Speeding up profile steering algorithms with caching for buffer type devices

Niels Overkamp
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
n.m.overkamp@student.utwente.nl

ABSTRACT

To enable the electricity grid to handle upcoming or present changes such as increasing penetration of electric vehicles and photovoltaics, a different approach of matching supply with demand is required. This approach must minimise local and global overproduction and shortages, and due to the increasingly distributed nature of both production and consumption, it must be scalable. One such alternative called profile steering iteratively and hierarchically communicates with consuming and producing devices to obtain a desired energy profile. It delegates the scheduling problem to these devices. Two important controllable devices for profile steering are electric vehicles and batteries. Research has been conducted into the theoretical optimisation of the algorithms to be run on these devices. Caching might provide a significant decrease in the execution time of these algorithms. In this research three algorithms are presented and their performance is measured. We show that these algorithms use half of the execution time of the original algorithm when the number of relevant devices is more than 9 and that the Full cache algorithm requires half of the execution time for any number of relevant devices.

1. INTRODUCTION

There is consensus among 90% to 100% of experts that global warming is caused by humans [1]. Countries are setting ambitious goals to reduce carbon emissions, such as the Paris Agreement [5] and renewable energy is providing a larger share of our energy consumption every year [4]. Renewable energy plays a big role in our political landscape and personal lives, but the current electricity grid is not yet able to accommodate this shift towards renewable energy.

Renewable energy is largely produced using energy sources outside of our control, such as the wind and the sun. This lack of control means that renewable energy is not as flexible as carbon based electricity generators that do not rely on external factors. This is a problem since also the consumption of electricity is primarily inflexible and uncontrollable using current methods. When both the production and consumption are uncontrollable matching supply with demand will become impossible and overproduction and shortage can occur in the electricity grid.

31th Twente Student Conference on IT Jul. 5$^{\text{th}}$, 2019, Enschede, The Netherlands.

Individuals and businesses are also increasingly using photovoltaics (PV) for personal energy generation. On moments of high production and low consumption the energy system of this consumer can become an energy producer. At first this does not seem to be problematic, however, the current electricity infrastructure is not build for this irregular upstream flow[8]. Traditionally energy production is centralised with a few large scale electricity generators distributing downstream to the consumers. Rooftop PV is in contrast very decentralised, causing local overproduction which cannot be easily matched with a consumer.

A potential solution to both these problems is to have more control of the consumption side. Consuming less energy when production is low and more when production is high has the potential to solve both problems mentioned above. Overproduction and shortage can be resolved by increasing or decreasing consumption at the right moments. When this consumption control allows for control of individual households or neighbourhoods, local extremes can be countered locally. One branch of this Decentralised Energy Management (DEM) functions by sending steering signals to appliances such that they can alter their behaviour. Such a system is not entirely novel. Certain regions have implemented or are planning to implement a system where the energy price is dependent on the short term demand and supply (i.e. the Netherlands [7]), motivating the consumption to follow the production.

But there are other methods to steer the consumption, such as profile steering. This method communicates with the consuming devices and negotiates an optimal schedule. On a regular time interval a controller requests an energy profile from all connected devices, constructs an overall energy profile and then attempts to improve this profile. It does not attempt any optimisation itself, but requests the devices to optimise the profile towards a more desired profile by changing their schedule. The controller selects the proposition which improves the profile the most and requests the device to apply this change. This process repeats until the improvements are below a certain threshold. The load is hereby distributed over the individual devices.

This method outperforms the variable pricing mentioned earlier. Van der Klauw et al. showed that this method can create a flatter energy profile than when using price steering when applied to the charging of electric vehicles [9][10].

This paper aims to lower the execution time of the buffer planning algorithm, the algorithm for making the charging planning for a pure buffer type device. This would reduce the required computation resources of the controller and thus making it cheaper to produce and use.

First the purpose and working of the algorithm to be op-

timised will be explained, followed by a short explanation on how caching could improve the execution time. We will then give a more in depth explanation of the aim of this research. The core of this paper is the Methodology explaining the workings and motivation behind the caching algorithms that could improve the BP algorithm. Finally we conclude with the setup of a simulation to validate these algorithms.

2. BACKGROUND

2.1 Buffer Planning

Below follows a short explanation of the algorithm for making the charging planning for a pure buffer type device which is the focus of this research. For a more in depth explanation the reader is referred to van der Klauw et al. [9].

Profile steering, as explained in section 1, is an approach where a controller iteratively requests a planning of the electricity usage from connected devices. It provides these devices with an objective function and these devices then run an algorithm to determine their improved planning. These algorithms have been developed for many different devices, but we will be looking at so called buffer type devices [6]. Buffer type devices can store energy such as heat or electricity. Within buffer type devices there is a distinction between pure storage devices and devices that require storage for a different use. A battery is a pure storage device, whereas an electric vehicles is a non-pure storage device. For both these sub-types an algorithm has been developed by van der Klauw et al [9].

For the algorithm to charge a non pure storage device it is assumed that they only draw energy from the grid and do not discharge back to the grid. van der Klauw et al developed two algorithms for planning these devices but their specifics are not relevant for this research. What is relevant is that these algorithms produce an optimal planning given an objective function, a charging rate constraint and the state of charge (SoC) to be reached at the end of the time horizon. It is noted that the planning is a vector over the time set to be planned on, with the energy to be used during a certain interval as elements. The objective function can be represented as a vector too, with a function for every interval. Applying these function to to planning results in a vector of values. The objective function is to minimise the sum of the values in this vector.

The algorithm for a pure storage device is different because while it can discharge back to the grid, it does not have a SoC to reach but it does have a upper and lower limit of its SoC. We will call this algorithm the buffer planning algorithm, or BP for short. Note that this algorithm is called optimal Buffer Charging (optBC) in van der Klauws research. The upper and lower limit are a vector like the other arguments to this algorithm. BP uses the algorithm for non pure storage devices first to obtain a candidate solution by leaving out the upper and lower SoC limits and by setting the SoC to be reached to be the upper bound value at the time horizon. If this solution does not fall within the upper and lower bounds it finds the interval k where these bounds are violated maximally. The algorithm now splits the problem into two sub-problems at interval k. The objective functions and the constraints are split at this interval and BP is called twice recursively with these sub-problems. The result of these two calls is combined and thus a solution is found.

2.2 Caching

The algorithms running on these pure storage devices have the potential to be made more efficient time-wise with a caching method. In an iteration of the controller the objective function changes on an interval of the time horizon. When this interval is smaller than the time horizon, a part of the objective function is unchanged. If k is unchanged too then some of the sub-problems will be identical to sub-problems encountered in the previous iteration. Caching these sub-problems and their results could thus lead to a decrease of the execution time of this algorithm.

3. PROBLEM STATEMENT

BP is required to be efficient in computation time, for it is made to be run on controllers of the consuming devices which typically are devices with limited computing resources. van der Klauw has conducted research on the worst case time complexity of these algorithms [9] but no research has been conducted to prune the recursion tree by storing previously obtained results. This research aims to contribute by assessing several caching approaches on their performance in computation time.

3.1 Research Questions

How do different caching algorithms affect the computation time performance of the BP algorithm?

A What are the possible caching algorithms that can be used to potentially improve the run time of the BP algorithm?

B What are the worst case theoretical time complexities of these algorithms?

C What is the execution time of these algorithms when simulating a group of households of varying sizes?

4. METHODOLOGY

The problems the BP algorithms solves can contain sub-problems that have been solved in a previous iteration of the problem as explained in subsection 2.2. We will call these: sequential repetitions of sub-problems. However, sequential repetitions of the full problem have been observed too. The reason for these repetitions is unknown to the author. Moreover, non-sequential repetitions of the full problem or of sub-problems have been observed as well, most likely simply due to chance.

Eliminating these three types of repetition could accelerate the execution of the BP algorithm. For this purpose three algorithms were constructed. To The first is called "Toplevel 1-cache", the second "Binary tree cache" and the last "Full cache". A brief overview of the functionality and implementation of these algorithms is discusses below. Pseudo code of these algorithms is included in Appendix A.

The Toplevel 1-cache simply stores the arguments with which BP was called by the device together with the result following from those arguments. If in the next call the arguments are identical it returns the stored result. Note that this is only done at the call to BP by the device, and not the subsequent recursive calls to BP. Therefore, this algorithm eliminates sequential repetitions of the full problem.

The Binary tree cache functions similarly, but it adds the functionality of caching sub-problems. This is achieved by passing a binary tree along to the recursive calls. This

Table 1. Theoretical time complexities of caching algorithms

	Miss	Hit
Toplevel 1-cache	$O(1)$	$O(1)$
Binary tree cache	$O(n)$	$O(1)$
Full cache - Best case	$O(n)$	$O(1)$
Full cache - Worst case	$O(nm)$	$O(m)$

binary tree represents the recursive calls of a previous execution of BP. The first call to BP receives the full tree, with as root node the arguments of the previous call and its result. The left sub-tree corresponds to the first sub-problem and is passed on to the first recursive call to BP. The root of this sub-tree corresponds to the arguments and result of the first recursive call of the previous call to BP. Every call the arguments of the call are compared to the arguments in the value of the current node. If these are equal the second value of the tuple is returned. The tree itself is a simple binary tree built using a class storing its value and its two children as references. This algorithm caches sequential repetitions of both the full problem and the sub-problems.

The Full cache stores all calls that a device makes to BP. It achieves this by constructing a hashmap from the arguments to the result and adding a argument-result pair to this hashmap at every call of BP. If a call to BP is made with arguments that are in this hashmap the algorithm returns the corresponding result. This algorithm thus caches all repetitions, sequential or non-sequential, full problem or sub-problem.

4.1 Complexity

The Full cache algorithm is expected to eliminate more repetitions than the Binary tree algorithm. The Binary tree algorithm in turn is expected to eliminate more repetitions than the Toplevel 1-cache. However their theoretical worst case complexity, displayed in Table 1, are ordered in the opposite order.

Table 1 contains the complexity of the caching algorithm of a call of the BP algorithm by the device. n signifies the number of recursive calls within a call by the device and m is the amount of *unique* recursive calls made prior to this call of BP. m is therefore equal to the number of entries in the hashmap. The complexity is split up in "Miss" and "Hit", the first being the problem and its subsequent recursive sub-problems not being in the cache, the latter being the top level problem being in the cache. Full cache is split up in two different categories because of the way hashmaps are implemented in CPython. The best case when the hashes of the keys of the hashmap are all unique the complexity of a lookup is $O(1)$. In the worst case the hashes of the keys are all identical and the complexity is identical to a search in a list, $O(m)$ with m being the number of entries in the hashmap.

Toplevel 1-cache is only called at the top level of the algorithm and not during the recursion steps, therefore its complexity is of order $O(1)$ for both the "Miss" and the "Hit" case. The Binary tree cache is called during every recursion steps, but does not add any additional complexity besides that. Because the tree is built using references, passing the sub-trees onto the recursive calls has a complexity of $O(n)$. The comparing of the arguments to the value of the tree has a complexity of $O(n)$ as well resulting in a total complexity of $O(n)$. As explained before a lookup in hashmap has a complexity of $O(1)$ best case and $O(m)$ worst case. The full cache algorithm is called during every recursion step, and at every recursion step a lookup is done, resulting in a complexity of $O(n)$ or $O(nm)$.

5. RESULTS

To measure the execution time improvements made by the caching algorithms we apply them to a simulation of a typical use case of the BP algorithm. This simulation simulates the electricity usages of a group of households by simulating the behaviour of its residents and their devices. This simulation uses the DEMKit software developed by Hoogsteen et al. [6] [3] to simulate this neighbourhood of households. To generate a load profiles for these households the software ALPG developed by Hoogsteen et al. [6] [2] is used. The households in this neighbourhood each have a household controller. This controller sends Profile Steering signals to the devices in the household. These household controllers are connected to a single higher level controller steering the households using the Profile Steering approach.

Some of the simulation software in DEMKit has been changed. The algorithms simulating the devices that use the BP algorithm were adapted slightly such that the BP algorithm was able to keep a state. This state was only kept within this device and was not shared with the other devices. Furthermore, calls were added to the `os` library to access the environment variables to allow for external control to change the number of households that were simulated and to select the type of caching algorithm that was used.

The timings were measured by the algorithms simulating the buffer type devices using the standard python library `time`. The current time was measured before the BP algorithm was called and after the BP algorithm returned. The difference between these measured times was summed up for every call to BP. This measured time therefore includes the computation time the caching algorithm has added and the computation time it has saved.

Besides the three algorithms and the original algorithm, a fifth algorithm was included in the measurements. The implementation of the original algorithm was slightly rewritten without modifying the functionality of the algorithm. This was done to be able to implement the caching algorithms. Because this rewrite might have an effect on the performance of the algorithm, its performance was included in the measurements to have a better control measurement. This rewritten implementation of the BP algorithm is called Rewritten in the measurements.

The simulation was run twice to increase the accuracy of the results. This number is still rather low due to to time constraints.

Measurements were made for every algorithm and for a number of households ranging from 2 up to and including 20. A household however can contain zero, one or two buffer type devices. A buffer type device runs the BP algorithm and thus increases the measured timing. Therefore using the number of devices as variable is a better representation of the size of the problem than the number of houses. The measurements can be seen in 1. In Figure 2 this same execution time is represented as a percentage of the rewritten algorithm.

The Full cache algorithm uses 55% or less of the execution time of the Rewritten algorithm in all cases. The Binary tree algorithm uses 94% in its worst measured case, but when the number of devices is bigger than 9 it uses 52% at worst. Toplevel 1-cache requires at least 5% percent more execution time than the Binary tree and the Full

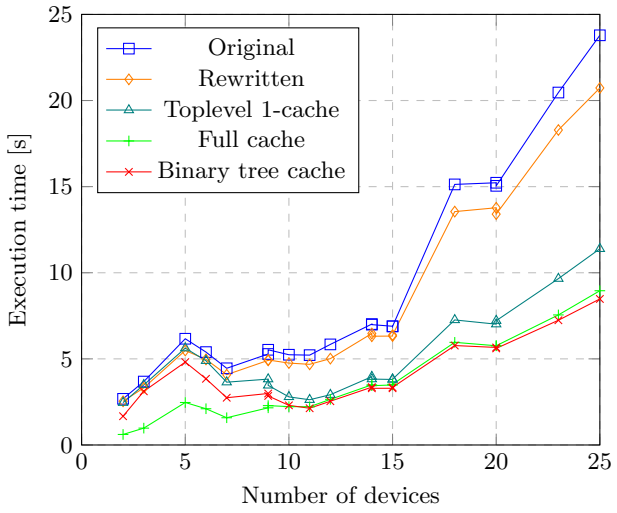

Figure 1. Execution time of BP for different caching approaches

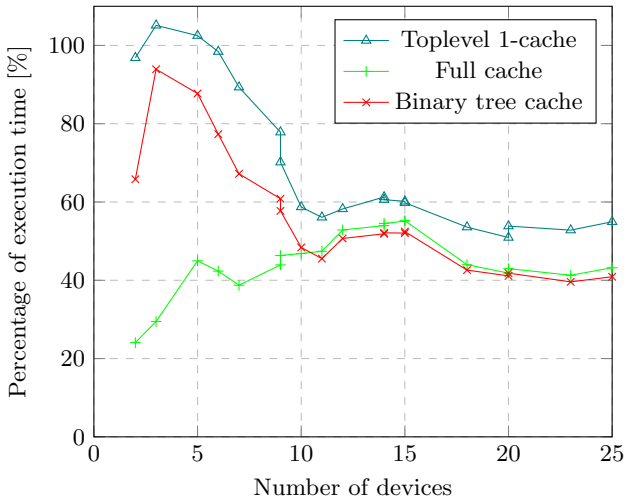

Figure 2. Percentage of execution time of Rewritten

cache algorithm. The execution time percentages of the Full cache algorithm and the Binary tree algorithm when the number of devices is larger than 9 differ 3% or less.

6. CONCLUSION

In this paper we first discussed three different caching algorithms for the BP algorithm eliminating three different types of repetition. The Toplevel 1-cache algorithm covers only sequential repetitions of the full problem, but has a theoretical complexity of $O(1)$. The Binary tree cache covers sequential repetitions of both sub-problems and full problems but has a complexity of $O(n)$. Lastly, the Full cache covers all repetitions but has a complexity of $O(nm)$.

The execution time of the BP algorithm with the different types of caching algorithms was measured. Looking at Figure 2 this is less than %50 percent for more than 9 devices. The Toplevel 1-cache algorithm has a worse performance in some measurements below 9 devices than the rewritten algorithm. The Full cache performs better than the rewritten algorithm, but only marginally for lower amounts of devices. The binary tree cache performs better at lower amounts of devices than at higher.

With these algorithms all using half of the execution time of the current BP algorithm for more than 9 pure buffer devices this approach can already significantly speed up simulations using this algorithm. With further research, see 7, the algorithms presented in this research could be used to reduce resources required to run this buffer planning algorithm on smart devices connected to profile steering network.

7. FUTURE WORK

Another import factor for algorithm running on devices with limited resources besides computation time is memory usage. Since the algorithms researched in this paper store significant different amounts of data, the memory usage might be too high to be practical for some of these algorithms. In a future research this would be an interesting metric to measure.

This research is also rather limited in the variation in the test cases. Every data point is produced using only two simulations. A higher accuracy would be obtained by running more simulations. The difference in computation time with different compositions of devices in the neighbourhood could also produce different results. Furthermore it would be interesting to see how these algorithms perform with more than 25 devices.

8. REFERENCES

[1] J. Cook, N. Oreskes, P. T. Doran, W. R. L. Anderegg, B. Verheggen, E. W. Maibach, J. S. Carlton, S. Lewandowsky, A. G. Skuce, S. A. Green, D. Nuccitelli, P. Jacobs, M. Richardson, B. Winkler, R. Painting, and K. Rice. Consensus on consensus: a synthesis of consensus estimates on human-caused global warming. *Environmental Research Letters*, 11(4):048002, apr 2016. DOI: 10.1088/1748-9326/11/4/048002.

[2] EEMCS Energy University of Twente. Artificial load profile generator. *https://www.utwente.nl/en/eemcs/energy/profile_generator/*. Accessed: 2019-06-28.

[3] EEMCS Energy University of Twente. DEMKit. *https://www.utwente.nl/en/eemcs/energy/demkit*. Accessed: 2019-05-05.

[4] Eurostat. Eurostat your key to european statistics. *ec.europa.eu/eurostat/web/energy/data/main-tables*. Accessed: 2019-05-03.

[5] Framework Convention on Climate Change. Adoption of the Paris Agreement. Technical report, 2015. *unfccc.int/resource/docs/2015/cop21/eng/l09r01.pdf*. Accessed: 2019-05-05.

[6] G. Hoogsteen. *A cyber-physical systems perspective on decentralized energy management*. PhD thesis, University of Twente, Enschede, The Netherlands, dec 2017. DOI: 10.3990/1.9789036544320.

[7] H. Kamp. Tijdpad uitrol slimme meter en dynamische leveringstarieven. *https://www.tweedekamer.nl/kamerstukken/brieven_regering/detail?id=2016Z04632&did=2016D09547*. Accessed: 2019-05-09.

[8] M. Karimi, H. Mokhlis, K. Naidu, S. Uddin, and A. Bakar. Photovoltaic penetration issues and impacts in distribution network - A review. *Renewable and Sustainable Energy Reviews*, 53:594–605, jan 2016. DOI: 10.1016/J.RSER.2015.08.042.

[9] T. van der Klauw. *Decentralized energy management with profile steering: resource allocation problems in energy management*. PhD thesis, 2017. DOI: 10.3990/1.9789036543019.

[10] T. Van Der Klauw, M. E. Gerards, G. J. Smit, and J. L. Hurink. Optimal scheduling of electrical vehicle charging under two types of steering signals. In *IEEE PES Innovative Smart Grid Technologies Conference Europe*, volume 2015-Janua, pages 1–6. IEEE, oct 2015. DOI: 10.1109/ISGTEurope.2014.7028746.

APPENDIX

A. CODE AND PSEUDO CODE

A.1 Toplevel 1-cache

A.1.1 Initialisation

```
self.previous_arg_key = None
self.previous_res = None
```

A.1.2 Before the non recursive call

```
arg_key = immutable copy of arguments
if arg_key == self.previous_arg_key:
    return list(self.previous_res)
else:
    self.previous_arg_key = arg_key
```

A.1.3 After the non recursive call

```
self.previous_res = tuple(result)
```

A.2 Binary tree cache

A.2.1 Initialisation

```
self.previous_call_graph = BinaryTree((None, None))
```

A.2.2 Before every recursive call

extra arg: `previous_call_graph`

```
prev_key, prev_res = previous_call_graph.get_val()
if arg_key == prev_key:
    return list(prev_res)
```

A.2.3 After every recursive call with no hit

extra arg: `previous_call_graph`

```
previous_call_graph.set_val((arg_key, tuple(result))
```

A.3 Full cache

A.3.1 Initialisation

```
self.memory = dict()
```

A.3.2 Before every recursive call

```
arg_key = immutable copy of arguments
if arg_key in self.memory:
    return list(self.memory[arg_key])
```

A.3.3 After every recursive call with no hit

```
self.memory[arg_key] = tuple(result)
```

A.4 BinaryTree

```
class BinaryTree:
    _val: T

    _left = None
    _right = None

    def __init__(self, val: T):
        self._val = val

    def get_val(self):
        return self._val

    def set_val(self, val: T):
        self._val = val

    def get_left(self):
        return self._left

    def get_right(self):
        return self._right

    def set_left(self, tree):
        self._left = tree

    def set_right(self, tree):
        self._right = tree
```