Evolutionary Algorithms: Final report

Fabian Denoodt (r0698535)

December 31, 2023

1 Metadata

- **Group members during group phase:** Sisheng Liu and Melvin Schurmans
- **Time spent on group phase:** 12 hours
- **Time spent on final code:** 168 hours or more (3 * 7d * 8h).
- **Time spent on final report:** 14 hours

2 Changes since the group phase (target: 0.25 pages)

1. **Genetic algorithm** from base project.
   (a) **Variation**
   More mutation and crossover procedures: introduce **scramble & inversion mutation** (preserve swap mutation from group project), introduce **order crossover**, preserve (edge crossover from group project).
   (b) **Diversity**
   Through **fitness sharing** and **island model local search**.
   (c) **Local search**
   Two-opt, Insert node at random location.
   (d) **Phase-based convergence**
   Split optimization in **exploration phase (1)** for first half of the run (eg 2.5 minutes) and then go over to **exploitative convergence phase (2)** for the remaining duration.
   (e) No self-adaptation or multi-objective optimization.
2. **Plackett-Luce Gradient Search (PL-GS) model**: For this part, the base evolutionary algorithm is discarded and an algorithm based on gradient descent in the discrete domain (proposed by (CeberioJosu and SantucciValentino, 2023; Santucci et al., 2020)) is used instead. This part includes the following efforts:
   (a) Reproduce the results from their paper to their benchmark. This includes writing entire source code, which was not publicly available.
   (b) Apply the PL-GS model to the Traveling Salesman Problem (TSP).
   (c) Explore alternative representations to represent Probability Mass Function (PMF) in PL-GS (based on Probabilistic Graphical Models).

We have made a modification to the structure of the report. Section 3 remains the final design of the algorithm. Here we only discuss the genetic algorithm, which is based on the group project. A new section (4) is added where we discuss the PL-GS algorithm. The performance of the algorithm is underwhelming and, therefore, not part of the final algorithm used for the performance evaluation of this course.

3 Final design of the evolutionary algorithm (target: 3.5 pages)

3.1 The three main features

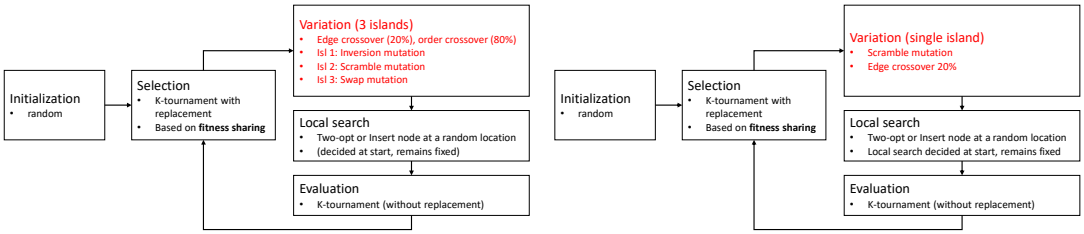
1. Order-crossover (only edge crossover is too exploitative, having a single child based on mutual edges from two parents resulted in too quick convergence to sub-optimal. Instead, order-

crossover, obtains 2 children per two parents, which ensures that population doesn't convergence to same path.)

2. Significant performance improvement through **local search** and **fitness sharing**.
3. Phase-based convergence
   - Encouraging diversity (due to multiple islands running in tandem and order-crossover creating larger populations) results in large computational costs, possibly hurting performance. Instead, we allow for an explorative scheme through multiple islands in the first phase (with migration and so on). In the second phase, islands merge to a single island, preserving only the candidate solutions that survive the elimination procedure. Additionally, we only run edge-crossover during this phase as it is a more greedy procedure, converging faster.

3.2 The main loop

We solely provide a figure for the genetic algorithm. Since the PL-GS algorithm does not fit in the "selection, variation, evaluation"-scheme, we believe a high-level view of the pseudo-code is more insightful.



(a) Phase 1: Explorative with multiple islands and mostly order crossover.

(b) Phase 2: Exploitative with single island and only edge crossover and scramble mutation.

Figure 1: Overview of the Genetic algorithm.

3.3 Representation

In both algorithms (Genetic algorithm and PL-GS), a route is indirectly represented using the adjacency representation $\sigma = (\sigma(1), \sigma(2) \ldots, \sigma(n)) \in \mathbb{S}^n$, where $\mathbb{S}^n$ represents the set of all permutations of length n.
Note: Implementation-wise, the permutations will be represented as numpy arrays with length $n-1$. The length can be set to $n-1$ by implicitly inferring that all representations start from city 1, hence this number does not need to be stored in the numpy array. This decision was made to enable a more computationally efficient implementation of constructing the "edge-table" (used in edge crossover and when computing the distance of a node to another in fitness sharing).

As discussed in the group report, we considered the cycle notation as representation. Although this is the exact representation required for the edge list in the crossover procedure, it posed challenges in ensuring that the path is a single cycle. We thus chose the first representation because we presumed that the performance cost of computing an edge table would be more manageable than having to deal with the overhead of invalid individuals.

3.4 Initialization

Population initialization: In both algorithms, population initialization is carried out randomly. Notes depending on the algorithm:

1. **Genetic algorithm**:
   Individuals are defined as random permutations, that do not pass through the same city twice. **No local search is applied during initialization**. We assume local search would be helpful at the start to remove infinity edges and give a small push towards the right solution. However, for the larger routes (e.g. 750 or 1000) our local search operators are quite computationally expensive. Instead, by a trick to the weights of infinite edges (see the paragraph below), enabling us to compare distances between the number of infinity edges, we can get away with not applying local search

in the first round. Being able to avoid the computational cost from local search at the start is nice for computational reasons, but also ensures that we do not need to think about whether local search would push all populations from each island into too narrow regions, resulting in faster convergence in the beginning but a sub-optimal solution in the end. Local search will be applied at every iteration afterwards.

2. **PL-GS**:
   Initialization of the population $\{\sigma^{(1)} \dots \sigma^{(\lambda)}\}$ is dependent on the definition of $p_{\mathbf{w}}(\cdot)$, which by default is defined as a uniform categorical distribution. Implementation-wise this is done by defining $\mathbf{w}$ as a numpy array containing n zeros.
   Alternative representations: Although not explored in this project, an island model could be considered in combination with the PL-GS algorithm, each with a different initialization for $p_{\mathbf{w}}$ than the uniform distribution. This allows for easily searching different regions of the solution space. The island model in combination with PL-GS was not explored because a migration procedure is not trivial to apply to probability density functions.

Distance matrix initialization:

1. An infinite edge D_{ij} is replaced by a real value according to the following equation:

$$D_{ij} \leftarrow \text{largest non-inf edge} \times n$$

   As such, we can compare paths containing infinite edges based on how many infinite edges they have. By multiplying by n in the equation, we ensure that choosing a path without infinite edges is always better than a path with infinite edges, regardless of how long the non-infinite path may be. This enables faster convergence in the early stages of the algorithm.
2. Secondly, since in the PL-GS algorithm parameter updates are based on how large f outcomes are, we normalize by dividing each edge distance by the largest non-infinite value. This also helps to prevent overflow errors caused by exploding gradients. Normalization is only applied in PL-GS.

3.5 Selection Operators

The selection process, k-tournament with replacement, remains unchanged from the group project. The value of the parameter k is predetermined and remains constant for a specific TSP, such as 750 or 1000 tours. While it is possible to extend the selection operators by adjusting k dynamically, for instance, starting with a low k for more randomness and gradually increasing it towards the end for a higher probability that the good candidates are always selected, we opted not to pursue this approach. This decision is based on the fact that we did not assign the responsibility of managing population diversity or promoting exploitation to the selection process. These aspects are already taken care off by other components in our algorithm, such as fitness sharing for diversity, local search for exploitation, and the phase-based convergence scheme (that works with varying numbers of islands and crossover functions depending on the need for diversity or exploitation). Moreover, maintaining a constant selection scheme adds predictability to the algorithm, in contrast to introducing a parameter that changes smoothly over time.

3.6 Mutation operators

Evolutionary Algorithm: We considered the following three mutation operators: **Inverse mutation, swap mutation and scramble mutation**. We observed that scramble mutation resulted in slightly better results compared to the other two algorithms. We therefore choose the following scheme:

1. Phase 1) Three islands run in tandem, each with its own mutation operator. The mutation rate remains fixed and is identical for each island.
2. Phase 2) At the beginning of phase 2, the islands merge and the best mutation operator is preserved (scramble mutation in this case).

Although we observed better results using scramble mutation, we preserved the other mutation techniques in phase one, encouraging islands to explore different regions. The mutation rate percentage is a parameter that is decided upfront, dependent on the number of tours.

3.7 Recombination operators

In the group project, we used **edge crossover** because the technique chooses the mutual edges between two parents, such that the "good properties" from a candidate would be preserved into the offspring. While being a good exploitative technique, from preliminary testing we observed that the entire population quickly converged to a single candidate solution, even after introducing other diversity mechanisms such as islands and fitness sharing. To combat this premature convergence, we introduced **order crossover** as well. Order crossover can create two offspring from two parents, while aiming to preserve the relative order from the parents. In our experiments, we observed that this less exploitative technique delays the entire population from converging to a single individual as quickly.

Based on these findings, we apply the following scheme:

1. Phase 1) All three islands have 80% of applying order crossover (for diversity) and 20% of edge crossover (for exploitation). Order crossover creates twice the number of offspring than edge crossover, but the elimination procedure then reduces the offspring population to the required population size.
2. Phase 2) The objective is now to obtain a single and optimal candidate. Hence in phase 2) edge crossover is constantly applied.

3.8 Elimination operators

The elimination operator remains k-tournament selection (without replacement), as in the group project. The parameter k is set to the same value as in selection. Elimination also considers the parents.

Although we did not explore it in this project, another interesting technique would be age-based elimination. This would be less computationally demanding, and the possible dangers of bad offspring are already reduced by local search. Using age-based elimination would allow us to compute local search only for the candidate solutions that are guaranteed to go through. In our current elimination operator (k-tournament), we risk creating offspring and applying expensive local search procedures to offspring that do not survive the following rounds.

3.9 Local search operators

We consider three operators:

1. **Two-opt:** our implementation is based on the pseudo-code found online[1]. There are two implementations to consider, two-opt for symmetric matrices or for asymmetric matrices. Although in theory, the symmetric variant is not applicable to our TSP, we observed it still significantly improved performance, while being much easier to compute than in the asymmetric case. Hence, we use the efficient two-opt for symmetric distance matrices, choosing two nodes, swapping them, and reversing the path between the two nodes.
2. **Insert node at random location:**
   Motivation: since our two-opt implementation is in theory not guaranteed to actually improve candidate solutions (due to symmetric distance matrix assumption), we also experiment with other local search procedures, that ensure a correct transformation, while remaining relatively easy to compute (in contrast to two-opt in the asymmetric case).
   Explanation: Given a permutation σ, this procedure considers a subset of random node pairs $(a = \sigma(i), b = \sigma(j))$ with $i \neq j$. For each pair, the procedure verifies if inserting b just in front a results in a better fitness (this can be calculated in constant time, not linear). If performance is improved, the node is inserted, resulting in the original nodes $\sigma(i+1) \dots \sigma(j-1)$ being moved one step to the right. The subset size is a parameter determined upfront.
3. **PL-GS:**
   Motivation: the algorithm (discussed in section 4), although initially introduced as a standalone evolutionary algorithm by Santucci et al., could in theory also be used a as local search operator for a single σ. By default PL-GS results in finding a PMF $p_{\mathbf{w}}(\cdot)$ where sampling from it should give us a σ for which $f(\sigma)$ is good. If we start the algorithm, not from the uniform categorical distribution $p_{\mathbf{w}}(\cdot)$, but instead, define $\mathbf{w}$ s.t. $p_{\mathbf{w}}(\cdot)$ is high for σ and low for other routes, applying the PL-GS algorithm for a few steps, would result in similar samples to σ which are hopefully

[1] https://en.wikipedia.org/wiki/2-opt

better.
Implementation-wise, given $\sigma = (\sigma(1), \sigma(2), \sigma(3), \ldots, \sigma(n))$, then $p_{\mathbf{w}}(\cdot)$ can be initialized by defining $\mathbf{w}_{\sigma(1)} = 1,\ \mathbf{w}_{\sigma(2)} = \frac{1}{2},\ \ldots,\ \mathbf{w}_{\sigma(n)} = \frac{1}{n}$, where $\mathbf{w}_i$ refers to the i'th component in the vector $\mathbf{w}$. Although the idea seems appealing in theory, our preliminary tests have shown that the PL-GS converges slowly, and a simple two-opt procedure obtains better results in a shorter duration. We therefore do not consider it as part of the final algorithm.

A single local search operator and its respective tuning parameters are predetermined and dependent on the TSP.

3.10 Diversity promotion mechanisms

1. **Fitness sharing**
   The distance metric d is defined as follows:

   $$d(\sigma_1, \sigma_2) = n - \#\text{mutual edges}(\sigma_1, \sigma_2)$$

   Here, n refers to the number of nodes. Hence, by constructing an edge table for σ_1, $d(\sigma_1, \sigma_2)$ can be computed in linear time complexity. We fix $\sigma_{\text{share}} = n$, s.t. given a candidate solution σ, its shared fitness score is dependent on the distance to every other node in the population. Setting $\sigma_{\text{share}} < n$ resulted in unpredictable results, as sometimes σ would be punished, and sometimes not. The parameter α is determined dependent on the number of tours (e.g.: 50 or 750).

   Time complexity and an efficient approximation: computing the average distance of a single permutation σ to all other nodes would require rougly $\Theta(n^2)$ computations. Hence, if this metric must be computed for every individual in the population, the algorithm quickly becomes slow. Instead, we do two things:

   (a) The fitness scores are computed for only a sub-population (eg 10%). The remaining 90% then receive a "fake" fitness score based on the average penalty of the sub-population.
   (b) For each individual part of the 10% sub-population, the distance metric is computed with a subset of nodes (for instance, 10%). We ensure that each node is guaranteed to be compared against itself as well.

   Both parameters are tuned based on the TSP problem.
2. **Multiple phases & islands in tandem** The initial algorithm starts from **three islands**, each island with its own mutation operator to encourage. Migration between islands occurs every 25 epochs. During migration, for each island i a predetermined percentage of its population (e.g. 10%) is moved to the next island ($(i+1)\%3$).

   Running multiple islands in tandem is significantly slower than running one island. It may also not be beneficial to maintain all islands during the entire execution of the algorithm. When diversity is required, then the computational cost of multiple islands may be acceptable. However, after a certain duration the goal is no longer diversity, but instead, to obtain a good solution. Hence, one island may be better. We therefore introduce two phases:

   (a) Phase 1) 3 islands in tandem, with migration after 25 epochs.
   (b) Phase 2) After a certain duration (e.g. 2.5 minutes), merge all islands into a single island (through the elimination procedure discussed above).

   The parameter when to go over to phase 2 is also determined in advance, dependent on the number of tours in the TSP. We explored multi-threading, but this did not seem to improve performance.

3.11 Stopping criterion

For the five-minute benchmarks, no stopping criterion is applied. Even if an algorithm's population may have converged to a single individual, there may still be a slim chance of obtaining a better result due to mutation.

For the histogram experiment with 500 runs, running the algorithm for 5 minutes each time, was not an option due to time constraints. Here we set a smaller time window that is roughly based on our

approximate "dissimilarity" metric. The metric is defined as follows:

$$\frac{1}{P} \sum_{i \in \{1 \ldots P\}} d(\sigma_i, \sigma_{(i+1)\%P})$$

Here, $P \in \mathbb{N}$ is the population size and $d()$ is the same distance function discussed in section 3.10. The metric can be computed with a time complexity of roughly $\Theta(P \times n)$. Intuitively, the metric compares each node to one other node and computes the average distance. Based on this metric we have an idea of how similar the population is, e.g.: 747.5 in a 750-tours TSP implies a lot of dissimilarity, while 0 implies that each node is the same.

Based on this population-similarity metric, we set the stopping time for the 50-tour TSP to 20 seconds per run. This results in approximately the final 50 iterations to be stuck at the identical population (50 iterations roughly corresponds to 5 seconds, with a single island, or 20% of the running duration).

3.12 Parameter selection

A grid search, trying every combination is not possible due to the large number of parameters running each for 5 minutes. Instead, we start from a base set of parameters and optimize one parameter at a time. The order in which we considered the parameters and the respective values we experimented with are shown in table 1. This procedure is applied to all TSPs. Afterwards, we tweak the parameters by hand based on intuition. The final parameters per number of tours is shown in table 2.

Parameter	**Values**
Population size	10, 50, 100, 200, 500, 1000
Offspring size multiplier	1, 2, 3
k (k-tournament in selection & elimination)	3, 5, 10, 25
Mutation rate (%)	0.05, 0.2, 0.4
Migrate every nb epochs	25, 50
Migration percentage	0.05, 0.1
Merge after percent time left	0.5, 0.75, 0.9
Fitness sharing subset percentage	0.05, 0.2, 0.5
Alpha (in fitness sharing)	1, 2, 0.5
Local search (either 2-opt or insert random node)	None, (2-opt, 1), (2-opt, 5), (insert, 0.1), (insert, 0.5), (insert, 1)

Table 1: Parameters for TSP in order of which one was optimized first.

4 Plackett-Luce Gradient Search - An advanced EDA scheme

Estimation of Distribution (EDA) schemes replace standard variation operators, and instead, consider a model (e.g.: a graphical model) of which samples are taken, evaluated, and based on these scores the model is adjusted to hopefully create better samples in the future Eiben and Smith (2015). In this section, we explore such a scheme based on Santucci et al.'s gradient search algorithm, "Plackett-Luce Gradient Search (PL-GS)".

4.1 Concise explanation of the algorithm

The paper introduces a gradient-based optimization scheme, applied to a discrete domain. In particular, to the space of permutations $\sigma \in \mathbb{S}^n$. Since $f : \mathbb{S}^n \to \mathbb{R}$ has a discrete domain, one cannot directly apply a gradient descent scheme to optimize f as follows:

$$\sigma \leftarrow \sigma + \eta \nabla f(\sigma)$$

Indeed, f's domain is discrete, and therefore, $\nabla f(\sigma)$ is not defined. Instead, the authors aim to optimize $f(\cdot)$ indirectly, based on the following objective function:

$$F(\mathbf{w}) = E_{\mathbf{w}}[f(\sigma)] = \sum_{\sigma \in \mathbb{S}^n} f(\sigma) p_{\mathbf{w}}(\sigma)$$

Here, $p_{\mathbf{w}}(\sigma)$ is a differentiable probability mass function with parameters $\mathbf{w} \in \mathbb{R}^n$. We discuss two possible representations in the following two sections. Note that finding the parameters $\mathbf{w}$ for which $F(\mathbf{w})$ is optimal, will also result in samples $\sigma \sim p_{\mathbf{w}}(\sigma)$ for which $f(\sigma)$ is good.

One can prove that the following statements hold true (Santucci et al., 2020):

$$\nabla F_{\mathbf{w}}(\mathbf{w}) = \sum_{\sigma \in \mathbb{S}^n} f(\sigma) \left[\nabla_{\mathbf{w}} \log p_{\mathbf{w}}\right] p_{\mathbf{w}}(\sigma) = E_{\mathbf{w}} \left[f(\sigma) \nabla_{\mathbf{w}} \log p_{\mathbf{w}}(\sigma)\right] \tag{1}$$

$$\approx \frac{1}{\lambda} \sum_{i=1}^{\lambda} f(\sigma^{(i)}) \nabla \log p_{\mathbf{w}}(\sigma^{(i)}) \tag{2}$$

Equation 1) tells us that the gradient of F can be computed, regardless of whether f is differentiable or not. We merely require that that $p_{\mathbf{w}}$ is differentiable with respect to $\mathbf{w}$. Equation 2) shows how to approximate $\nabla F_{\mathbf{w}}(\mathbf{w})$ based on λ samples.

Based on the above details, we obtain the following algorithm to find σ for which $f(\sigma)$ is good:
PL-GS: *high-level view of the pseudo-code:*
Objective: Optimize $F(\mathbf{w}) = E_{\mathbf{w}}[f(\sigma)] = \sum_{\sigma \in \mathbb{S}^n} f(\sigma) p_{\mathbf{w}}(\sigma)$
Initialization: Define the uniform categorical distribution $p_{\mathbf{w}}(\sigma)$ s.t. $\forall\, \sigma : p_{\mathbf{w}}(\sigma) = \frac{1}{|\mathbb{S}^n|}$
Repeat until convergence:

1. Compute λ samples: $\sigma^{(i)} \sim p_{\mathbf{w}}(\sigma)$.
2. Based on samples $\sigma^{(1)}, \ldots, \sigma^{(\lambda)}$, calculate $\nabla_{\mathbf{w}} F(\mathbf{w})$.
3. Update $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla F(\mathbf{w})$.

Return $\sigma \sim p_{\mathbf{w}}(\sigma)$.

4.1.1 Plackket-Luce ranking model as PMF

As a representation for $p_{\mathbf{w}}$, the authors consider an extension on the Plackett-Luce ranking model. Let $p_{\mathbf{w}}(\sigma(i) \mid D)$ denote the probability of traversing the i'th node $\sigma(i)$, given a set of nodes D which we have not yet traversed, then $p_{\mathbf{w}}(\sigma(i) \mid D)$ is defined as follows:

$$p_{\mathbf{w}}(i = \sigma(i) \mid D) = \frac{\mathbf{w}_i}{\sum_{\mathbf{w}_j \in D} \mathbf{w}_j}$$

Here, $\mathbf{w} \in \mathbb{R}^n$ is a vector where the components $\mathbf{w}_i$ contain the non-normalized probabilities of sampling node i. Hence, using this model, one can model a PMF over the permutation space $\mathbb{S}^n$ as follows:

$$p_{\mathbf{w}}(\sigma) = p_{\mathbf{w}}(\sigma(1) \mid \{\sigma(1) \ldots \sigma(n)\}) \cdot p_{\mathbf{w}}(\sigma(2) \mid \{\sigma(2) \ldots \sigma(n)\}) \cdot \ldots \cdot p_{\mathbf{w}}(\sigma(n) \mid \{\sigma(n)\}) \tag{3}$$

$$= \prod_{i=1}^{n-1} \frac{\mathbf{w}_{\sigma(i)}}{\sum_{j=i}^{n} \mathbf{w}_{\sigma(j)}} \tag{4}$$

Intuitively, a sampling procedure for this PMF is obtained by sampling n nodes from a categorical distribution with parameters $\mathbf{p} = (\mathbf{w}_1 \ldots \mathbf{w}_n)$ without replacement. Such sampling procedure can be efficiently implemented using the "Gumbel trick" Kool et al. (2019). Our Python implementation includes this trick.
Note: the definition of $p_{\mathbf{w}}(\sigma)$ in the paper is more mathematically involved, including multiple $\exp()$ functions to ensure that $\mathbf{w}$ consists of positive values. We omit this notation in the equation here for simplicity.

As shown in equation (1), the gradient of $\nabla F_{\mathbf{w}} \in \mathbb{R}^n$, requires the computation of $\nabla \log p_{\mathbf{w}}(\sigma)$. By applying Calculus rules, one can derive the following partial derivative (Santucci et al., 2020):

$$\frac{\partial \log p_{\mathbf{w}}(\sigma)}{\partial \mathbf{w}_{\sigma(i)}} = 1 - \exp \mathbf{w}_{\sigma(i)} \sum_{k=1}^{i} \frac{1}{\sum_{j=k}^{n} \exp \mathbf{w}_{\sigma(j)}}$$

Note that the paper contains an error. The equation shown here is correct.

4.1.2 An extension on the PMF

Here, we explore another representation for $p_{\mathbf{w}}$ which is not part of Santucci et al. (2020); CeberioJosu and SantucciValentino (2023).

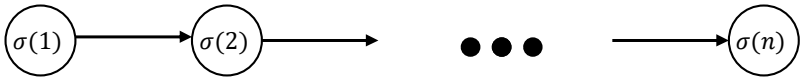


Figure 2: Bayesian network: our representation for $p_{\mathbf{w}}$

Motivation: A noticeable limitation of the $p_{\mathbf{w}}$ representation in 4.1.1 is the independence assumption between nodes. E.g.: traversing (= sampling) node a will not influence whether the next node to be traversed is node b or c. However, this is an incorrect assumption in the TSP which could drastically hurt the performance. For instance, when the algorithm has learned the optimal path to be $a, b, c, \ldots$. Now consider an unlucky node being sampled at the start, e.g. b is sampled first. Using the representation from the $p_{\mathbf{w}}$, the next node is most likely to be a, however, is no longer the best next node the traverse, resulting in reduced scores. Instead, c would have been the correct node.

Another note is related to the number of learned paths. A single $p_{\mathbf{w}}$ learns a single optimal path, but perhaps it could be interesting to also consider other paths. We therefore represent $p_{\mathbf{w}}$ as a first order Markov chain network, where sampling a node a at time step t is dependent on the previously sampled node at time step $t-1$. Its Bayesian network is depicted in figure 2. Formally, $p_{\mathbf{w}}$ is defined as follows:

$$\begin{aligned} p_{\mathbf{W}}(\sigma) &= p\left(\sigma(1)\right) \cdot \prod_{t=2}^{n-1} p\left(\sigma(t) \mid \sigma(t-1)\right) \\ &= \frac{1}{n} \prod_{t=2}^{n-1} \frac{\mathbf{W}_{\sigma(t)|\sigma(t-1)}}{\sum_{j=t}^{n} \mathbf{W}_{\sigma(j)|\sigma(t-1)}} \end{aligned}$$

Here, $\mathbf{W}$ is an $n \times n$ matrix where $\mathbf{W}_{ij} = \mathbf{W}_{i|j}$ denotes the probability of sampling node i given that j has just been sampled. We assume the probability of traversing the first-node as uniform, hence the $\frac{1}{n}$ in the second equation.

The gradient $\nabla F_{\mathbf{W}} \in \mathbb{R}^{n \times n}$ is now a square matrix, based on $\nabla \log p_{\mathbf{W}}(\sigma)$. The partial derivative is defined similar as before for $t \in \{2 \ldots n\}$:

$$\frac{\partial \log p_{\mathbf{w}}(\sigma)}{\partial \mathbf{w}_{\sigma(t)|\sigma(t-1)}} = 1 - \exp \mathbf{w}_{\sigma(t)|\sigma(t-1)} \sum_{k=2}^{t} \frac{1}{\sum_{j=k}^{n} \exp \mathbf{w}_{\sigma(j)|\sigma(t-1)}}$$

In addition,

$$\frac{\partial \log p_{\mathbf{w}}(\sigma)}{\partial \mathbf{w}_{i|j}} = 0$$

when i does not succeed j.

4.2 Considerations for future work

1. The algorithm as introduced by Santucci et al. does not contain any **diversity population** mechanisms. However, approaches similar to fitness sharing could be adapted to this algorithm as well. A penalty could be added to the fitness $f(\cdot)$ in the form of a regularization term, based on the average distance of λ samples. We could express the resulting function $g(\cdot)$ as follows:

$$g(\sigma^{(i)}) = f(\sigma^{(i)}) + \frac{1}{\lambda} \sum_{\substack{\sigma^{(j)}, \sigma^{(k)} \\ \in \{\sigma^{(1)} \ldots \sigma^{(\lambda)}\}}} d(\sigma^{(j)}, \sigma^{(k)})$$

2. Although selection is not directly applicable to PL-GS since there is no population to apply variation operators on, one could pose the question which samples are allowed to influence the gradient $\nabla F\left(\mathbf{w}\right)$, used to update $\mathbf{w}$:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla F\left(\mathbf{w}\right)$$

By default all $\{\sigma^{(1)} \ldots \sigma^{(\lambda)}\}$ are used with equal weighting. However, also here, **selection** procedures such as k-tournament could be applied to, for instance, only select the good samples to update $\mathbf{w}$, while eliminating the bad ones and thus preventing them from updating $\mathbf{w}$ in a bad direction.

5 Numerical experiments (target: 1.5 pages)

5.1 Metadata

All parameters are presented in table 2. Overall, the parameters remain relatively constant depending on the number of tours. There are two exceptions; local search and k in k-tournament. Although two-opt obtained good results for smaller tours, it does not scale to larger tours. We did experiment with applying two-opt to only a subset of nodes in a path, but although less computationally demanding, it did not always result in good candidate solutions. This may be a side effect of the symmetric matrix assumption we've made to improve time complexity. Two-opt reverses the path between two nodes, and as the number of tours increases, the length of these reversed lists also increases. This may result in the error due to the symmetric distance matrix assumption to also increase, resulting in a higher probability of faulty solutions. As a compensation for the lack of a good local search procedure in the larger tours, we increase k to increase convergence speed, as the good individuals in the populations have a larger chance of surviving.

PC characteristics:

1. Processor Intel i7-4790 CPU 3.60GHz (Cores: 4)
2. Memory 16GB
3. Python version 3.9

5.2 tour50.csv

Figures 3 and 4 display the performance graphs. The algorithm is run for exactly 20 seconds. The first 10% in the convergence graph (fig 4) have been removed. The graph depicts the results from a single island. Note that the duration in the graph is not entirely linear. The first 25 iterations take 10 seconds, while the remaining 100 iterations also take 10 seconds. This is due to the first iterations being more computationally demanding due to having three islands in tandem.

To gain some insight in a single run, we refer to table 3. Due to the two-opt local search, the algorithm finds a better solution than the greedy heuristic provided on Toledo within 25 iterations. Meanwhile, due to the order crossover and other diversity mechanisms, the population can remain diverse in phase one (the maximum distance is 50 for 50-tours). Phase two behaves similarly to the egg-holder algorithm where the entire population quickly converges to a single solution. This is due to the edge crossover operator used in phase 2. We observe that the average distance between individuals quickly converges to zero, indicating that the entire population has become the same individual.

The histogram (figure 3) indicates quite consistent results. The algorithm consistently finds candidate solutions which are better than the greedy heuristic from Toledo. An example candidate solution with score of 25975.70 is the following: (0 26 21 32 17 40 34 33 44 36 23 28 27 12 25 39 37 20 49 22 6 9 43 30 48 5 8 46 35 16 4 15 47 42 11 3 19 38 10 13 7 41 14 1 18 24 45 2 31 29).

Parameter	**50 Tours**	**100 Tours**	**500 Tours**	**1000 Tours**
Popul Size	50	50	50	50
Offspring size mult.	1	1	1	1
K (k-tournament)	3	3	10	25
Mutation rate	0.1	0.2	0.2	0.2
Migrate after epochs	25	25	25	5
Migration %	0.05	0.05	0.05	0.05
Merge islands after % time left	0.5	0.5	0.5	0.5
Fitness sharing sub-population %	0.05	0.05	0.05	0.05
Local search	2-opt	2-opt	IRN, 0.5	IRN, 0.5

Table 2: Parameter settings for different tours. IRN refers to the Insert Random Node operator and 0.5 is the number of nodes to consider, as discussed in section 3.9

5.3 tour100.csv

Figure 5 shows a typical convergence graph of the algorithm running for five minutes. Similar as discussed before, due to islands running in tandem in phase 1), the first 125 roughly take 2.5 minutes and the remaining 375 iterations, the other 2.5 minutes. Here, the difference between the two phases

Phase	Island	Iteration	Best Fitness	Avg Fitness	Fit Shared	Avg Dist	Mutation
1	0	24	27025.15	146528.29	59479.35	46.86	Inversion
	1	24	26792.80	126835.99	66089.59	46.57	Swap
	2	24	26736.30	124162.91	122564.17	46.47	Scramble
2	NA	49	25957.36	26759.92	1066930.26	0.69	Scramble
	NA	74	25957.36	26020.75	981034.99	1.02	Scramble
	NA	99	25981.84	25981.82	1065255.44	0.00	Scramble
	NA	124	25981.84	25981.84	1065255.44	0.00	Scramble

Table 3: Example run on the 50 tours TSP, 20 seconds. Both phases run for 10 seconds each.

is clearly noticeable, having phase one with a more exploration objective, while phase aims to exploit, resulting in a behavior similar to the egg-holder algorithm. We observe that the algorithm quite easily outperforms the simple greedy heuristic from Toledo. We observe that similar to the 50-tours problem, the algorithm does not require the entire 5 minutes to find a good solution.

5.4 tour500.csv and tour1000.csv

Figures 6 and 7 depict the convergence graphs over 5 minutes. With the larger maps, we observe that problems start to arise, as we can no longer beat the greedy heuristic from Toledo within the 5 minute interval. This behavior can be partly explained due to the local search operator. As discussed in section 5.1, our two-opt local search operator is computationally demanding, and does not scale well to the larger tours. We therefore, need to use a less demanding local search operator (insert random node), which clearly has impact. As a consequence of the slow convergence, we end up having to increase the selection pressure to $k = 10$ for 500 tours and $k = 25$ for 1000 tours. This results in slightly faster due to less randomness, but such large k values are unusual as the population size is only 50. Most likely, in the long run, this procedure, in combination with the egg-holder effect in phase 2 will almost guarantee we will eventually get stuck in local optimum. However, we also experimented with less selection pressure, but this would converge slower, resulting in a worse overall sore at the end of the 5 minute mark.

5.5 PL-GS

Finally, we discuss the two PL-GS algorithms for the 50 tours TSP. The results are shown in figures 8a (PMF with independence assumption) and 8b (PMF with conditional assumption (ours)). Both algorithms ran for one minute. While both algorithms seem nice in practice, the graphs demonstrate that both algorithms are immediately outperformed by even a simple evolutionary algorithm (eg without diversity promotion or local search). The independent PMF obtained a final best fitness of $76,999.85$ and population average fitness of $158,960,90$, the conditional PMF a final best fitness of $83,706.50$ and population average fitness of $1,316,909.39$ indicating that both algorithms do converge but slowly.

While techniques exist to speed up convergence, e.g.: using a larger learning rate, the PL-GS algorithm is inherently unstable due to the representation of both PMFs. Hence, choosing to increasing the learning may increase the chances of overflow errors. This is due to the design of the Plackett-Luce model, assigning a probability score of sampling a node at time step t. A converged model, which consistently results in similar samples thus requires the probability weight of sampling the first node to be large (towards infinity), while later nodes should have a lower probability, with the final approaching 0. As the length of the permutation becomes longer, this problem becomes even more prominent. Hence, scalability is a major issue for this model.

We also observe that our proposed PMF representation, converges slower than the independent PMF. This can be explained due to the number of parameters which must be optimized, which is now n^2, instead of n. A second reason, is that a sample $\sigma \sim p_{\mathbf{w}}(\sigma)$ can only influence the parameters $\mathbf{W}_{t|t-1}$ for $t \in \{2 \dots n\}$. The remaining weights can only be updated if they are part of another permutation sample.

6 Critical reflection (target: 0.75 pages)

Three positive points:

1. Having a phase-based scheme with a tweakable a parameter on when to go to phase two allows

for an easily adaptive algorithm depending on the problem. For easy problems, with few local minima, phase 2 can start early allowing to find optimum quickly. For complicated problems, the first phase can go on for a long time without unwillingly converging to a population of identical individuals. Secondly, since only phase 1) contains multiple islands, we gain the population diversity benefits, while the computational drawbacks can be tempered as phase 2) merges all islands and preserves the best ones.

2. Our two-opt local search operator which assumes a symmetric matrix can still perform well for smaller TSP problems, resulting in the algorithm quickly converging to a good solution. The symmetric matrix assumption allows for a much faster implementation than if the asymmetry must be considered.
3. Our fitness sharing implementation encourages diversity, while not being a big burden on speed due to a fast approximation of the shared fitness values of the population.

Three weak points:

1. Since our best local search operator (two-opt) did not scale to larger problems we have a less powerful operator for the larger tours. This can be observed in the performance graphs of the larger tours, as the algorithm converges much slower. As a consequence, we aim to increase convergence speed in other ways due to an unusually large k value, which may increase the chances of getting stuck in a local minimum.
2. While the objective of the second phase is to quickly converge to the optimal solution, given the series of good candidates found in phase 1), we found that phase 2) may behave too much like the egg-holder algorithm, quickly eliminating any diversity. This is largely due to the edge crossover operator, only preserving mutual edges between individuals.
3. The island model, although resulting in improved performance, remains computationally demanding. Multi-threading could help preserve the benefits while omitting the computational costs.
4. While we do have parameters depending on the number of tours, the parameters remain relatively static during the entire run (apart from the two phases). It would be interesting to investigate more dynamic parameters that change throughout the run. For instance, a dynamic k value in k-tournament, requesting more diversity (low k) at the start and more selection pressure towards the end. Alternatively, a fitness-sharing scheme that only "kicks in" when the population diversity becomes too low, could also be interesting to explore. These dynamic models come at the cost of less predictability and this is the main reason that we did not explore them in this project.

I initially started this assignment with the mindset of avoiding evolutionary algorithms at all costs, as they are known to be slow. Therefore, I implemented the gradient descent scheme, PL-GS. However, it quickly became apparent that such a scheme on its own would never be able to find an absolute optimum, as it kept getting stuck in local optima. In contrast, even the simplest evolutionary algorithm could outperform a greedy metric, although, with perhaps a slightly longer running time.

This experience has truly shown how evolutionary algorithms benefit from utilizing diversity schemes to avoid getting stuck in local optima. Additionally, they can employ local search schemes to find solutions relatively quickly. We can easily choose a balance in the trade-off between speed (exploitation) and finding the optimal answer (exploration). This is a benefit not immediately applicable to solely gradient descent schemes, for instance.

References

CeberioJosu and SantucciValentino (2023). Model-based Gradient Search for Permutation Problems. *ACM Transactions on Evolutionary Learning and Optimization*. Publisher: ACM New York, NY.

Eiben, A. and Smith, J. (2015). *Introduction to Evolutionary Computing*. Natural Computing Series. Springer, Berlin, Heidelberg.

Kool, W., van Hoof, H., and Welling, M. (2019). Stochastic Beams and Where to Find Them: The Gumbel-Top-k Trick for Sampling Sequences Without Replacement. arXiv:1903.06059 [cs, stat].

Santucci, V., Ceberio, J., and Baioletti, M. (2020). Gradient search in the space of permutations: an application for the linear ordering problem. In *Proceedings of the 2020 Genetic and Evolutionary Computation*

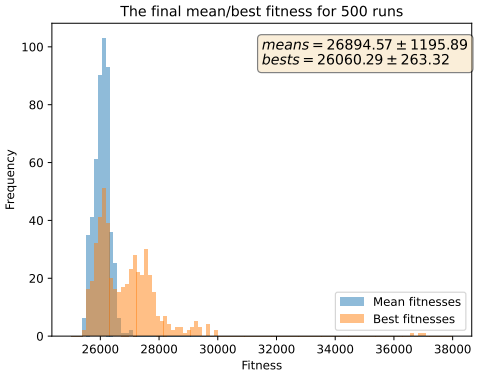


Figure 3: 50 tours: 500 runs, averages/standard deviations are displayed in the top right corner.

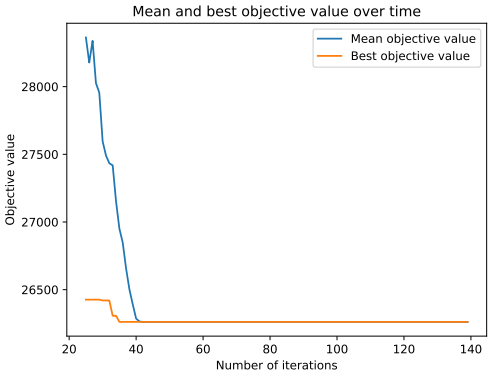


Figure 4: 50 tours: 20 seconds, first 10% have been skipped.

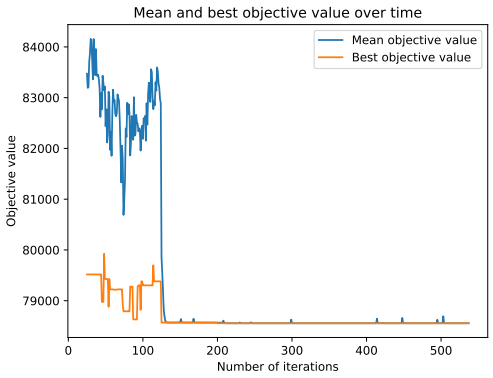


Figure 5: 100 tours

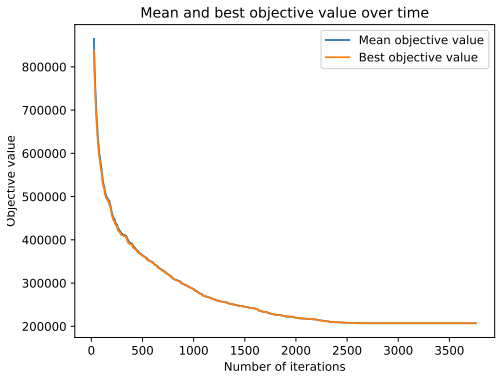


Figure 6: 500 tours

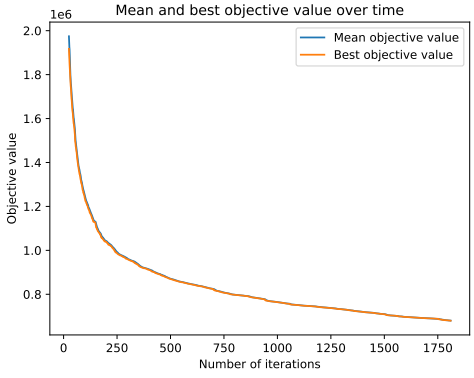
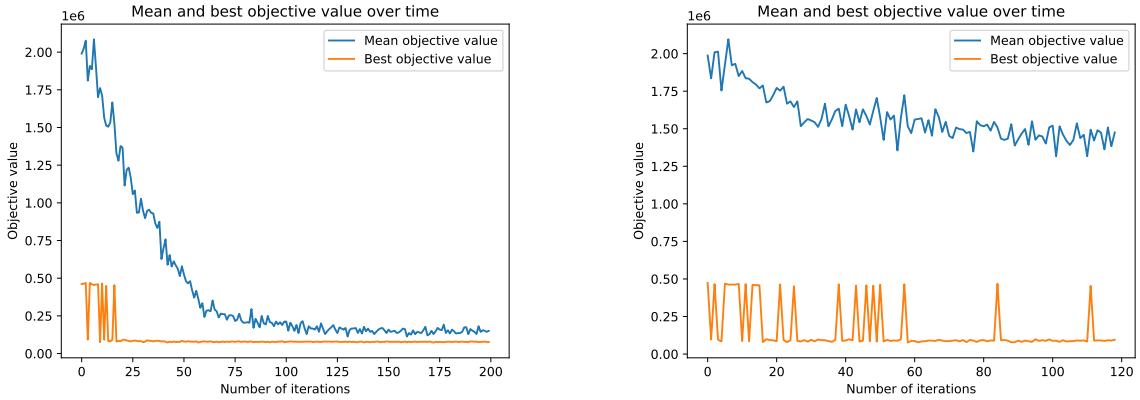


Figure 7: 1000 tours



(a) PL-GS (Independence assumption)

(b) PL-GS (Conditional assumption)

Figure 8: PL-GS Figures

Conference Companion, GECCO '20, pages 1704–1711, New York, NY, USA. Association for Computing Machinery.