A Computational Model of Heuristic Decision Making

Alexandra Kirsch

A Computational Model of Heuristic Decision Making

Alexandra Kirsch
Department of ComputerScience
Eberhard Karls Unversität Tübingen
Tübingen, Germany

Keywords: heuristic decision making, computational social choice

Abstract

Heuristics seem to lie at the core of people's ability to make timely and adequate decisions. The study of heuristic decision making is thus of interest both for psychology and artificial intelligence. Several authors have recognized the need to structure heuristics and to develop general models of them. This paper proposes a computational model that encompasses the existing descriptive models of heuristics. We set special focus on the instantiation of this model with respect to aggregating cue values by reviewing some methods from the field of computational social choice. To show its applicability in the context of artificial intelligence we present a case study of computational problem solving.

Introduction

The human capacity of making appropriate, timely decisions in changing, partially unknown environments has been fascinating for cognitive scientists and researchers in artificial intelligence alike. Heuristics seem to be the key to this unique capacity. Despite occasional flaws (Ariely, 2010), heuristics generally lead to ecologically valid decisions (Gigerenzer & Brighton, 2009).

Several heuristics have been identified in a variety of contexts. Systematic research of which heuristic is applied when, and which cognitive processes underlie heuristics, is, however, still in its infancy. Svenson (1979) classifies decision rules along several dimensions. Gigerenzer (2001) has proposed the "Adaptive Toolbox", characterizing heuristics along three modules: search rules, stopping rules and decision rules. Shah and Oppenheimer (2008) classify heuristics in an effort-reduction framework, where a basic approach to decision making is analyzed with respect to cognitive costs and how heuristics reduce such costs.

All of these attempts are descriptive in style. In this article, I incorporate these models into a modular computational model that can be instantiated to arbitrary heuristics from the literature. The purpose is on the one hand to offer a more precise description of the processes underlying heuristic decision making, and on the other hand to provide an implemented decision procedure to engineers that may help to make more human-like decisions and thus improve human-machine

With the support of the Bavarian Academy of Sciences and Humanities.

interaction. The engineering aspect is related to the tradition of cognitive architectures (Langley, 2017; Kotseruba & Tsotsos, 2016), which try to model human intelligence as well as use the models in artificial intelligence applications.

The proposed model opens a wide range of parameter choices. We will focus on one specific aspect: the aggregation of cues over several alternatives. We review voting methods from the field of computational social choice, which make it possible to aggregate ordinal-scale cues, in addition to the well-known numeric cues that are typically aggregated with a weighted or unweighted sum.

In an engineering context, the flexibility of the model requires the choice of parameters. We present a case study from the domain of problem solving, where each step can be modeled as a decision. The purpose of the case study is no attempt to provide any general insight into which parameters are best, but rather an example of how the model can be used for computational decision making and how the choice of parameters might be tackled.

The Computational Model

Heuristics are usually described as distinct strategies for decision making. Some frameworks, as mentioned in the introduction, have tried to classify heuristics and explain heuristic decision-making by basic principles that are adapted according to the task, the situation and the decision maker. We first transform two approaches from the literature into a more formalized form. Then we derive a general computational decision procedure that incorporates both approaches and discuss how its components form building blocks of different heuristics. We then discuss the model with respect to the adaptive toolbox (Gigerenzer, 2001) and Svenson's (1979) classification.

Formalization of Heuristics

A decision consists of choosing an *alternative* a^* from a set of alternatives $a_i \in A$ (which may be given or may be retrieved by the decision-maker). The quality of each alternative is assessed by a number of *cues* $c_i \in C$. We formalize cues as functions in two ways: 1) as a *cue value function* mapping alternatives to real values, $v : A \to [0.0, \dots 1.0]$. The restriction to the interval between 0 and 1 is just a convenience to make sure that cues map to a comparable range, otherwise the range of a cue could imply an implicit weight. 2) as a *cue ranking function* returning a preference ranking, which is a linear ordering $\succeq$ of A.

Shah and Oppenheimer (2008) build their effort-reduction framework on the weighted additive rule and divide the decision process into five tasks:

0. A vector of alternatives is given $\vec{a} = (a_1, a_2, \dots, a_n), a \in A$[1]
1. "Identifying all cues—all relevant pieces of information must be acknowledged." (Shah & Oppenheimer, 2008, p. 1, item 1). We formalize this with a function GET-CUES that returns a vector of cues (i.e. functions) $\vec{c} = (c_1, c_2, \dots, c_m)$
2. "Recalling and storing cue values—the values for the pieces of information must either be recalled from memory or processed from an external source." (Shah & Oppenheimer, 2008, p. 1, item 2). The cue values in this quote correspond to the result of a cue value function, thus, for the weighted additive rule we restrict cues to cue value functions. Ap-

[1](Shah & Oppenheimer, 2008) do not explicitly mention this step, we added it as step 0 for consistency with our generalized model.

plying the cues to all alternatives is equivalent to filling a decision matrix M:

$$\begin{array}{c|cccc|}
 & a_1 & a_2 & \dots & a_n \\
\hline
c_1 & c_1(a_1) & c_1(a_2) & \dots & c_1(a_n) \\
c_2 & c_2(a_1) & c_2(a_2) & \dots & c_2(a_n) \\
\dots & \dots & \dots & \dots & \dots \\
c_m & c_m(a_1) & c_m(a_2) & \dots & c_m(a_n) \\
\hline
\end{array}$$

3. "Assessing the weights of each cue—the importance of each piece of information must be determined." (Shah & Oppenheimer, 2008, p. 1, item 3). This is another retrieval process. We treat weights as a special case of parameters that may be necessary in the process (such as parameters of the cue functions). Therefore, we call the retrieval function GET-PARAMS, which returns (possibly among other things) a vector of weights $\vec{w}$, $|\vec{w}| = m$.
4. "Integrating information for all alternatives—the weighted cue values must be summed to yield an overall value or utility for the alternative." (Shah & Oppenheimer, 2008, p. 1, item 4). This corresponds to a multiplication of the filled decision matrix with the weight vector, resulting in a vector of utilities $\vec{u} = M^T \cdot \vec{w}$, $|\vec{u}| = n$.
5. "All alternatives must be compared, and then the alternative with the highest value should be selected." (Shah & Oppenheimer, 2008, p. 1, item 5). This is the mathematical operation of choosing the alternative with the highest utility value $a^* = \max_{\vec{u}} \vec{a}$.

The weighted additive rule is also the standard approach in computational decision making in all types of search and optimization tasks. By making the above steps explicit, Shah and Oppenheimer (2008) point out ways in which heuristics reduce the cognitive effort, such as examining only a subset of available cues or omitting the weights.

Other simplifications require a modification of the process to allow for several iterations of decisions. The QuickEst heuristic (Hertwig, Hoffrage, & Martignon, 1999) is a good example of such an approach. Here is our formalization:

1. A vector of alternatives is given or retrieved from memory or the environment $\vec{a} = (a_1, a_2, \dots, a_n)$.
2. Cues are ranked according to their "coarseness" (Hertwig et al., 1999, p. 223), which means those cues that can eliminate many alternatives quickly are used first. We thus get the cue vector $\vec{c}_\succ = (c_1, c_2, \dots, c_m)$ with ordered cues. In the QuickEst heuristic, cues are assumed to be boolean functions, thus mapping any alternative to the value 0 (false) or 1 (true), meaning that an alternative either has a certain property or not.
3. Repeat, starting with $i \leftarrow 1$ and $\tilde{a} \leftarrow \vec{a}$:
   (a) Apply cue c_i to alternatives in $\tilde{a}$, resulting in a set of alternatives with positive evaluation $\tilde{a}^+$ and alternatives with negative evaluation $\tilde{a}^-$.
   (b) If $|\tilde{a}^+| = 1$, return the only element in $\tilde{a}^+$, otherwise repeat with $i \leftarrow i + 1$ and $\tilde{a} \leftarrow \tilde{a}^+$

The stopping criterion in step 3 is incomplete. One can define different ways of dealing with situations where not single alternative fulfills all available cues. QuickEst is just one example of many heuristics that decide in iterations, changing the cues and/or the alternatives in each iteration.

```
def DECIDE ($\vec{a}$, $\vec{c}$, $\vec{p}$):
  $M \leftarrow$ FILL-DECISION-MATRIX($\vec{a}, \vec{c}$)
  $\vec{r} \leftarrow$ AGGREGATE-AND-ORDER($M$)
  $a^* \leftarrow$ FIRST($\vec{r}$)
  if ACCEPTABLE ($a^*, \vec{r}, M$):
    return $a^*$
  else:
    DECIDE (UPDATE-ALTERNATIVES ($\vec{a}$), UPDATE-CUES ($\vec{c}$), UPDATE-PARAMS ($\vec{p}$))
```

Figure 1. : General model of heuristic decision-making. The functions and parameters are explained in the text.

The Integrated Model

We combine the concepts from the weighted additive rule with iterative procedures into one algorithm, shown in Figure 1. The initial call is

DECIDE (GET-ALTERNATIVES (), GET-CUES (), GET-PARAMS ())

The GET-... functions are special cases of the UPDATE-... functions without initial values. These functions retrieve alternatives, cues and parameters (such as weight vectors) from memory or from the environment, the GET- functions without any prior knowledge (except for the situation and task context), the UPDATE- functions with knowledge about the values of the last iteration.

Thus, DECIDE starts with an initial vector of alternatives $\vec{a}$, an initial vector of cues $\vec{c}$ and an initial set of parameters $\vec{p}$. Let us assume that alternatives and cues contain at least one element, whereas $\vec{p}$ can be empty, depending on the configuration of the algorithm. The function FILL-DECISION-MATRIX calls all cues in $\vec{c}$ on all alternatives in $\vec{a}$. The cue functions may be value or ranking functions (more on this in the next section).

The function AGGREGATE-AND-ORDER combines the judgements of different cues and different alternatives and sorts alternatives so that the collectively highest ranked alternative is first. AGGREGATE-AND-ORDER may need a weight vector or other parameters, depending on the used aggregation function, as discussed in the following section. The sorting is not strictly necessary and seems very unlikely as a model of human thinking, it just facilitates the next step, which is just to take the first alternative in the ranked vector $\vec{r}$. Without the sorting, the function FIRST would have to be replaced with respective functionality. But usually the question of which alternative is best with respect to the combined evaluations is uncontroversial, and the typically low number of alternatives in human-like decision procedures leads only to a slight computational overhead.

This step can contain many special cases, which are not explicitly handled in the algorithm in Figure 1 and can be set according to the heuristic to be modeled or used. If there is only one cue in $\vec{c}$, the step contains only the sorting of alternatives according to the cue values or the alternatives are already sorted by the cue rating function. If there is just one alternative in $\vec{a}$, this step is trivial. The preceding step of filling the matrix, however, should not be omitted, because the step to decide whether to accept the alternative is still to come. If several alternatives share the first rank, they will be sorted in some way in the ranked vector $\vec{r}$, maybe using a random order for equal evaluations. Again, the next step may decide to run another iteration to make a more informed choice.

The function ACCEPTABLE decides whether the best alternative is good enough to be the result of the decision process. This function can base its decision on the ranking $\vec{r}$ or the pure cue values from the decision matrix $\vec{M}$. Thus, an alternative may be judged unacceptable when the next-best alternative in the ranking has an equal or very close utility score. Or it may be rejected as being rated too low on certain cue values. For practical reasons, it is advisable to also include a counter in ACCEPTABLE and set a maximum number of iterations, otherwise the decision process could run infinitely. On top of these considerations is the question of how difficult or simple it would be to generate more cues or alternatives. With a high effort to get more cues, one might be more inclined to guess between equally ranked alternatives.

If ACCEPTABLE is not satisfied with the best alternative, DECIDE is called again, but the input parameters may be changed. The vector of alternatives might be diminished by alternatives that did not satisfy some cue criterion (as in the QuickEst heuristic) or that falls below as certain threshold (as in the elimination by aspects heuristic (Svenson, 1979)). Alternatives may also be added if none of the previous alternatives was good enough. This could mean additional effort of memory retrieval or perception. In the same way cues can be reduced or added to. Several heuristics such as QuickEst or Take-the-best (Gigerenzer & Goldstein, 1999) use one cue per iteration, thus UPDATE-CUES would return the next cue. One may re-consider cues (see Rieskamp and Hoffrage (1999, p. 150)) that have been used in previous iterations. The weight vector is usually assumed to be static, but the algorithm leaves the possibility open to also adapt weights or other parameters.

Discussion of the Model

We motivated the model by formalizing the basic decision procedure by Shah and Oppenheimer (2008) and the QuickEst heuristic (Hertwig et al., 1999) as an instance of iterative heuristics. Let us first make sure that the algorithm in Figure 1 really comprises the two.

For the Shah and Oppenheimer model, a list of fixed alternatives can be provided with the function GET-ALTERNATIVES (step 0); GET-CUES corresponds to the retrieval of cues (step 1), GET-PARAMS models the retrieval of the weight vector (step 3). The retrieval and storage of cue values (step 2) corresponds to FILL-DECISION-MATRIX, and their aggregation (step 4) as a weighted sum is one way of instantiating AGGREGATE-AND-ORDER. The choice of the best alternative (step 5) is simplified by the ordering included in AGGREGATE-AND-ORDER. The Shah and Oppenheimer (2008) model assumes only one decision cycle. Therefore, ACCEPTABLE will accept the alternative with the highest aggregated cue value, the UPDATE- functions are irrelevant.

Step 1 in QuickEst again corresponds to GET-ALTERNATIVES being instantiated to returning a list of fixed alternatives. For GET-/UPDATE-CUES we need a vector of ordered cues $\vec{c}_{\succ}$. The functions GET-/UPDATE-CUES return one element per round in the order given in $\vec{c}_{\succ}$. FILL-DECISION-MATRIX is reduced to applying the one cue chosen for the iteration to all the alternatives, filling the matrix with boolean values (or integers 0 and 1). The function ACCEPTABLE accepts a choice when only one alternatives comes out true, otherwise it requires another iteration. UPDATE-ALTERNATIVES returns all alternatives that were rated as true in the last iteration.

With respect to Gigerenzer's (2001) adaptive toolbox, the algorithm in Figure 1 can be mapped to the tools in the following way:

- *Search rules*: Gigerenzer distinguishes the search for alternatives, as modeled in the GET-/UPDATE-ALTERNATIVES functions, and the search for cues, as in the GET-/UPDATE-CUES functions. The options he lists ("random search, ordered search (e.g., looking up cues

according to their validities), and search by imitation of conspecifics, such as stimulus enhancement, response facilitation, and priming" (Gigerenzer, 2001, p. 44)) would have to be implemented by different versions of these functions.
- *Stopping rules* are implemented by the function ACCEPTABLE. They can, for instance, be based on absolute aspiration levels as in satisficing models (Simon, 1956) or if a cue is found that favors one alternative as in QuickEst or other one-good-reason heuristics (Gigerenzer & Goldstein, 1999).
- *Decision rules* correspond to AGGREGATE-AND-ORDER, i.e. the aggregation of cue values. Gigerenzer (2001) describes them after the stopping rules, whereas our model performs this step before calling ACCEPTABLE. If ACCEPTABLE is implemented in a way that it can decide without knowing the aggregated values, the algorithm may be modified to first test for acceptability and then call AGGREGATE-AND-ORDER only in the affirmative case. In the case study below we use an acceptability criterion that compares the aggregated value of the best choice to the aggregated value of the second best choice. Thus, the algorithm in Figure 1 is slightly more general, but also less efficient in cases where ACCEPTABLE works with unaggregated values.

Svenson (1979) classifies decision rules along three dimensions, which correspond to our model in the following ways:

- *Metric level of cues*: We have differentiated between cue ranking and cue value functions, where ranking corresponds to an ordinal scale and values to ratio or interval scale. However, the ordinal scale is not treated as a mere ordering by Svenson, some rules use criterion values corresponding to absolute thresholds. The instantiation of AGGREGATE-AND-ORDER must be consistent with the output of the cue functions. Different ways of combining ordinal values are presented in the next section.
- *Lexicographic order*: This dimension says whether cues are used all at once (which would mean that no order is necessary) or whether they are used in an iterative fashion, one per iteration, in which case the cues must be ordered. We have shown in the QuickEst heuristic, how this can be implemented with our model.
- *Commensurability*: This dimension classifies rules according to whether a high score by one cue can counterbalance a low score by another. Weighted sums are an example of commensurable aggregation, whereas QuickEst is non-commensurable (because once an alternative has been eliminated it will never be the final choice, no matter how well it would be rated by subsequent cues). This quality is determined by an interplay of AGGREGATE-AND-ORDER, ACCEPTABLE and the control of the iterations by UPDATE-ALTERNATIVES and UPDATE-CUES.

Svenson (1979) points out that decision rules could change or be adapted during the decision process. Our model provides no explicit support for this, but it could well be built into the different functions. For example, one might have several implemented options for UPDATE-CUES, such as using the next cue from a list of lexicographically ordered cues, using a random cue, or using the cues that one can restore from memory in a certain time frame. The function UPDATE-CUES could contain a conditional that decides which of the three possible functionalities to use in the current situation.

Aggregation of Cues

If the decision matrix has more than one column and more than one row, we have to aggregate the judgements of all cues to determine the overall best alternative. The classical approach is a weighted sum, or the simpler variant of an unweighted sum (which corresponds to a weighted sum when all weights have the same value). But still we would need all cues to be implemented as a cue value function, which means they have to provide numeric values. In many cases it seems simpler to just rank the alternatives by preference without giving them explicit values. But how can we aggregate rankings?

The field of computational social choice (Brandt, Conitzer, Endriss, Lang, & Procaccia, 2016) is concerned with developing fair voting systems. The assumption there is that each voter from a finite set of voters $V, |V| = m$ is given a finite set of alternatives $A, |A| = n$. Each voter i casts a *ballot* (or ranking), which is a linear ordering $\succeq_i$ of A. A *profile* $P = (\succeq_1, \succeq_2, \dots, \succeq_m)$ specifies a ballot for all voters in V; $\mathcal{L}(A)^m$ denotes the set of all such profiles for a given m. A *social choice function* maps a profile to a single combined preference ranking scf : $\mathcal{L}(A)^m \rightarrow \mathcal{L}(A)$[2].

If we now replace the set of voters V by the set of cues C in our decision procedure, we can use social choice funtions to combine cue rankings into an overall ranking of the alternatives. However, there is a fundamental difference in the tasks of voting and cue aggregation that one should not forget: voting assumes equality of voters (each has one vote), whereas cues are often assumed to be of unequal importance. For example, Russo and Dosher (1983) assume that the least important cue is “the one with the smallest dimensional difference” (Russo & Dosher, 1983, p. 683). Huber (1979) mentions the relation of his weighted-sets-of-dimensions model to social choice functions and suggests an interpretation of $\succeq$ as a “partial order of social power” (Huber, 1979, p. 163).

Cue value functions can be transformed into cue ranking functions by sorting the alternatives for each cue according to the value assigned by the cue. Thus, a value aggregation function (i.e. summing) can be used as a social choice function, by sorting the alternatives according to the combined score. The reverse case (from ranking to value) is possible, but under-specified. Scoring functions offer different ways of assigning cue values to ranked alternatives.

Scoring Scoring rules assign a value to each alternative in a ranking, thus transforming a ranking function into a value function. The methods differ in the way they assign the numbers, here are two examples:

- Borda count: $f(r) = n - r$, where r is the rank of the alternative and n is the number of all alternatives. Thus, the highest ranked alternative receives value $n - 1$, the second highest $n - 2$, the lowest one 0.
- Formula One Championship: The values 25, 18, 15, 12, 10, 8, 6, 4, 2, 1, 0, $\dots$0 are assigned to the ranked alternatives in the given order, i.e. the highest ranked alternative is assigned the value 25, the second highest 18, etc.

For a comprehensive list of scoring methods we refer to Zwicker (2016, p. 36 ff.). The transformed cue values are aggregated with an unweighted sum.

[2]The notation and definitions are taken from Zwicker (2016).

Voting Voting methods combine rankings directly without the intermediate step of assigning scores. For two alternatives and odd number of cues, majority rule is the uncontroversial choice. (Brandt et al., 2016, p. 34, proposition 2.2)

Most voting rules are based on the concept of *net preference*. The net preference of alternative a_1 over alternative a_2 is defined as

$$Net(a_1 > a_2) = |\{c \in C | a_1 \succ_c a_2\}| - |\{c \in C | a_2 \succ_c a_1\}|$$

i.e. the number of cues that rank a_1 over a_2 diminshed by the number of cues that rank a_2 over a_1

Note that even though the decision matrix is filled row-wise (cue-wise), the net preference compares alternatives, resulting in a matrix of alternative preferences:

$$\begin{array}{c|cccc|}
 & a_1 & a_2 & \dots & a_n \\
\hline
a_1 & - & Net(a_2 > a_1) & & \\
a_2 & Net(a_1 > a_2) & - & & \\
\dots & & & - & \\
a_n & & & & - \\
\hline
\end{array}$$

An alternative a that defeats every other alternative in the strict pairwise majority sense $a > a_i$ for all $a_i \neq a$ is called a *Condorcet winner*. It is generally assumed that a Condorcet winner is a fair choice in an election. However, not every profile has a Condorcet winner, but it seems desirable that a social choice function returns the Condorcet winner if one exists. Such a function is called a *Condorcet extension*. The Borda and Formula-One-Championship scoring methods are not Condorcet extensions. Here are two examples of voting rules that are:

- Copeland method: Compute the Copeland score for each alternative: $Copeland(a_i) = |\{a_j \in A | Net(a_i > a_j)\}| - |\{a_j \in A | Net(a_j > a_i)\}|$, i.e. count all alternatives that are worse than a_i and subtract the number of alternatives that are better than a_i according to the net preference. Select the alternative with the highest Copeland score.
- Sequential Majority Comparison is a pairwise comparison of alternatives:

```
1. pick some order of alternatives: $(a_1, a_2, \dots a_n)$
2. winner ← $a_1$
3. for $i = 2 \dots n$: if Net($a_i$ > winner) then winner ← $a_i$
4. return winner
```

  We treat Sequential Majority Comparison as an atomic social choice function. In the decision model, one could treat it as an iterative method, in which alternatives are excluded in each iteration. However, Sequential Majority Comparison assumes that all alternatives are compared sooner or later so that the iterative approach does not give any benefit here.

More on voting rules and Condorcet extensions can be found in Zwicker (2016, p. 33 ff.).

Multiround rules Another class of voting rules compares alternatives in rounds, removing alternatives in each round and re-ranking the remaining alternatives. This approach is similar to the elimination by aspects heuristic (Svenson, 1979, p. 90), which we use below. For example, the Nanson rule (a Condorcet extension) eliminates in each round the alternatives with a Borda score below the average Borda score. For more details on multiround rules refer to Zwicker (2016, p. 37).

Case Study

The goal of our computation model is on the one hand to integrate existing models of heuristic decision-making, as we have already discussed. On the other hand, we want it to be usable as a component in computational decision-making systems. In this section we present experiments for a specific task, the Traveling Salesperson Problem. We first introduce the task, then we address some implementation details, and finaly show some experiments with different parameter combinations of the algorithm.

The Traveling Salesperson Problem

The Euclidean Traveling Salesperson Problem (TSP) is the task of finding the shortest possible tour through a set of given 2D points. Apart from being a representative of the class of NP complete problems (i.e. problems where the number of potential solutions grows exponentially with the size of the problem instance), it has been an object of research on human problem solving (J. N. MacGregor & Chu, 2011). Optimization programs can nowadays solve problems with thousands of points optimally and problems with millions of points near-optimally. These approaches usually tackle the whole task at once, whereas people start with one point and add connections one after the other (Tenbrink & Wiener, 2009) and find surprisingly good solutions (J. MacGregor & Ormerod, 1996). For related tasks that in volve user interation, a human-like approach seems desirable. Therefore, we treat a TSP solution process as a sequence of decisions of which point to add next to the tour.

One characteristic of TSPs is that the instances differ a lot, so that it is hard to find one parameterization that solves any instance equally well (Kirsch, 2011, 2012). This also puts some limit on the optimization of parameters, because a set of parameters might work well for one TSP instance and bad for another.

For the experiments we use an open dataset [3] that contains over 25,000 human solutions to 90 TSP problems (Rach & Kirsch, 2016).

Implementation

We implemented the general decision algorithm in a system called Heuristic Problem Solver (HPS) (Kirsch, 2016). The source code used for the experiments below is available at `https://bitbucket.org/kirschalexandra/heuristicproblemsolver/commits/tag/compSoC-TSP`.

A TSP instance is represented by a set of 2D points. A state in the solution process is a partial solution, which is initially empty and contains all the points plus one (the starting point has to be the same as the end point) in the end. The last point in a partial solution is the current point, from which the next step of the tour has to be decided (Figure 2).

The implementation of GET-ALTERNATIVES is constant in all experiments. In HPS alternatives are generated by so-called *producers*, functions that return alternatives according to the state of the world. We used three producers that decide, which point should be the start point (Rach & Kirsch, 2016), one that returns the three closest unvisited points to the current point, one that suggests the next point on the convex hull (i.e., the outline of the problem), and one that suggests the starting point to return to when the rest of the tour has been constructed.

[3] `http://www.wsi.uni-tuebingen.de/lehrstuehle/human-computer-interaction/home/code-datasets/tsp-dataset/perlentaucher-2.html`

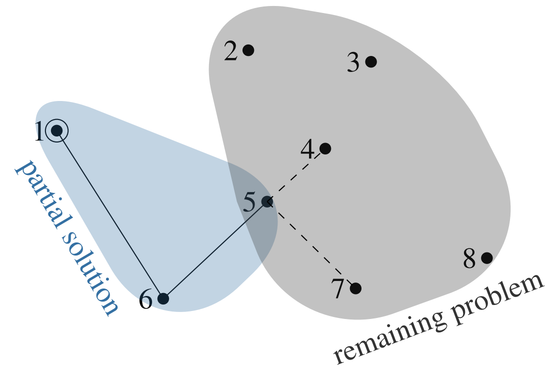


Figure 2. : Representation of TSP instances and states in the solution process. The numbers next to the points are labels. The partial solution starts at point 1 and includes points 6 and 5. Point 5 is the current point from which to decide the next steps. Points 4 and 7 are alternatives for the next decision.

The cues are represented in HPS by so-called *evaluators*, functions that receive all alternatives in the iteration and return an assignment of alternatives to real values between 0 and 1. We have a total of 10 evaluators available (the abbreviations are used in Figure 6):

- *nearest-neighbour (nn)* prefers alternatives that are close to the current point.
- *last-chance (lc)* tries to avoid that a point skips its nearest neighbour. Sometimes points are "left out" on the way and need to be collected in the last steps, which inevitably leads to long tours with crossings.
- *remaining-acc-dist (rd)* estimates the accumulated distance to the remaining points and chooses the one with the lowest estimated future path length. This is a similar rationale as the heuristic function in an A^* search.
- *no-intersections (ni)* rejects alternatives that would lead to an intersection by adding the next edge.
- *avoid-splitting (as)* punishes points that would lead to a line through the middle of the problem, thus splitting it into two halves that could only be combined later by a crossing line.
- *start-intersection (si)* avoids alternatives whose direct connection to the starting point would cause an intersection.
- *follow-lines (fl)* prefers alternatives that lead to a wide angle of the edges at the current point, rather than alternatives that lead to sharp turns.
- *convex-hull (ch)* prefers moving along the convex hull.
- *regions (rg)* uses a clustering of points into regions. In the TSP literature, hierarchical approaches have long been discussed (Graham, Joshi, & Pizlo, 2000). This expert prefers alternatives that are in the same region as the current point.
- *regions-convex-hull (rc)* like *region* prefers points in the same region, but also points in the region that follows the current one on the convex hull.

Not all cues are equally useful, and it may depend on the aggregation of cues, which ones should be used. Therefore, in the experiment we ran an optimization for each cue aggregation function to find the best set of cues, and for weighted sums the weight for each cue. In the first experiment, the GET-CUES function returns a fixed set of evaluators, depending on the used aggregation function. GET-PARAMS returns an association of cues to weights, if necessary.

In the first set of experiments, the functions UPDATE-ALTERNATIVES and UPDATE-CUES are irrelevant, because only one iteration is run. For the second set of experiments, the functionality of these functions is explained below.

Different options for AGGREGATE-AND-ORDER are tested in the first set of experiments. ACCEPTABLE is by default a function that accepts any alternative. In the second experiments we use another definition that allows for an iterative process.

Aggregation Functions

First, we test the rather classical approach of evaluating a given number of alternatives with a given number of cues, thus filling the whole decision matrix once. We look at the effect of different aggregation functions to determine the best choice:

- summing: *weighted sum*, *unweighted sum* (weighted sum with equal weights)
- ranking: *copeland*, *sequential majority comparison*
- scoring: *borda*, *formula-one championship*

All of these six strategies have parameters: weighted sums assign a value between 0 and 1 to each cue, for simplicity we assume steps of 0.1, which results in a parameter space with a size in the order[4] of 10^{10}. For the other methods, there is only the choice of whether to use a cue or not, thus each cue is assigned a Boolean value, resulting in a parameter space of $2^{10} - 1 = 1023$ (the empty set of cues is excluded).

Parameter Optimization From the 90 problems in our TSP data set, 20 are pairs of problems with identical geometrical layout, but slight variations in presentation (e.g. one task is presented with differently colored points, the other with one color for all points). From these pairs, one version per pair was used as a training set, thus we had 10 training tasks for the optimization process. The remaining 70 tasks served as test problems.

The solution quality of the Traveling Salesperson Problem is usually measured by the percentage above the optimal tour length (PAO). For each TSP instance, only a finite number of specific PAO values is possible (since there is a finite number of tours). Therefore, the PAO values of different TSP instances are not comparable and should not be aggregated. To allow for some aggregation of the 70 PAOs for each tour, we normalize the PAO value among all solutions that were generated in the following experiments with a student transformation: $p'_i = \frac{p_i - m_i}{s_i}$, where p_i is the percentage above optimum for a specific solution of TSP instance i; p'_i is the normalized value; m_i is the mean PAO of all solutions for instance i, and s_i is the (descriptive) standard deviation of PAO values over all solutions of instance i. This normalization allows for a comparison among different instantiations of the decision procedure, but it is not appropriate or intended for assessing the absolute solution quality.

For the five conditions that are controlled only by the number of cues used, we used the brute force method of solving each of the 10 training tasks with each of the 1023 parameter sets,

[4] 10^{10} is an upper bound, because some configurations are equivalent, for example when all weights have the same value.

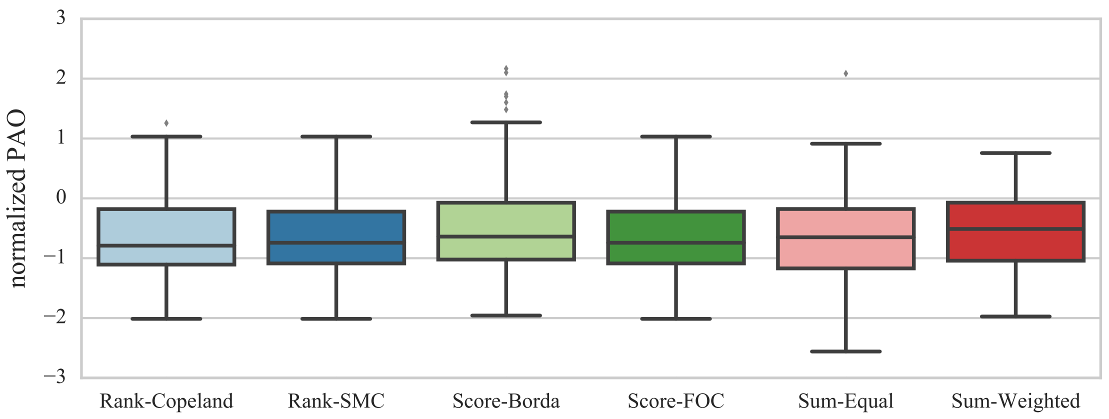


Figure 3. : Quality with optimized configurations over 70 test tasks. Lower values denote shorter, i.e. better, tours.

normalizing over the solutions per task, aggregating the PAO values for each parameter set, and choosing the one with the lowest relative PAO. This takes about 20 minutes per condition on a desktop computer (Intel Core i5-2520M (2.50GHz), 8 GB RAM).

For the weighted sum, as the parameter space is far too large for the brute force approach, we used a genetic algorithm (Banzhaf, Nordin, Keller, & Francone, 1998) with a population size of 16 and 64 generations, resulting in 1024 tested parameter combinations, thus having an optimization effort comparable to the other conditions. The fitness of each parameter combination was again assessed by aggregating normalized PAO values.

To evaluate the robustness of the aggregation functions, we ran the same optimizations with two other measures of tour quality: 1) a rating that measures the similarity of a solution to human solutions of the same task, and 2) the count of how many participants in the data set chose the same solution for a tour. In addition, for the weighted sum we experimented with different numbers of generations of the genetic algorithms to assess the necessary effort for optimization.

Quality of Decisions Figure 3 shows the comparative results over the 70 test problems, lower values denote shorter, and therefore better, tours. The median under all conditions is below 0 (which would be the average relative performance per tour), which shows that in general the performance is on some common level, but that all conditions can produce exceptionally long tours. The performance in all conditions is practically the same when the normalized PAO is aggregated. This does not mean, however, that they all produce the same results on the same tasks. Figure 4 shows the unnormalized PAO values per task for the first 20 tasks from the test set.

Optimization Effort For this experiment we more or less arbitrarily decided on using 1024 parameter settings in the optimization. Possibly, a similar level of performance could be reached with less optimization effort, or an additional effort could boost the performance. Figure 5a shows the performance for the weighted sum on the test data when trained in 16, 32, 64 and 128 generations. The performance is practically identical for all variants. To better understand the reason, Figure 5b

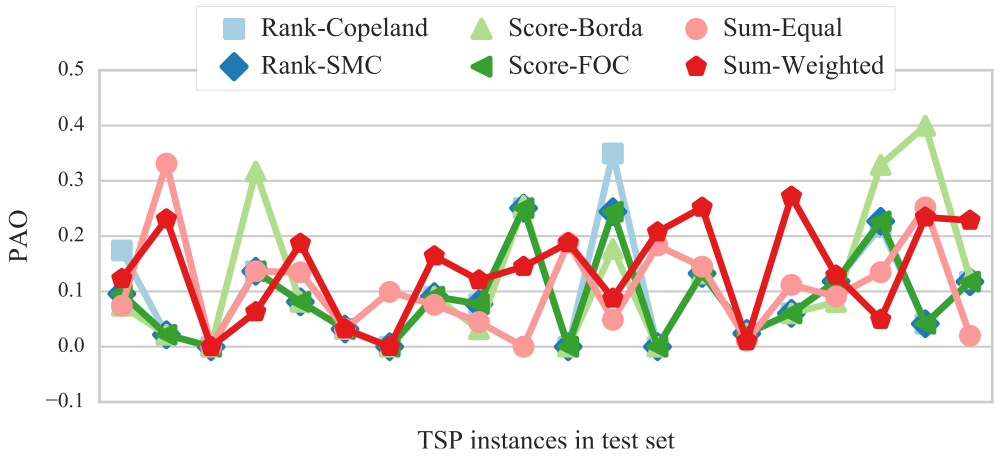
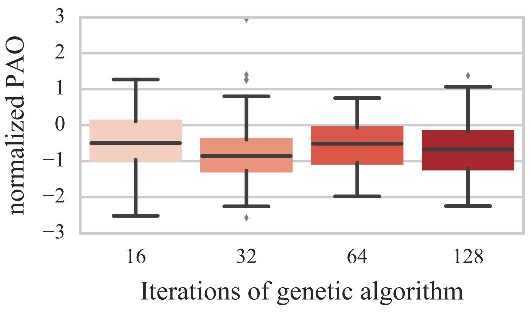
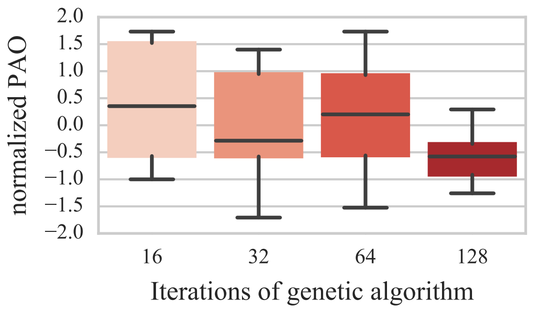


Figure 4. : (Unnormalized) PAO of optimized configurations for 20 test tasks.



(a) Tested on test data.



(b) Tested on training data.

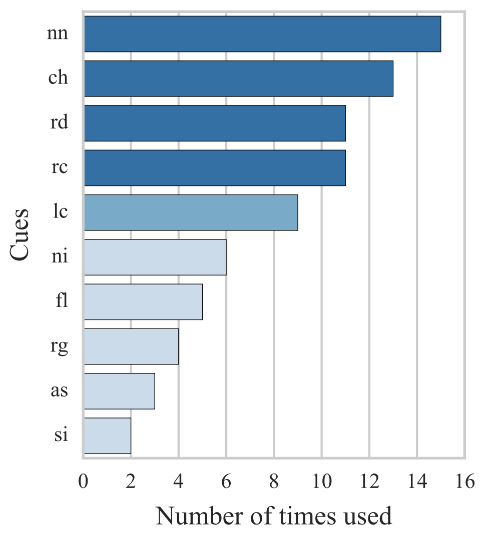
Figure 5. : Results for weighted sum after different number of iterations in genetic algorithm. 64 iterations correspond to the optimization effort of the non-weighted aggregation functions.

shows the performance on the training data. This shows that the algorithm is improving on the training data with more generations, but this has no effect on the performance on the test data.

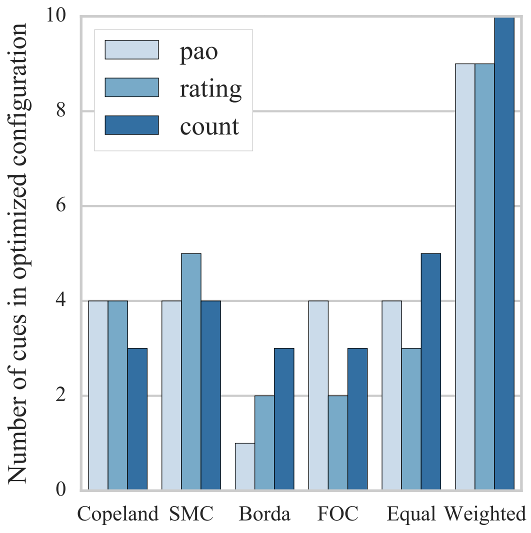
This overfitting can be explained by the high diversity of TSP instances. In such a setting, a costly optimization seems not to pay off, because instances require different parameters. This point is part of the discussion section.

Influence of Parameter Choice Another question is how important the parameters are for making good decisions. In the optimization, all variants of aggregation functions faced the same challenge of the diversity of TSP instances. But how much is the performance influenced when the value to be optimized for is a different one than that is tested for? Or how does the performance change with a non-optimized configuration?

Figure 6 analyzes which cues were used in the optimized configurations. It shows that there are some cues that were used more often than others. It also shows that the weighted sum used 9 or 10 cues, no matter for which target value it was optimized, whereas the conditions with a boolean



(a) Number of times expert was chosen in the 18 optimized configurations.



(b) Number of experts in optimized configurations.

Figure 6. : Expert usage in optimized configurations.

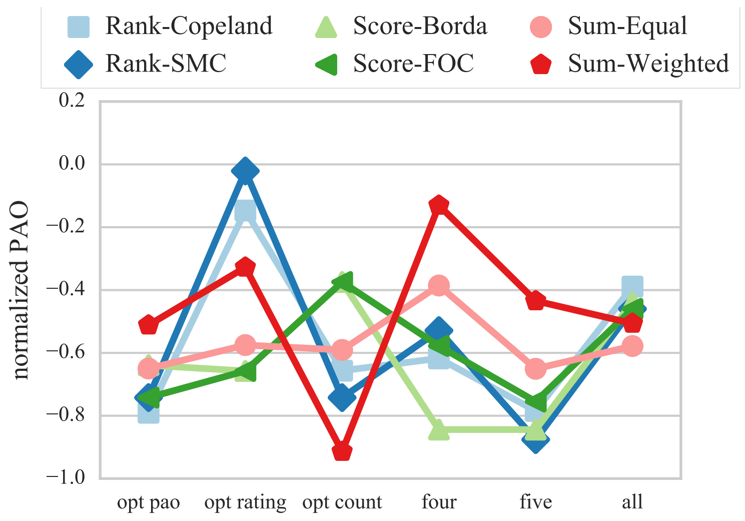
choice of cues mostly used 3–5 cues.

From these observations we constructed the following parameter sets:

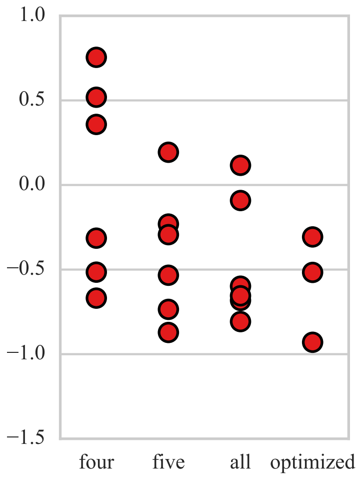
- *opt pao/ rating/ count*: the individual parameter set for the aggregation function when optimized for PAO, rating of similarity to human solutions, and the count how often a solution was chosen by participants
- *four/ five/ all*: Using the four/ five cues that were used most often in the optimization process (*four*: dark blue cues in Figure 6a, *five*: dark and medium blue cues in Figure 6a); *all* uses all available cues. For the weighted sum for each condition, six randomly generated sets of weights were used; Figure 7a shows the median of those six variants, Figure 7b shows the single parformances.

Figure 7a shows that the scoring methods are the most robust with respect to parameter choice. The ranking methods are mostly stable, but perform notably worse when trained on the rating value. The unweighted sum shows a similar level of stability as the scoring methods, but in most cases performs slightly worse. The most fluctuation is shown in the weighted sums aggregation function. Interestingly, it performs best (tested on PAO) when trained on the number of times a solution was chosen by participants. Figure 7b suggests that this fluctuation is due to the choice of particular weights. In all three predefined parameter sets, there is a set of weights that outperforms or at least compares to the scoring methods, but there are always other choices of weights that make the performance notably worse.

Summary of Aggregation Functions The method of aggregation is mostly irrelevant in the specific task regarded here. The optimization effort we used could probably be reduced even further



(a) Configurations of optimization process and three constructed ones (for weighted sum: median of six random configurations).



(b) Single results of configurations with weighted sum.

Figure 7. : Robustness with respect to configuration. Lower values denote shorter tours.

for all the aggregation functions. However, the weighted sum seems to need slightly more attention to the parameters, the scoring methods seem to be the most stable ones. In less diverse tasks or with an adequate classification of TSP instances, the effort to learn weights for a weighted sum might pay off.

Iterative Decision Methods

We now test methods that use more flexibility of the decision model. In the last experiment, we varied the setting of AGGREGATE-AND-ORDER, now we look at some combinations of the functions ACCEPTABLE, GET-/UPDATE-CUES and UPDATE-ALTERNATIVES, considering only one cue at a time. Thus, the decision matrix has only one line per iteration, which means that AGGREGATE-AND-ORDER just orders the alternatives according to the one value provided. The other parts are configured as follows:

- ACCEPTABLE: We accept an alternative if $\frac{c(a^*)-c(a^+)}{c(a^+)} > \eta$, where $c(a^*)$ is the (single) cue value of the best alternative, $c(a^+)$ is the cue value of the second-best alternative, η is a parameter, which is varied in the experiment between 0.01 and 2. If there is only one alternative left or no more cues, the best (or only) alternative is chosen.
- GET-/UPDATE-CUES: We mimic two one-good-reason heuristics (Gigerenzer & Goldstein, 1999), which use one cue per iteration:
  - The *Minimalist heuristc* uses cues in random order and makes use of the recognition heuristic whenever possible (i.e. prefers the option that it recognizes). The recognition heuristic makes no sense in TSPs, so we just use the cues in random order.
  - The *Take the Best* heuristic assumes that the decision maker has some knowledge about the importance of cues for a specific task and uses the cues in the order from most important to least important. Determining the “best” order of cues is similar to determining

condition	UPDATE-CUES	UPDATE-ALTERNATIVES
min	minimalist	keep all
elim-min	minimalist	elimination by aspects
ttb	take-the-best	keep all
elim-ttb	take-the-best	elimination by aspects

Table 1:: Conditions for iterative decision procedure.

weights. We used for all tasks the order of the cues as they were included in the optimized configurations shown in Figure 6a. When several cues have the same relevance (as rd and rc), the order is chosen randomly.

- UPDATE-ALTERNATIVES:
  - Keeping all alternatives by passeing them on the the next round.
  - The elimination by aspects heuristic of Svenson (1979, p. 90). In each iteration it removes low-ranked alternatives. We defined "low-ranked" as being below the mean of the cue value (unless all alternatives have the same value, then all are kept). We experimented with other parameters, such as one or two standard deviations below the mean, the results were very similar to the ones shown below.

The conditions are summarized in Table 1. GET-/UPDATE-CUES now contains some randomness (even for take-the-best as some cues have the same relevance). Therefore, each task was now solved 20 times per condition.

Acceptance parameter The approach has several parameters, some of which are fixed and were justified in the preceding paragraph. For the parameter η of the ACCEPTABLE function, the choice is not so obvious. Figure 8 shows the performance of the *ttb* condition for different choices of η (the other conditions reveal a similar picture).

The data suggests that it is beneficial to use a low acceptance threshold. For the take-the-best update function, this makes sense as the (presumably) better cues are used first and if their decision is not used, the less reasonable cues make the decision. For the minimalist update function, this argument does not hold, but since the cues are used in random order, waiting for other cues does not in general bring any benefit either. Also from a computational point of view, few iterations are preferable.

Quality Figure 9 shows the result of testing the four conditions of Table 1 with $\eta = 0.1$, the left figure showing the best runs from each of the 20 runs per task, and the right figure the worst runs. Be aware that the x-axis shows different ranges, on the left all median values are below 0, on the right, most of them are above 0.

Condition *min* can lead both to very good and very poor results, depending on the random choice of cues. The elimination of alternatives stabilizes the behavior significantly. The poorest results of condition *elim-min* are only slightly worse than the best ones. But this stabilization also removes the positive outliers of condition *min*. Possibly, in some cases the best decision is one that

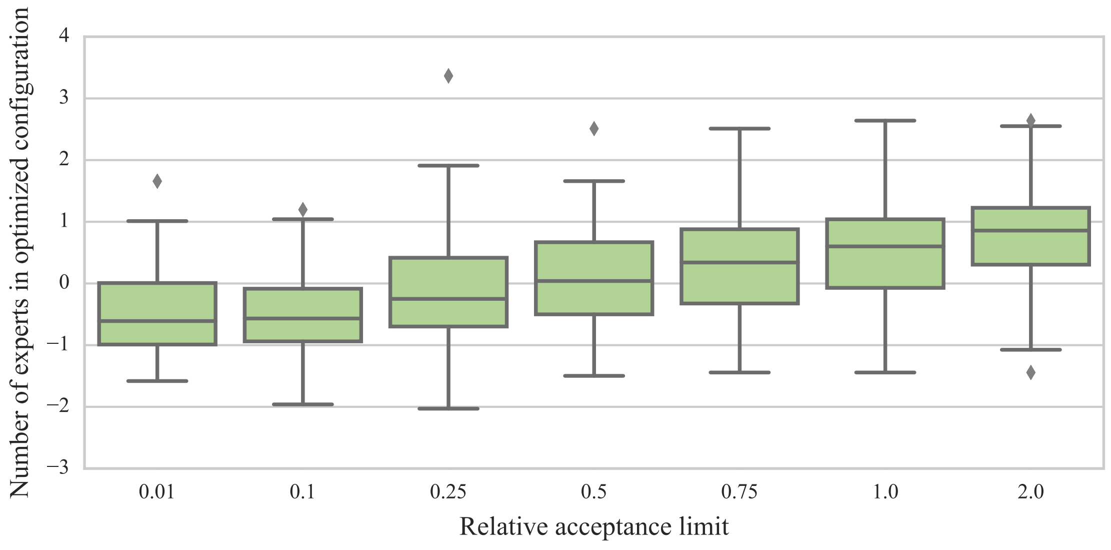


Figure 8. : Acceptance parameter η for *ttb* condition.

is evaluated poorly by most cues and good by few. By eliminating alternatives, a kind of aggregation sets in, which removes the positive as well as the negative outliers.

Condition *ttb* performs comparably to *elim-min*. Interestingly, adding elimination of aspects leads to results similar to those of *min*. One might conclude that good results can be obtained either by a good ranking of cues (*ttb* vs. *min*) or by elimination of alternatives (*elim-min*), when such knowledge is not available. The question remains of how to obtain the positive outliers (or produce more of those) of *min* and *elim-ttb*.

Summary of Iterative Methods The parameterization of the cue ordering and whether to eliminate unpromising alternatives, has a notable effect in the TSP domain. With the general decision procedure many more variations are possible, like using two cues per iteration, combined with dif-

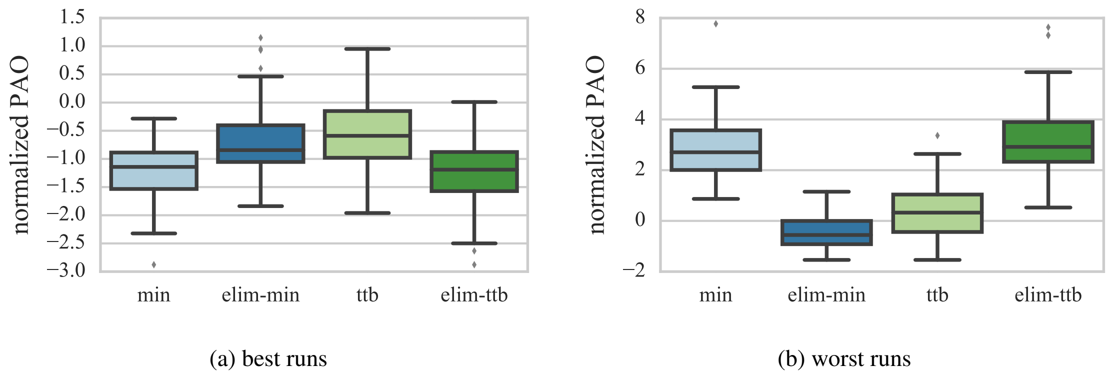


Figure 9. : Different variants of one-good-reason heuristics with acceptance parameter $\eta = 0.1$.

ferent aggregation functions for those cues. Also the generation of alternatives could be explored in different directions.

Discussion and Conclusion

We have presented a computational model of heuristic decision making and shown how it incorporates other generalizing frameworks of heuristics. By instantiating the functions in the model, different heuristic strategies can be implemented. But are the parameters of the model instantiated? Some of the factors seem to be the decision task (including available time, importance of decision, necessity to justify decision); accessibility/ presentation of alternatives and cues; familiarity with the task; an explicit trade-off between error and effort (see (Russo & Dosher, 1983, p. 694)).

The framework is intended as a better means to explore, model and simulate different strategies. The instantiation options of the functions are far from trivial. In this paper we have given some ideas of how to represent cue functions and how to aggregate different cues using methods from computational social choice. Some of these methods resemble the aggregation methods in the literature on heuristic decision making.

Implementing functions for getting and updating alternatives and cues needs insights on memory retrieval, perception and attention. The presented framework makes this connection explicit and invites further research. For replicating studies with given alternatives and cues, the instantiation of these functions is straightforward.

The model does not contain an explicit learning functionality, but as argued in the discussion of the model, any function can work as a meta-function, including several implementations that are selected according to the situation. One could go further and change the functions in each step or after each decision. For example in iterative procedures with one cue per iteration, the order of the cues can depend on the outcome of previous decisions. Simple decision methods, such as aggregation by voting, may also serve as a bridge to more refined methods, such as weighted sums: for a new task the cruder method can be applied, while each decision provides more experience, from which, in the long run, weights or other parameters for more sophisticated methods can emerge.

Svenson (1979, p. 92) argues in the opposite direction. He proposes that over time, decision makers develop simplifying methods and apply the more sophisticated methods in new or important situations. The two views can be reconciled in that decision makers might learn over time how to configure the decision process and to know in which situations more costly decision methods pay off. Svenson (1979, p. 93) notes that "[...] a decision problem may be more fruitfully viewed as a problem of classification". This is certainly true in our model. It allows a wide variety of parameters that may best be set by classifying the situation and deriving the appropriate instantiation of the parameters.

From a computational point of view, the model provides a structure that may help to make decisions that are better understandable for people. A program might explicitly display the alternatives and cues considered. This can also open possibilities of mixed-initiative problem solving of collaborating humans and computers. However, parameters such as aggregation function or weights, are hard to determine for users. The different choices for the components of the model constitute new parameters that again have to be optimized for or that have to be set by people. Learning or classification approaches are necessary to relieve users or developers from this choice.

We have presented a specific example of a computational decision-making process for TSP solving. The results are not intended to be generalized to other tasks and we have by far not explored the whole spectrum of instantiations of the algorithm. One especially interesting path is to examine noncompensatory aggregation methods for cues. Compromises as in weighted sums (which are the predominant method of attribute integration in artificial intelligence) can lead to acceptable average results, but may fail in special cases. As an example, we have used the Heuristic Problem Solver for robot navigation (Kirsch, 2017), where the alternatives are possible control commands and cues contain considerations such as moving towards the goal, moving forwards (instead of side- or backwards), and staying away from obstacles. If the robot starts in a narrow space with its back to the goal position, it sometimes doesn't move at all, because the urge of moving towards the goal is canceled out by the urge to move forward and to stay away from obstacles. A noncompensatory strategy might help to make sure that the robot moves at all. And again, a classification of the situation could improve the behavior.

In sum, our model provides a general framework for research on human heuristics as well as decision making in artificial intelligence. We have given some ideas for instantiating the framework and using it in a computational context. There are many more ways in which this model can be instantiated and used in the future.

References

Ariely, D. (2010). *Predictably irrational: The hidden forces that shape our decisions*. New York: Harper Perennial.

Banzhaf, W., Nordin, P., Keller, R., & Francone, F. (1998). *Genetic programming — an introduction*. San Francisco, CA: Morgan Kaufmann.

Brandt, F., Conitzer, V., Endriss, U., Lang, J., & Procaccia, A. D. (Eds.). (2016). *Handbook of computational social choice*. Cambridge University Press.

Gigerenzer, G. (2001). The adaptive toolbox. In G. Gigerenzer & R. Selten (Eds.), *Bounded rationality: The adaptive toolbox.* Cambridge, MA: MIT Press.

Gigerenzer, G., & Brighton, H. (2009). Homo heuristicus: Why biased minds make better inferences. *Topics in Cognitive Science*, *1*, 107–143. doi: 10.1111/j.1756-8765.2008.01006.x

Gigerenzer, G., & Goldstein, D. G. (1999). Betting on one good reason: The take the best heuristic. In G. Gigerenzer, P. M. Todd, & the ABC Research Group (Eds.), *Simple heuristics that make us smart.* Oxford University Press.

Graham, S. M., Joshi, A., & Pizlo, Z. (2000). The traveling salesman problem: A hierarchical model. *Memory & Cognition*, *28*(7), 1191–1204.

Hertwig, R., Hoffrage, U., & Martignon, L. (1999). Quick estimation: Letting the environment do the work. In G. Gigerenzer, P. M. Todd, & the ABC Research Group (Eds.), *Simple heuristics that make us smart.* Oxford University Press.

Huber, O. (1979). Nontransitive multidimensional preferences: Theoretical analysis of a model. *Theory and Decision*, *10*, 147–165.

Kirsch, A. (2011). Humanlike problem solving in the context of the traveling salesperson problem. In *Aaai fall symposium on advances in cognitive systems.*

Kirsch, A. (2012). Hierarchical knowledge for heuristic problem solving — a case study on the traveling salesperson problem. In *First annual conference on advances in cognitive systems.*

Kirsch, A. (2016). Heuristic decision-making for human-aware navigation in domestic environments. In *2nd global conference on artificial intelligence (GCAI).*

Kirsch, A. (2017). A modular approach of decision-making in the context of robot navigation in domestic environments. In *3rd global conference on artificial intelligence (GCAI).*

Kotseruba, I., & Tsotsos, J. K. (2016). *A review of 40 years of cognitive architecture research: Core cognitive abilities and practical applications.* arXiv:1610.08602.

Langley, P. (2017). Progress and challenges in research on cognitive architectures. In *Proceedings of the thirty-first aaai conference on artificial intelligence (aaai-17).*

MacGregor, J., & Ormerod, T. (1996). Human performance on the traveling salesman problem. *Perception & Psychophysics*, *58*(4), 527–539.

MacGregor, J. N., & Chu, Y. (2011). Human performance on the traveling salesman and related problems: A review. *The Journal of Problem Solving*, *3*(2).

Rach, T., & Kirsch, A. (2016). Modelling human problem solving with data from an online game. *Cognitive Processing*, *17*(4), 415–428. Retrieved from `http://dx.doi.org/10.1007/s10339-016-0767-4` doi: 10.1007/s10339-016-0767-4

Rieskamp, J., & Hoffrage, U. (1999). When do people use simple heuristics and how can we tell? In G. Gigerenzer, P. M. Todd, & the ABC Research Group (Eds.), *Simple heuristics that make us smart.* Oxford University Press.

Russo, J. E., & Dosher, B. A. (1983). Strategies for multiattribute binary choice. *Journal of Experimental Psychology: Learning, Memory, Cognition*, *9*(4), 676–696.

Shah, A. K., & Oppenheimer, D. M. (2008). Heuristics made easy: An effort-reduction framework. *Psychological Bulletin*, *134*(2), 207–222.

Simon, H. A. (1956). Rational choice and the structure of the environment. *Psychological Review*, *63*(2), 129–138.

Svenson, O. (1979). Process descriptions of decision making. *Organizational Behavior and Humand Performance*, *23*, 86–112.

Tenbrink, T., & Wiener, J. (2009). The verbalization of multiple strategies in a variant of the traveling salesperson problem. *Cognitive Processing*, *10*, 143–161.

Zwicker, W. S. (2016). Intoruction to the theory of voting. In *Handbook of computational social choice* (chap. 2). Cambridge University Press.