Contents

Chapter 1

Introduction

1.1 Why use algorithms?

The human life today certainly has an accelerated change. If one were to compare the type and quality of our lives even with those from some ten years ago would find many deviations. Although looking back a decade later to the current years would probably feel the same. The majority agrees that changes are mostly due to technical and scientific improvements which usually come hand in hand. While the source of said developments is generally common the application and utilization of them can happen basically everywhere. No matter what field we are talking about there is always need for new ideas and to make existing solutions smooth and automatic even more adjusting to current standards. Indeed this is the reason why there are so many algorithms out there manipulating things which seem so simple and straightforward at first look that we tend to take it for granted. In reality these are presumably complicated tasks needed to be optimized in a sophisticated way.

We live in a world full of online data storage places. Without letting our imagination narrowed down we could think many different examples. Among the widely recognized sites, there are GitHub helping programmers to share their projects, Stack Overflow to help solving recurring problems during coding, scientific journals providing information about the current state of research and Amazon or EBay to make physical items available for purchase worldwide. They are all just a different manifestation of an online place storing specific type of data. It is necessary for all of them to keep up with the improvements and introduce algorithms in their workflows in order to maintain their status on the market against similar sites. For instance Stack Overflow needs to have a smart searching algorithm in order to offer solutions to related problems and scientific journals have definitely a difficult job to find connections in a large pool of articles. This study however will focus on the likes of Amazon and EBay and the algorithms that can be utilized there. These sites are generally called online marketplaces.

Predicting movie scores is just as important as speech and sign recognition. Well one could definitely argue with that. However in the sense that in both cases an algorithm operates behind the scenes indeed they can be considered equally interesting to a 21st-century scientist. Naturally some applications are more valuable than others in terms of how much they contribute and are driven by a certain good cause. Although having a good cause could be very motivating, defining it explicitly comes with difficulties in many cases. However the good cause of online marketplace platforms is something we can define right away. In short they tear down geographical obstacles making any kind of physical or intellectual data, service and information available for the whole world, while forming and squeezing them into a shape where they can be most easily accessed and utilized.

1.2 Online marketplaces

As mentioned earlier from now on we would like to focus on a specific type of online data storage places. These are the so-called virtual marketplaces that have extended the role of common markets in the past few years. The simplest description of these kind of sites would be that they bring together buyers and sellers while controlling some activities between them. More precisely consumer transactions are processed by the marketplace itself and delivered to the participating retailers or wholesalers (generally called sellers) which then fulfill them. Marketplaces aggregate products from a wide range of providers thus selection is usually more versatile compared to a vendor-specific online store, which is very similar to how markets and supermarkets operate compared to individual sellers. Their online version however can have several additional advantage apart from offering a larger variety of available products. These extra capabilities include reverse or forward auctioning, handling the ordering and delivery process and placing vendor specific advertisements. All these factors contribute to the leading role and competitiveness of online markets in our world.

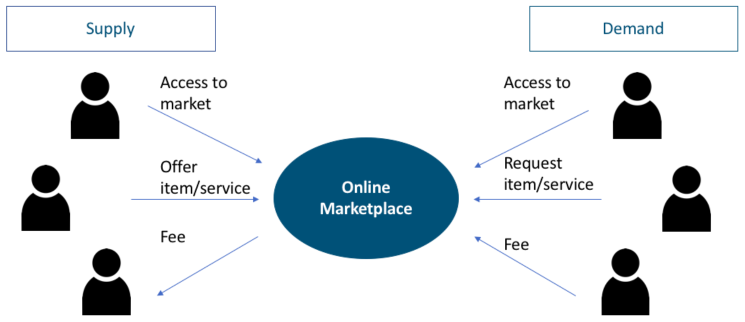


Figure 1.1: The mechanism of a peer-to-peer online marketplace (source: medium.com)

As seen from figure 1.1 the three parties (marketplace, buyer and seller) are brought together by the marketplace itself. Demand comes from the end of the buyer and it will be met by one of the available sellers in exchange for a fee. This fee is paid by the requester and then traverses through the marketplace reaching eventually the provider. For all this to happen the marketplace needs to host the two other participant and has to make sure the workflow is smooth enough. It includes adjustments to both directions keeping them satisfied by taking on their jobs in terms of procurement and sales. Hence an ideal version of a marketplace consists of optimized solutions towards both customer and supplier maintaining its competitiveness against other marketplaces. A very simple optimization would be to choose seller (i.e. product or source) for all requested items in a smart way corresponding to the expectations of the buyer. Usually the most sensitive detail for a customer is the total cost that has to be paid to initialize the transaction. Moreover this could be the decisive feature during source selection as well. Simplifying the task the marketplace now has to just select the cheapest seller in case of each requested item resulting in the lowest possible total cost the buyer needs pay at the end. It all seems very simple. What is even here to optimize? Why would we need an algorithm for this kind of work?

Turns out there is a certain type of online marketplace where problems are even more interesting and can be related to many known phenomena in the field of physics. Finding a solution could be an interesting task for a modern day physicist. In the following chapter we will get to know these problems in more detail.

1.3 An interesting problem

Let us image a marketplace where the offered products are more or less equally valuable. Moreover let us also assume that these products are required and ordered in bulk quantities, i.e. single customer demand usually consists of many items. Provided that sellers are very much aware of the exact needs of buyers they tend to manipulate their pricing schemes to account for this behaviour and attract as many customers as possible. Having acknowledged the nature of this particular field of business a natural reaction from suppliers is to incorporate quantity discounts and introduce per item prices as a function of the total number of products ordered from them. This feature can be manifested in a form seen in table 1.1. Taking into account quantity discounts we get a so-called price table which we will refer to many times in the following chapters.

Quantity	Price (USD)
1	10
2	7
5	4
10	1

Table 1.1: A simple price table

A price table shows us how much an item will cost depending on how many products we would like to order from the corresponding seller. May it concern only a particular item or applies to each one of them, it does not really matter at this point. What matters is that in general the more we order from one seller the lower price we have to pay for its products. Now keeping this in mind let us rephrase the question from the previous section. How to choose seller/product for each item in order to reach the lowest possible transaction fee? Well, it is not that obvious anymore. What were considered previously as independent product selections have just become an entangled system of choices. By choosing the cheapest product for every item we neglect the possibility for price reduction enabled by quantity discounts. Better to say our optimization procedure does not really utilize said opportunity. We could of course end up with a product selection which will be suitable to exploit some quantity discounts, however the main direction of the optimization is different this way.

1.4 Real world examples

We present here two distinct applications of quantity discounts appearing on an online marketplace. Most common feature of those sites is that they offer almost equally valuable items which are generally purchased in bulk quantities. Under these circumstances it follows naturally to have quantity discounts introduced by the sellers.

A very good example of a purchase site of this kind is *Bricklink*[1], an online marketplace of Lego blocks. It clearly affirms our assumption. Before finalizing the order a certain Lego element is usually just as useful as others. However ending up in the hands of need obviously comes with an increase of value. Moreover acquiring many Lego items at the same time could as well be the reason behind using a site of this sort. Available optimization algorithms include *Wanted-List-Auto-Finder* and *Easy buy*[2] which are simultaneously used indirectly by customers to provide guidance during seller selection.

A more challenging example for a scientist could be found in the field of early-phase drug discovery. Prior to carrying out tests in the laboratories of pharmaceutical research companies it is a standard procedure to run computer simulations narrowing down the pool of drug candidate small molecules. By doing so a relatively small group of chemical compounds are advanced to be tested on cells, microorganisms and later on living biological entities. The preselection is done via molecular mechanical calculations. When the targeted disease is associated with a protein thousands of small molecules are being tested whether they can attach to the macromolecule modifying its functionality. While it is inevitable to run in-silico calculations to filter down the chemical space, very often a large number of molecules can yield promising results at the end. Thus further laboratory analysis will be carried out for all those who survived the prefiltering phase.

Mcule[3] is an online drug discovery platform combined with a webshop of molecules. On one hand one can obtain the most promising drug candidates by filtering through the chemical space using molecular modelling tools. On the other hand the most interesting structures can be ordered immediately since multiple sellers have uploaded their databases here providing the necessary chemical diversity. Customer requests are therefore consist of bulk quantities. More precisely many compounds needed at the same time allow-

[1]Bricklink website
[2]Guide for using algorithms on Bricklink
[3]Mcule webstite

ing for the utilization of quantity discounts. What was true for Lego blocks also stands in the world of molecules, i.e. seller selection is not as trivial as it seems at first glance. *Mcule* provides its so-called *Instant Quote*[4] feature which uses a built-in algorithm to find the best combination of sellers for each requested compound in terms of price or delivery time as seen in figure 1.2.

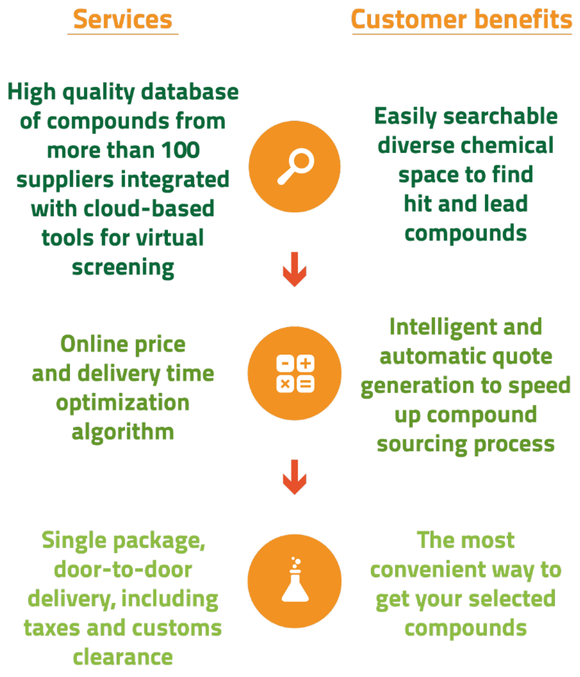

Services
Customer benefits
High quality database of compounds from more than 100 suppliers integrated with cloud-based tools for virtual screening
Easily searchable diverse chemical space to find hit and lead compounds
Online price and delivery time optimization algorithm
Intelligent and automatic quote generation to speed up compound sourcing process
Single package, door-to-door delivery, including taxes and customs clearance
The most convenient way to get your selected compounds

Figure 1.2: Mcule's model: combination of molecular modelling and compound sourcing (source: mcule.com)

We have seen so far a couple of real-world examples on the usage of quantity discounts. These sort of price reductions have been introduced by the sellers which are themselves embedded in an online marketplace. Therefore the task to find the most adequate combination of providers often requires a sophisticated algorithm. In the next chapters we will try to approach these problems in a more scientific way and put them into a mathematical formulation.

[4]Mcule's Instant Quote feature

Chapter 2

Problem description

In this chapter we would like to continue the investigation in more depth. We will present here a more accurate description of the problem in question focusing on general patterns we can find in other optimization tasks. Realizing the common features we can have a clear view on possible algorithms that proved to be useful in solving similar problems. From here we can start working up our way towards a problem-specific solution.

2.1 Mathematical formulation

Let us consider a collection of items that we will be working with. Say $N_{\mathrm{P}} \in \mathbb{N}$ places or positions need to be filled and for each spot we can choose from a large pool (denoted by I) containing $N_{\mathrm{I}} \in \mathbb{N}$ items in total. It has to be emphasized in the very beginning that while available items for each position construct a disjoint set (i.e. the collection I can be divided into N_{P} non-overlapping parts spanning the representation I_{P}), it will not necessarily result in N_{P} independent choices if there is a final goal to be achieved during the selection process. In fact there is a value ($V_k \in \mathbb{R}$) associated with each possible selection we can end up with which provides the base for the optimization procedure. To build up some connection with the concepts introduced in the previous chapter we ought to interpret the value as price. Consequently our goal will be to find or at least approximate the minimum of V_k. We also need to mention that each item has a source (earlier called seller) associated with it, more precisely there are N_{S} sources from where we can select items. We can therefore group the items according to their sources building up the source-based representation I_S as opposed to the position-based representation I_P. Now let us also represent a certain selection of items with $S_k \in \mathbb{N}^{N_p}$ belonging to the k-th combination. Moreover the length of each disjoint set I_p is indicated as n_p. Using the notations introduced earlier we can easily write down some useful relations. From the non-overlapping property of the division of the whole set of items we have

$$|I| = N_\mathrm{I} = \sum_{p=1}^{N_\mathrm{P}} n_p, \tag{2.1}$$

while the number of possible combinations is

$$K = \prod_{p=1}^{N_\mathrm{P}} n_p, \tag{2.2}$$

where the size of a disjoint set can be written as

$$n_p = |I_p| \; \forall \, I_p \in I_\mathrm{P}, \tag{2.3}$$

since the total set of items is the union of all non-overlapping subsets

$$I = \cup \, I_p. \tag{2.4}$$

The item selection corresponding to combination k is a set of positive integer numbers with restricted domain ranges in each position, i.e.

$$\begin{aligned} S_k &= \{k_1, k_2, ..., k_{N_\mathrm{P}}\} \\ k_p &\in \{1, 2, ..., n_p\} \; \forall \, p \in \{1, 2, ..., N_\mathrm{P}\}. \end{aligned} \tag{2.5}$$

The value associated with this selection reads

$$V_k = V(S_k) = V(k_1, k_2, ..., k_{N_\mathrm{P}}), \tag{2.6}$$

which is in fact a discrete function of N_P variables:

$$V : \mathbb{N}^{N_\mathrm{P}} \mapsto \mathbb{R}. \tag{2.7}$$

Due to the constraints on each domain range we end up with a discrete optimization problem. A naive approach to approximate these kind of problems is to apply a so-called relaxation. By doing so the integrality constraint of an integer program is removed allowing the variables to take value from a continuous range. This way we can gain information about the original integer program, although it is not completely straightforward how to utilize them to solve the initial problem [1].

Consider our value introduced in Eq. (2.6). Let us convert it to a continuous function of N_P variables and forget the integrality constraints for now:

$$V : \mathbb{R}^{N_\mathrm{P}} \mapsto \mathbb{R}. \tag{2.8}$$

Moreover we introduce all our additional restrictions as inequalities and put them in the form of Lagrange multipliers. Let us also reformulate the selection associated with the k-th combination as an N_P dimensional vector:

$$\mathbf{S_k} = \begin{bmatrix} k_1 \\ k_2 \\ \vdots \\ k_{N_\mathrm{P}} \end{bmatrix}.$$

We introduce another N_P dimensional vector accounting for the limited domain ranges:

$$\mathbf{L} = \begin{bmatrix} n_1 \\ n_2 \\ \vdots \\ n_{N_\mathrm{P}} \end{bmatrix}.$$

Then the constraints can easily be written as $\mathbf{C_k} = \mathbf{L} - \mathbf{S_k}$, thus we can introduce our optimization task as the following:

$$\min(V(k_1, k_2, ..., k_{N_\mathrm{P}}) + \lambda^\mathbf{T} \cdot \mathbf{C_k}), \tag{2.9}$$

where λ is non-positive vector to ensure that the optimal solution of the relaxed problem will not be higher than the optimal solution of the original problem upon minimization. This method is called Lagrangian relaxation [2].

Although we have reached an intriguing formulation of our problem with some restrictions embedded into a Lagrange multiplier, there is still some work to do. We still need to figure out a way to formulate V as a proper function. If one were to do that a variety of optimization techniques will become applicable. Among others we could exploit the large pool of gradient type methods which are effective for finding extrema of continous and differentiable functions.

Splitting V to components corresponding to N_{P} positions we have:

$$V(k_1, k_2, ..., k_{N_{\mathrm{P}}}) \;=\; v_1(k_1) + v_2(k_2) + \cdots + v_{N_{\mathrm{P}}}(k_{N_{\mathrm{P}}}). \tag{2.10}$$

Unfortunately in this basis the individual v_p terms are not proper functions, at most they could be interpreted as mappings between k_p and their associated value v_p. It is because the values do not depend continuously on which item we select for a particular position. Meaning that the awaited improvements by linear programming (LP) relaxation cannot be achieved in this representation. Furthermore the representation in Eq. (2.10) is something we cannot really apply due to the entanglements of choices. What is definitely feasible though is to express V in the following way:

$$V \;=\; v_1(k_1, k_2, \ldots, k_{N_{\mathrm{P}}}) + v_2(k_1, k_2, \ldots, k_{N_{\mathrm{P}}}) + \cdots + v_{N_{\mathrm{P}}}(k_1, k_2, \ldots, k_{N_{\mathrm{P}}}), \tag{2.11}$$

although it does not yield any progress for us. We will account for this formula in the next section.

Since our position-based representation I_{P} implied that V could not be expressed as a proper function preventing us to use LP relaxation and analytical optimization methods, the need for a different type of item classification has increased. Let us reconsider therefore the grouping of items I by introducing N_{S} new subsets and the source-based representation I_{S} replacing the position-based representation I_{P}. Both composition is the set of all subsets which covers I, thus we have

$$I_{\mathrm{P}} \;=\; \{I_1, I_2, \ldots, I_{N_{\mathrm{P}}}\} \quad\Longleftrightarrow\quad I_{\mathrm{S}} \;=\; \{I_1, I_2, \ldots, I_{N_{\mathrm{S}}}\}, \tag{2.12}$$

$$I = \cup\, I_p \;\; \forall I_p \in I_{\mathrm{P}} \quad\Longleftrightarrow\quad I = \cup\, I_s \;\; \forall I_s \in I_{\mathrm{S}}.$$

We can see the difference of I_{S} and I_{P} on Fig. 2.1, with the subsets being:

$$I_{p1} = \{\text{Item 1, Item 2, Item 3}\} \qquad I_{p2} = \{\text{Item 4, Item 5, Item 6}\},$$

$$I_{s1} = \{\text{Item 1, Item 4}\} \quad I_{s2} = \{\text{Item 2, Item 5}\} \quad I_{s3} = \{\text{Item 3, Item 6}\}.$$

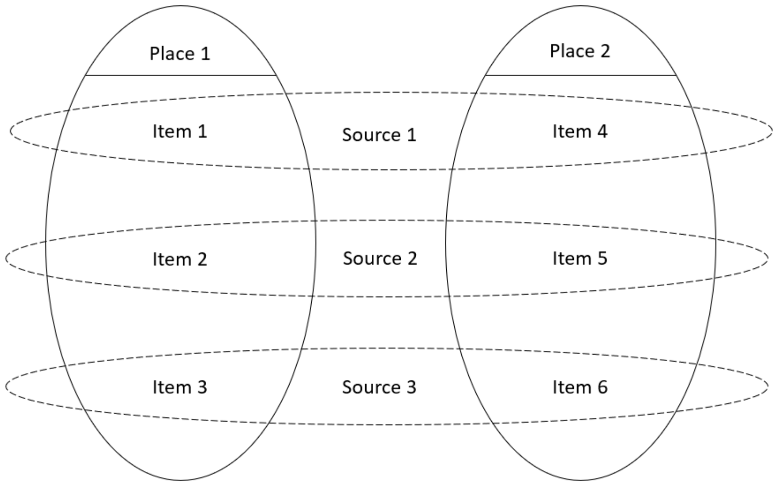


Figure 2.1: Difference between position-based and source-based representation

These subsets will provide the value function V to be separable analogously to Eq. (2.10) with N_S replacing N_P. Similarly transforms the dimension of S_k and the meaning of k_s and n_s (previously k_p and n_p) while both maintaining their role. $k_s \in \{1, 2, \ldots, n_p\}$ now represents the number of elements to be selected from $I_s \subseteq I$, and not the index of a single item as before. This is in fact the reason why we can eliminate the issue related to our value function V. Even though the source-based v_s values will not be a continuous function of k_s at first, it still linearly depends on the value of k_s since the more item we choose from a source the higher price we need to pay for them. Thus we can easily do some manipulation to achieve continuity. However there is a restriction which is yet to be accounted for. Namely the original grouping has been constructed in a way where only a single item was required to select from each subset. Changing the group structure does not modify this requirement. That said we need still need to somehow keep account of the original subsets. The associated logic can clearly be embedded into the Lagrange multiplier.

We can now write an optimization objective analog to Eq. (2.9) with N_S replacing N_P in the formulas. Additionally the Lagrange multiplier has to be extended to contain new restrictions. The only problem is that we work on a different basis than needed to express the additional constraints. Let us generalize S_k from Eq. (2.5) to be a subset of items which fulfils the criteria of a selection, that is:

$$|S_k \cap I_p| = 1 \quad \forall\, I_p \in I_{\mathrm{P}}. \tag{2.13}$$

It is hardly convenient to keep track of all I_p subsets while working on the basis of I_{S}. We could definitely squeeze these into the Lagrange multiplier, however it would enormously increase its dimension. Not to mention that we would like to be solving problems for very large N_{I}, N_{P} and N_{S} values. The number of places in a selection could be in the range of $10^3 - 10^4$, while the number of sources varies around $10^2 - 10^3$. In the case when all source can provide an item for each position, we could end up with 10^7 items altogether. Eventually the problem size taking into account each constraint would exceed the limitations of analytical methods. That said we will turn our focus into the direction of discrete optimization. In the next section we will spend some time motivating ourselves why choose source-based I_{S} representation as the basis for combinatorial optimizations.

2.2 Optimization with source-based representation

Previously we have come to the conclusion that neither the position-based I_{P} nor the source-based I_s representation is of use when one wants to apply continuous optimization on a problem of this type. I_{P} failed due to the value V, which is in fact not a proper function. Later with I_{S} we had dimensionality problems, nevertheless V was said to be modifiable to become a continuous function and on the top of that we were able to separate it into N_{S} independent terms similarly as in Eq. (2.10):

$$V(k_1, k_2, ..., k_{N_{\mathrm{S}}}) \;=\; v_1(k_1) + v_2(k_2) + \cdots + v_{N_{\mathrm{S}}}(k_{N_{\mathrm{S}}}). \tag{2.14}$$

It is perhaps time to explain why the value function once looks like this and another time reads as in Eq. (2.11). Having already introduced the concept of quantity discounts earlier we can now utilize its effect on the form of the value function. Since values (prices) are determined by sources (sellers) the quantity discounts can only be accounted for when we group the items according to I_{S}. This way each source has an associated value function which depends only on the number of items in the selection belonging to them. Working in the basis of I_{P} we are not aware how many items we have selected from one particular source, therefore we cannot really know its corresponding value. In fact if it weren't for quantity discounts it wouldn't be necessary to bother with optimization at all. We would just choose sources with the lowest price for each N_{P} positions. This temptation has already been warded off though in the previous chapter.

Before moving on we will show two important features of representation I_{S}. First of all while the value function could be split according to Eq. (2.14), the optimization procedure cannot be carried out in N_{S} parts independently. A decision for the number of items coming from source s clearly affects other subsets as well since it will block k_s original (i.e. position-based) groups and their included items no matter which source they are associated with. Another interesting property is again generated by quantity discounts. Namely one is only concerned with the boundaries of value jumps and not with the actual values since they remain constant between jumps. Introducing a normalization on the values we have:

$$v_s := \frac{v_s}{n_s}, \tag{2.15}$$

where v_s and n_s are the original value and item count belonging to source s. Now we can clearly see (Fig. 2.2) v_s has a step function structure if we plot the data corresponding to Table 1.1.

Based on these characteristics a working solution could be to cleverly investigate combinations of value jumps. Actually the objects (s,j) constructed as the pair of source s and value jump j (a single quantity number to be reached for value reduction) will act as bases for the optimization. For instance if Table 1.1 corresponds to source s we can say that four pairs exist, i.e. (s,1), (s,2), (s,5), (s,10).

It is very important to emphasize that while we look for the minimum of the value function there is actually another very severe condition regarding the time allowed for the searching algorithm to run. In fact our task could be transformed to finding the minimum in a way where we are continuously getting closer to the objective. Aiming for this will ensure the algorithm to visit the most promising states under finite time t no matter how large or small it might be.

Having decided that the optimization will be carried out with respect to combinations of value jumps only an intelligent method is needed to sample

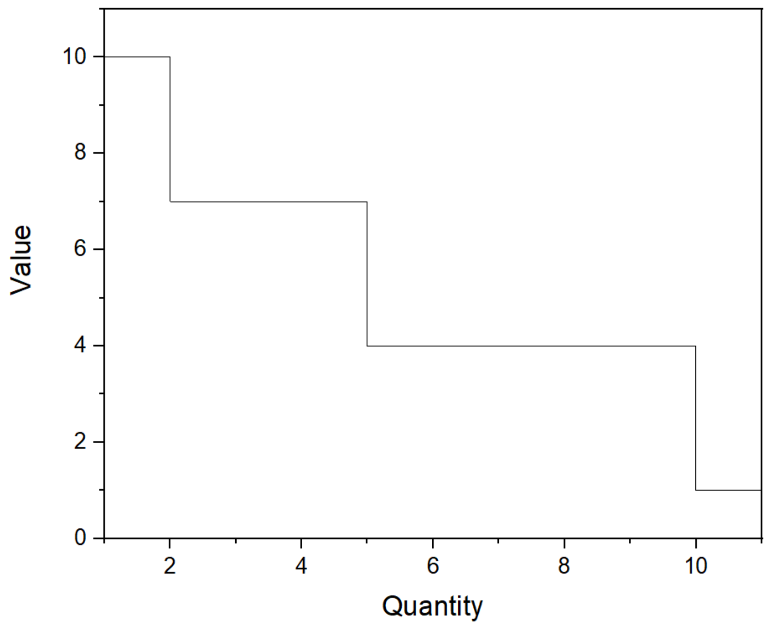


Figure 2.2: Value decrease by quantity discounts

from the possible states and yield the best combinations within given time. The k-th combination or state will be denoted by c_k and reads:

$$c_k = \{(s, j)_1, (s, j)_2, \ldots\}, \tag{2.16}$$

which is in fact a set of several source-jump pairs. We will not give an explanation for the undefined length now, let us just accept that each combination could consist of different number of pairs. One may observe that this is somehow contradicts to the condition introduced in Eq. (2.13), but things will be clarified later on. We can also draw parallels between c_k combination of source-jump pairs and S_k selection of items, although they are not identical.

In the next chapter we will draw similarities with other commonly known discrete optimization tasks from where we could borrow ideas how to solve these kind of problems. We will also become familiar with some well-known approaches in the field of combinatorial optimization such as simulated annealing and Monte-Carlo tree search.

Chapter 3

Auxiliary optimization techniques

In this chapter we will reveal how some existing machinery might help us along the way with our optimization task. More precisely two well-known methods will be presented here in order to get a grip on how these techniques could be advantageous to use and comprehend the basic concepts of their applicability. Although let us first get familiar with a very essential problem which is commonly known among physicist and mathematicians.

3.1 Travelling salesman problem

The classical problem is defined as: given a list of cities to be visited and the distances between them, what is the shortest route to visit all cities and return to the starting point? It is an NP-hard problem in combinatorial optimization, and often considered as a benchmark for other similar problems. There are a large number of heuristics and exact algorithms that can be used as a starting point when one deals with a comparable optimization task.

In terms of mathematical formulation we have C set of cities:

$$C = \{c_0, c_1, \ldots, c_n\}, \tag{3.1}$$

and our job is to visit all cities starting and ending the tour at c_0. According to the commonly known formulation of the problem as a linear integer program [3], let us denote the distances between cities i and j as d_{ij} and introduce

$$x_{ij} = \begin{cases} 1 & \text{the tour includes going from city } i \text{ to city } j \\ 0 & \text{otherwise} \end{cases}, \tag{3.2}$$

thus we can write the minimization objective:

$$\min \sum_{i=1}^{n} \sum_{j \neq i, j=1}^{n} x_{ij} d_{ij}. \tag{3.3}$$

The travelling salesman problem is a touchstone for many heuristical approach including genetic type algorithms and simulated annealing. We will continue by familiarizing ourselves with the details of how and why simulated annealing could work for such optimization problems [4].

3.2 Simulated annealing (SA)

Simulated annealing is a probabilistic approach to approximate the global optimum of a given function. It is especially useful when one has to deal with a large discrete search space reducing the applicability of exact optimization algorithms. For problems where finding an approximate global optimum is more important than finding a precise local optimum in a fixed amount of time, simulated annealing may be preferable to alternatives such as gradient descent. It belongs to the class of metaheuristics whose main objective is to find or generate a partial search algorithm (heuristic) that may provide sufficiently good solution to an optimization problem [5]. Even though these type of algorithms do not guarantee that a globally optimal solution can be found on some class of problems, they are very useful when there is a limited computational capacity. Metaheuristics take sample from a set of solutions which is too large to be completely sampled.

Annealing refers to the original technique in metallurgy or material science where the goal is to perfect the formation of a crystal [6]. It includes reaching for a larger size and reducing defects which can occur inside the crystal. Both properties of the material depend on its thermodynamic free energy which in fact is affected by heating and cooling. Thus annealing involves heating a material above its recrystallization temperature, maintaining a suitable temperature for sufficient amount of time, then it is followed by a controlled cooling schedule. We understand now the name "simulated annealing" since the whole idea has been inspired by the original physical problem. In this simulated version the usual task is to find the state s minimizing a given function $E(s)$ which is analogous to the internal energy of the crystal. Starting from an arbitrary initial state the goal is to reach the state with the lowest possible energy. In the followings let us walk through on the steps taken by the algorithm.

The algorithm

Iteration At each step the simulated algorithm heuristic considers a neighbouring state s^* and decides between moving there or staying at the current state s. The decision is based on probabilities which is a function of the so-called energies $E(s)$ and $E(s^*)$. These probabilities ultimately lead the system to move to states of lower energy. Typically this step is repeated until the system reaches a state that is good enough for the application, or until a given computation budget has been exhausted.

Neighbouring states Neighbourhood selection strategy is a critical step in simulated annealing. Generally neighbours should be constructed by slightly altering the current state. For example, in the travelling salesman problem each state is typically defined as a permutation of the cities to be visited, and its neighbours are the set of permutations produced by reversing the order of any two successive cities. The current state is altered to produce its neighbours (a new set of states) by a well-defined *move*. Different moves generate different set of neighbours, therefore a single type of move should be applied throughout the iterations. However due to its criticality multiple moves can be used simultaneously to compensate some drawbacks of the algorithm, such as being stuck at local minima [7].

Acceptance probabilities The probability function $P(E(s), E(s^*), T)$ controls the likelihood of a transition from s to s^*. Apart from the two neighbouring states it also depends on a time-varying parameter T, which in fact can be considered as temperature. Looking for the minimum, states with smaller energy are more probably visited. In accordance the probability function returns 1 when we have a downhill move. If it is the other way around, our move produces a state s^* with higher energy, that is $E(s^*) > E(s)$. The probabilities however must be positive even in this case preventing the algorithm to stuck at a local minimum which is worse than the global one. If this is the case the probability function should depend on the energy difference $E(s^*) - E(s)$ and on the temperature T controlling how much this energy difference contributes to probability. Analogously to the ordinary annealing high temperature results in accepting states that does not seem to be good choices at first. Decreasing T the probability of refusing a move producing a higher energy state increases. The limit $T = 0$ corresponds to the greedy algorithm which only takes into account downhill transitions. Given these properties, the temperature T plays a crucial role in controlling the evolution of the state s of the system with regard to its sensitivity to the variations of system energies. To be precise, for a large T, the evolution of s is sensitive to coarser energy variations, while it is sensitive to finer energy variations when T is small.

Cooling schedule The name and inspiration of the algorithm demand an interesting feature related to the temperature variation to be embedded in the operational characteristics of the algorithm. This necessitates a gradual reduction of the temperature as the simulation proceeds. Starting with an initial temperature which is usually high (or infinity) the annealing schedule continuously decreases it reaching $T = 0$ towards the end of the time budget allocated for our given task. This way the system is expected initially to wander around a broad range of search space then drift towards low-energy regions which as the algorithm proceeds become increasingly narrower. It can be shown that as the annealing schedule is extended any finite problem converges to the global minimum [8]. In practice however there are always limitations on the allowed time budget an algorithm can use which almost always prevents the complete search of a solution space diminishing the importance of this seemingly useful theoretical result.

Parameter selection

In order to apply the simulated annealing method to a specific problem, one must specify the following parameters: the state space, the energy function, the candidate generator procedure, the acceptance probability function, and the annealing schedule and initial temperature. These choices can have a significant impact on the method's effectiveness. Unfortunately, there are no choices of these parameters that will be good for all problems, and there is no general way to find the best choices for a given problem. Now that we are aware of the essential characteristics of a simulated annealing program it is time to dwell on the questions we left open so far. It is still undecided how to generate neighbours for a current state and what kind of temperature decrease should be applied starting from the also yet-to-be-determined initial temperature.

Generator function As for the generator function we have some requirements that should be met. A simulated annealing can be modeled as random walk on a graph whose nodes correspond to the available states and their relationships are represented with edges. Thus neighbouring states will be depicted as nodes with an edge connecting them. As a consequence of the defining properties of the simulated annealing algorithm there is an important requirement concerning the search space graph. For each node there has to belong a sufficiently short path connecting it with the node representing the global minimum. Since finding the global minimum is the task of the algorithm we do not know which node corresponds to it in advance, therefore our modified requirement is that any two nodes should have a relatively short path connecting them. As an example let us consider again the travelling salesman problem with n cities. Defining the *move* as a swap between two consecutive cities in the current tour (state), we can reach any other state within $\frac{n(n-1)}{2}$ steps. While the total number of states is $n!$, which is significantly higher. As opposed to the pair-swaps of consecutive cities a move of changing two random cities in a tour results only a maximum of $n-1$ edges between any two nodes. However from the point of view of another restriction, pair-swaps of consecutive cities are much more favourable. That is we expect from the algorithm that after a few iterations the current state has much lower energy than a random state. And this can be provided if one considers moves between states of similar energies with higher probability. This will result in excluding promising moves as well as excluding very bad moves, the former one happens a bit less often. This is actually the main principle of the so-called Metropolis-Hastings algorithm [9] [10].

Temperature decrease The ideal cooling rate unfortunately cannot be determined beforehand, hence should be adjusted empirically for each problem. However the physical analogy that is used to justify simulated annealing demands that the cooling rate should be low enough for the probability distribution of the states to be near thermodynamic equilibrium at all times. In reality though the relaxation time strongly depends on the structure of the energy function and on the current temperature [4]. One way to overcome this uncertainty is to embed the cooling schedule into the search progress, i.e. to optimize with respect to the initial temperature and temperature decrease. A so-called adaptive simulated annealing algorithm [11] is used when one utilizes this idea to find the best values for all unknown parameters.

Applying simulated annealing

Having introduced the principles of simulated annealing we can now use it for our optimization task. For this it is necessary to formulate the conditions and optimization objectives according to the key steps of the algorithm.

Let us start with defining the state space. It is surely all possible combination of source-jump pairs, where one combination is built up of some of these pairs according to Eq. (2.16). Then the whole state space C can be written as the set:

$$C = \{c_1, c_2, \dots, c_{N_\mathrm{C}}\}, \tag{3.4}$$

where N_C denotes the number of possible states or combinations. It is not a surprise that the value associated with a selection S_k (Eq. (2.6)) will have to play the role of the energy. It is because the mapping from C to S, i.e. from the set of combinations (states) to the set of selections is a function:

$$f : C \mapsto S, \tag{3.5}$$

thus we can utilize the values V_k as energies of c_k:

$$\begin{aligned} E &: C \mapsto \mathbb{R}, \\ E_k &:= V(f(c_k)). \end{aligned} \tag{3.6}$$

States are in fact partial selections where the item selection occurs based on the constructing source-jump pairs, which cannot necessarily sum up to N_P, the number of positions to be filled. At this point we will neglect how the function f operates and transforms a state to a full item selection and instead continue defining the variables and parameters for the simulated annealing algorithm.

Justified with the analogy of a physical system in thermodynamic equilibrium [9], the acceptance probability of a move from state c_j to c_k is defined as 1 if $E_k < E_j$ and $\exp(-(E_k - E_j)/T)$ otherwise, where T is the prevailing temperature. This choice apart from motivated by the actual physical problem fulfils the requirements set for the acceptance probability in the previous section. In case of a seemingly good move the probability is 1, otherwise it depends on the energy difference between the two states and is reduced with time as the temperature decreases.

As for the cooling rate we can hardly decide anything about it at this point. However let us consider how neighbour generation might work. First it is useful to emphasize again that the combinations (states) can have differing lengths, that is the number of source-jump pairs used to construct them is not generally defined. On one hand it is not always possible to build up a whole selection S_k independently of the length of combination c_k. In other words even above some particular length f cannot be considered as the identity function. On the other hand treating the combinations hierarchically connected to each other could be very advantageous due to the expansion of the energy similarly to that of the value function according to Eq. (2.14):

$$E((s,j)_1,(s,j)_2,\ldots) \;=\; e_1((s,j)_1)+e_2((s,j)_2)+\ldots. \qquad (3.7)$$

Therefore properties (including the value) of a combination constructed by almost the same elements as another state with fewer elements could be approximately deduced from the shorter one. Exploiting these results during neighbour generation puts us in front of a decisive choice. We could either take neighbours by replacing one element from a given combination or it is also possible to move up one level above and consider neighbours with a higher length (l+1). In the latter case parents are those l-length combinations which are all built up by the same pairs except one.

Going in both direction seems to be crucial for the algorithm. We would need somehow a two dimensional implementation of simulated annealing. Moreover looking for alternative solutions which could capture the tree-like structure of our search space could also be a reasonable step to take at this moment. With this in mind we will introduce a method called Monte-Carlo tree search in the next section.

3.3 Monte-Carlo tree search (MCTS)

We are still dealing with heuristic search algorithms which can shift us towards adequate approximating solutions for our problem. Monte-Carlo tree search (MCTS) is generally used for some kind of decision processes, most notably those employed in game play. The original idea dates back to 2006 [12], when the first version of MCTS was introduced for computer Go, which is a program aided with artificial intelligence dedicated to play the traditional board game Go. Ever since it has been widely applied to multiple board games like chess and shogi [13] and computer programs playing games with incomplete information such as poker have also been beneficiaries of the algorithm [14]. In 2015 AlphaGo developed by Google became the first Computer Go program to beat (without handicaps) a professional human Go player on a full-sized 19x19 board [15]. AlphaGo represents a significant improvement over previous Go programs as well as a milestone in machine learning as it uses Monte Carlo tree search with artificial neural networks.

The focus of MCTS is on the analysis of the most promising moves, while expanding the search tree based on random sampling of the search space. The application of Monte-Carlo tree search in games is based on many playouts (also called rollouts). In each playout, the game is played out to the very end by selecting moves at random, and the final game result is then used to weight the nodes in the game tree so that better nodes are more likely to be chosen in future playouts. We can understand the backpropagation of a playout result as a learning feature, which indeed has some resemblance with standard reinforcement learning (RL) [16].

Steps of MCTS

The most basic way is to apply the same number of playouts after each legal move of the current player, then choose the move which led to the most promising result, as was done in the first use of a Monte-Carlo program for evaluating the current state in 9x9 Go [17]. The efficiency often increases with time as more playouts are assigned to the moves that were frequently included in successful simulations previously. Accordingly each game decision is preceded by the MCTS algorithm attempting to find out the best move from that position. In order for the decision to be reasonable every time a large number of iterations are carried out. A single round of this repetition process consists of the following four steps.

1. Selection Say the game is currently at a root position R, from where the best move should be selected. Then we traverse downwards the tree by selecting optimal child node until a leaf node L is reached. If we are at the first iteration it means no children have been visited so far, thus we stay at R.

2. Expansion If L is a not a terminal node, i.e. it does not terminate the game decisively then create one or more child nodes according to available actions at the current state (node), then select one of these new nodes denoted by C. This node will then become part of the tree implying that it has been already visited.

3. Playout Complete one random playout or rollout from node. A playout may be as simple as choosing uniform random moves until the game is decided. Let us use E for denoting the game ending node.

4. Backpropagation Upon completing a playout, a result is returned from node E. All nodes from C up to R will be updated by applying this result on their value. The states in which the rollout traversed through are not considered visited, thus they are not even part of the current tree yet.

Exploration and exploitation

One could observe that during *selection* there is bias regarding the selection of child nodes. Starting at R, the first child node C is selected in the first iteration. In each following rounds however node C will be selected if its backpropagated value surpasses the initial value of sibling nodes at the current branching point. In other words we do not give any chance for the algorithm to explore new, unvisited nodes, instead there is a maximal exploitation of the route that is believed to be the best. This irregularity results in expanding the same route over and over not allowing to deviate from any previously visited nodes. In fact the main difficulty in selecting child nodes is maintaining some balance between the exploitation of deep variants after moves with high value and the exploration of moves with few simulations. The first solution for balancing between exploitation and exploration in games have been introduced as the upper confidence tree (UCT) [18]. In a UCT it is recommended to choose, in each branching point of the game tree, the node for which the expression

$$w_i + c\sqrt{\frac{N_i}{n_i}} \tag{3.8}$$

has the highest value. In this formula:

- w_i stands for the value of the node considered after the i-th iteration,
- n_i stands for the number of simulations for the node considered after the i-th iteration,
- N_i stands for the total number of simulations after the i-th iteration ran by the parent node of the one considered,
- c is the exploration parameter, in practice usually chosen empirically.

The first component of the formula above corresponds to exploitation, while the second one supports exploration. It is because nodes with few simulations have higher value for the second part of this expression accounting for their relatively small contribution to the routes investigated up to that point. In most recent implementations of MCTS, the optimization is generally based on upper confidence trees.

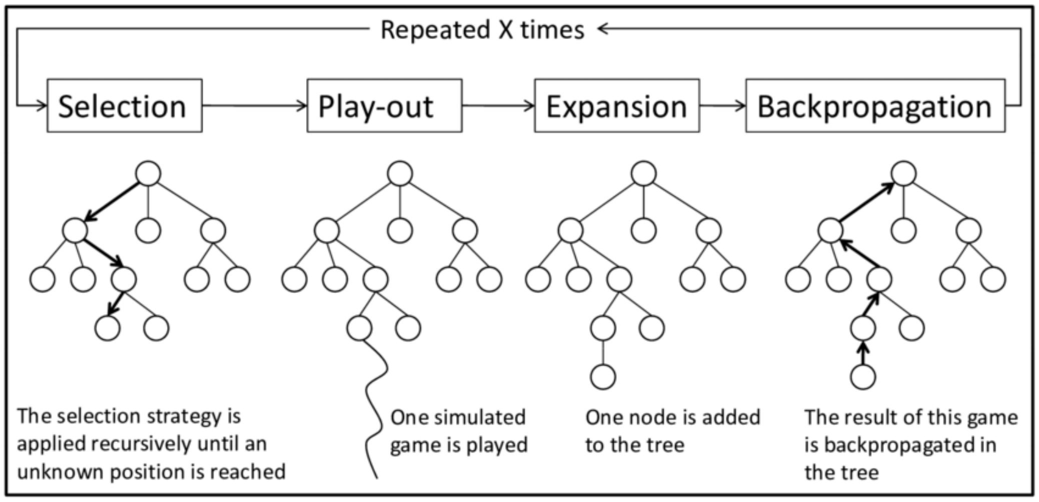

Repeated X times
Selection
Play-out
Expansion
Backpropagation
The selection strategy is applied recursively until an unknown position is reached
One simulated game is played
One node is added to the tree
The result of this game is backpropagated in the tree

Figure 3.1: Outline of Monte-Carlo tree search [19]

Chapter 4

Sampling with SA-MCTS

Let us briefly recap where we left off earlier. A state space has been defined according to Eq. (3.4) with the associated energies (Eq. (3.6)). A single state is built up with several source-jump pairs (Eq. (2.16)). Since a mapping (Eq. (3.5)) from state space to the set of selections have been introduced, we can utilize the value function (Eq. (2.6)) as energy:

$$C = \{c_1, c_2, \dots, c_{N_\mathrm{C}}\},$$

$$c_k = \{(s,j)_1, (s,j)_2, \dots\},$$

$$f : C \mapsto S,$$

$$E_k := V(f(c_k)),$$

$$V_k = V(S_k) = V(k_1, k_2, ..., k_{N_\mathrm{P}}).$$

A very important property of the energy function is decomposability based on distinct sources (Eq. (3.7)):

$$E((s,j)_1, (s,j)_2, \dots) \;=\; e_1((s,j)_1) + e_2((s,j)_2) + \dots,$$

which has been motivating enough to shift our considerations towards the source-based representation when the task is to take samples from the state space in the most clever way possible. That is return states with the lowest energy from the population. Having gained the necessary insights on the structure of the energy function and on the state space itself we could argue that exploiting a method that somewhat resembles to a tree-search-like approach is beneficial. It would be able to capture the hierarchical construction of our state space. For this reason we have familiarized ourselves with

Monte-Carlo tree search. In the followings we will continue to build up a new method called SA-MCTS to help us solve our initial problem. As the name suggests it is a symbiotic combination of two distinct metaheuristics: simulated annealing and Monte-Carlo tree search. This merging has been proposed recently for evaluating Partially Observable Markov Decision Processes (POMDPs) [20], and in the field particle physics as well. In high energy physics extremely large expressions are obtained when evaluating the corresponding Feynman diagrams, simplifying them turns to be more easily achieved when temperature control is applied for UCT [21].

4.1 Tree structure

It was one of our findings earlier that our state space has a hierarchical structure. However we are yet to be able to illustrate or even explain this phenomenon. In order to do that we need to take care of some negligence carried along the way for a while now. Earlier it was mentioned that the length of a state, i.e. the number of source-jump pairs constructing it had not been defined. It is in fact because this so-called length could vary from one state to another. More precisely we consider all combinations equal starting from the shortest one with a single source-jump pair up to the one which is built up by N_S (number of sources) pairs. The upper limit is due to the fact that a state indicates the number of items that should be selected from certain sources. Thus it can contain a single source at most once. Let us take a look at how the first and the last combination looks like:

$$\begin{aligned} c_1 =& \{(s,j)_1\}, \\ c_{N_\mathrm{C}} =& \{(s,j)_1, (s,j)_2, \ldots, (s,j)_{N_\mathrm{S}}\}. \end{aligned}$$

Let us assume that there are five jump values for each source equivalently, i.e. $N_J = N_J^s = 4 \,\forall\, s$. Indicating the absence of a source in a combination as another jump value, N_C could be theoretically around $(N_J+1)^{N_\mathrm{S}}$. It would yield $\sim 10^{70}$ states eventually with $N_\mathrm{S} = 100$. Although not all combinations are feasible, in fact most of them are not. These cannot be transformed into a valid N_P-length selection, but we will revisit this issue later.

As hinted earlier each node in the tree corresponds to a state. There are certain levels on the tree and the association of nodes on the same level is called a branch. These nodes can have parents and children which is determined in the following way. A node n on level l has $\binom{l}{l-1}$ parents on level $l-1$ such that they correspond to $l-1$-length subsets of c_n. An example might help digesting the idea. We have the node

$$c_n = \{(s,j)_1, (s,j)_2, (s,j)_3\}, \tag{4.1}$$

which has parents:

$$c_{p_1} = \{(s,j)_1, (s,j)_2\},$$
$$c_{p_2} = \{(s,j)_1, (s,j)_3\},$$
$$c_{p_3} = \{(s,j)_2, (s,j)_3\},$$

who are in fact children of $(s,j)_1, (s,j)_2$ and $(s,j)_3$ in the same way. The corresponding tree is illustrated in Fig. 4.1, where a simpler notation is used to denote the states.

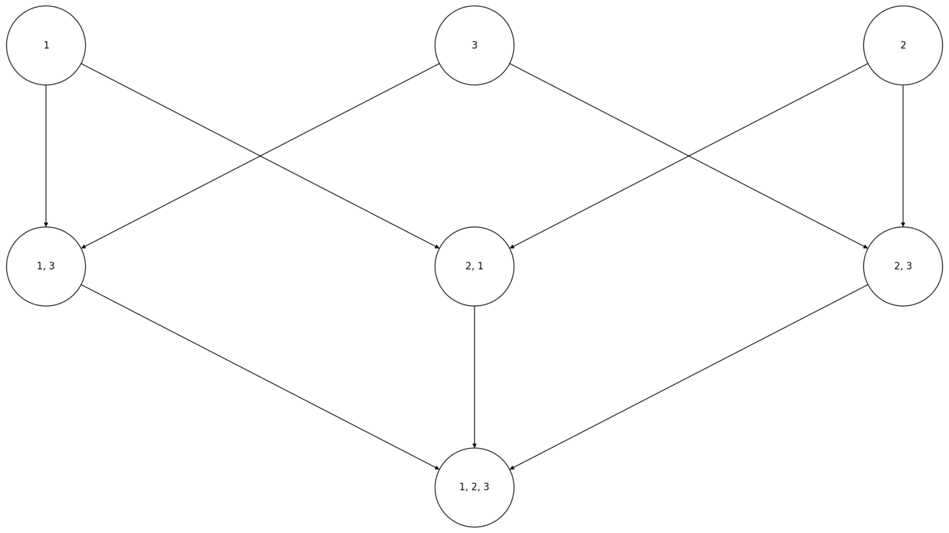


Figure 4.1: A simple tree

4.2 Tree search

It is very important to emphasize again that a combination or state simply bears information on how many items should be selected from some particular source. We still do not know which positions they belong out of the N_P possibilities. What we do know though is that a position p can be represented only once in a combination putting additional restrictions on the items to be chosen. We will get back to this later how to select items fulfilling a certain state, for now let us encapsulate this into function f. This function from Eq. (3.5) has yet to be defined, however it is known that it maps a state to a complete selection of items. That is N_P items are selected, each for one position. Now on one hand we recognize f as the part of the machinery which is responsible for item selection fulfilling a combination. Although the intermediate result will not be a complete selection since it does not necessarily contain items for all N_P. Actually we can tell exactly how many positions will be filled with items upon performing an item selection for a particular state. Combination c_n from Eq. (4.1) offers exactly $j_1 + j_2 + j_3$ positions to be filled by node n. We will denote this by N_P^n later on. And this is the other task function f has been assigned to do. To extend a partial selection with length N_P^n to a complete one with N_P items. Now that we have cleared the air around some essential parts of the algorithm it is time to lay down the steps of our tree search.

The algorithm

Analogously to the steps of the original Monte-Carlo tree search we have *selection*, that is which node should be investigated next, *playout*, to reach a game ending state, *expansion* of the tree and *propagation* of values to update the nodes. However the context and details of these steps will be a bit different as will be demonstrated in the followings.

Node value In common applications of MCTS we have a defined value for the nodes. In terms of games it is usually the average score reached after the simulation step or the win ratio in a zero-sum game. In our case nodes have an associated energy, however energies should be subject to minimization. Therefore it cannot be considered as a proper value for the MCTS algorithm. It should not be much trouble to consider the lowest value as best from now on, but instead we will work with inverse energies. It has to be emphasized that only after a node gets selected will it have a value based on the actual energy of the state. The soul of the whole idea breaks down to the propagation of this value to parents and children as well providing us to rationally compare nodes which are yet to be visited or have been visited long time ago.

Initialization Construct a one-length node for each source-jump pair. They will expand the first branch of the tree. We have to consider them identical (in terms of value) at the beginning, and all of them have to be selected once. Upon completion of the first branch a new branch with two-length combinations will be created.

Selection The next node to be chosen is always the one with the highest value. Until a branch is not completely visited or inactive, i.e. all nodes have been selected at least once, the selection process starts on that branch. In other words the first active branch is the starting point of the selection process. When a node is not chosen for the first time, the process continues focusing on the children of this node until it reaches a new node. Precisely for this reason expansion step is necessary.

Expansion When the first node has been chosen from a new branch, all feasible children nodes are created on a new branch. By doing so the process is prepared for the immediate exploitation of the same route.

Playout As mentioned earlier a node or state is just an indicator regarding the required item count from certain sources. It is not a proper selection of N_{P} items. To be able to calculate energies we need to map from the state space to the set of selections. This is done via function f. First it carries out an item selection for the N_{P}^{n} designated positions, then it does a simulation to choose the rest of the $N_{\mathrm{P}} - N_{\mathrm{P}}^{n}$ items pseudo-randomly.

Energy calculation When a node is picked and playout has been performed to transform it into an item selection, the iteration stops until its energy is calculated. Once there is something to be returned, the algorithm proceeds forward with the goal of spreading this new information among ascendants and descendants.

Forward-update Since the energy function can be decomposed as in Eq. (3.7), we have been motivating ourselves to utilize the tree structure of the state space. Now this is the part where we take advantage of it. During expansion not only new nodes are created, but in fact they will possess an initial value from the very beginning. This initialization is done by the parents, whose current values are averaged and propagated to the children on the new branch. In general anytime a node gets a new value, it will be propagated forward to the children who are yet to be visited by the algorithm. This continues for all active branches forward. The nodes who have not been selected yet have estimated values based on those of their parents. In case of nodes visited already, information in the future takes precedence, since their value now is based on the energy of the corresponding state. As for the so-called *unvisited* nodes information in the past takes precedence, and every time a parent is updated the information is shared with them.

Backward-update There is information sharing in the other direction as well called backward-update. When a node is selected, its new value then will be propagated backwards to the parents who have been already selected. Their value will be updated based on those of their children taking their average. The update then proceeds backwards until the last active branch. These parents have values reflecting the true energy of their states. Their only role from now on is to optimize the path of deeper node selections. Again information in the future takes precedence when it comes to visited nodes.

4.3 Annealing part

During node selection the one with the highest value is considered next. Node values were said to be inversely proportional to the energy of the corresponding states:

$$w_n = \frac{1}{E_n}, \tag{4.2}$$

where w_n and E_n is the value and energy for node n respectively. (To avoid some confusion: E_n, the energy of the state or node has been defined as equal to V_n, the value of the corresponding item selection, w_n is the value of the node itself, which is in fact the inverse of V_n, whereas v_k, or denoting the actual state v_k^n came from the decomposition of V_n.) If we were to compare this expression with Eq. (3.8), it is apparent that the exploration part is missing from our algorithm and w_n only accounts for exploitation of known routes in the tree.

Consider for instance a new branch with estimated node values coming from forward-update. Say a node is selected and achieves a value which is higher than that for all sibling nodes in the same branch. As a consequence the same node will be selected in the next round as well, however since it was already selected once the process continues considering now only the children of this particular node. Eventually a chain of identical events occurs resulting that the algorithm has only explored a single route but with maximal exploitation, that is travelling as far as it could on the route that was picked initially. MCTS keeps a balance of exploration and exploitation by introducing a so-called *upper confidence bound* (UCB) value. It forces the algorithm not to neglect nodes only because they are rarely visited and therefore their values are lower. In our case we would only need to mimic this feature if the estimated node values could not be completely trusted. Otherwise the previous example exploiting only a single route entirely would not be a bad solution at all. In this case lower estimated values would surely lead to worse results, and allocating time for investigating them would be inadvisable.

As for the tree-search part, the key concept behind it was the tree-like structure of our state space so that we could predict values for combinations built up with more elements based on those built up with just a few pairs. Now the main idea is that while this prediction is useful, it cannot be taken granted. In fact if one were to think this through it can be realized that in general an estimated value should be an upper bound for the real one. Consider two pairs belonging to different sources and say these are two states on branch 1: $n_1 = (s_1, j_1)$, $n_2 = (s_2, j_2)$, and also assume that branch 2 would contain the node: $n_3 = \{(s_1, j_1), (s_2, j_2)\}$. It is easy to see that upon playout for collecting items for the rest of the $N_\mathrm{P} - N_\mathrm{P}^n$ positions, we will end up with a more promising selection for node n_3 in terms of energy. At least two distinct value reductions can be reached simultaneously, while for nodes in branch 1 it is only one by default. (It has to be mentioned that both value jumps could be reached even on branch 1 upon the simulation, and that the estimated value is only an upper limit when the presence of these source-jump pairs together is advantageous. Otherwise we can reach a halting point during the run, where the child node does not yield any improvement in terms of energy compared to its parents. We will go into more details about halting points in a later section.) Eventually we would be missing out on investigating worthwhile states if we were to completely trust the estimated node values. Therefore we need exploration.

Analogously to Eq. (3.8) we can introduce the exploration term:

$$expl_n = c\sqrt{\frac{vis_b}{vis_n}}, \tag{4.3}$$

where vis_b and vis_n is the number of visits for branch b and its node n respectively, c is still a constant as in MCTS. The UCB value then becomes

$$u_n = w_n + expl_n. \tag{4.4}$$

We have been mentioning that the set of positions (denoted by p_n), a combination (node n) is assigned to choose items for, should not contain the same position more than once. However it is not a requirement to include all N_P positions. Thus we can write the following conditions:

$$\begin{aligned} \arg\max_p \ | \{p \in p_n\} | \ &= \ p_n, \\ 1 \leq \ N_\mathrm{P}^n \ &\leq N_\mathrm{P}. \end{aligned}$$

Satisfying the first condition is not always trivial. Ideally the whole tree is built up with states which does not seem to violate the rules. However it is not something we can easily decide in advance. It largely depends on the structure of the source-based and the position-based representation and on the connection between them (Eq. (2.12)). Therefore non-feasible states could appear in the tree and it will only be discovered during runtime. These states should be punished and the algorithm should avoid their descendants afterwards. The same applies to halting points introduced previously. In this case the particular combination is feasible, however it does not seem worth continuing the investigation for the corresponding part of the tree. (This is very similar to the well known branch-and-bound method in the field of combinatorial optimization [22]). Non-feasible combinations are more likely to appear in higher branches of the tree when a relatively large number of value jumps should be reached simultaneously. This indicates that as we proceed forward in the tree exploitation of known routes becomes more important and significant than exploring sibling states. It is because a satisfied combination on a high branch is very valuable, thus we should not be looking for a sibling combination with the same length, rather we must utilize entirely what we have in hand and try to expand it. According to previous arguments the more pairs a combination consists the more likely it will possess a lower energy. Approaching from the opposite direction, at the very beginning the algorithm does not have a clear picture on the state space. Exploring and getting a grip on it is much more advantageous initially than exploiting the same route over and over. As a result a slight modification of our method is necessary to capture the initial importance of exploration and the subsequent shift towards exploitation. This resembles very much to an annealing schedule.

To account for the requirements mentioned above we introduce a time-dependent temperature parameter into our algorithm. Modifying the UCB value expression we have

$$u_n = w_n \; + \; T(t) \; \sqrt{\frac{vis_b}{vis_n}}, \tag{4.5}$$

when $T(t)$ is the time-dependent temperature. The annealing schedule should ensure that we start off with a temperature large enough to enable the exploration part of our expression to dominate. While as we proceed forward the significance of the exploitation term (that is the raw value of the node) should gradually increase. As with other metaheuristics, some parameters are not pre-defined and could be adjusted for specific tasks. For now details regarding the initial temperature, the annealing schedule and criteria for halting points are neglected.

4.4 States to partial and complete selections

As mentioned before a state sets up requirements for certain sources regarding how many positions they should occupy in a partial selection. Upon fulfilling this requirement an intermediate result is reached. Then it gets extended to a complete item selection by function f (Eq. (3.5)). This part as a whole is conveniently named *playout* in the SA-MCTS scheme. Let us start with how a partial selection might be obtained.

Building up partial selections

If the algorithm is currently at node n, an investigation regarding its importance is necessary. In the first step combination c_n will be transformed into a partial selection, that is items will be selected to N_P^n positions meeting certain criteria. Precisely the source-jump pairs constructing c_n will bear information about how many items should be allocated to each source. Filling up the places with items in a sophisticated way will be dealt here in more detail.

We have overall N_I items available for selection at the beginning. The set of items can be split into N_P parts (subsets I_p) according to Eq. (2.4), this was called position-based representation. There is another type of grouping called source-based representation with subsets I_s. Their comparison is seen in Eq. (2.12). Now if we filter I_p-s with the source-based subsets in question we can create a collection of items which is more convenient to work with. The interesting I_s subsets will be those that belong to sources included in our current combination c_n:

$$I^n = \cup\, I_s \;\forall\; s \in c_n,$$

where I^n is the whole set of items that are in fact available for us to use at the current node. Now we introduce new I_p subsets (denoted by I_p^*) that are filtered through I^n:

$$I_p^* = I_p \cap I^n \;\;\forall\;\; p \in \{1, 2, \ldots, N_\mathrm{P}\}. \tag{4.6}$$

Sorting out immediately those new subsets which are empty, we will do the same procedure with source-based representation. This time, filtering with $\cup I_p^*$, we end up with new subsets I_s^*.

Now the first items to be chosen will be those that belong to I_p^* subsets consisting of only one element. Basically we have filtered out those positions where item selection is trivial in the case where we only consider a restricted number of sources. We also delete these items from the new subsets. It is also useful to keep account of the progress of item selections for each source. Therefore we continuously subtract one from the $j_1, j_2, \ldots$ jump values whenever an item is picked. Let us call this *remaining count* and denote it by r_s, which will correspond to source s. Having done with the terminology we have everything in our possession to introduce a systematic way for the item selection.

There is an order regarding the sources for which $r_s \neq 0$ (sources who do not participate in the current combination will have automatically $r_s = 0$ remaining counts). The algorithm will prefer those with larger r_s values and leave over the ones who still have a great variety of available items. Mathematically this rule could be captured as the so-called source-score:

$$X_s = \frac{r_s}{|I_s^*|}.$$

Before picking an item X_s will be calculated for each source. The one with the highest value will be our next choice. Naturally zero in the denominator can occur which either implies that the item selection is finished or we reached a non-feasible node (we ran out of available items before reaching each jump in the selection). Now that we have the next source (say s) for which an item needs to be chosen, we can shift our attention and focus on the set I_s^*. The preference of our algorithm will be to use up less valuable items first and leave more valuable ones for later. What do we mean by that? Well, it is not particularly the item itself whose importance is of interest, rather the subset I_p^* including it. The larger its size is the more sources could be assisted to reach their jumps, thus the position p itself is more valuable. Hence choosing an item for source s from I_p^* practically means the simultaneous loss of possibility to increment the item count for several other sources. On the other hand if $|I_p^*|$ is currently one there is no question whether its single item is the most perfect choice for source s. Again a certain score would be helpful to define:

$$X_p^s = \sum_{s'}^{I_p^*} X_{s'} - X_s,$$

where the left-hand side is the so-called position-score. This score can only acquire a meaning once we have selected the next source candidate. According to the right hand side of the equation above we take the source-scores of all sources for which there is an item in I_p^* as well (i.e. $|I_s^* \cap I_p^*| \neq 0$), and the actual source-score calculated in the previous step is subtracted from their sum. This time however our choice will be the item with the lowest score, that is the least valuable one. We can thus conclude that the iterative procedure of item selection consists of the following steps:

1. Calculate $X_s \; \forall \; s : r_s \neq 0$, choose the source with the highest score. Halt if there is zero in the denominator for any source. If $\sum_s r_s = 0$ the combination is valid, otherwise it will be declared as a non-feasible state.

2. Calculate X_p^s for the source selected, choose the item with the lowest score.

3. Update the status of our data by deleting I_p^* and subsequently creating new I_s^* subsets, additionally subtract one from r_s.

4. Goto 1

When the algorithm proceeded without halt and terminated successfully we have generated a partial selection.

It has to be mentioned that determining whether a node is feasible or not during the run is solely based on this procedure of item selection specified above. It means our assumptions are as good as much the scoring system yields the most optimal order of items.

Upgrading to full selections

From Eq. (2.5) we know the form of a complete item selection S_n. This will belong to state n upon playout in the SA-MCTS algorithm. Until this point however we do not have S_n in our hand, only a partial selection has been obtained previously. Let us denote it by S'_n. As stated earlier this part is also covered by function f. More precisely we could decompose f into two components:

$$f = f_1 + f_2,$$

where f_1 is in charge of creating partial selections out of combinations:

$$f_1 : C \mapsto S' \qquad S'_n = f_1(c_n),$$

and f_2 is responsible for the expansion:

$$f_2 : S' \mapsto S \qquad S_n = f_2(S'_n).$$

We have been saying a lot about f_1, now it is time to elaborate on the mechanism of f_2. Its job is to pick items for the remaining $N_\mathrm{P} - N_\mathrm{P}^n$ positions. It was also mentioned that it selects items pseudo-randomly. What we mean by that is something more robust than a clearly random procedure, however fast enough to maintain the smoothness of our algorithm. According to Eq. (2.11) the value V for the whole selection S_n can be decomposed into N_P parts. It depends on which item we selected for other positions, however provides a way to choose remaining items. For all p empty positions we do the following:

1. Calculate $v_p({}^*S'_n, k_p)$ individual values (where ${}^*S'_n$ is the unpacked partial selection) for all available items.
2. Choose the item yielding the lowest value.
3. Go to the next empty position.

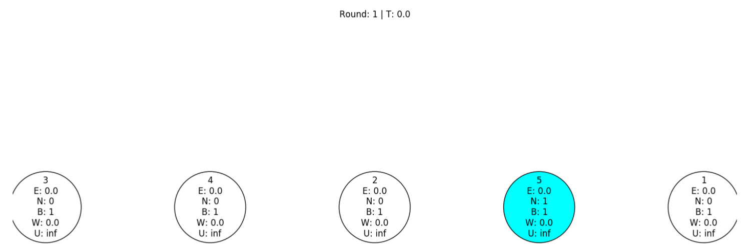
This ensures a static environment where only those items are varied which are chosen for the position in question. Hence provides a simple way to compare choices without expanding the current state. Having completed the iteration for the remaining spots we end up with a complete selection covering all positions.

We reached a part where most of the details about our algorithm have been explained and specified. Hopefully it gives us an adequate insight into the machinery behind the scenes. In the next chapter illustrative examples will be provided about the results achieved when using the sampling technique of SA-MCTS.

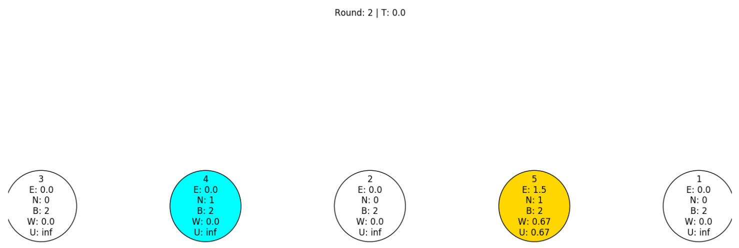
Chapter 5

Illustrative examples

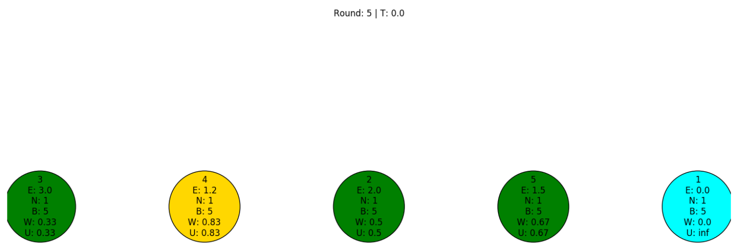
In this chapter we present a walk-through example of how SA-MCTS operates. We will particularly focus on how the algorithm navigates in the tree generating item selection candidates. Although the exact way of transforming a state to an item selection will be neglected. The number of sources and their corresponding jumps will be restricted in order for the key steps of the algorithm to be apparent. To be more precise, our little example consists of five sources ($N_{\mathrm{S}} = 5$), and each source has a single jump assigned to it. The amount of items and available positions are not even necessary to be specified, additionally the exact item count where jumps (value reductions) occur and their corresponding values will also be unspecified. Let us take a look at the tree in the beginning of the process.



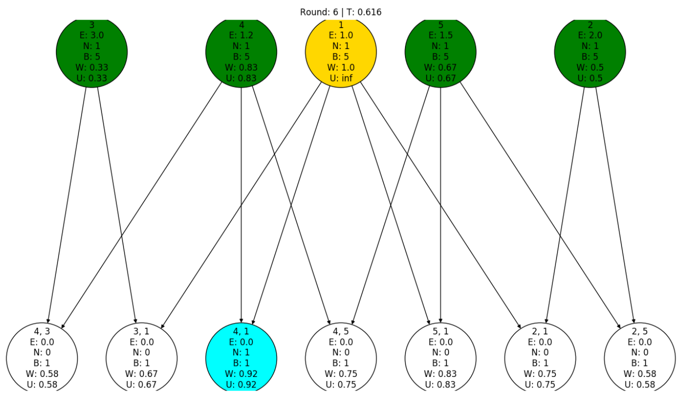
This is the first round of the process. What can be seen here is the first branch consisting of the five source-jump pairs in question. Source identifiers, energies (E), node (N) and branch (B) visit counts, values (W) and UCB values (U) are indicated for each node. What happens now is that the initialization part of the algorithm has started, accordingly each node on the first branch have to be visited first. We do not know anything about the nodes at this point, hence the infinite UCB values. The node corresponding to source 5 has been chosen indicated with light blue. At the second round we have the following structure of the tree.



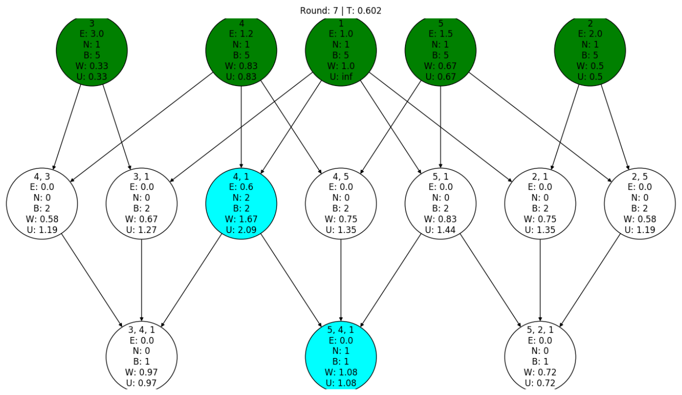
Gold node indicates the best state in terms of energy up to that point. It is obviously what we chose first, since there has been a playout and a value calculation hidden behind the scenes. One can observe that the temperature is currently zero, thus UCB values coincide with the actual values for each state. After five rounds the first branch is completed.


Round: 5 | T: 0.0
3 E: 3.0 N: 1 B: 5 W: 0.33 U: 0.33
4 E: 1.2 N: 1 B: 5 W: 0.83 U: 0.83
2 E: 2.0 N: 1 B: 5 W: 0.5 U: 0.5
5 E: 1.5 N: 1 B: 5 W: 0.67 U: 0.67
1 E: 0.0 N: 1 B: 5 W: 0.0 U: inf

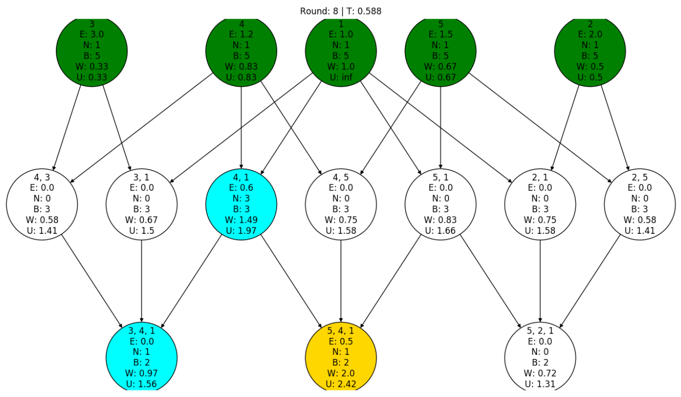
Green nodes mark those already visited. The next round investigates further down on a new branch which will be created by then.


Round: 6 | T: 0.616
3 E: 3.0 N: 1 B: 5 W: 0.33 U: 0.33
4 E: 1.2 N: 1 B: 5 W: 0.83 U: 0.83
1 E: 1.0 N: 1 B: 5 W: 1.0 U: inf
5 E: 1.5 N: 1 B: 5 W: 0.67 U: 0.67
2 E: 2.0 N: 1 B: 5 W: 0.5 U: 0.5
4, 3 E: 0.0 N: 0 B: 1 W: 0.58 U: 0.58
3, 1 E: 0.0 N: 0 B: 1 W: 0.67 U: 0.67
4, 1 E: 0.0 N: 1 B: 1 W: 0.92 U: 0.92
4, 5 E: 0.0 N: 0 B: 1 W: 0.75 U: 0.75
5, 1 E: 0.0 N: 0 B: 1 W: 0.83 U: 0.83
2, 1 E: 0.0 N: 0 B: 1 W: 0.75 U: 0.75
2, 5 E: 0.0 N: 0 B: 1 W: 0.58 U: 0.58

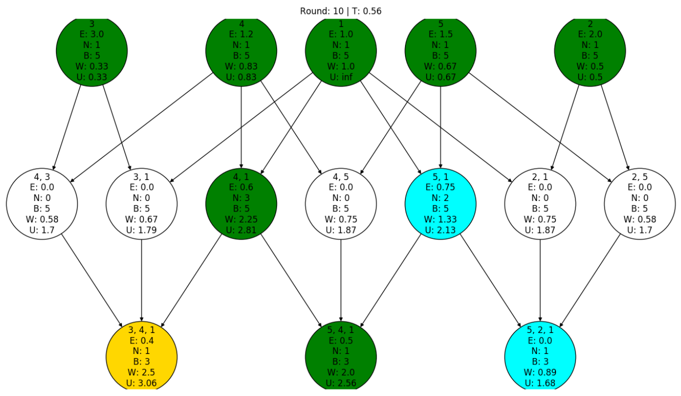
We see the best candidate from the first branch is eventually the node belonging to source 1. Upon completion a new branch is created by the expansion step and values are immediately assigned for each node by forward-update. There is an embedded logic regarding the feasibility of states resulting that only probable pairs are presented in the tree. Missing energies are due to the fact that these nodes are newly created, thus they have not been investigated yet. There is also a non-zero temperature now, which will undergo an annealing schedule. As for the UCB values, they still entirely coincide with the regular values in the presence of temperature. According to the expression of our UCB value (Eq. (4.5)), the exploration term depends on visit counts as well, which are currently zero in this new branch. This explains the previous observation. Nevertheless the next candidate to be chosen is the one with highest UCB value. Since it depends on the temperature, calculation for each node should be at the same time prior to picking the most promising state. This is the reason why UCB calculation for the last node on branch 1 has been omitted, since no further comparison will take place there. From now on branch 2 will be the first active branch, marking the start of the node selecting procedure. The next node belongs to the combination of source 1 and source 4, and it had the largest estimated value upon forward-update.


Round: 7 | T: 0.602
3 E: 3.0 N: 1 B: 5 W: 0.33 U: 0.33
4 E: 1.2 N: 1 B: 5 W: 0.83 U: 0.83
1 E: 1.0 N: 1 B: 5 W: 1.0 U: inf
5 E: 1.5 N: 1 B: 5 W: 0.67 U: 0.67
2 E: 2.0 N: 1 B: 5 W: 0.5 U: 0.5
4, 3 E: 0.0 N: 0 B: 2 W: 0.58 U: 1.19
3, 1 E: 0.0 N: 0 B: 2 W: 0.67 U: 1.27
4, 1 E: 0.6 N: 2 B: 2 W: 1.67 U: 2.09
4, 5 E: 0.0 N: 0 B: 2 W: 0.75 U: 1.35
5, 1 E: 0.0 N: 0 B: 2 W: 0.83 U: 1.44
2, 1 E: 0.0 N: 0 B: 2 W: 0.75 U: 1.35
2, 5 E: 0.0 N: 0 B: 2 W: 0.58 U: 1.19
3, 4, 1 E: 0.0 N: 0 B: 1 W: 0.97 U: 0.97
5, 4, 1 E: 0.0 N: 1 B: 1 W: 1.08 U: 1.08
5, 2, 1 E: 0.0 N: 0 B: 1 W: 0.72 U: 0.72

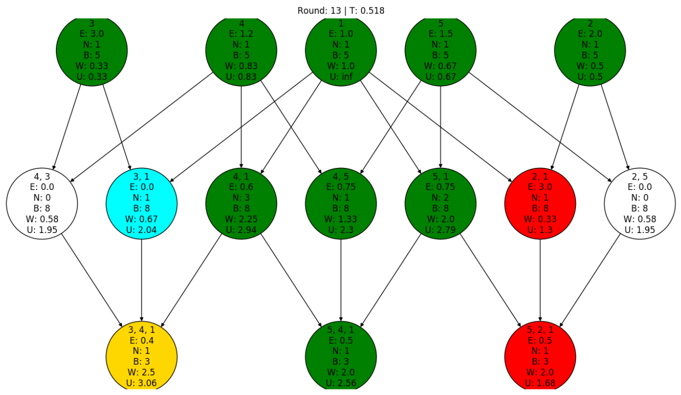
Round 7 presents an immediate exploitation of the previously selected route. This behaviour could not be avoided even with temperature in present. What happened exactly is that the same node was chosen as in the last round on branch 2. Its actual value (upon playout and energy calculation) turned out to be better than the estimated one. However since it had been already visited, the algorithm moved forward considering its children. Again a new branch was created by forward-update, and out of the two (feasible) child nodes, the one with highest estimated value was chosen.


Round: 8 | T: 0.588
3 E: 3.0 N: 1 B: 5 W: 0.33 U: 0.33
4 E: 1.2 N: 1 B: 5 W: 0.83 U: 0.83
1 E: 1.0 N: 1 B: 5 W: 1.0 U: inf
5 E: 1.5 N: 1 B: 5 W: 0.67 U: 0.67
2 E: 2.0 N: 1 B: 5 W: 0.5 U: 0.5
4, 3 E: 0.0 N: 0 B: 3 W: 0.58 U: 1.41
3, 1 E: 0.0 N: 0 B: 3 W: 0.67 U: 1.5
4, 1 E: 0.6 N: 3 B: 3 W: 1.49 U: 1.97
4, 5 E: 0.0 N: 0 B: 3 W: 0.75 U: 1.58
5, 1 E: 0.0 N: 0 B: 3 W: 0.83 U: 1.66
2, 1 E: 0.0 N: 0 B: 3 W: 0.75 U: 1.58
2, 5 E: 0.0 N: 0 B: 3 W: 0.58 U: 1.41
3, 4, 1 E: 0.0 N: 1 B: 2 W: 0.97 U: 1.56
5, 4, 1 E: 0.5 N: 1 B: 2 W: 2.0 U: 2.42
5, 2, 1 E: 0.0 N: 0 B: 2 W: 0.72 U: 1.31

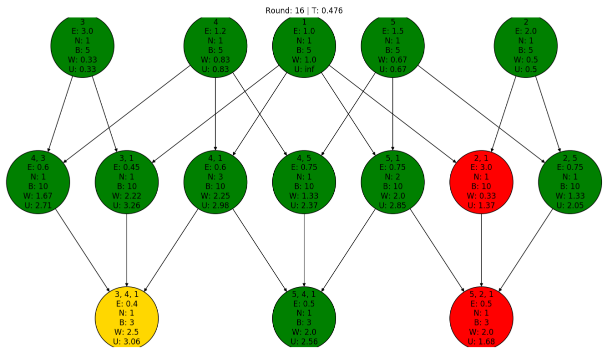
It has been beneficial to exploit the same route since we found a new leader. It is apparent that the tree could not grow further inserting an additional branch on level four, therefore it did not necessitate a forward-update. In return the new value has been propagated back to node (4,1) (but not to (4,5) and (5,1) complying to the rules of backpropagation), whose value now depends on its children. The three visits so far has worsened its UCB value, not as much though to endanger its superiority on branch 2. Thus the other child is the next candidate of the algorithm in round 8.


Round: 10 | T: 0.56
3 E: 3.0 N: 1 B: 5 W: 0.33 U: 0.33
4 E: 1.2 N: 1 B: 5 W: 0.83 U: 0.83
1 E: 1.0 N: 1 B: 5 W: 1.0 U: inf
5 E: 1.5 N: 1 B: 5 W: 0.67 U: 0.67
2 E: 2.0 N: 1 B: 5 W: 0.5 U: 0.5
4, 3 E: 0.0 N: 0 B: 5 W: 0.58 U: 1.7
3, 1 E: 0.0 N: 0 B: 5 W: 0.67 U: 1.79
4, 1 E: 0.6 N: 3 B: 5 W: 2.25 U: 2.81
4, 5 E: 0.0 N: 0 B: 5 W: 0.75 U: 1.87
5, 1 E: 0.75 N: 2 B: 5 W: 1.33 U: 2.13
2, 1 E: 0.0 N: 0 B: 5 W: 0.75 U: 1.87
2, 5 E: 0.0 N: 0 B: 5 W: 0.58 U: 1.7
3, 4, 1 E: 0.4 N: 1 B: 3 W: 2.5 U: 3.06
5, 4, 1 E: 0.5 N: 1 B: 3 W: 2.0 U: 2.56
5, 2, 1 E: 0.0 N: 1 B: 3 W: 0.89 U: 1.68

Despite the lower estimated value, the second child has been identified as the new leader. Skipping the next round, we can see that another exploitation occured at round 10. First (5,1) was selected initiating a forward update to (5,2,1) (but not to (5,4,3), which was visited already once; as it is on the last branch it does not have much significance), then (5,2,1) was picked.


Round: 13 | T: 0.518
3 E: 3.0 N: 1 B: 5 W: 0.33 U: 0.33
4 E: 1.2 N: 1 B: 5 W: 0.83 U: 0.83
1 E: 1.0 N: 1 B: 5 W: 1.0 U: inf
5 E: 1.5 N: 1 B: 5 W: 0.67 U: 0.67
2 E: 2.0 N: 1 B: 5 W: 0.5 U: 0.5
4, 3 E: 0.0 N: 0 B: 8 W: 0.58 U: 1.95
3, 1 E: 0.0 N: 1 B: 8 W: 0.67 U: 2.04
4, 1 E: 0.6 N: 3 B: 8 W: 2.25 U: 2.94
4, 5 E: 0.75 N: 1 B: 8 W: 1.33 U: 2.3
5, 1 E: 0.75 N: 2 B: 8 W: 2.0 U: 2.79
2, 1 E: 3.0 N: 1 B: 8 W: 0.33 U: 1.3
2, 5 E: 0.0 N: 0 B: 8 W: 0.58 U: 1.95
3, 4, 1 E: 0.4 N: 1 B: 3 W: 2.5 U: 3.06
5, 4, 1 E: 0.5 N: 1 B: 3 W: 2.0 U: 2.56
5, 2, 1 E: 0.5 N: 1 B: 3 W: 2.0 U: 1.68

Three rounds later we reached a halting point (indicated by red) on (2,1). Its calculated value was worse than that of at least one of its parent. In this case all of its descendants should be avoided later on. However there is only one child that belongs to (2,1), moreover it has already been considered by the algorithm. Still, for visualization purposes, it is also marked red. In fact the general rule regarding the identification of halting points is the following: if at least one parent has lower value than the child node in question, it will be considered a halting point.


Round: 16 | T: 0.476
3 E: 3.0 N: 1 B: 5 W: 0.33 U: 0.33
4 E: 1.2 N: 1 B: 5 W: 0.83 U: 0.83
1 E: 1.0 N: 1 B: 5 W: 1.0 U: inf
5 E: 1.5 N: 1 B: 5 W: 0.67 U: 0.67
2 E: 2.0 N: 1 B: 5 W: 0.5 U: 0.5
4, 3 E: 0.6 N: 1 B: 10 W: 1.67 U: 2.71
3, 1 E: 0.45 N: 1 B: 10 W: 2.22 U: 3.26
4, 1 E: 0.6 N: 3 B: 10 W: 2.25 U: 2.98
4, 5 E: 0.75 N: 1 B: 10 W: 1.33 U: 2.37
5, 1 E: 0.75 N: 2 B: 10 W: 2.0 U: 2.85
2, 1 E: 3.0 N: 1 B: 10 W: 0.33 U: 1.37
2, 5 E: 0.75 N: 1 B: 10 W: 1.33 U: 2.05
3, 4, 1 E: 0.4 N: 1 B: 3 W: 2.5 U: 3.06
5, 4, 1 E: 0.5 N: 1 B: 3 W: 2.0 U: 2.56
5, 2, 1 E: 0.5 N: 1 B: 3 W: 2.0 U: 1.68

Finally, at round 16 the algorithm could not find any available node to pick, thus the iteration has been completed. While forward and backward updates, halting points and other basic features of the algorithm could be easily evaluated on the figures produced during runtime, a quantitative analysis focusing solely on the temperature dependence would require a different plot. This time we only need to be aware of node values and how quickly or slowly the algorithm reached that point, it is not relevant which state they belonged to originally. This graph can be seen in Fig. 5.1.

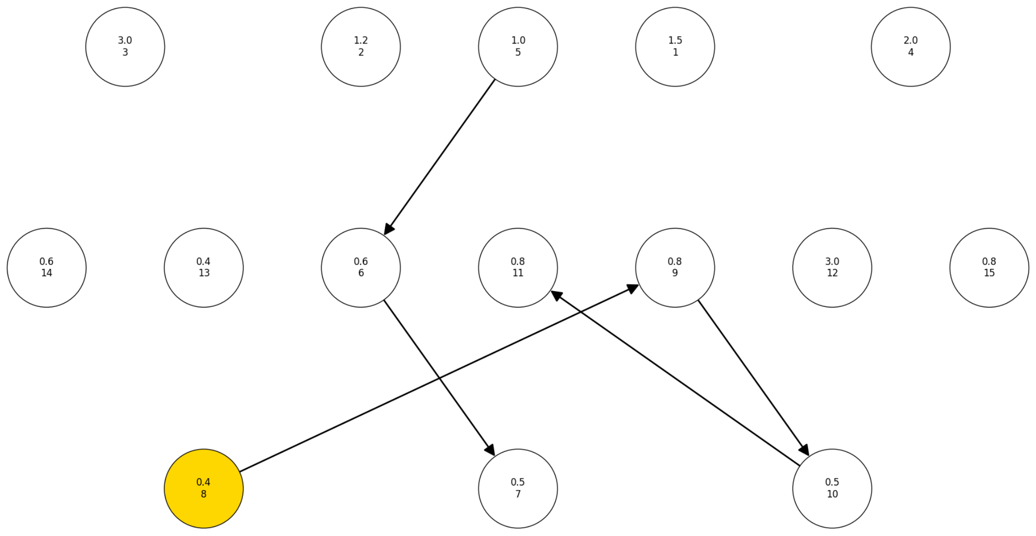
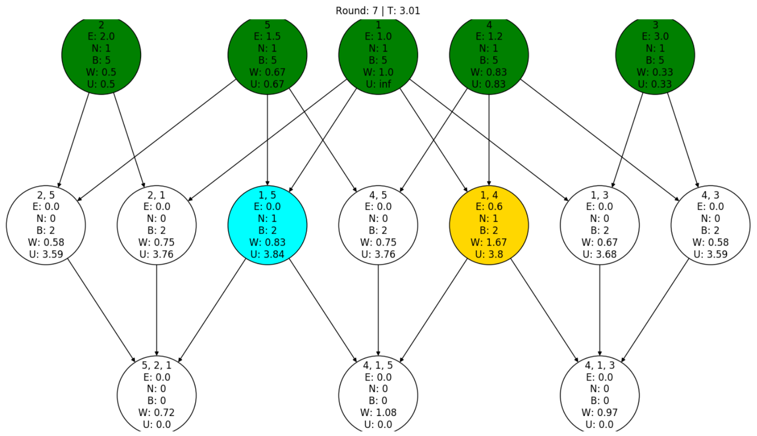
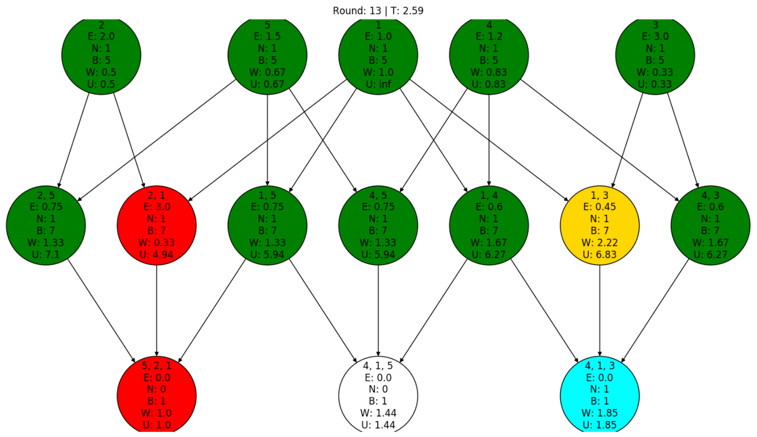


Figure 5.1: Final result of the iteration

The best node was picked in round 8, although it could not have been chosen much earlier, at most round 7. The different layers represent branches of the tree, and the arrows drawn between them represent two consecutive node selections, which did not happen on the same branch. This provides information regarding the role of the temperature in the scheme. In fact in this little example we have not been able to account for the importance of having a temperature dependent expression. The truth is this tree - created by demonstration purposes only - is too small to notice the importance of the annealing schedule. However as for the temperature itself, since it affects UCB values, having larger values throughout the process would tip the balance in favor of exploration over exploitation. In this example we would expect only a single passage between neighbouring branches. Let us raise the temperature values and rerun the program.


Round: 7 | T: 3.01
2 E: 2.0 N: 1 B: 5 W: 0.5 U: 0.5
5 E: 1.5 N: 1 B: 5 W: 0.67 U: 0.67
1 E: 1.0 N: 1 B: 5 W: 1.0 U: inf
4 E: 1.2 N: 1 B: 5 W: 0.83 U: 0.83
3 E: 3.0 N: 1 B: 5 W: 0.33 U: 0.33
2, 5 E: 0.0 N: 0 B: 2 W: 0.58 U: 3.59
2, 1 E: 0.0 N: 0 B: 2 W: 0.75 U: 3.76
1, 5 E: 0.0 N: 1 B: 2 W: 0.83 U: 3.84
4, 5 E: 0.0 N: 0 B: 2 W: 0.75 U: 3.76
1, 4 E: 0.6 N: 1 B: 2 W: 1.67 U: 3.8
1, 3 E: 0.0 N: 0 B: 2 W: 0.67 U: 3.68
4, 3 E: 0.0 N: 0 B: 2 W: 0.58 U: 3.59
5, 2, 1 E: 0.0 N: 0 B: 0 W: 0.72 U: 0.0
4, 1, 5 E: 0.0 N: 0 B: 0 W: 1.08 U: 0.0
4, 1, 3 E: 0.0 N: 0 B: 0 W: 0.97 U: 0.0

Deviation from the previous example occurs at round 7. Now instead of a child node on branch 3, the current branch was investigated further.


Round: 13 | T: 2.59
2 E: 2.0 N: 1 B: 5 W: 0.5 U: 0.5
5 E: 1.5 N: 1 B: 5 W: 0.67 U: 0.67
1 E: 1.0 N: 1 B: 5 W: 1.0 U: inf
4 E: 1.2 N: 1 B: 5 W: 0.83 U: 0.83
3 E: 3.0 N: 1 B: 5 W: 0.33 U: 0.33
2, 5 E: 0.75 N: 1 B: 7 W: 1.33 U: 7.1
2, 1 E: 3.0 N: 1 B: 7 W: 0.33 U: 4.94
1, 5 E: 0.75 N: 1 B: 7 W: 1.33 U: 5.94
4, 5 E: 0.75 N: 1 B: 7 W: 1.33 U: 5.94
1, 4 E: 0.6 N: 1 B: 7 W: 1.67 U: 6.27
1, 3 E: 0.45 N: 1 B: 7 W: 2.22 U: 6.83
4, 3 E: 0.6 N: 1 B: 7 W: 1.67 U: 6.27
5, 2, 1 E: 0.0 N: 0 B: 1 W: 1.0 U: 1.0
4, 1, 5 E: 0.0 N: 0 B: 1 W: 1.44 U: 1.44
4, 1, 3 E: 0.0 N: 1 B: 1 W: 1.85 U: 1.85

The algorithm just went from one extreme to the other, this time there has been no exploitation at all. The branch on level 3 was beginning to be discovered only when the upper branch had been entirely explored.

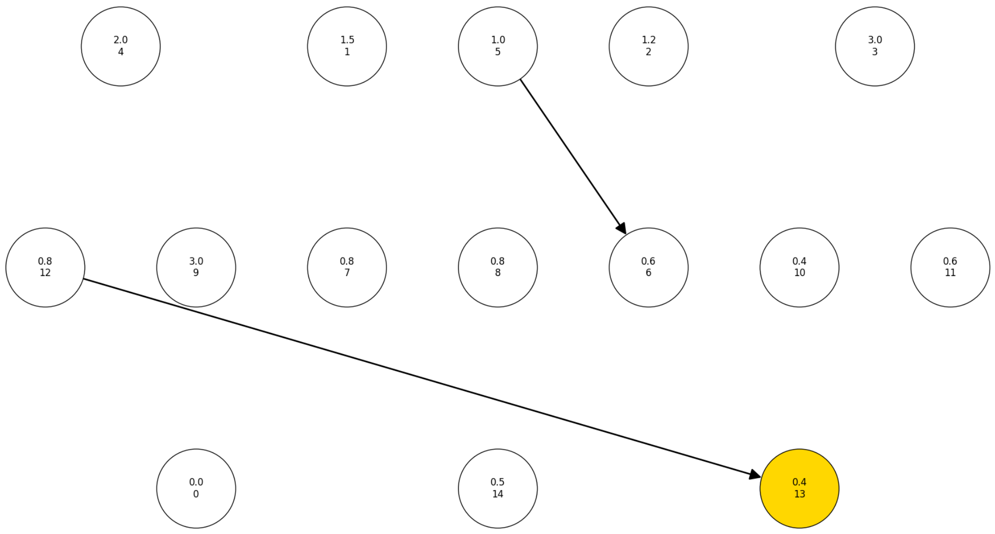


Figure 5.2: Second iteration with higher temperatures

The final result is shown in Fig. 5.2. A great decay in performance can be observed in terms of the order of node visits. Instead of round 8, the algorithm proposed the best candidate in round 13.

In spite of the extremely simple setup, some features of the algorithm could be captured even on this walk-through example. When the tree is much larger, both in terms of length and width, it is easy to imagine how much they could influence the paths taken by the algorithm. While the propagation of new information and the temperature control are undoubtedly important parameters, playout and the subsequent energy calculation play a crucial role as well in the machinery. This was however not demonstrated here.

Chapter 6

Conclusions and future prospects

At the beginning of the project we were faced with an exciting problem regarding the utilization of value jumps and the optimization of item selections in an online marketplace. Upon laying down the mathematical foundations of the problem, it was apparent that it does not belong to a unique category of optimization tasks. After spending some time investigating the possibility of the linear relaxation of an integer programming approach, we concluded that a shift of focus towards discrete solutions and metaheuristics would be more profitable. In fact we had the knowledge about the domain structure of the state space in our possession once the basis of states has been defined. With that in mind, we proposed two commonly known algorithms: simulated annealing and Monte-Carlo tree search. By combining the two ideas, we can control the exploration-exploitation ratio in time. This suits better a domain where the hierarchical structure is partnered up with an increasing importance for exploitation. If we forget about the rest of the algorithm for a moment, the applicability of SA-MCTS is not limited to online marketplaces. Whenever there are two distinct principal directions in the domain structure of states and connections between them can provide a way for the propagation of energies, a customized SA-MCTS could be applied for an optimization task. The ability for customization originates in the soul of metaheuristics. In our case, combining the parameters of SA and MCTS, we end up with a handful of adjustable variables. This includes the choice for energy and node value, the exact form of the exploration term, directions and methods for information propagation, identification of halting points, initial temperature and the annealing schedule. Even though some of these parameters were fixed in our case, they can be modified if it is necessary.

Previously we set goals for investigating large trees, and as mentioned before we have a flexibility to choose parameter values. Because of these and the complexity of the task there is still some work to do. Especially the initial temperature and the annealing schedule have been so far the victim of negligence, and a thorough analysis is required to fine-tune them. The task was initially defined with a constraint on running time, which definitely affects the correct temperature range. A benchmark method could be some genetic-type algorithm (GA), which can also be helpful in similar problems. Thus comparing results achieved by SA-MCTS and GA can provide a way to evaluate our algorithm before launching it in production.

Apart from the technical improvements, there is certainly some space for further development regarding the underlying theory. Nevertheless we are quite hopeful that we are on the right track to create something very useful in the world of online marketplaces.

Bibliography

[1] Jiří Matoušek and Bernd Gärtner. Integer programming and LP relaxation. In *Universitext*, pages 29–40. Springer Berlin Heidelberg.

[2] Arthur M. Geoffrion. Lagrangian relaxation for integer programming. In *50 Years of Integer Programming 1958-2008*, pages 243–281. Springer Berlin Heidelberg, November 2009.

[3] G. Dantzig, R. Fulkerson, and S. Johnson. Solution of a large-scale traveling-salesman problem. *Journal of the Operations Research Society of America*, 2(4):393–410, November 1954.

[4] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[5] V. Černý. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51, January 1985.

[6] M.K. Banerjee. 2.1.5 annealing. In *Comprehensive Materials Finishing*, pages 20–27. Elsevier, 2017.

[7] Saed Alizamir, Steffen Rebennack, and Panos M. Improving the neighborhood selection strategy in simulated annealing using the optimal stopping problem. In *Simulated Annealing*. InTech, September 2008.

[8] V. Granville, M. Krivanek, and J.-P. Rasson. Simulated annealing: a proof of convergence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(6):652–656, June 1994.

[9] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by

fast computing machines. *The Journal of Chemical Physics*, 21(6):1087–1092, June 1953.

[10] W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, April 1970.

[11] L. Ingber. Very fast simulated re-annealing. *Mathematical and Computer Modelling*, 12(8):967–973, 1989.

[12] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *In: Proceedings Computers and Games 2006*. Springer-Verlag, 2006.

[13] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[14] Jonathan Rubin and Ian Watson. Computer poker: A review. *Artificial Intelligence*, 175(5-6):958–987, April 2011.

[15] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[16] Tom Vodopivec, Spyridon Samothrakis, and Branko Šter. On monte carlo tree search and reinforcement learning. *J. Artif. Int. Res.*, 60(1):881–936, September 2017.

[17] Bernd Brügmann. Monte carlo go. Technical report, Physics Department, Syracuse University, 1993.

[18] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Lecture Notes in Computer Science*, pages 282–293. Springer Berlin Heidelberg, 2006.

[19] Guillaume Chaslot, Mark Winands, H. Herik, Jos Uiterwijk, and Bruno Bouzy. Progressive strategies for monte-carlo tree search. *New Mathematics and Natural Computation*, 04:343–357, 11 2008.

[20] Kai Xiong and Hong Jiang. Simulated annealing monte carlo tree search for large POMDPs. In *2014 Sixth International Conference on Intelligent Human-Machine Systems and Cybernetics*. IEEE, August 2014.

[21] Ben Ruijl, Jos Vermaseren, Aske Plaat, and H. Jaap van den Herik. Combining simulated annealing and monte carlo tree search for expression simplification. *CoRR*, abs/1312.0841, 2013.

[22] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497, July 1960.