*-Tree/l, B*-Tree/SA – approaches using the same B*-tree representation searching the space of all possible floorplans by means of local search algorithm and simulated annealing, respectively.

- **Main achievements**
  - POEMS generates floorplans comparable and better than the ones generated by B*-Tree/l and B-Tree/SA algorithms.
  - POEMS outperforms CompaSS algorithm w.r.t. the quality of the generated floorplans on some test data.
Examples of results obtained for 300 blocks.

3.8% dead space

4.6% dead space

Floorplanning

Visualization of a POEMS run on data with 300 blocks.
“A hyper-heuristic can be seen as a (high-level) methodology which, when given a particular problem instance or class of instances, and a number of low-level heuristics (LLH), or their components, automatically produces an adequate combination of the provided components to effectively solve the given problem(s).” [Burke et al., 2009]

Classification of hyper-heuristics

- heuristic selection - methodologies for choosing or selecting existing heuristics
- heuristic generation – methodologies for generating new heuristics from components of existing heuristics

Each category can be further divided into

- construction heuristics – build a solution incrementally, starting with an empty solution.
- perturbation heuristics - start with a complete solution, try to iteratively improve the current solution.
“A hyper-heuristic can be seen as a (high-level) methodology which, when given a particular problem instance or class of instances, and a number of low-level heuristics (LLH), or their components, automatically produces an adequate combination of the provided components to effectively solve the given problem(s).” [Burke et al., 2009]

Classification of hyper-heuristics

- heuristic selection - methodologies for choosing or selecting existing heuristics
- heuristic generation – methodologies for generating new heuristics from components of existing heuristics

Each category can be further divided into

- construction heuristics – build a solution incrementally, starting with an empty solution.
- perturbation heuristics - start with a complete solution, try to iteratively improve the current solution.

Approaches

- Iterated Local Search, Variable Neighborhood Search, Tabu Search, Simulated Annealing, Genetic Algorithms, Hill-climbing, Case-based Reasoning, ...

generate \textit{currentSolution}

choose \textit{LLHs}

\textit{candidateSolution} \leftarrow \text{apply\_on}(\textit{LLHs}, \textit{currentSolution})

\textit{currentSolution} \leftarrow \text{accept}(\textit{candidateSolution}, \textit{currentSolution})

\text{stopping condition fulfilled?}

\text{return } \textit{currentSolution}
HyFlex [Ochoa et al., 2012] – software interface for hyper-heuristics that provides solution representation, evaluation function and a set of low-level heuristics for several hard combinatorial problem domains.

User’s part in designing the hyper-heuristic is to implement high-level strategies that intelligently combine the set of heuristics provided.

Problems: Max-SAT (SAT), Bin Packing (BP), Personnel Scheduling (PS), Flow Shop (FS), Traveling Salesman Problem (TSP), Vehicle Routing Problem (VRP)

Low-level heuristics
- Local search
- Mutation
- Ruin-recreate
- Crossover

where the local search LLHs are controlled with the “depth of search” parameter and the mutation and ruin-recreate LLHs are controlled with the “intensity of mutation” parameter. Both parameters take values from the interval (0.0, 1.0).

Iterated Search Driven by EA (ISEA)

- **Prototype is a fixed-length sequence of LLHs.**

  Ex.: [LLH₄(DoS, IoM)] [LLH₇(DoS, IoM)] [LLH₂(DoS, IoM)] [LLH₃(DoS, IoM)]

Prototype is initialized with a specified structure:
- Only local search LLH can be chosen at the first and the last position.
- LLHs of any type can be generated at inner positions.

Idea behind the structure: By default, when trying to improve any solution, one might try to
- locally optimize it first, then
- modify it somehow (using the mutation and rui-recreate operators) and finally
- locally optimize it again.

The prototype is being modified by evolved action sequences, so that different sequences of LLHs can be created and applied on the current solution at different stages of the run.
Three basic actions for modifying the prototype

- **addLLH(position, type, parameters)** – adds a new LLH of certain type with specified parameters to the prototype at given position.
  
  Ex.: `addLLH(position=2, type=local_search, {name=1, DoS=0.7})`

- **removeLLH(id)** – removes LLH with given id from the prototype.
  
  Ex.: `removeLLH(id=1)`

- **changeLLH(id, parameters)** – modifies the LLH with given id in the prototype according to the specified parameters.
  Changes either the type of the LLH, or chooses a new name of the same LLH type, or changes parameters DoS and IoM.
  
  Ex.: `changeLLH(id=5, {name=5})`
ISEA: Action Sequence Evaluation

Action sequence:

```
addLLH(2, L, name=1, DoS=0.3)  nop  nop  removeLLH(id=3)  nop
```

Apply action sequence on prototype

Prototype:

```
0   1   2   3
LLH₄(DoS, IoM) LLH₇(DoS, IoM) LLH₂(DoS, IoM) LLH₅(DoS, IoM)
```

Modified prototype:

```
0   1   2   3
LLH₄(DoS, IoM) LLH₇(DoS, IoM) **LLH₁(DoS=0.3)** LLH₅(DoS, IoM) LLH₇(...)
```

Evaluate modified sequence of LLHs on working sol.

Fitness assignment: The quality of the best solution observed during applying the LLHs of the modified prototype.
ISEA: Adaptive Perturbation

initialize(evaluation_solution)

best_so_far ← evaluation_solution
evaluations ← 0

lastPerturbationTime ← 0; lastImprovementTime ← 0;
initialize(prototype)
initialize(actionSequencesPopulation)

while(!hasTimeExpired())
    calculateGenerationEA()
    update(P_reinit, T_exp, T_min)
    evaluations ← evaluations + 2
    if((currentTime() − lastPerturbationTime > T_exp) or
       ((currentTime() − lastPerturbationTime > T_min)
        and (rand() < P_reinit)))
        evaluation_solution ← perturb(best_so_far)
        lastPerturbationTime ← currentTime()
    if(evaluations > maxEvaluations)
        evaluations ← 0
        initialize(prototype)
        initialize(actionSequencesPopulation)

Return best_so_far
ISEA: Experimental Setup

Cross-domain Heuristic Search Challenge (CHeSC 2011)

- Six optimization problem domains (5 instances per domain):
  General problem-independent hyper-heuristic is sought.
  - Max-SAT (SAT)
  - Bin Packing (BP)
  - Personnel Scheduling (PS)
  - Flow Shop (FS)
  - Traveling Salesman Problem (TSP)
  - Vehicle Routing Problem (VRP)
- Total running time per run 600s
- 31 runs per instance
- Performance indicator: Median from the set of 31 best-of-run values
- ISEA variants compared against 19 CHeSC 2011 competing algorithms.
- The Formula 1 point system used for calculating competitors’ score on each problem:
  the winner takes 10 points, the 2nd takes 8 points, then 6, 5, 4, 3, 2 and 1 point respectively.
When compared to the CHeSC 2011 competitors, the ISEA-adaptive would ranked as the second best hyper-heuristic with the total score of 145.75 points.

<table>
<thead>
<tr>
<th>algorithm</th>
<th>Total</th>
<th>SAT</th>
<th>BP</th>
<th>PS</th>
<th>FS</th>
<th>TSP</th>
<th>VRP</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISEA-chesc</td>
<td>rank</td>
<td>8</td>
<td>9</td>
<td>2</td>
<td>5</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>points</td>
<td>71.0</td>
<td>6.0</td>
<td>30.0</td>
<td>14.5</td>
<td>3.5</td>
<td>12.0</td>
</tr>
<tr>
<td>ISEA-adaptive</td>
<td>rank</td>
<td>2</td>
<td>14</td>
<td>1</td>
<td>4</td>
<td>2-3</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>points</td>
<td>145.75</td>
<td>0.25</td>
<td>42.0</td>
<td>22.5</td>
<td>34.0</td>
<td>24.0</td>
</tr>
</tbody>
</table>
Multiobjective Optimizations

- **Multiobjective optimizations involve conflicting objectives**
  - How to compare/rank solutions?
  - Which are the optimal solutions?
- **Dominance** – a solution \( x \) dominates the other solution \( y \), if \( x \) is no worse than \( y \) in all objectives and \( x \) is strictly better than \( y \) in at least one objective.
- **Pareto-optimal set** – contains all solutions of the entire search space \( P \) that are not dominated by any other solution of \( P \).
- **The goal in multiobjective optimization** is to find a set of solutions
  1. as close as possible to the Pareto-optimal front, and
  2. as diverse as possible.
- **Multi-objective Evolutionary Algorithms (MOEAs)** are very suitable for solving multi-objective optimization problems.
mPOEMS is an optimization algorithm based on the dominance concept.

SolutionBase is a set of best solutions found so far. Best-of-run solutions are the non-dominated solutions of the SolutionBase.

Function choose_prototype() chooses a prototype as one of the non-dominated solutions of the SolutionBase.

MOEA takes into account quality and uniqueness of generated neighbours of the current prototype.
**mPOEMS: Multiobjective EA**

**input:** *Prototype, SolutionBase*

**output:** Set of evolved action sequences

1. `generate(OldPop)`  // generates starting population of action sequences
2. `calculate_objectives(OldPop)`
3. `evaluate(OldPop, SolutionBase)`  // assigns single-valued fitness
4. Repeat
   // create NewPop by means of selection, crossover, mutation
   5. `NewPop ← evolutionary_cycle(OldPop)`
   6. `calculate_objectives(NewPop)`
   // use the best of OldPop and NewPop to create new OldPop
   7. `OldPop ← merge(OldPop, NewPop)`
8. `evaluate(OldPop, SolutionBase)`  // assigns fitness values
9. until (EA termination condition is fulfilled)
10. return `OldPop`  // returns evolved action sequences (*Prototype* modifications)
In each iteration the *Prototype* is chosen among non-dominated solutions of the current solution base so that all partitions of the non-domination set are sampled with an equal sampling rate.

**Selection schema:**

1. A set $S$ of $k$ candidate prototype solutions are selected from the non-dominated set of the current solution base by using a greedy heuristic.
2. In each iteration a new *Prototype* is randomly selected from the set $S$ (being removed from the set $S$ to avoid repetitions).
   - If any solution $s_i$ in $S$ becomes dominated by new solutions introduced to the solution base, then $s_i$ is removed from $S$.
   - If set $S$ is empty a new sample of the non-dominated solutions of the current solution base is selected.
Set $S$ is constructed iteratively:

1. Empty $S$, $\text{counter} = 0$.
2. Add extreme solutions to $S$ and update $\text{counter}$ accordingly.
3. While ($\text{counter} < k$)
   - Use a tournament selection to choose a non-dominated solution from solution base that maximizes a normalized Euclidean distance $d_{ij}$ (in the objective space) to the nearest solution in $S$.
   \[
   d_{ij} = \sqrt{\sum_{k=1}^{m} \left( \frac{o_k^{(i)} - o_k^{(j)}}{u_k - l_k} \right)^2}
   \]
   - Add the solution to $S$.
   - Increment($\text{counter}$).
Merging New Solutions and Solution Base

1. New solutions produced by action sequences evolved by the MOEA are merged with the current solution base resulting in a temporary population $P_{\text{temp}}$ of size $\text{PopulationSize} + \text{SolutionBaseSize}$.

2. Non-dominanted fronts of $P_{\text{temp}}$ are ranked according to a crowding distance metric (NSGA-II) and only the best solutions are added to the new SB.
# mPOEMS: Multiobjective 0/1 Knapsack

## Coverage of two sets

<table>
<thead>
<tr>
<th></th>
<th>NSGA-II</th>
<th>SPEA2</th>
<th>mPOEMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>n=2</td>
<td>NSGA-II</td>
<td>-</td>
<td>0.604</td>
</tr>
<tr>
<td></td>
<td>SPEA2</td>
<td>0.283</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>mPOEMS</td>
<td>0.708</td>
<td>0.966</td>
</tr>
<tr>
<td>n=3</td>
<td>NSGA-II</td>
<td>-</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>SPEA2</td>
<td>0.887</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>mPOEMS</td>
<td>0.955</td>
<td>0.431</td>
</tr>
<tr>
<td>n=4</td>
<td>NSGA-II</td>
<td>-</td>
<td>0.006</td>
</tr>
<tr>
<td></td>
<td>SPEA2</td>
<td>0.844</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>mPOEMS</td>
<td>0.961</td>
<td>0.971</td>
</tr>
</tbody>
</table>

## Size of the space covered

<table>
<thead>
<tr>
<th></th>
<th>NSGA-II</th>
<th>SPEA2</th>
<th>mPOEMS</th>
</tr>
</thead>
<tbody>
<tr>
<td>n=2</td>
<td>0.497</td>
<td>0.492</td>
<td><strong>0.490</strong></td>
</tr>
<tr>
<td>n=3</td>
<td>0.689</td>
<td>0.689</td>
<td><strong>0.688</strong></td>
</tr>
<tr>
<td>n=4</td>
<td>0.820</td>
<td>0.822</td>
<td><strong>0.818</strong></td>
</tr>
</tbody>
</table>

## Non-dominating solutions for 2 k

![Graph showing non-dominating solutions for 2 k](image)
Agents Clustering within MAS

- **Problem:** Static agents clustering within multi-agent system from messages. The problem of reconstructing the DNA sequence is known to be NP-hard.
  - Given: Data with records about the number of messages sent and received between every pair of agents.
  - Goal: To find a partitioning of agents into \( N \) clusters such that communication within clusters is maximized and inter-cluster communication is minimized and the clusters are of similar size in order to efficiently distribute agents across multiple execution units.
  - Approach: Multiobjective optimization approach - mPOEMS.
  - Practical impact of this problem:
    - Agent Distribution Optimization
    - MAS Observation improvement
    - Data mining for unknown systems

- **Prototype** solution (agents partitioning) is represented as an integer vector of size \( N \), where the number at position \( i \) refers to a cluster assigned to the \( i \)-th agent.
Agents Clustering within MAS

- **Action** (variation operators used as building blocks of the action sequences)
  - *ChangeCluster* – assigns specified agent to a new cluster.

- **Compared algorithms**
  - Fuzzy clustering algorithm – clusters are represented by Gaussian membership functions. For each pattern, the algorithm determines whether it belongs to a sufficient degree to some existing cluster. If yes, the pattern is added to that cluster and the cluster center and volume is updated. Otherwise, a new cluster is created.
  - Hierarchical agglomerative algorithm.

- **Main achievements**
  - mPOEMS exhibits its ability to generate a set of tradeoff solutions with various distinct values of the difference in clusters sizes criterion from which a user can choose the best one in the end.
  - mPOEMS outperforms both the compared algorithms, as it finds better solutions with respect to both considered criteria.

Example: MAS, where agents are used for control of the transportation of products or discrete materials on the factory’s shop floor using a network of conveyor belts. Goal is to find an optimal partitioning of 67 agents into 4 clusters.
mPOEMS: Initialization of SB
mPOEMS: After 1st iteration
mPOEMS: After 2\textsuperscript{nd} iteration
mPOEMS: After 5\textsuperscript{th} iteration
mPOEMS: After 10th iteration
mPOEMS: After 20\textsuperscript{th} iteration
mPOEMS: After 100\textsuperscript{th} iteration
mPOEMS: Final SB

Solutions spread along the lower-right boundary of the solution space.
Problem: Large software companies have to plan their project portfolio to maximize potential portfolio return and strategic alignment, while balancing various preferences, and considering limited resources.

- Given: A pool of $N$ projects, each described by a set of attributes such as a return generated by the project, resources needed to successfully finish the project, synergy effects to other projects, etc.
- Goal: Select a set of projects (i.e. choose the project portfolio) such that multiple criteria such as
  - potential revenue,
  - strategic alignment,
  - resource usage distribution metric (to be ),
  - risk,
  - synergy
are optimized, while considering limited resources.

Prototype solution (i.e. the project portfolio) is represented as an integer vector of size $N$, where the number at position $i$ specifies the timeframe when the $i$-th project starts (value 0 means the project is not selected to the portfolio).
**Actions** (variation operators used as building blocks of the action sequences)

- *ChangeMonth* – changes the starting month of a given project,
- *SwitchOnOff* – adds given project to the portfolio if it is not included there yet, and vice versa.

**Compared algorithms**

- NSGA-II and SPEA2 – two state-of-the-art multiobjective optimization evolutionary algorithms.

**Main achievements**

- mPOEMS proved to be capable to deal with a complex set of goals and restriction.
- mPOEMS exhibits better capabilities of finding sets of good trade-off solutions than the compared state-of-the-art algorithms.

---

POEMS is a general-purpose optimization approach for hard combinatorial, discrete, and real-valued optimization problems with huge search space.

It is well suited for solving problems of the following classes:

- **Black-box** optimization problems where no information about the optimized function is available (such as gradients, properties of the fitness landscape, etc.).
- Problems where standard evolutionary algorithms fail due to a large solution representation (long chromosome, large solution graph or network) and for which it is hard to design an efficient crossover operator.
- **Design of hyper-heuristics** - described as "heuristics to choose heuristics", are search methods or learning mechanisms for selecting or generating heuristics to solve computational search problems.
- **Dynamic optimizations** - are a class of problems whose specifications, evaluation function and/or problem-specific constraints, change over time, resulting in continuously moving optima.
- **Problems involving a refinement** of an initial existing solution, representatives of which are re-planning and re-scheduling.

(m)POEMS is easy to implement as both main strategies – the iterative optimization framework and evolutionary algorithm – are conceptually very simple.

mPOEMS in Java: https://github.com/mr-stateradio/mpoems-in-java

Any interesting problem for (m)POEMS?
References


