Technische
Universität
Braunschweig

Institut für Betriebssysteme
und Rechnerverbund
Algorithmik

Algorithm Engineering
Lecture 12: Metaheuristics

(Meta)heuristics

Previous lectures:

- Given some hard problem, solve instances optimally
- Or at least produce very high-quality solutions (with good bounds)

Often:

- Small(ish) instances can be solved like this
- This can suffice (thanks to powerful solvers & modeling techniques)
- Easy to implement, reasonable first step
- Very large instances typically impractical to solve
- Often, the optimal solution is not necessary
- Theory: approximation algorithms, practice: heuristics & meta-heuristics

Heuristics & Metaheuristics

Heuristic:

- A heuristic is an algorithm that attempts to solve a problem ‘well enough’
  - In general: need not be successful
  - In general: no guarantees on solution quality
  - Should be efficient (more efficient than an exact solve)

Metaheuristic:

- A design pattern for heuristics that works for several problems
- Normally uses problem-specific subroutines
- We will see some of the most important/well-known metaheuristics
- Performance of resulting heuristics: depends on problem, instance, hyperparameters

Interplay of Metaheuristics & Exact Optimization

Heuristics informing solvers:

- MIP starts (bounds for pruning)
- Phases (near-feasible solutions)
- Good incumbent solutions feed solver-internal heuristics (RINS, Cut Selection, …)

Solvers used in heuristics:

- Large Neighborhood Search (LNS) & variants: can directly use solvers
- Analysis of heuristic quality, missed opportunities

Plan for Today

Overview over (meta)heuristic approaches & types:

- Constructive heuristics
- Trajectory-based algorithms
- Population-based algorithms
- Orthogonal: heuristics for feasibility problems

Constructive Heuristics

Typical constructive heuristics:

- Greedy-like ad-hoc/problem-specific heuristics
- LP rounding/iterated rounding
- GRASP
- Pilot method/rollout
- Beam search
- Squeaky wheel/construct-analyze-reconstruct
- …

Usually hybridized with trajectory- or population-based methods!

GRASP

GRASP (Greedy Randomized Adaptive Search Procedure):

- Multi-start heuristic with randomized greedily constructed starting points
- Each iteration: special randomized greedy
  - Do not pick the ‘best’ (according to greedy) element, but randomly from RCL
  - Restricted Candidate List (RCL): list of ‘best’ (according to greedy) elements
  - Size of RCL can be adapted by GRASP
- Afterwards, solutions are improved via some trajectory-based algorithm
- Then, the next iteration starts
- Often extended/hybridized; can easily be combined with other approaches

Pilot method/Rollout

Observation: greedy uses a ‘myopic’ score to score each candidate extension.

Alternative: use some cheap base heuristic to ‘pilot’ each candidate to completion. Then, take the candidate resulting in the best objective.

Points to note:

- Can easily become too expensive; normally, candidates are filtered first (e.g., RCL)
- Alternatively, we can truncate the base heuristic and estimate the rest.
- But: this can destroy ‘comparability’ of scores (all scores are complete solutions).
- Better base heuristic can be *worse* overall; ranking of candidates is what is important!

Beam Search

- Visits a partial search tree
- Basically, on each level/at each depth, only the best B nodes are kept
- Set at level $i \rightarrow$ expand, score, prune $\rightarrow$ set at level $i + 1$
- Typically, two scoring functions (filtered beam search):
  - One should be very fast and scores every child at level i
  - One can be slower and scores only the B' best children per node
- Score should be an estimate of the total cost of a complete solution
- Issue to keep in mind: in many problems, partial solutions at level i can be differently far away from complete; biases many obvious scoring functions!

Squeaky Wheel Heuristics

For some problems, like graph coloring:

- Greedy can be optimal
- If the choices are made in the right order
- Any arbitrary order gives a solution, quality changes

Idea:

- After an iteration, analyze the solution; find ‘squeaky wheels’ (who is to blame)
- Increase their priority (so the ‘squeaky wheels’ are no longer the worst parts)
- Start the heuristic again

- Also called construct-analyze-reconstruct

Trajectory-based Algorithms

Trajectory-based heuristics: move from solution to solution in solution space.

- Local Search
- Simulated Annealing
- Path Relinking
- Variable Neighborhood Search — VNS
- (Adaptive) Large Neighborhood Search — (A)LNS
- Limited Discrepancy Search
- …

Local Search

Local search:

- Have seen multiple examples of this (e.g., 2-opt for TSP)
- Given an initial solution (e.g., from constructive heuristics):
  - Interpret the problem as optimization problem over the solution space
  - Each solution is a point in this space
- While searching for improvements:
  - Move from one solution to a ‘neighboring’ solution
  - Simplest variant (‘Hill Climbing’): only move to a neighbor that is better

Fundamental question/subroutine: What are the neighbors of a given solution?

Fundamental problem: Local optima

Local Search

What are the neighbors of a solution in k-OPT?

- Tours that can be reached by replacing k edges by k other edges.
- Alternative (not k-OPT): tours we can reach by removing and reinserting k points
- In general, many neighborhoods are possible

Treatments for local optima:

- Repetition from different initial solutions (Multi-Start Heuristic, GRASP, …)
- Perturbation of locally-optimal solutions (‘kicks’) → Iterated Local Search (ILS)
- Accepting ‘worsening’ moves
- Adaptive neighborhoods (larger/different neighborhoods can allow an escape)

Simulated Annealing

Simulated annealing:

- Origin: energy-efficient, stable, near-optimal crystal after cooling hot metals
- **Idea:** initially high ‘temperature’ allows strongly worsening moves
- Successive cooling lowers the acceptance probability for worsening moves
- We try to emulate this physical process
- Parameters:
  - Neighborhood (as in local search)
  - Decreasing temperature sequence $T_0, \ldots, T_t, \ldots$ over time steps
  - Fitness function (e.g., objective) $f(x)$ to evaluate solutions

Simulated Annealing — Algorithm Template

Simulated annealing (for minimization):

- Start from initial solution x, $x^* \leftarrow x$, $t \leftarrow 0$, $T \leftarrow T_0$
- **In every step:**
  - Choose (e.g., uniformly at random) a neighbor y of x
  - If $f(y) \leq f(x)$: $x \leftarrow y$, $x^* \leftarrow \arg\min(f(x^*), f(y))$

    Else: $x \leftarrow y$ with probability $\exp\left(\frac{f(x) - f(y)}{T}\right)$
  - $t \leftarrow t + 1$, $T \leftarrow T_t$
- Until stopping criterion is satisfied (e.g. good enough, #iterations, time limit)
- Again: multiple starting points, perturbation and other extensions possible
- **Important:** Normalizing fitness function vs. temperatures

Simulated Annealing — Parameter Tuning

Parameters: Simulated annealing requires well-tuned parameters.

- Normally, the step to a neighbor is repeated H_t times before incrementing t, and thus changing temperature etc.
- Initially, H_t is small; it is increased while running the algorithm.
  Idea: H_t can be considered a ‘neighborhood size’
- Acceptance probability of a medium worsening solution ~40-60%
- Cooling schedule: typically geometric ($T_{i+1} = \gamma \cdot T_i,\ \gamma \leq 1$), γ often close to 1
- Alternative cooling (slower): $T_{i+1} = T_i / (1 + \varepsilon T_i)$

Simulated annealing (and variants) have many successful applications

Relatively general algorithm template; parameter tuning can be tricky and time-consuming.

Simulated Annealing — Caveats

Bad neighborhood definition:

- Many options for neighborhoods
- Not all are good; there are trade-offs between efficiency and size/diversity

Temperature drops too quickly:

- Getting stuck in local optima

Temperatur drops too slowly:

- We hit points near the global optimum, but do not stay there
- More computational effort than necessary

Initial temperature is too low:

- Strong dependence on the initial solution

Path Relinking

During a multi-start local optimization, store ‘elite’ solutions.

- Walk from one elite solution to another through the neighborhood structure
- Depending on neighborhood/problem, this can be fairly straightforward
- Search for improvements on the path or in its neighborhood
- Sometimes also called ‘intensification’
- Can profit from *diversification* in the elite solution pool:
  - Penalties or incentives (changes to scoring function)
  - Dropping solutions from the elite pool that are too close to others
  - Can lead to better coverage of the search space and avoid local optima

Variable Neighborhood Search

Variable neighborhood search:

- Often, multiple neighborhoods are available
- Or neighborhoods are parameterized with some size parameter
- Some are more expensive then others
- A local optimum may only be a local optimum for a few neighborhoods
- A local optimum can stop being locally optimal for sufficiently large neighborhoods
- A global optimum is an optimum under all neighborhoods

Idea of VNS:

- We have multiple, different neighborhoods, or a neighborhood with size parameter
- We regularly change between neighborhoods
- Some neighborhoods can be more expensive without slowing us down too much
- Can avoid local optima, in particular if the neighborhoods are very different
- Can be combined/hybridized with other approaches (simulated annealing, …)

Ruin & Recreate and LNS

Ruin & recreate — basic idea:

- Destroy operators
  - Take a solution and partially destroy it
  - Can have multiple destroy operators; typically have size parameters
- Repair operators
  - Take a destroyed solution and repair it
  - Can have multiple repair operators
- Accept the solution according to some acceptance criterion (e.g., simulated annealing)
- Allows relatively large neighborhoods to be explored.
- Repairs can be heuristic, but can also be exact (CP-SAT, MIP, SAT, …)
- In the latter case: we are searching through a (exponentially) large neighborhood (LNS)
- Allows use of solvers, expensively trained neural operators, … on limited sizes

LNS and Exact Repair

Exact search repair operators in LNS:

- Repair runtime scales exponentially with destruction magnitude/size
- Runtime is very hard to predict, depends on instance and destruction
- A time limit for individual repair operations is usually in place
- Typically, the destruction magnitude is taken from an interval, adapted dynamically:
  - On timeouts, reduce upper bound
  - On stagnation, increase lower/upper bound
  - The time limit may also be modified over time (allow larger operations later on)
- Sometimes, an ‘improve by one’ approach is possible (feasibility problem to repair)
- Exact repair and heuristic repair can coexist

Adaptive LNS

Adaptive LNS: Select repair/destroy operators & parameters adaptively

- Multiple repair operators/destroy operators can exist
- Can have different costs/different strengths (depending on instance, problem, destroy)
- This diversity alone can be beneficial
- ALNS tries to tune the selection of which operator to use while running
- Multiple schemes exist; in many cases, uniform random selection is good enough
- Otherwise, selection schemes from the Multi-Armed Bandit problem can be used
- For exact repair, tuning by time/success often dominates
- For portfolios: parallel solver $\leftrightarrow$ multiple single-threaded solvers

Limited Discrepancy Search

Another option to use exact solvers:

- Search a solution in a ‘neighborhood’ of a given solution $\hat{x}$
- Basically, taking an existing MIP/CP-SAT formulation:
  - Add a $|\hat{x} - x| \leq \delta$ constraint (limit the discrepancy δ, hence the name)
  - Often much easier to solve (search space severely limited) than the full formulation
  - Can be worthwhile; can also be quite costly
- Does not limit the search to specific variables like a LNS repair subproblem does, thus can be a bit broader

Population-based Algorithms

Previous methods: Trajectory-based

Now: Population-based

- Population of multiple solutions
- Typically updated generation-to-generation
- Want to improve solutions over time
- Want to keep diversity → better exploration of the search space

Examples:

- Evolutionary algorithms
- Particle swarms
- …

Evolutionary Algorithms

Evolutionary algorithms:

- Covers the entire field of algorithms loosely based on genetics/natural selection
- Applied to optimization, it is normally called genetic algorithms

Genetic algorithms:

- Population of solutions (tens to thousands), ideally diverse
- Fitness function to evaluate (e.g., objective or objective-derived)
- In each generation, we select a subset of ‘parents’
- We often also choose a few less fit parents (to maintain diversity)
- Create children from parents (new generation):
  - By combining parents (crossover or recombination)
  - Possibly additionally changed by a mutation operator
- Depending on the precise setup, some parents can also be kept

Evolutionary Algorithms: Parameters

Parameters:

- Crossover operator, mutation operator, fitness function
- Parents per child, keeping parents, mate selection
- Population size, number of populations (e.g., male/female, …)
- Genotype vs phenotype
- Mutation probability
- Additional operators (migration, extinctive pressure, …)
- Speciation (prevention of too ‘similar' mates, ‘incest’)
- Niche penalty/filters (filter or penalize too many too similar members)

Genetic Algorithms: TSP Example

For tours: just use objective value as fitness

Mutation: perform 2-OPT (or just some 2-OPT moves)

Biggest question: crossover?

Edge Assembly Crossover (Nagata):

- Start at a node; parent (1) blue, parent (2) red
- Alternate between blue and red edges
- Results in a single subtour S (3)
- Take the blue tour, remove blue edges from S
- Insert red edges from S; keeps degree 2 everywhere
- Merge subtours via 2-OPT moves

Yields competitive tour heuristics (for some instances)!

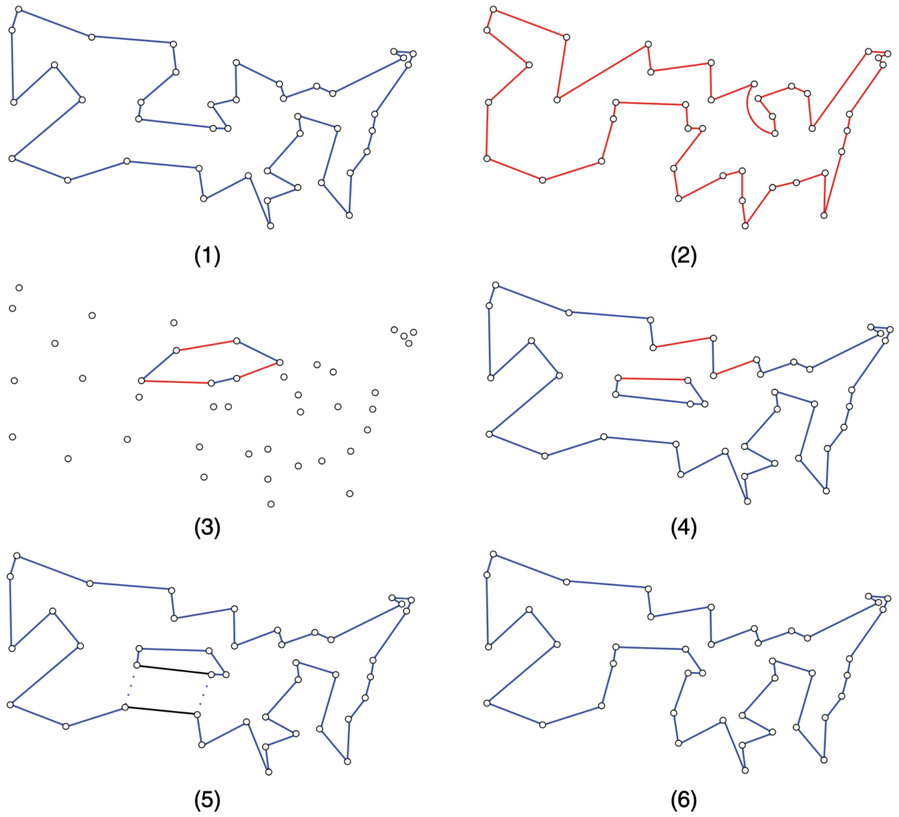


Figure 4.27, William J. Cook:
In Pursuit of the Traveling Salesman

Examples: Art via TSP

Memetic Algorithms

Observation:

- Evolutionary algorithms often explore quite well
- They can struggle converging to local optima
- Local search strategies converge (exploit) better, but explore less well

Idea: Combine evolutionary & local search algorithms.

- Apply local search to some offspring in the genetic algorithm
- Parameters:
  - Intensity, frequency, how many/which offspring?
  - Which local operators?
  - Changing only the objective or also the ‘genome’?
  - Challenge: avoiding convergence too early, keeping diversity alive

Other Approaches

- Ants, Bats, Bees, Bacteria, Cats, Hawks, Wolves, Fireflies, Root Growth, River Formation, Gravity, Swarm Intelligence, Empires, Harmonies, …
- A lot of ‘nature-inspired’ algorithms… many very similar to each other.

1. The optimization literature is rich with heuristic search methodologies for both discrete and continuous spaces. Proposing new paradigms is only acceptable if they contain innovative basic ideas, such as those that are embedded in classical frameworks like genetic algorithms, tabu search, and simulated annealing. The Journal of Heuristics avoids the publication of articles that repackage and embed old ideas in methods that are claimed to be based on metaphors of natural or man-made systems and processes. These so-called “novel” methods employ analogies that range from intelligent water drops, musicians playing jazz, imperialist societies, leapfrogs, kangaroos, all types of swarms and insects and even mine blast processes (Sörensen, 2013). If a researcher uses a metaphor to stimulate his or her own ideas about a new method, the method must nevertheless be translated into metaphor-free language, so that the strategies employed can be clearly understood, and their novelty is made clearly visible. (See items 2 and 3 below.) Metaphors are cheap and easy to come by. Their use to “window dress” a method is not acceptable.

Heuristics for (Disguised) Feasibility Problems

With the TSP:

- Many different solutions, rarely the same objective
- Quality changes in neighborhood are clear; $f(x)$ straightforward

Not always the case — consider, e.g., coloring!

- Objective value is no good indicator for improvement.
- Often better: make initial solution invalid and minimize some conflict metric

Concrete example (coloring):

- Pick color class and discolor it (or recolor it randomly)
- Conflicts: uncolored vertices or monochromatic edges

At 0 conflicts: one color eliminated; continue with the next color class

Example: Coloring Intersection Graphs of Line Segments

CG:SHOP Help Competitions Login Register

CG:SHOP 2022

Download Submit Solution

Organized by: Sándor Fekete (TU Braunschweig),
Phillip Keldenich (TU Braunschweig),
Dominik Krupke (TU Braunschweig),
Stefan Schirra (University of Magdeburg)

40 teams participating
Sept. 19, 2021, midnight (UTC) - Jan. 19, 2022, 11:59 p.m. (AoE)

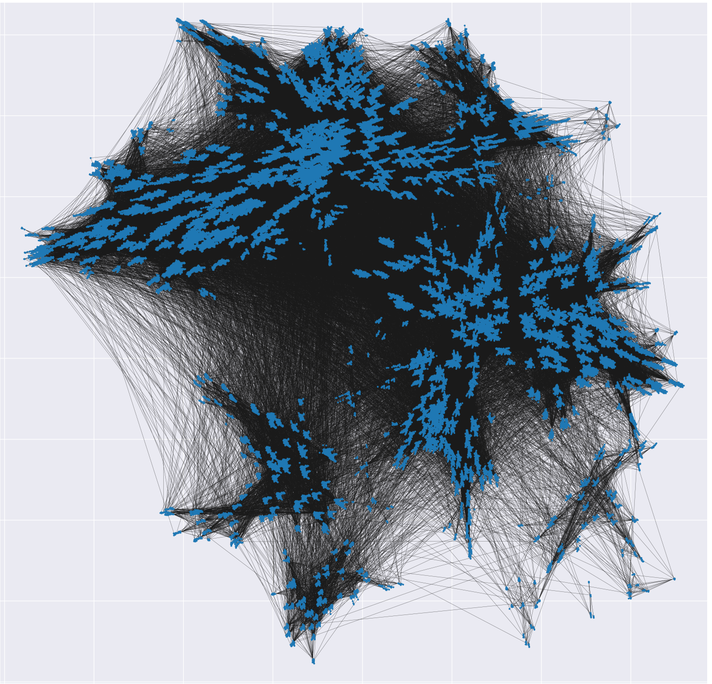
view all competition news

The winners of the 2022 Challenge will present their approach at SoCG, Thursday, June 09, 15:30-16:30 CET. See the full ... 05/28/2022 11:25 a.m.

read more ...

The top teams have been notified via mail. 01/20/2022 11:48 a.m.

The competition has ended. To view your score and the score of the best teams, please refer to the ranking tab.



Conclusion

We covered:

- Heuristics to find & improve solutions
- Not many details; high level. You should know what to search for.
- Some guidance, principles, pitfalls
- If you do algorithms: high likelihood you will eventually implement heuristics
  - For example if practical instances/problems become too large
  - Or if you want to compete in a challenge
  - Or if your thesis needs some (extra) material