Music Generation Using Machine Learning **MUS-17**

Michael Krause

July 5, 2017

The problem

This report deals with computer generated music. In particular, it discusses two recent approaches for automated melody composition using machine learning methods. Apart from melody, music needs accompaniment (e.g. chords played by different instruments) and must be performed, possibly by a synthesizer. Computer generated music can be used where music is required on-demand (e.g. video games) or as an aid for human composers.

History

A comprehensive survey of different approaches for algorithmic composition can be found in [3]. This section will give a brief overview.

One of the earliest examples of semi-automatic composition is Mozart's dice game (see appendix A), where a minuet is stitched together from different pre-composed parts depending on the rolls of two dice. Since the individual parts are human-made, they are guaranteed musical validity and appeal, but their random combinations may sound unnatural.

More sophisticated approaches that were very popular in the 1970s and 1980s use formal grammars to encode knowledge about music. Non-terminal symbols may refer to different parts of a piece and production rules encode the relationship between these parts. Terminals may correspond to pre-made measures, notes with pitch and duration or chords for accompaniment. A simple musical grammar in Prolog can be found in appendix B.

Formal grammar approaches are particularly suitable for music with strong formal requirements. They have been used very successfully for tasks like harmonization of Bach chorals, see [3] and [4]. As an illustration, a piece in sonata form can be expressed as a formal rule "Sonata $\rightarrow$ ABACABA" and the non-terminals A, B, C can be expanded further. In so-called L-systems, the same rule is used to expand all instances of a non-terminal at a time, which creates repetition and regularities. Variety can be added through stochastic rules that choose different musical elements with varying probabilities.

An alternative approach uses Markov models, where states of the model correspond to musical events (like notes or chords) and the next event is chosen according to a probability distribution at each state. A simple example is given in appendix C. Markov chains have been used as early as the 1960s by composers like Iannis Xenakis, but cannot model higher level structures (like the sonata form). Some researchers have combined Markov models with formal grammars in hybrid approaches.

A very influential composition system was EMI, developed by David Cope in the 1990s (see [3], [4] and also appendix A for sound examples). Cope encoded the structure of different kinds of pieces (e.g. Nocturnes by Chopin, sonata movements by Beethoven) as grammars and then used pattern matching on a number of these pieces to obtain characteristic sequences of intervals that the composer used frequently. A new piece could be obtained by stitching together these common sequences according to the hand-made grammar. However, EMI required a high level of user interaction to classify the sequences correctly.

Papers

Modern artificial intelligence research moves away from symbolic approaches like grammars in favor of machine learning based techniques. These techniques learn properties of given data. Applied to music, they can create music in a style given a lot of examples from that style.

In this section, two papers on melody generation (see [1] and [2]) will be presented, which employ two distinct machine learning methods.

Unit selection using deep learning

The authors in [1] follow an approach they call *unit selection*: a large database of existing melodies is divided into groups of consecutive measures, called units. These units are then concatenated in new orders to create novel music. In order to decide which units get concatenated, two neural networks are trained.

The database consists of over 4000 melodies, all transposed to a common key. This dataset is first augmented by adding transpositions of the same melodies for all twelve tones and then divided into units. In the process, previous information on which units fit together is necessarily lost. To decide, which units should be concatenated, the system takes into account information on the semantic similarity of units (determining, whether two units together make sense musically) and on note level transitions (determining, whether the change from one unit to the next sounds natural or abrupt).

Semantic similarity for melody units is encoded using a *Deep Structured Semantic Model*, a special neural network architecture that was originally designed for semantic web search. In the original application, a DSSM would encode input words (represented as 1-0-characteristic-vectors of dictionary words) as continuous vectors. These continuous vectors represent the input words in a semantic space, meaning that the distance between semantic vectors for semantically similar words like “beach” and “summer” would be very small, whereas the distance between words with very different meanings (e.g. “parliament” and “potato”) would be very large. Applying this to melody generation, a DSSM is trained to assign similar semantic vectors to units that were connected in the original dataset.

The DSSM architecture used in [1] is illustrated in figure 2. The input (in sequential note-by-note description) is transformed into a bag-of-words feature vector. Instead of the exact order of notes, this bag-of-words vector contains information like the number of notes with a certain pitch and duration, the number of rests, counts of two consecutive pitches etc. (see [1, p. 3]).

To train the DSSM, one first defines the cosine similarity

$$\mathrm{sim}(A, B) = \frac{A^T \cdot B}{|A||B|} \tag{1}$$

for two real vectors A, B of equal length and then maximizes the likelihood of the training data using gradient descent, which is

$$\prod_{(A,B)} \mathrm{sim}(A, B) \tag{2}$$

where the product is over the semantic vectors A, B of all consecutive units in the dataset (in fact, the authors also include semantic vectors for randomly chosen other units using softmax, see [1, equation 2]).

Due to the bag-of-words encoding of the input units, the DSSM does not learn information about exact note order. Furthermore, as it assigns one semantic vector per unit, it allows only to compare a unit A with a unit B depending on their semantic similarity, but cannot say whether unit A should follow B or vice versa. Therefore, a second network that can learn information about sequences of notes is needed. This second network is based on the *Long Short-Term Memory* recurrent neural network architecture, which can learn sequences of events: upon an event, the LSTM produces an output which also becomes part of the input when the next event is processed. In addition, an LSTM stores information about all previous events in its “cell state”, which is dynamically updated while the sequence is received.

In this application, the LSTM receives music as a sequence of notes defined by pitch and duration. After having seen a sequence $\mathbf{x}$ of notes, it predicts a probability distribution over possible next notes, i.e: its output gives $\Pr(y|\mathbf{x})$ for all possible next notes y. During training, the correct y^* is known and backpropagation through time is used to maximize $\Pr(y^*|\mathbf{x})$. After training, the LSTM predictions can be used to evaluate the quality of note transitions from one unit to the next: given a sequence of notes $\mathbf{x}$ from previous measures and y the first note of the next unit, the LSTM gives the likelihood of the transition from $\mathbf{x}$ to y. See figure 3 for an illustration.

After both networks have been trained, they can be used to generate new melodies. The first measure(s) can be chosen randomly or by a human user. Thereafter, the most likely unit given the previous measures is chosen by ranking all units firstly according to the similarity of their semantic vector to the previous measure (see equation 1) and secondly (if they are among the most similar units) depending on the likelihood of the note transition. This is computationally demanding, since it requires comparing with every single unit in the database each time a new unit gets concatenated. Appendix D links to a video with melodies composed using the approach.

Data-based evolutionary optimization

The technique used in [2] is very different from that discussed in the previous subsection. The authors apply an evolutionary optimization algorithm to melodies that are encoded as trees. In these trees, leafs correspond to notes and rests, whose lengths depend on their distance to the root (see figure 4 for an example). The fitness of these "tree melodies" is defined by several fitness criteria, which are learned from a dataset of existing melodies.

The optimization starts with an initial set of candidate tree melodies that are generated randomly (since the trees follow a clear syntax expressed as a context-free grammar, this is done by simply randomly evaluating the grammar's rules). The fittest candidates are then crossbred by interchanging tree nodes between two candidates. With each exchange, two new candidates are obtained (see figures 6 and 7). This operation is repeated until the size of the population has doubled. Among the newly generated candidates, the fittest ones are mutated with a small probability, which means some of their subtrees are replaced by randomly generated trees (see figure 8). From the resulting set of candidates, one retains only the fittest such that the original population size is obtained again. The whole procedure is iterated for a number of epochs and the final set of candidates are the composed melodies.

It remains to discuss what the fitness of a candidate is. This should ideally correspond to the musical quality of the candidates as a human perceives them. However, the large population size and number of evolutionary iterations effectively prohibit the use of user interaction for determining fitness. Therefore, fitness must be calculated automatically. To that end, the authors implement 12 different fitness functions, whose parameters are learned from a training set of existing melodies. Here, some of these fitness functions will briefly be summarized.

One fitness function considers a vector representation of tree melodies that is similar to the bag-of-words representation from the last subsection. The vector stores some properties of a melody like the total number of notes, average pitch intervals, average durations, rate of syncopated notes, and more (see [2, p.8]). The fitness of a candidate melody is calculated as the negative squared distance of its vector representation to a centroid of the vector representations of melodies from the training set.

Another group of fitness functions used in [2] considers a statistical language model learned from the training set. Similar to the LSTM from the last subsection, such models consider the probabilities of some symbol occurring after a given sequence of previous symbols. As symbols, the authors don't choose notes (with pitches and durations) but groups of two or three consecutive pitch intervals and inter-onset durations. For example, one symbol might stand for "a perfect

fourth interval of duration $\frac{1}{8}$ followed by a major third of duration $\frac{1}{2}$". Every melody can be decomposed into such symbols. This is done for each melody in the training set. Then, all n-grams (sequences of n symbols, where n is a parameter of the function, e.g. 3 or 4) in the training set are counted to sample their probability. The likelihood of a candidate melody under the language model is the probability of its symbol sequence given the sampled n-gram probabilities. So for a symbol sequence $S = (s_1 \dots s_{|S|})$ of some melody:

$$\Pr(S) = \mathrm{b}(s_1 \dots s_{n-1}) \prod_{i=n}^{|S|} \mathrm{c}(s_{i-n+1} \dots s_{i-1} s_i),$$

where $\mathrm{b}(s_1 \dots s_{n-1})$ is the fraction of times a melody in the training set begins with $s_1 \dots s_{n-1}$, and $\mathrm{c}(s_{i-n+1} \dots s_{i-1} s_i)$ is the fraction of occurrences of n-gram $s_{i-n+1} \dots s_{i-1} s_i$ compared to all other n-grams that begin with $s_{i-n+1} \dots s_{i-1}$. Some additional adjustments are necessary to assign probabilities to n-grams that do not appear in the training set and to account for melodies of different length (since under the formula above, longer melodies necessarily have lower likelihood), see [2] for details.

Some of the fitness functions chosen by the authors also take into account music theoretic analysis. For instance - given an algorithm that counts musical segments in melodies - the average number of segments in the training set is used as the parameter λ of a Poisson distribution. The distribution then gives the fitness of a candidate melody depending on the number of its segments. Another fitness function takes into account harmonic categories of notes.

All these individual fitness functions are different objectives for the evolutionary optimization explained in the beginning. In order to determine the overally fittest among the candidates, the *Non-dominated Sorting Genetic Algorithm II* is used. This algorithm first lists all candidates that are *non-dominated* (also called *pareto-optimal*), where a candidate is *dominated*, if there exists a different candidate that has better or equal fitness values for all individual functions and strictly better fitness for at least one of them. Among the non-dominated candidates, it prefers those from low-density regions in the space defined by the non-dominated candidates (i.e: it prefers candidates that are as far apart as possible from the rest in terms of their fitness function values). See figure 9 for an illustration. If the number of non-dominated candidates is not sufficient to reach the required population size, NSGA-II proceeds iteratively by selecting candidates that are non-dominated once the previous non-dominated candidates have been removed etc.

The authors use an initial population size of 100 candidates and run the evolution for 1000 epochs. Appendix E links to melodies composed using the approach, including sound examples.

Conclusion

As machine learning methods become more powerful, they will be increasingly useful for music generation. The techniques presented in this report - unit selection using neural networks and tree melodies obtained by evolution - are capable of learning characteristics of given music and creating new pieces. However, their subjective quality varies.

A major obstacle is the lack of clearly defined quality measures for algorithmic composition approaches. The authors in [2] only evaluate their system qualitatively for different choices of fitness criteria. The authors in [1] also evaluate their system subjectively and additionally show that it is likely to choose the correct measures when tasked to reconstruct a melody beginning from a held out test set. However, since the technique in [2] does not use pre-made measures and cannot take into account a given beginning of a melody, it cannot be evaluated this way. It is therefore difficult to objectively compare the effectiveness of the two approaches.

Finally, as mentioned in the beginning, melody generation is only one component of successful music generation. The machine learning methods presented here might be extended to learn harmonization, instrumentation etc. from given music to generate something new.

Appendix

References

[1] Mason Bretan, Gil Weinberg, and Larry Heck. A Unit Selection Methodology for Music Generation Using Deep Neural Networks. arXiv:1612.03789 [cs.SD], 2016.

[2] Pedro J. Ponce de León, José M. Iesta, Jorge Calvo-Zaragoza, and David Rizo. Data-based melody generation through multi-objective evolutionary computation. *Journal of Mathematics and Music*, 10(2):173–192, 2016.

[3] Jose David Fernández and Francisco Vico. AI Methods in Algorithmic Composition: A Comprehensive Survey. *Journal Of Artificial Intelligence Research*, 48:513–582, 2013.

[4] Gerhard Nierhaus. *Algorithmic composition: paradigms of automated music generation.* Springer Science & Business Media, 2009.

A History: Supplementary material

The original publication of Mozart's dice game can be found on IMSLP. An implementation that runs in a web browser can be found here.

David Cope's website includes additional information on EMI as well as audio samples of EMI's compositions. For instance, one can listen to this piece composed by EMI in a style very similar Johann Sebastian Bach's two part inventions for key instruments. As a reference, compare it to this original piece by Bach.

Cope's YouTube channel contains more music composed by EMI and Emily Howell (an improved version of EMI).

B Formal grammar example

Here is a very small definite clause grammar written in Prolog that encodes melodies which consist of 16 quarter notes and begin and end with a C. A cadence is encoded by choosing notes from the tonic (C-Major), then the subdominant, then the dominant and finally the tonic again.

```
melody --> tonic1,subdominant,
    dominant,tonic2.
tonic1 --> [c1],t,t,t.
tonic2 --> t,t,t,[c1].
subdominant --> s,s,s,s.
dominant --> d,d,d,d.
t --> [c1].
t --> [d1].
t --> [e1].
t --> [f1].
t --> [g1].
t --> [c2].
s --> [f1].
s --> [g1].
s --> [a1].
s --> [b1].
s --> [c2].
s --> [f2].
d --> [g1].
d --> [a1].
d --> [b1].
d --> [c2].
d --> [d2].
d --> [g2].
```

A Prolog interpreter like SWI-Prolog can be used to enumerate all possible melodies that are obtained using this grammar. In particular, the query

```
?- phrase(melody,[c1,e1,g1
   ,e1,f1,g1,a1,g1,b1,a1,
   g1,d2,c2,g1,e1,c1]).
```

returns "true". Therefore, the following melody (among many others) is generated by the grammar:

C Markov model example

The Markov chain in figure 1 generates a rhythm as a sequence of note durations.

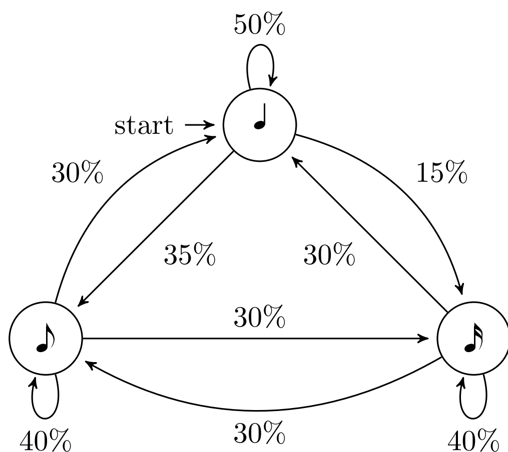

Figure 1: A simple Markov chain for rhythm generation. States correspond to note durations. The edge descriptions define for each state a probability distribution over all successive states. Therefore, if the last note was an eighth note, with probability 30% the next note will be a quarter note or with probability 40% another eighth note etc.

For example the rhythm

has likelihood

$$0.35 \cdot 0.3 \cdot 0.4 \cdot 0.3 \cdot 0.35 \cdot 0.4 \approx 0.0018$$

under this Markov chain.

D Unit selection using deep learning: Supplementary material

The authors of [1] have created a video showcasing some music composed using their system (from 4:20 onwards). The video showcases four different configurations of the system: The unit length (number of measures per unit) is a parameter and can be set to 1, 2 or 4 measures. Additionally, the authors test a system that consists only of an LSTM and does not use unit selection. Instead, that system only outputs the most probable note given the previous sequence of notes.

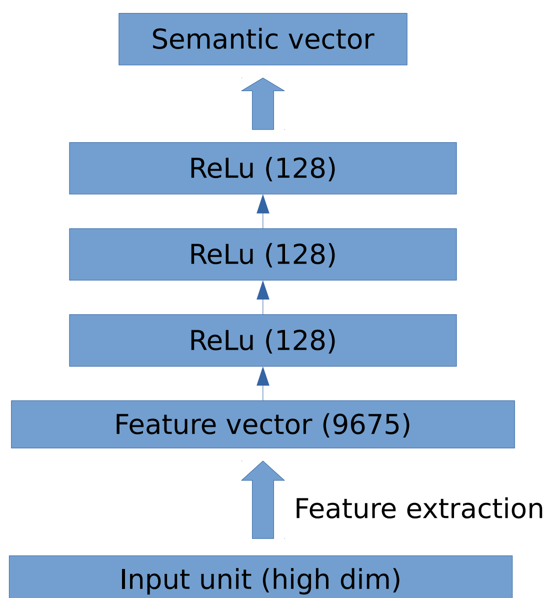

Figure 2: The DSSM network architecture for learning semantic similarity of units.
The input unit (consisting of measures of a melody) is first transformed into a bag-of-words feature vector and then processed through several hidden layers with leaky rectified linear units (ReLu, $\alpha = 0.001$). The output of the upper ReLu layer is the semantic vector for the input unit.
The numbers in parentheses are the number of neurons per layer. For regularization, the network is trained using 0.5 dropout (in each training iteration, half the neurons are deactivated). The learning rate used is 0.005.

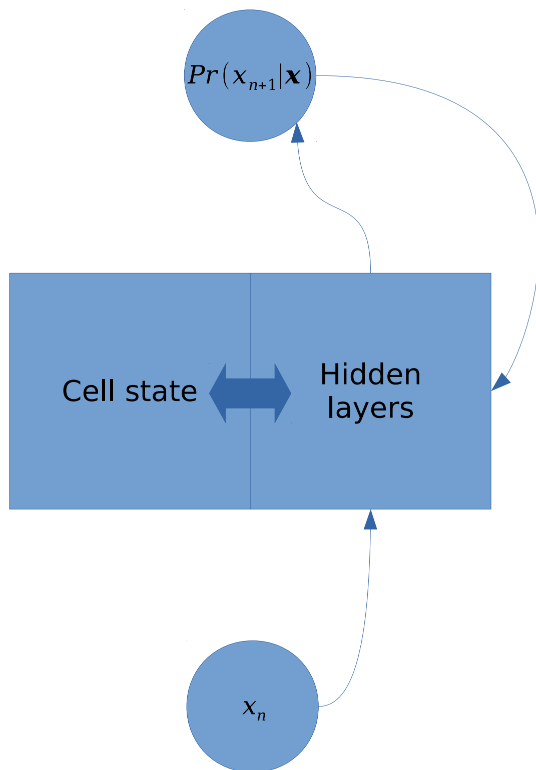

Figure 3: An overview of the LSTM that learns note transitions.
The network receives a sequence of notes $\mathbf{x} = (x_1, \ldots, x_m)$ as input, one by one. On input of the n-th note x_n, the output of the network is a prediction of the $(n+1)$th note, i.e.: $\Pr(x_{n+1}|\mathbf{x})$ for all possible next notes x_{n+1}. This prediction is looped back into the LSTM when the actual next note x^*_{n+1} is received.
Over the course of the whole input sequence, the LSTM also maintains a cell state, which stores information on the sequence and can be updated depending on the current input and previous prediction. Furthermore, the system used in the paper is multi-layered, so the output also becomes input to another LSTM which is again stacked under another LSTM etc.
The authors choose the length of note sequences $\mathbf{x}$ ad-hoc to be 36 notes. They do not elaborate on the precise layout of their LSTM (choice of hidden layers and number of stacked LSTMs).

E Data-based evolutionary optimization: Supplementary material

There exists an online supplement for [2]. This contains melodies in different styles composed by their evolutionary technique. I have arranged some of these melodies in MuseScore here, so they can be heard.

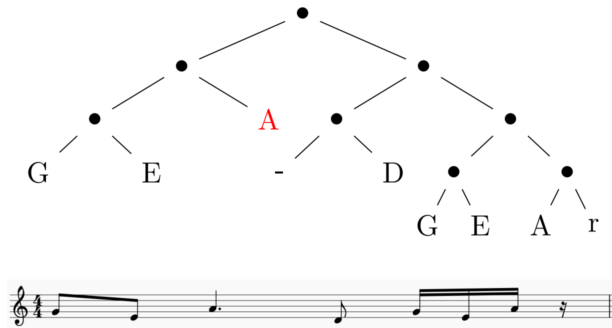

Figure 4: A melody as a tree.

The tree representation works for melodies in even rhythm (e.g $\frac{4}{4}$) and encodes note durations using a hierarchy. Each measure is represented by a subtree connected to the root (here: only one measure).

A leaf corresponds to a note (G, E, A, . . .) or rest (“r”) and an internal node (“•”) divides the hierarchy into parts of equal length. A leaf on the first level of the subtree therefore corresponds to a whole note, on the second level to a half note, on the third to a quarter note etc. A leaf can also contain a continuation symbol (“-”), to allow note durations of a length that is not an inverse power of two (see the third note in the example). This is used for dotted notes or tie-overs.

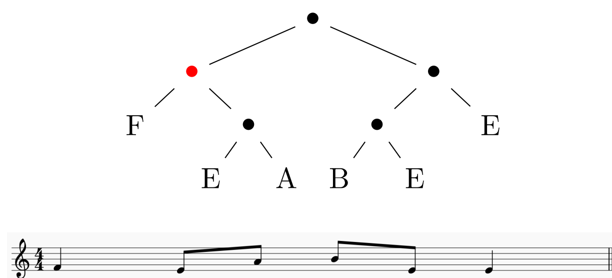

Figure 5: Another measure represented as a tree.

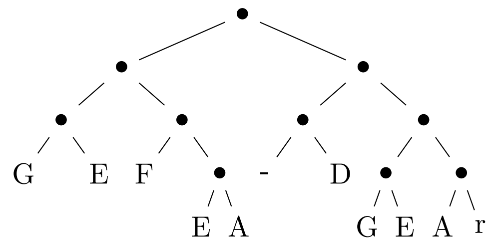

Figure 6: Crossover operation: A new measure is obtained by substituting the red node from figure 5 into the red node from figure 4. Notice the change of note durations.

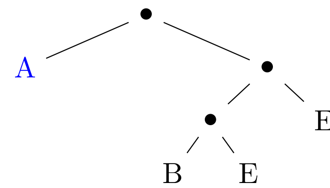

Figure 7: Crossover operation: Another new measure is obtained by substituting the red node from figure 4 into the red node from figure 5.

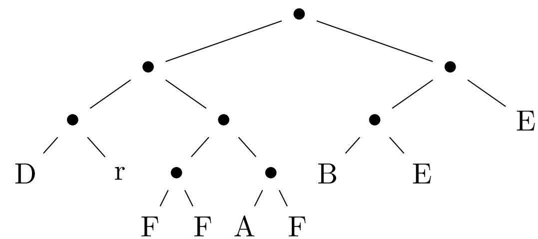

Figure 8: Mutation operation: The measure from figure 7 is mutated by randomly regenerating the blue node.

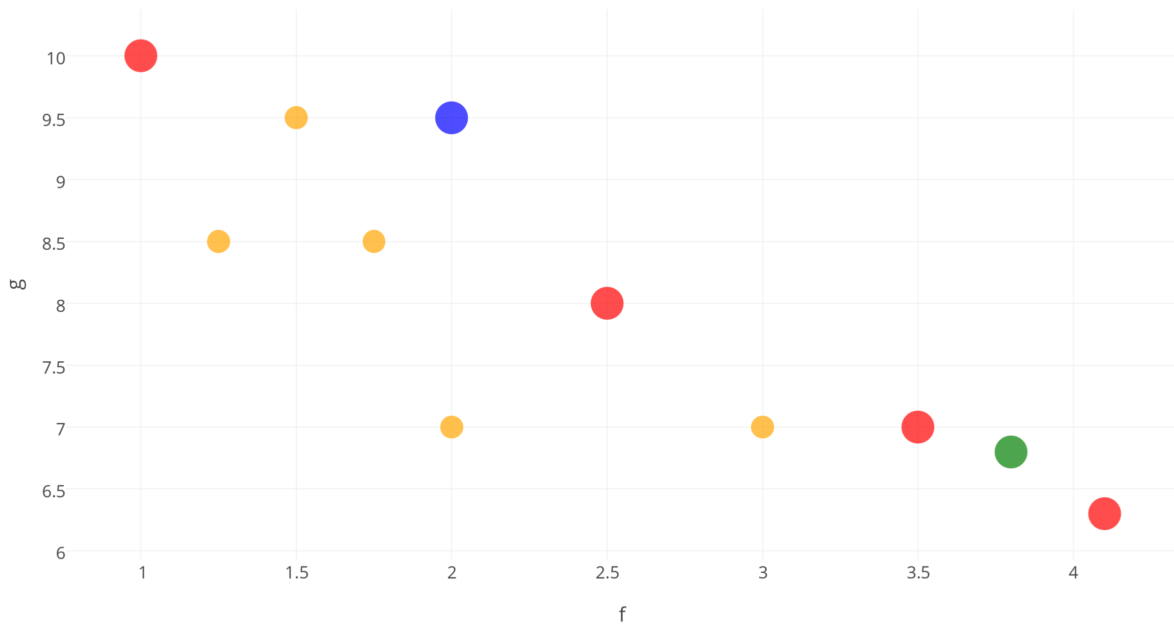

Figure 9: Illustration of the NSGA-II algorithm for two fitness functions, g and f.
The points represent candidates with their respective fitness values. Candidates in red, blue and green colors are non-dominated (cf. page), whereas candidates in orange are dominated. The blue candidate is preferred over the green one, as it has a higher distance to its closest non-dominated neighbors (in other words, the blue candidate comes from a lower density region of non-dominated candidates).
Although the illustration shows only a two-dimensional space defined by two fitness functions, the same concepts apply to higher dimensional spaces for multiple fitness functions.