Increasing profit from fees of cryptocurrencies

Noa Oved noaoved3@gmail.com
Besart Dollma besi7dollma@gmail.com
Advisor: Itay Tsabary

September 23, 2018

1 Introduction

When cryptocurrencies are transfered from one party to another transactions are created. These transactions are stored in a memory pool until a miner chooses to validate them by including the transactions on the next block of the blockchain. Each transaction has its fee, size and a list of other transactions upon which it is dependent. In order to validate a transaction, each of the other transactions it is dependent upon needs to be validated as well. The miner receives as profit the fee of each transaction he validated, hence he strives to maximize the sum of fees of the transactions he includes in the block. This needs to be done under the size contraint of the block and under the dependency constraints of the transactions.
This problem is known as the dependency knapsack problem. The version without the dependencies is known as the knapsack problem. The decision problem forms of both problems are NP-complete. In this paper we implemented four algorithms which solve the knapsack problem, two of which are optimal and infesiable for real data, and two approximations. Since the optimal solutions are infesiable, we implemented only the approximations for the dependency knapsack. We developed an incremental solution for one of the approximations.
We compared between all the solutions on mock and real data. The greedy approach is always the fastest and on real data it yielded the same results as the other approximation. The incremental solution of the algorithm shortens the running time in most cases, hence we would recommend using it in real applications.

2 Cryptocurrencies

2.1 Cryptocurrencies

A cryptocurrency is digital asset designed to work as a medium of exchange that uses strong cryptography to secure financial transactions, control the creation of additional units, and verify the transfer of assets. Cryptocurrency is a kind of digital currency, virtual currency or alternative currency. Cryptocurrencies use decentralized control as opposed to centralized electronic money and central banking systems. The decentralized control of each cryptocurrency works through distributed ledger technology, typically a blockchain, that serves as a public financial transaction database. Specifically, we will focus on Bitcoin.

2.2 Blockchain

The validity of each cryptocurrency's coins is provided by a blockchain. A blockchain is a continuously growing list of records, called blocks, which are linked and secured using cryptography. Each block typically contains a hash pointer as a link to a previous block, a timestamp and transaction data. By design, blockchains are inherently resistant to modification of the data. It is an open, distributed ledger that can record transactions between two parties efficiently and in a verifiable and permanent way. For use as a distributed ledger, a blockchain is typically managed by a peer-to-peer network collectively adhering to a protocol for validating new blocks. Once recorded, the data in any given block cannot be altered retroactively without the alteration of all subsequent blocks, which requires collusion of the network majority. Every 10 minutes on average, a new block is added to the blockchain. A real blockchain block has a size of 1 MB.

2.3 Transactions

When transferring cryptocurrency from a party to another party a transaction is created and added to the Mempool (explained below). Among other things, transactions contain:

ID	672e2c74d410d0adb9884d62a5b689902aee6c48e79c180c5e875155098c9a39
Fee	0.00015820 BTC
Size	224 bytes
Depends	[]

1. The ID is unique per transaction, generated by SHA-256 (a hash function).
2. Transaction fees are a fee that spenders may include in any Bitcoin transaction.
3. The size contains the size of the transaction in bytes.
4. Depends is a list that contains ID's of transcations on which this transaction is dependent upon. When a transaction is included in the block, all the transcations in Depends have to be included as well.

2.4 Mempool

Mempool (a compound of two words, 'Memory' and 'Pool') is a pool of memorized, held data. The data that is being stored on the Mempool are unconfirmed transactions that are currently stuck on the network.

2.5 Miners

In cryptocurrency networks, mining is a validation of transactions. For this effort, successful miners obtain new cryptocurrency as a reward. For each transcation the miner includes in a block, he collects its fee. Hence, the miner's motivation is to maximize the sum of the fees of the transactions that he includes in the block, under the size and dependency constraints, by selecting the optimal set of transactions. This set can be obtained by solving the dependency knapsack problem which we will discuss in the next section.

3 The knapsack and the dependency knapsack problems

3.1 The knapsack problem

The knapsack problem or rucksack problem is a problem in combinatorial optimization: Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items. The knapsack problem is hence a sub-problem of the dependency knapsack problem, as the dependencies only make the problem harder. As such the decision problem form of the dependency knapsack problem is NP-complete as well. The decision problem form of the knapsack problem is NP-complete, thus there is no known algorithm both correct and fast (polynomial-time).

3.1.1 Definition

We are given a block of size W and n transactions $\{a_1, a_2, ..., a_n\}$. Each transaction a_i has size $s_i > 0$ and fee $f_i \geq 0$. We are to find $I \subseteq [n]$ such that:

$$I = \arg\max_{J \subseteq [n]} \{\sum_{j \in J} f_j\} \; s.t \; \sum_{j \in J} s_j \leq W$$

3.2 The dependency knapsack problem

As mentioned in the introduction of the paper, transactions are dependent upon other transactions. In order for us to collect the transaction, we must collect with it all the transactions it is dependent on. As mentioned before, the dependency knapsack is a generalization of the knapsack problem, and as such it is harder to solve. We will use a directed graph to visualize the dependencies.

3.2.1 Definition

We define the dependency knapsack problem as follows: the input is a set of transactions, each with a size and fee, a set of dependencies between the transactions, where no circular dependencies exist and a total knapsack capacity of W. The goal is to find a set of transactions with a total weight that does not exceed W and where dependencies constraints are preserved. We will treat the input as a directed acyclic graph $G = (V, E)$. Each node v represents a transaction that has fee $f(v) \geq 0$ and size $s(v) > 0$. Each edge (i, j) represents that transaction j is dependent on transaction i. Transaction j can be selected if transaction m is selected for all $(m, j) \in E$. The goal is to find $\bar{V} \subseteq V$ of transactions such that

$$\bar{V} = \arg\max_{J \subseteq V} \{\sum_{v \in J} f(v)\} \; s.t \; \sum_{v \in J} s(v) \leq W$$

and the dependency constraints are preserved $\forall v \in \bar{V}$.

4 Solutions

4.1 Solutions for the knapsack problem

We will discuss four possible solutions for the knapsack problem. The first two are optimal and the other two are approximations.

4.1.1 Exhaustive search

The trivial approach is to check all the possible subsets $J \subseteq [n]$ while saving the subset with the maximal profit seen so far. This approach is optimal but very slow as its running time is exponential, $O(2^n)$. The size of the mempool can be $n \approx 16000$, hence it would be infesiable.

4.1.2 Dynamic Programming

For this solution we work under the assumption that $s_1, s_2..., s_n, W \in \mathbb{N}$, which actually holds in real life as the size of each transaction is given in bytes. For each $k \in \{0, 1, ..., n\}$ and $w \in \{0, 1, ..., W\}$ define $F(k, w)$ to be the maximal profit when choosing from transactions $\{a_1, a_2, ..., a_k\}$ and the size of the block is w. The solution to the problem would then be $F(n, W)$. We will try to give a recursive function for $F(k, w)$. When looking at the transaction a_k:

- If we can't add this transaction to the block, $s_k > w$, then the optimal solution is $F(k-1, w)$.
- Otherwise, we can decide to either add the transaction a_k to the block or no.
  - If we decide not to add it, then the best solution would be $F(k-1, w)$.
  - If we decide to add it, then we can earn $f_k + F(k-1, w-s_k)$.

Therefore it holds that:

$$F(k,w) = \begin{cases} 0 & k = 0 \vee w = 0 \\ F(k-1,w) & w, k > 0 \wedge s_k > w \\ \max\{F(k-1,w), f_k + F(k-1, w-s_k)\} & w, k > 0 \wedge s_k \leq w \end{cases}$$

This approach is optimal and faster than exhaustive search. The complexity is $O(nW)$ space and time, hence it is pseudo-polynomial as only $\log(W)$ bits are needed to represent W.

4.1.3 Greedy approximation algorithm

As for most NP-complete problems, it may be enough to find workable solutions even if they are not optimal. Preferably, however, the approximation comes with a guarantee on the difference between the value of the solution found and the value of the optimal solution. The greedy approximation algorithm is such an algorithm. It sorts the transcations by $\frac{f_i}{s_i}$ ratio in descending order. Then going through the sorted transcations it adds to the block the transcations until there is no space left for the next transaction\. The greedy approximation algorithm doesn't provide us with the optimal solution but with a 2-approximation. It means that if the optimal solution is c^*, the algorithm will output a value greater than $\frac{c^*}{2}$ and smaller than c^*. The advantage of this approach is its runtime, as it is very fast with only $O(n \log n)$.

4.1.4 $(1+\epsilon)$ approximation algorithm

Differently from the previous algorithms, the input of the $(1+\epsilon)$ approximation algorithm includes furthermemore a value $0 \leq \epsilon \leq 1$. The $(1+\epsilon)$ approximation algorithm is a combination of two of the algorithms mentioned above. Denote by *greedySol* the solution value of the greedy approximation algorithm. Denote

$$a = \epsilon \cdot greedySol$$

$$V_a = \{a_i \mid f_i < a\}$$
$$V_a^C = \{a_i \mid f_i \geq a\}$$

The next steps are:

- For each subset $J \subseteq V_a^C$ such that $|J| \leq \frac{2}{\epsilon}$
  - Run the greedy approximation algorithm on V_a with block size $W' = W - \sum_{j \in J} s_j$ and denote the solution by I_J.
- Output $I_J \cup J$ with maximal profit.

The $(1+\epsilon)$ approximation algorithm is a $(1+\epsilon)$ approximation to the knapsack problem as the name suggests. It means that if the optimal solution is c^*, the algorithm will output a value greater than $\frac{c^*}{(1+\epsilon)}$ and smaller than c^*. The running time of this algorithm is $O(n^{1+\frac{2}{\epsilon}} \cdot \log n)$. One can pay attention to the following cases:

- If $\epsilon \to 0$, V_a^C would contain all the transactions and we receive the exhaustive search algorithm. $\frac{2}{\epsilon} \to \infty$ hence we would receive a non-polynomial solution.
- If $\epsilon \to 1$, V_a^C would contain all the transactions that their fees are bigger or equal than *greedySol*, therefore probably $V_a^C = \emptyset$, (if $V_a^C \neq \emptyset$ its elements won't be in the solution). Hence in this case we would receive the greedy approximation algorithm.

4.2 Solutions for the dependency knapsack problem

The two approximations can be adapted to solve the dependency knapsack problem. We will use a directed acyclic graph to visualize the dependencies. Each node represents a transaction and we will use the names node and transaction coherently.

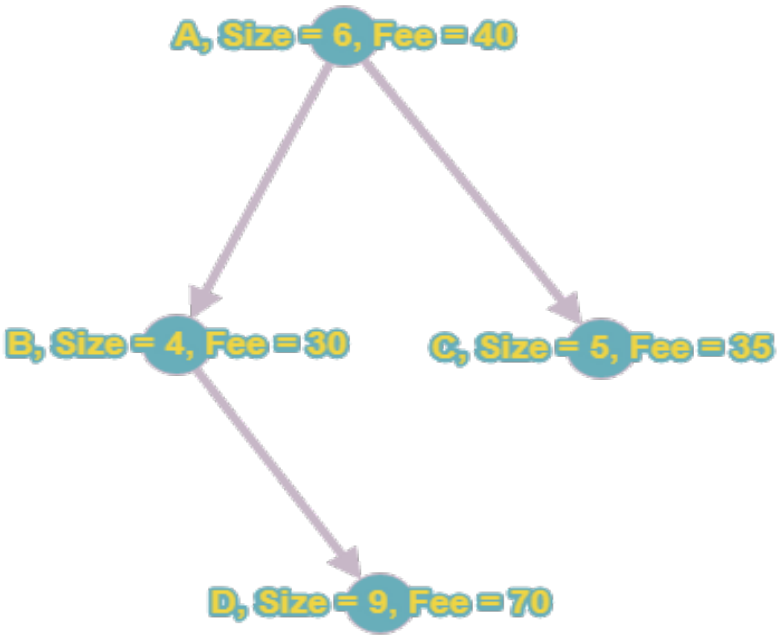


Figure 1: Example of a Knapsack with dependencies

We will use this example in order to explain the algorithms. In the example there are 4 transactions named, A, B, C, D. Suppose the knapsack size is $W = 11$. We pay attention that all transactions are dependent upon transaction A, therefore A will surely be in the solution.

4.2.1 Greedy approximation algorithm

The idea of the algorithm remains the same, pick the items with the best fee over size ratio, however this time we will need to preserve the dependency constraints. Therefore for each node (transaction), we create the set of its ancestors

$$Ancestor(v) = \{j \mid \text{there exists a path from } j \text{ to } v \text{ in } G\}$$

We notice that for all $v \in V$ it holds: $v \in Ancestor(v)$. The algorithm:

1. Calculate $Ancestor(v)$ for all $v \in V$.
2. Pick $Ancestor(v)$ with the maximal
$$\frac{\sum_{j \in Ancestor(v)} f(j)}{\sum_{j \in Ancestor(v)} s(j)}$$
ratio and add the transactions of the set in the knapsack.
3. Remove the transaction we just added in the knapsack from other sets and continue from 2 until we can't fit anything else in the knapsack.

In the example it holds that

$$Ancestor(D) = \{D, B, A\}, Ancestor(C) = \{C, A\},$$

$$Ancestor(B) = \{B, A\}, Ancestor(A) = \{A\}$$

and the respective ratios sorted in descending order are $R(D) = \frac{140}{19}$, $R(B) = \frac{70}{10}$, $R(C) = \frac{75}{11}$, $R(A) = \frac{40}{6}$. Since the size of the knapsack is 11, $Ancestor(D)$ can't be picked therefore the greedy approximation will pick $Ancestor(B)$ hence it will yield a solution of 70. The running time complexity of this algorithm is $O(n^3)$ where $n = |V|$ is the number of transactions.

4.2.2 $(1+\epsilon)$ approximation algorithm

The $(1+\epsilon)$ approximation algorithm for the dependency knapsack is very similar to the standard $(1+\epsilon)$ approximation algorithm we saw before in this paper with the adaption to dependencies using *Ancestors* as in the greedy approximation algorithm. Let us concentrate in the example. The greedy solution yielded 70, let $\epsilon = 0.1$. Hence it holds that

$$a = \epsilon \cdot greedySol = 7$$

therefore

$$V_a^C = \{A, B, C, D\}$$

and this means that the algorithm will check all the subsets of V_a^C since $\frac{2}{\epsilon} = 20$. The subset $\{A, C\}$ has the highest profit, 75 and therefore the algorithm will pick it. The running time complexity of this algorithm is $O(n^{3+\frac{2}{\epsilon}})$.

4.3 An incremental solution to the greedy approximation algorithm

Suppose at time t we have the solution of the dependency knapsack problem using the greedy approximation algorithm (maybe some other miner solved it). Now suppose at time $t+1$ we added some transactions to the mempool, and didn't remove any. Suppose also that the added transactions may be dependent on transactions from the previous sample but not vice versa. This is the case in the real life application. We want to explore the possibility of using the solution (the subset of the transactions) of a previous sample (time t) in order to solve the current sample (time $t+1$). First we pay attention that since the transactions from the previous sample can not be dependent on the transactions that were added in this sample, the $Ancestor(v)$ sets for all the transactions in the previous sample remain identical. The idea behind the incremental solution algorithm is:

1. Calculate $Ancestor(v)$ for all v.
2. Denote by
$$\alpha = \max_v \frac{\sum_{j \in Ancestor(v)} f_j}{\sum_{j \in Ancestor(v)} s_j}$$
the maximal ratio of fee over size of the transactions that weren't in the previous solution.
3. Add to the solution all the $Ancestor(\cdot)$ sets from the previous solution with a fee over size ratio bigger than α and remove them from the mempool. Use the greedy approximation algorithm on what is left on the mempool with the new size of the block.

Pay attention that the idea of the incremental solution is to improve the running time and not the solution value of the greedy approximation. The solution value should be the same in both algorithms, the standard greedy algorithm and the incremental solution algorithm.

5 Implementation

We implemented the algorithms using Python 3.7. We used graphs to represent the transactions and their dependencies. Graphs aren't build in in Python hence we used the networkx library. The implementation of the algorithms and some tests can be found in our github. Through out the tests, the effect of cache was disabled in order to measure the runtime properly. All tests were run on the same processor, hence one should consider the results relatively, since their exact values are dependent on the processor.

6 Results

6.1 The knapsack problem

We ran some tests on mock data with the algorithms we implemented for the knapsack problem.

Experiment I

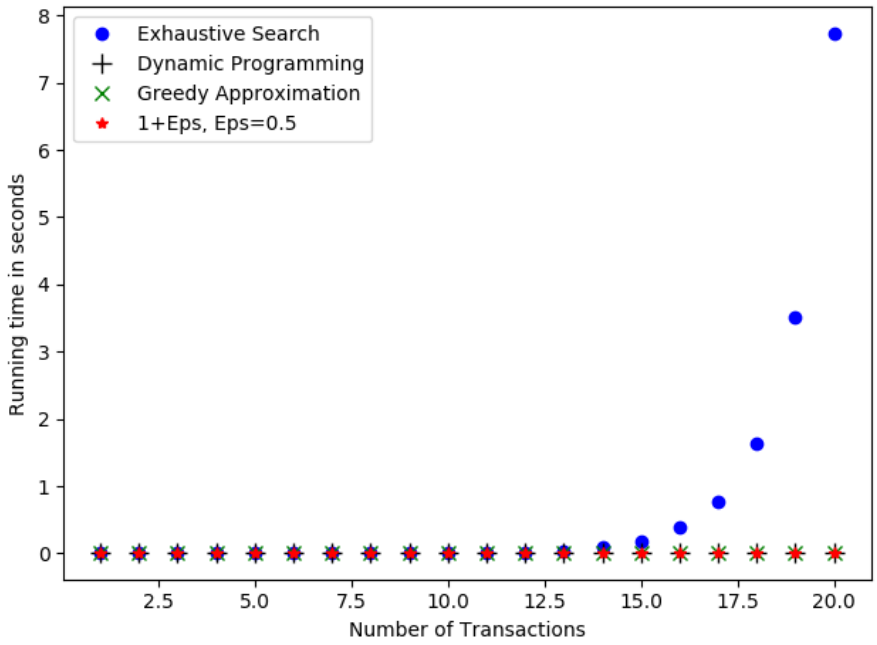


Figure 2: Runtime in seconds as a function of the number of transactions

For each $1 \leq n \leq 20$, we generated each time n random transactions a_i such that $1 \leq s_i, f_i \leq 100$ and $W = 1000$. For each number of transactions we calculated the runtime of each algorithm as the average of 50 random runs. Figure 2 describes the runtime in seconds as a function of the number of transactions. The graph shows that the exhaustive search runtime is $O(2^n)$ and it is not a feasible solution. For example it holds that for $n = 18$ the running time is ≈ 1.7 seconds, for $n = 19$ the running time is ≈ 3.5 seconds and for $n = 20$ it is higher than 7 seconds. The runtime of the other algorithms is indistinguishable and close to 0 for such small n.

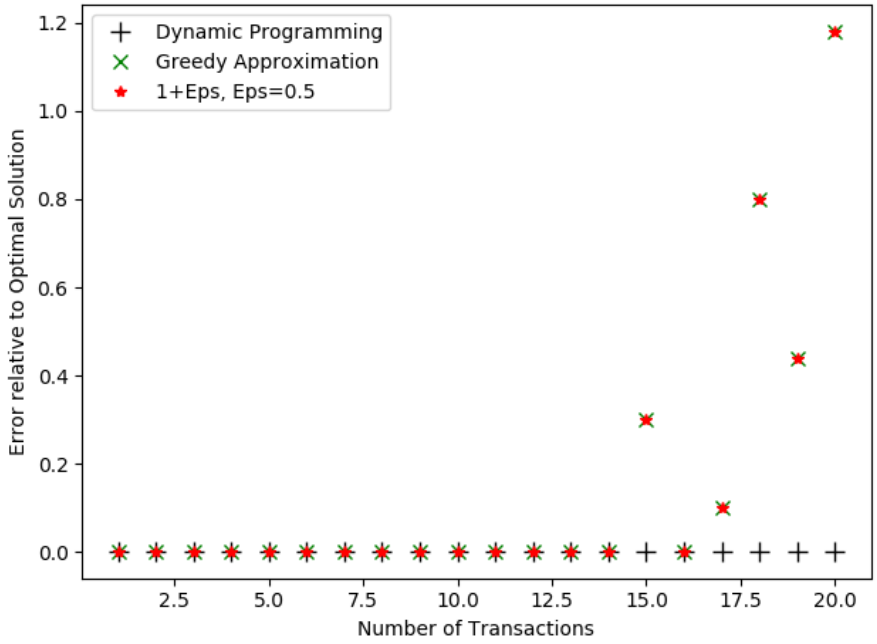


Figure 3: Relative error as a function of the number of transactions

Now we explore the quality of the solution from the previous test. We will compare the profit calculated from each algorithm relatively to the exhaustive search solution which is optimal, meaning the data points were calculated as the difference between the optimal value and the value obtained by each algorithm. As can be seen from Figure 3, the dynamic programming algorithm yields an optimal solution, while the greedy approximation and the $(1 + \epsilon)$ approximation in this case have the same error.

Experiment II

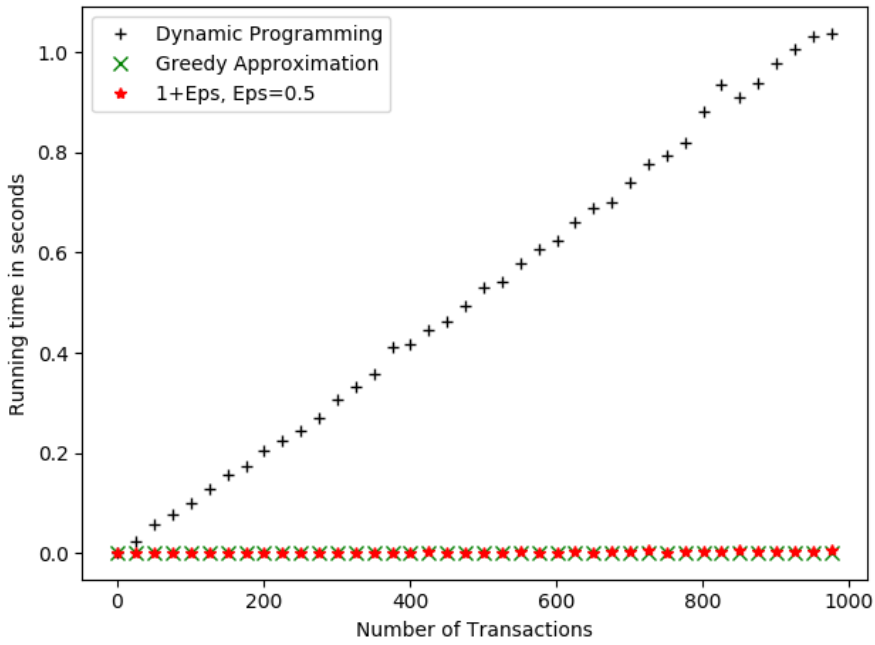


Figure 4: Runtime in seconds as a function of the number of transactions

But just how good the dynamic programming algorithm is? From the previous results, we got the optimal solution in no time. There are two major problems which make the dynamic programming algorithm infesiable. The first problem is the size of the mempool. As mentioned earlier the size of the memory pool can be up to $n \approx 16000$. We ran the tests from the previous experiment (same parameters as before) and increased the number of transactions up to $n = 1000$. As it can be seen in Figure 4, the runtime of the dynamic programming algorithm is considerably greater than the runtime of the approximations. To solve an instance of $n = 1000$ transactions it takes on average 1 sec, which might not seem to be a big problem but it will be, as we will soon see. It is worth to mention that the graph confirms the linear dependency of runtime as a function of the number of transactions, $O(nW)$.

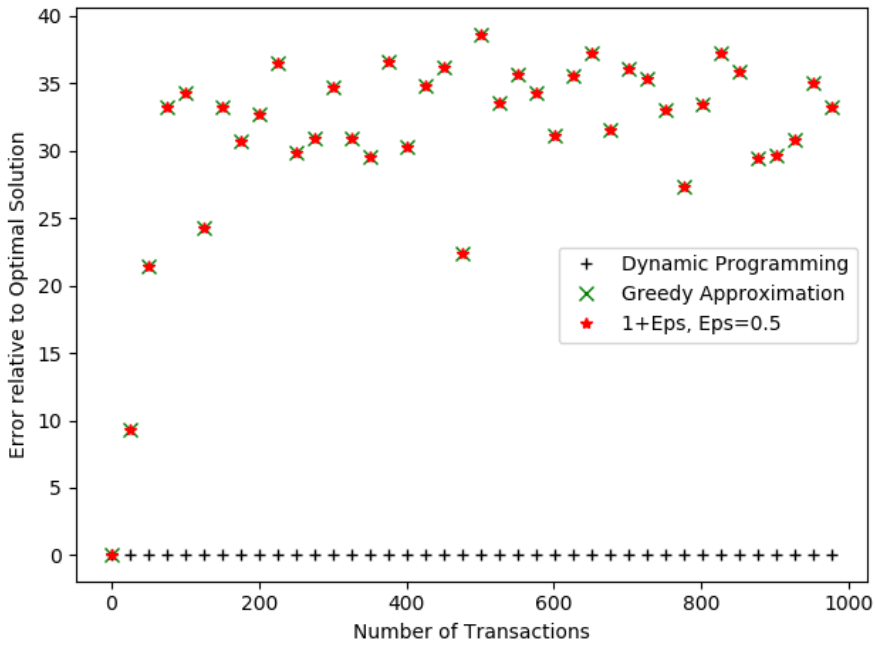
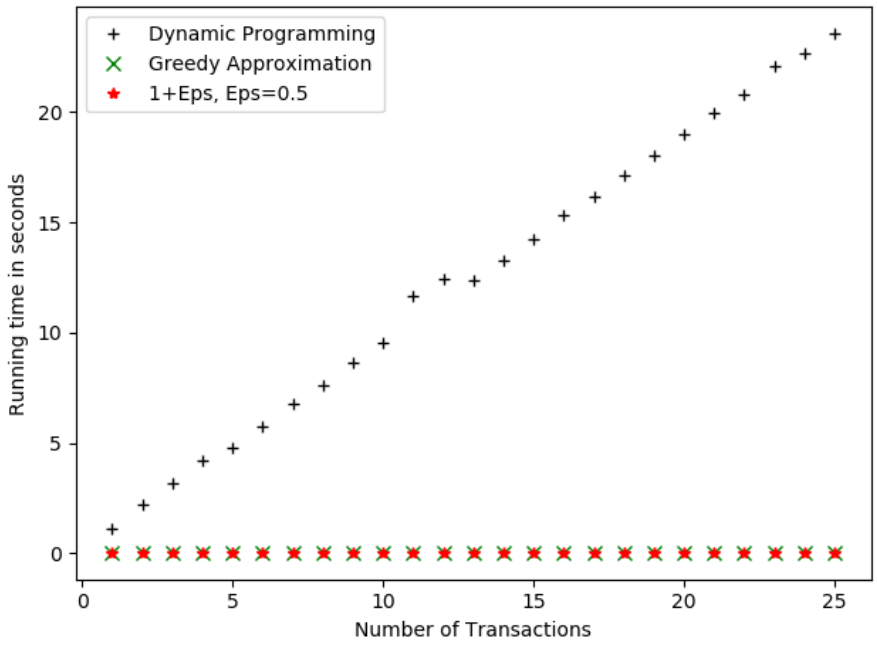


Figure 5: Relative error as a function of the number of transactions

Let us check the error relative to the optimal solution for the previous test. For $n \approx 1000$ we have no way to compute the optimal solution except for the dynamic programming algorithm. Hence, the data points were calculated as the difference between the dynamic programming solution value and each algorithm's output value. It can be seen from the graph that the dynamic programming solution value is higher than the approximations solution value, since the error values are positive. Both of the approximations yielded the same output, and this is because $\epsilon = 0.5$ was not small enough.

Experiment III



We

Figure 6: Runtime in seconds as a function of the number of transactions

The second problem with the dynamic programming algorithm starts showing when the block size W is big. Indeed, it is the only algorithm from those we mentioned whose running time is a function of W. In practice, a block of the blockchain has a capacity of 1 MB, that means that $W = 10^6$. For the upcoming graphs, we alter the parameters of the transactions a bit. Figure 6 shows the results when for each $1 \leq n \leq 25$ we randomly picked n transactions a_i such that $W = 10^6$, $1 \leq s_i \leq 500$ and $1 \leq f_i \leq 2000$. The data points are computed as the average of 25 runs. The graph shows that the running time of the dynamic programming algorithm becomes infeasible because for $n = 25$ it runs on average 23 seconds. We remind that in the previous test solving an instance of the problem where $n = 1000$ took on average 1 second. Once again we can see the linearity of the runtime as a function of the number of transactions, $O(nW)$.

Experiment IV

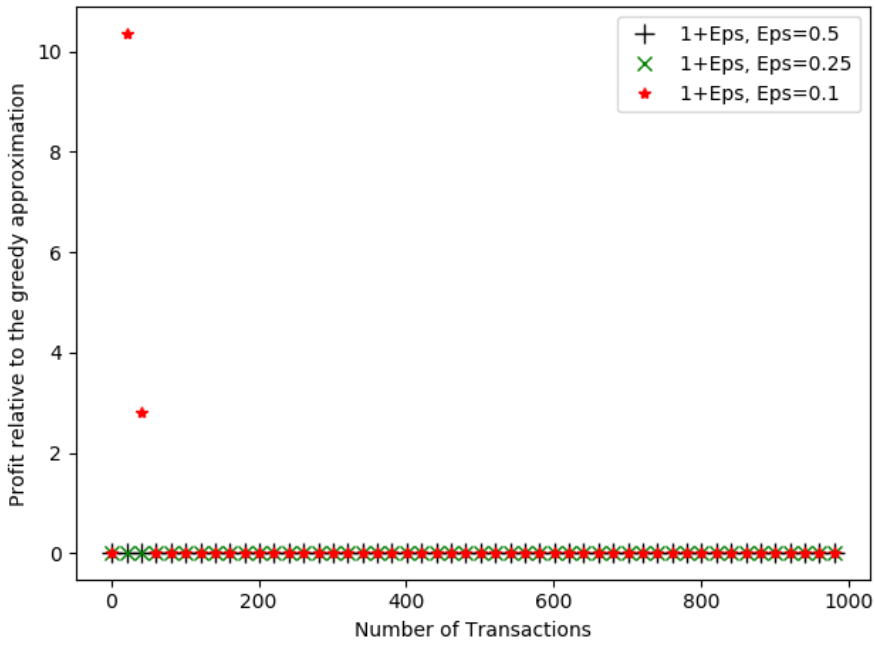


Figure 7: Profit compared to greedy approximation as a function of number of transactions

The previous tests showed us that exhaustive search and dynamic programming are not an option in real applications. Hence we are left with the approximations. However, we also saw that the greedy approximation achieved the same profit as the $(1+\epsilon)$ approximation for $\epsilon = 0.5$ within the same time. The next test shows that picking the right ϵ makes a difference. In this test, for each n we randomly picked n transactions a_i such that $W = 1000$, and $1 \leq s_i \leq 200$, $1 \leq f_i \leq 100$. The data result points are calculated as the average of 50 runs. Figure 7 shows that for small n the $1+\epsilon, \epsilon = 0.1$ can offer some profit in comparision to the greedy approximation. The data points are the difference between the $(1+\epsilon)$ solution value and the greedy solution value. Furthermore, the right value for ϵ is dependent on n as can be seen from the graph. If n increases than the solution value increases, and hence the set V_a^C will eventually become empty for a fixed ϵ.

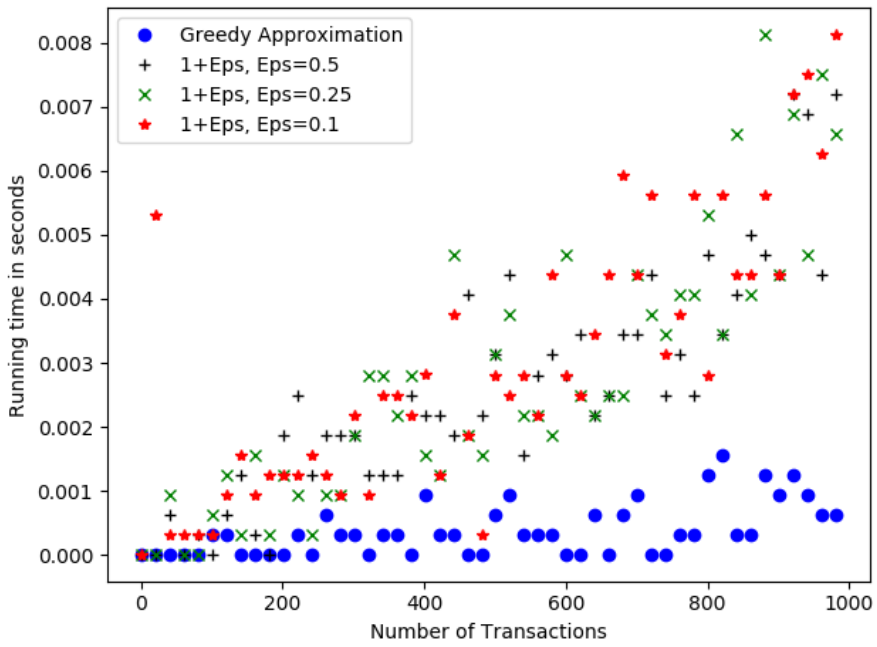


Figure 8: Runtime in seconds as a function of number of transactions

However, the running time of the $(1+\epsilon)$ approximation is higher than the running time of the greedy approximation as Figure 8 shows. This can be easily explained by the fact that the $(1+\epsilon)$ approximation algorithm uses the greedy approximation as a subroutine.

Experiment V

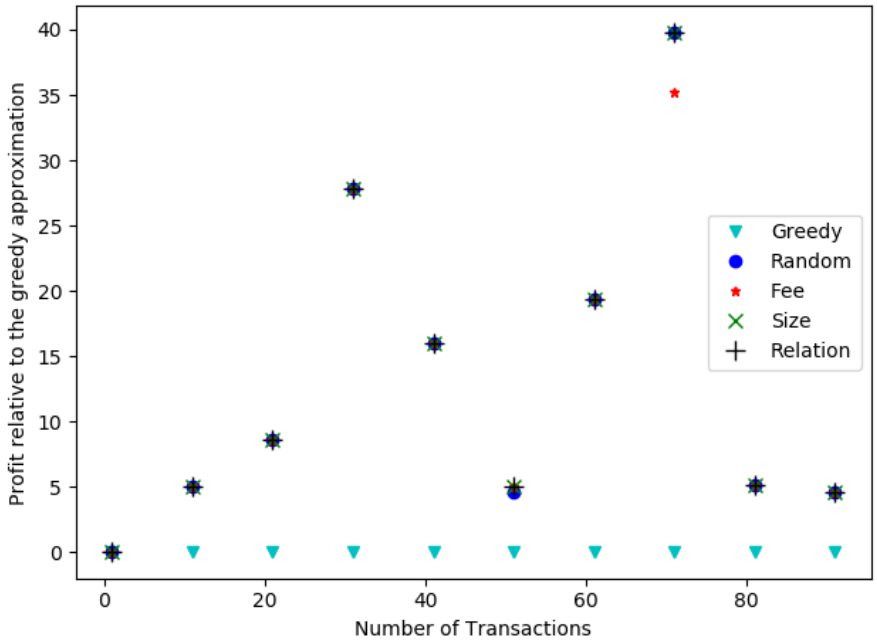


Figure 9: Profit relative to the greedy approximation as a function of the number of transactions

The $(1+\epsilon)$ approximation algorithm is a very delicate algorithm. It is very important to find the correct domain for ϵ in which the size of V_a^C isn't too small, yet not too big. The algorithm checks all the subsets $J \subseteq V_a^C$ such that $|J| \leq \frac{2}{\epsilon}$. For a lot of ϵ's it will hold that $V_a^C = \emptyset$, especially if n is big. That means we need to pick small values for ϵ. Picking small values for ϵ, means that $\frac{2}{\epsilon}$ becomes big and we remind that the runtime depends exponentially on $\frac{2}{\epsilon}$. Hence, the size of V_a^C needs not be too big because we check all the subsets of size less or equal $\frac{2}{\epsilon}$ of V_a^C. In order to keep running tests and check if we can still make an improvement, in case V_a^C becomes too big, we reduce it to a smaller set of size 20 and move the rest of the transcations from V_a^C to V_a. The number 20 was picked because more than 20 becomes infesiable in our computer in terms of memory and time. We however lose the correctness of the approximation now, it doesn't neccesary hold that the solution will be a $(1+\epsilon)$ approximation. An intereseting prospect to research is how to make the reduction: which are the 20 transactions that we should keep in V_a^C. We explored the following approaches:

- Pick 20 random transactions.
- Pick the 20 transactions with the highest $\frac{f_i}{s_i}$ ratio.
- Pick the 20 transactions with the highest fee.
- Pick the 20 transactions with the biggest size.

In this test we try to compare between these options. For each $n = 1, 11, 21..., 91$ we randomly picked n transactions a_i such that: $W = 1000$, $1 \leq s_i \leq 200$ and $1 \leq f_i \leq 100$. The value $\epsilon = 0.05$. The data is the average of 5 runs. The data points in Figure 9 are calculated as the difference between the output value of the algorithm depending on the reduction criterion and the greedy solution.

Experiment VI

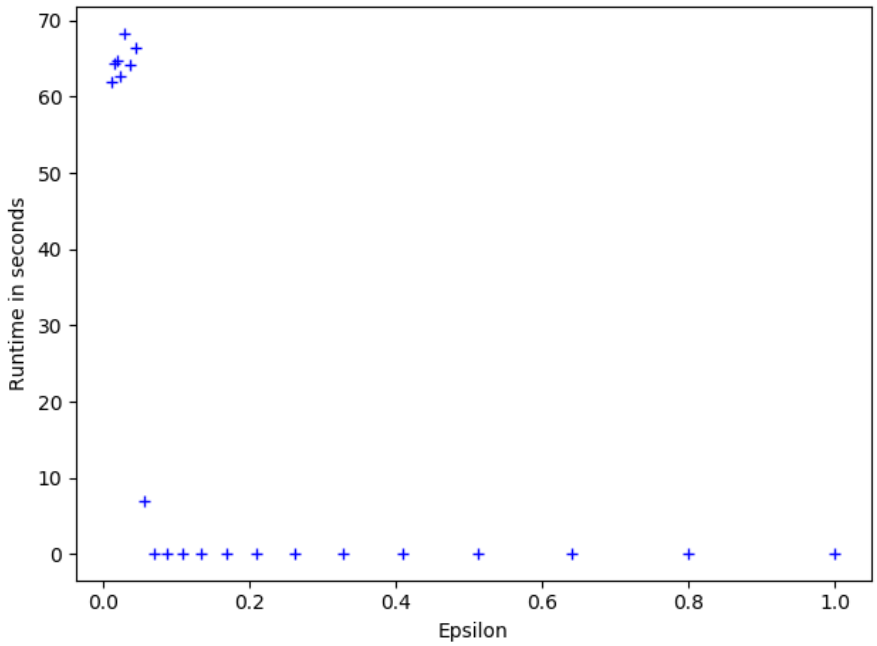


Figure 10: Runtime in seconds as a function of epsilon

We already mentioned that the $(1+\epsilon)$ approximation is a delicate algorithm and we must be careful to find the right ϵ. Figure 10 show this. For this test we randomly picked 50 transactions such that $1 \leq s_i, f_i \leq 200$. The block size is $W = 1000$. In the test ϵ is decreased by the following formula: $\epsilon = \epsilon \cdot 0.8$ while $\epsilon \geq 0.01$.

We can see that for small ϵs the running time becomes longer, those are ϵs such that $V_a^C \neq \emptyset$. Since the size of V_a^C is reduced to 20, we can expect that from some point the running time will be the same.

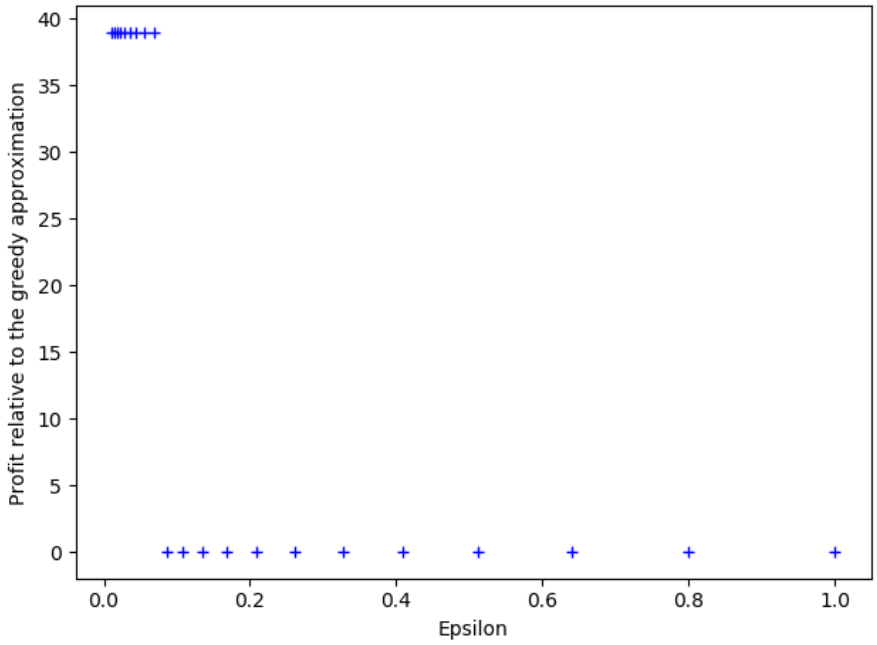


Figure 11: Profit compared to greedy as a function of epsilon

Figure 11 compares the solutions from the previous test in relation to the greedy approximation value. Specificially in this test the greedy approximation value is 2379 and so is the $(1+\epsilon)$ approximation value for $\epsilon \geq 0.085$. However for $\epsilon \leq 0.068$ the $(1+\epsilon)$ approximation value is 2418, hence the profit relative to the greedy approximation is $2418 - 2379 = 39$ as can be seen in the graph. We can see that for small ϵ we can actually make some profit compared to the greedy approximation, even with the reduction of the set V_a^C to size 20.

6.2 The dependency knapsack problem

First we compare between the greedy approximation with and without the dependencies to see that the dependencies make the problem harder, and as such it takes more time to solve. Later we will compare between the two approximations for the dependency knapsack. We mentioned that the runtime of the greedy approximation algorithm is $O(n \log n)$ for the knapsack problem and $O(n^3)$ for the dependency knapsack problem.

Experiment I

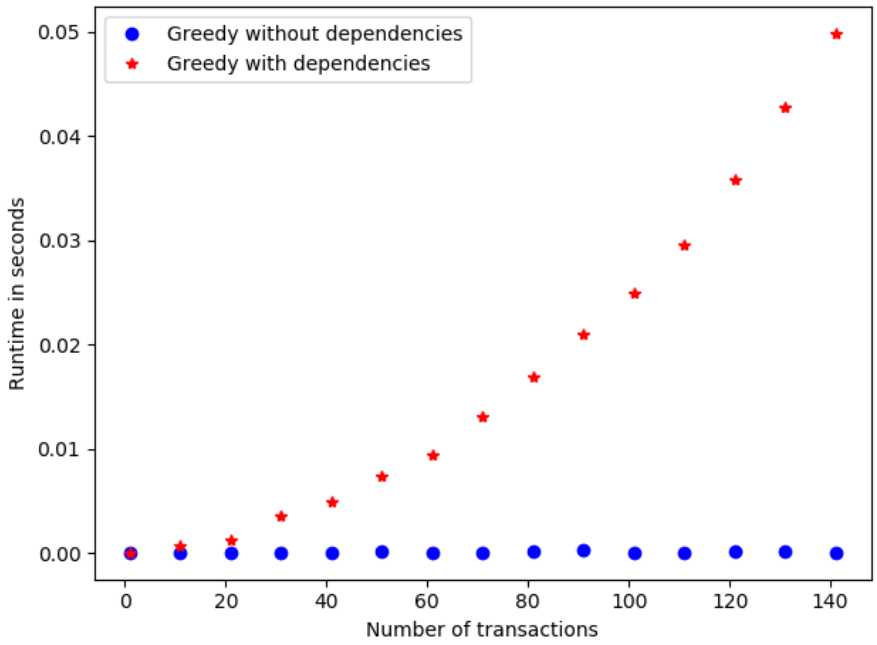


Figure 12: Runtime in seconds as a function of number of transactions

In this test, for each n we generated n random transactions such that $1 \leq s_i, f_i \leq 100$ and $W = 10000$. The dependencies were also generated randomly but such that the graph is a directed acyclic graph, meaning that there are no circular dependencies. For each number of transactions we calculated the data points as an average of 10 random runs. We can see from the figure that the runtime of the greedy approximation algorithm for the dependency knapsack problem is indeed $O(n^3)$.

Experiment II

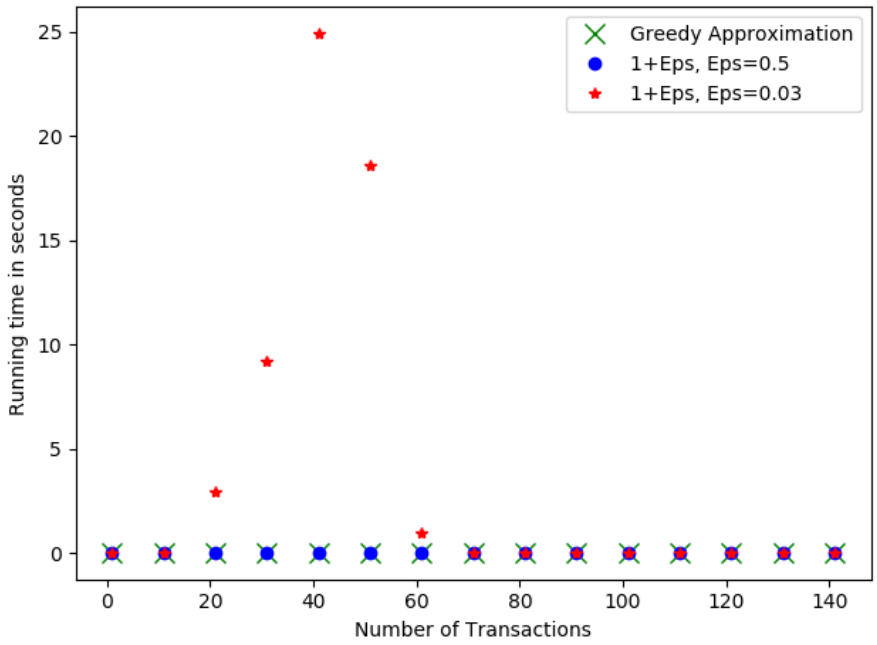


Figure 13: Runtime in seconds as a function of number of transactions

The next thing we want to check is the $(1+\epsilon)$ approximation algorithm. For each $n = 1, 11, 21..., 141$ this test creates a random instance of the dependency knapsack problem such that $100 \leq s_i \leq 200$ and $1 \leq f_i \leq 100$ and $W = 10000$. We remind however that in order to solve this we reduce the size of V_a^C as mentioned before. The reducement was done by the criterion of fee and V_a^C was reduced to 15 transactions. We also remind that the $(1+\epsilon)$ approximation algorithm for the knapsack problem uses inside the loop the greedy approximation algorithm of the knapsack problem. However the $(1+\epsilon)$ approximation algorithm for the dependency knapsack problem uses the greedy approximation for the dependency knapsack problem. We saw in the previous test that the greedy approximation with dependencies runs longer than the greedy approximation without the dependencies, hence we reduced the size of V_a^C to 15 instead of 20.

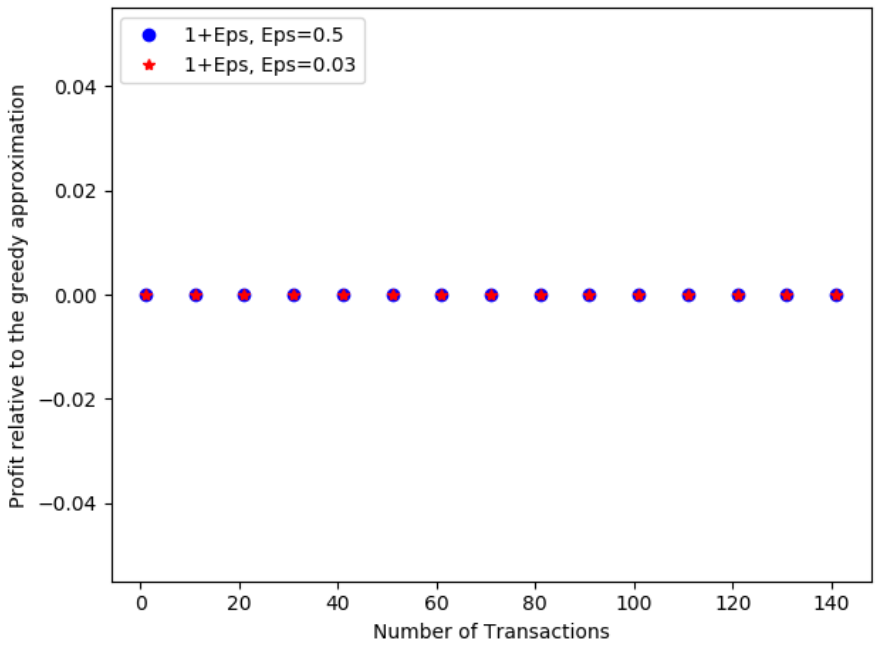


Figure 14: Profit relative to the greedy approximation as a function of the number of transactions

We can learn from Figure 13 that when we are talking about $21 \leq n \leq 61$, for $\epsilon = 0.03$ it holds that $V_a^C \neq \emptyset$. Hence we would expect a better solution at this point. Figure 14 shows the profit relative to the greedy approximation as a function of the number of transactions. Meaning, every data point is calculated as the difference between the $(1+\epsilon)$ approximation value and greedy approximation value. Inspite of the fact that for $\epsilon = 0.03$, $V_a^C \neq \emptyset$ the difference is always 0, therefore the solution is identical to the greedy approximation. This means that there is no profit compared to the greedy approximation.

6.3 Real data

We continued with the analysis by running the above algorithms on real data. The data was collected using the Bitcoin API, through out different days, a few times per day. The API function returns all the added and removed transactions relative to the last time the data was sampled, while in the beginning of the day the transactions in the added section are the current mempool. It usually holds that the removed section is empty, however the added section is not. Also, it holds that the added transactions may be dependent on transactions from the previous sample but not vice versa. Through out this section we analyze the $(1+\epsilon)$ approximation algorithm on real data, which ϵs can offer miners some profit compared to greedy approximation. We also analyze the running time and the profit through out different days. Now, the size of the set V_a^C was reduced furthermore to 10 because of the infesiable runtime.

Experiment I

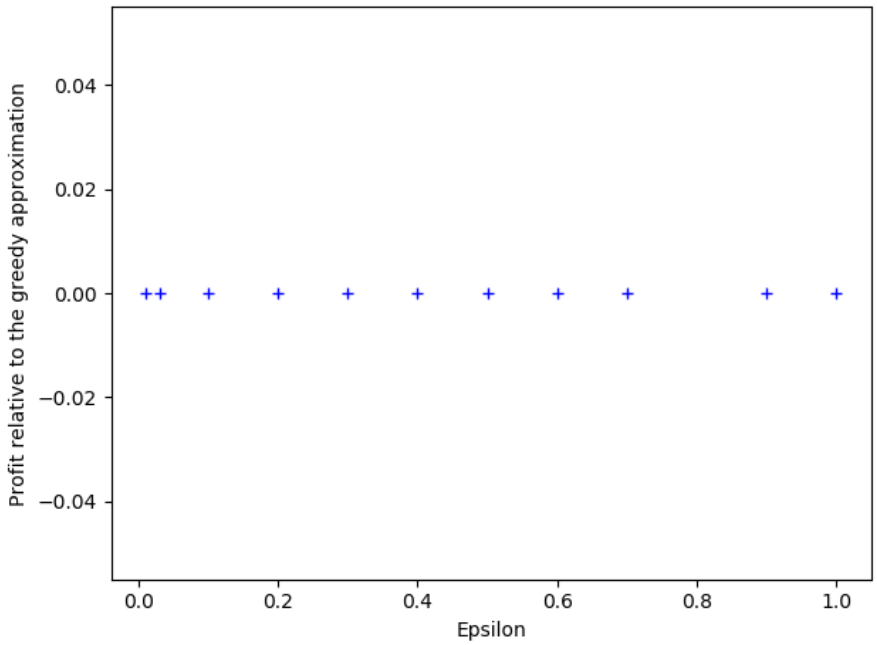


Figure 15: Profit relative to the greedy approximation as a function of ϵ

Figure 15 was build on the real data that was collected on 28/08/2017 on the beginning of the day. For all $\epsilon > 0.03$ it holds that the set $V_a^C = \emptyset$ because the greedy approximation value $\gg$ the fees of the transactions. Hence, we must use very small ϵ values, and this is bad news as we will see in the next graph because we remind that the runtime of the $(1+\epsilon)$ approximation algorithm depends exponentially on $\frac{2}{\epsilon}$. However, even for the values $\epsilon = 0.01$ and $\epsilon = 0.03$ we couldn't get any profit in comparision to the greedy algorithm. The data points are the difference between the $(1+\epsilon)$ approximation value and the greedy approximation value and since the difference is 0, it means that the two algorithms yielded the same solution for each ϵ we tested.

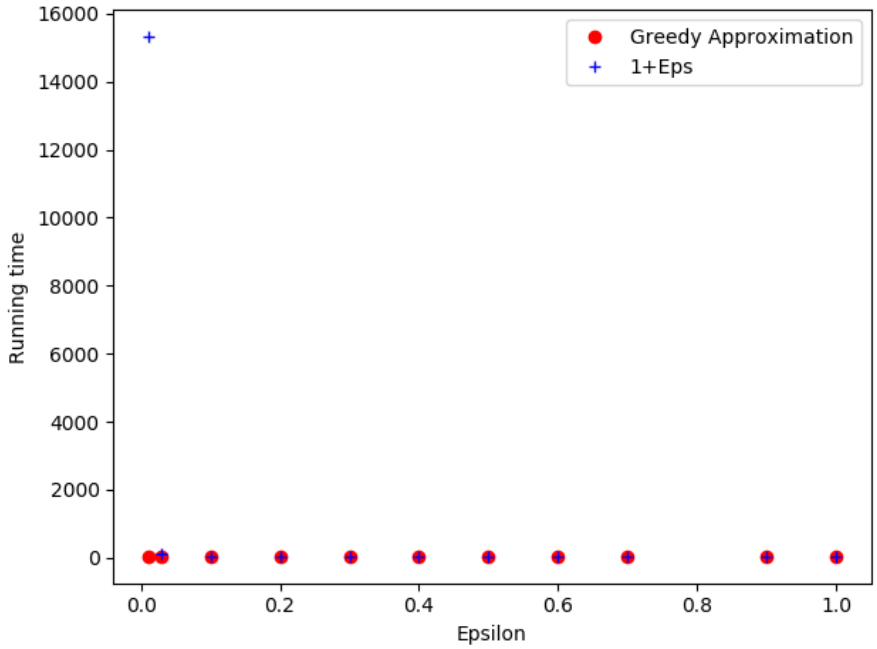


Figure 16: Runtime in seconds as a function of ϵ

This graph confirms that $V_a^C = \emptyset$ when $\epsilon > 0.03$. The runtime of the greedy approximation is something in between 15 to 16 seconds. When $\epsilon = 0.03$ it actually holds that $|V_a^C| = 3$ therefore we run the greedy approximation $2^3 = 8$ times, which explains the runtime of the $(1+\epsilon), \epsilon = 0.03$ approximation algorithm, $8 \cdot 15 \approx 120$ seconds. When $\epsilon = 0.01$ it holds that $|V_a^C| = 42$. We reduced it to 10 using the fee criterion. The runtime in this case would be $2^{10} \cdot 15 \approx 15360$ seconds which is close to what we actually received in the graph.

Experiment II

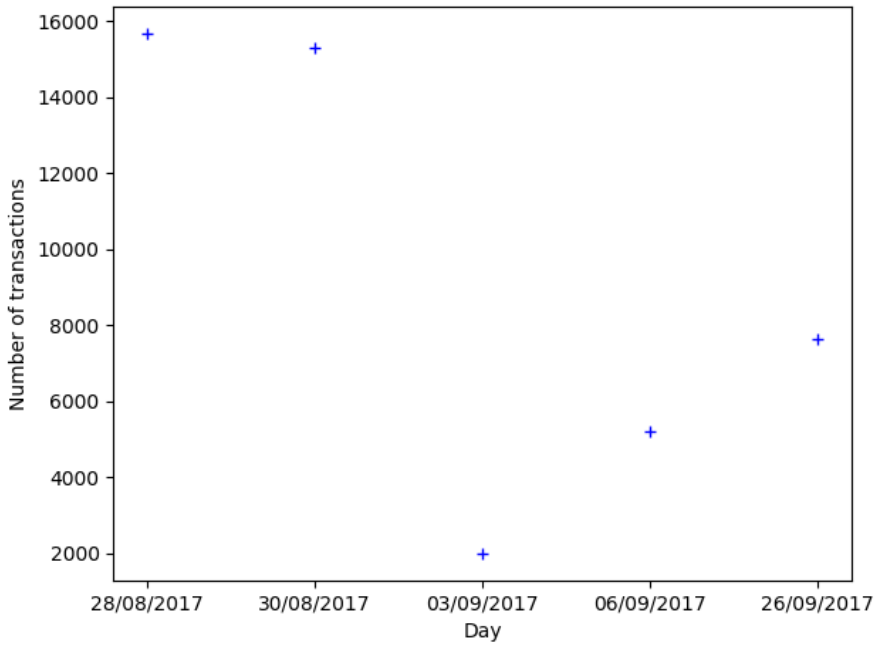


Figure 17: Number of transactions in the memory pool at the first sample per day

Figures 17 – 22 were build on real data through out the dates mentioned in the graph, while the data was sampled in the beginning of the day, hence it contains the whole mempool. Figure 17 shows the number of transactions that were in the memory pool in the beginning of each day.

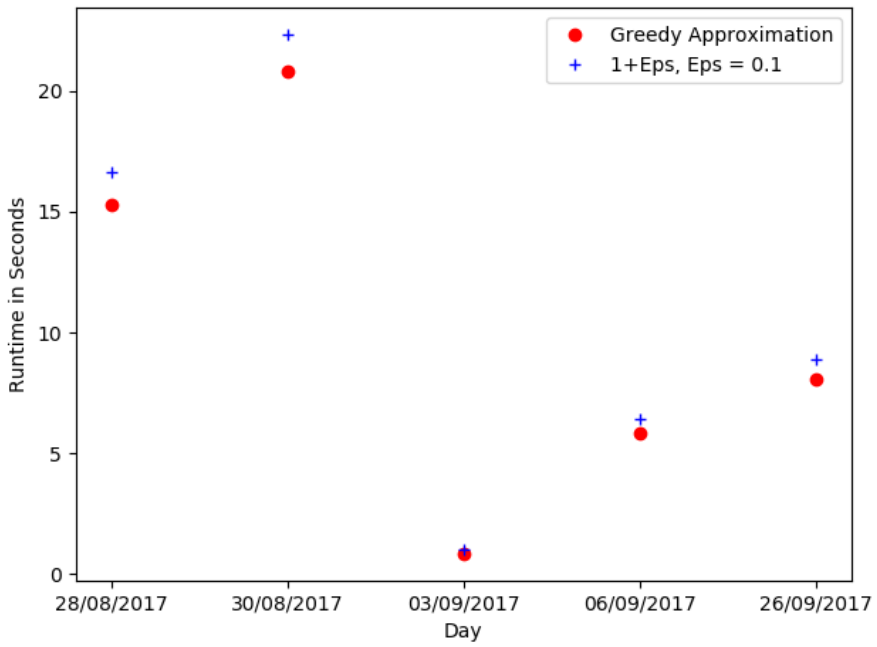


Figure 18: Runtime in seconds per day

First we checked the $(1+\epsilon)$ approximation algorithm for $\epsilon = 0.1$. Figure 18 shows that for $\epsilon = 0.1$ it holds that $V_a^C = \emptyset$ and hence the running time of the two algorithms is almost the same, with the $(1+\epsilon)$ approximation algorithm being slightly higher because the $(1+\epsilon)$ approximation uses the greedy approximation and calculates the sets V_a, V_a^C before returning the same solution. We conclude that for $\epsilon > 0.1$ it is more convinient to use the greedy approximation.

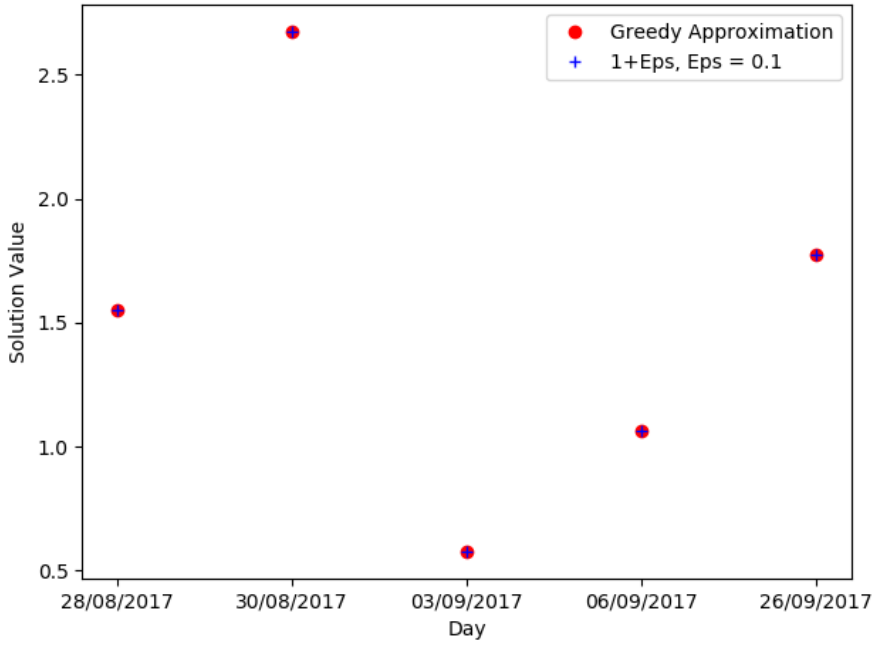


Figure 19: The solution value per day

We can see in this figure and the next that it doesn't matter which day, the value $\epsilon = 0.1$ is not small enough and yields the same solution as the greedy approximation.

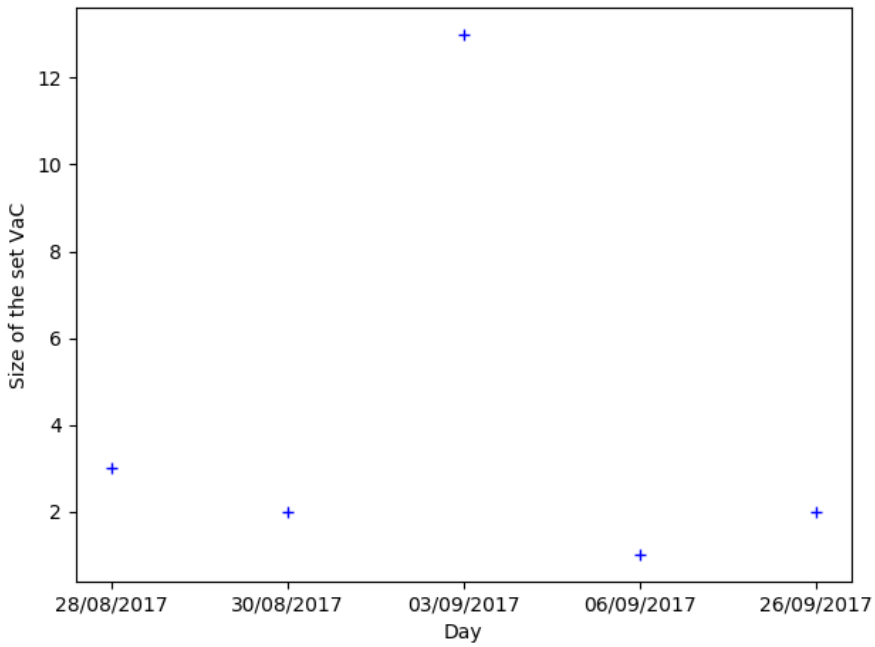


Figure 20: Size of the set V_a^C per day when $\epsilon = 0.03$

We saw that for $\epsilon = 0.1$ it holds that $V_a^C = \emptyset$ each day. Hence we decided to check $\epsilon = 0.03$. Figure 20 shows the size of the set V_a^C for $\epsilon = 0.03$ each day. Figure 17 shows that on 03/09/2017 the number of transactions is the smallest in comparision to other days. We will see soon that this implies that the value of the greedy approximation is the smallest on this day, hence the size of V_a^C is the biggest on this day. It held that $|V_a^C| = 13$ but it was reduced to 10 as mentioned before.

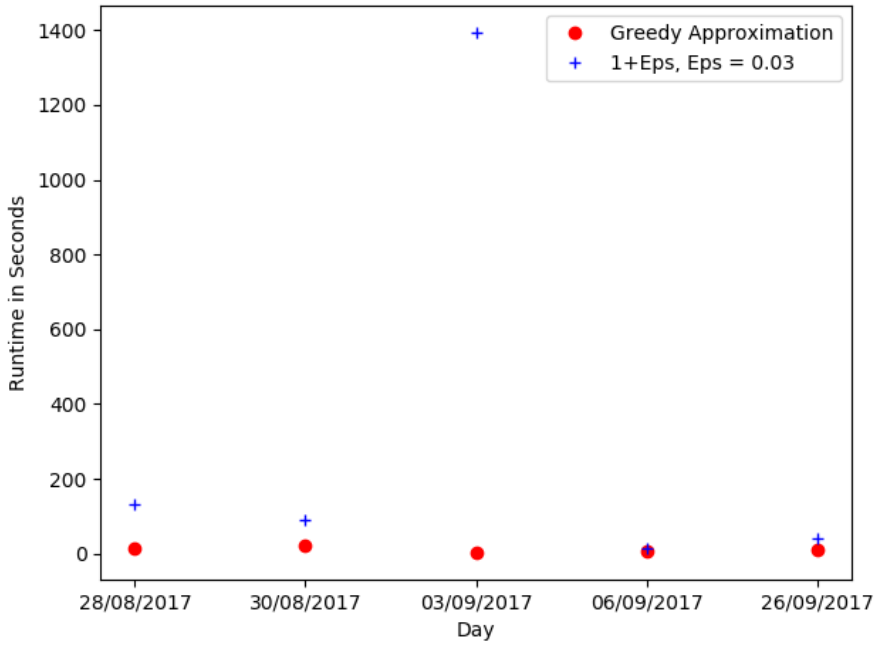


Figure 21: Runtime in seconds per day

We remind that the $(1+\epsilon)$ approximation algorithm uses the greedy approximation algorithm $2^{\min(|V_a^C|, \frac{2}{\epsilon})}$ times. Since it holds that $\epsilon = 0.03$ then $|V_a^C| < \frac{2}{\epsilon}$. This explains why the runtime of the $(1+\epsilon)$ algorithm behaves as the size of the V_a^C set in Figure 20. The conclusion is that reducing the size of the V_a^C set to 10, means that the runtime of the $(1+\epsilon)$ algorithm will likely be more than $2^{10} = 1024$ times the runtime of the greedy approximation algorithm. If the runtime of the greedy approximation for the dependency knapsack is more than 1 second (that would be a logical assumption assuming $n \approx 16000$) then the runtime of the $(1+\epsilon)$ would be at least 1024 seconds, approximately 17 minutes, which is more than the time the miner has to add the block.

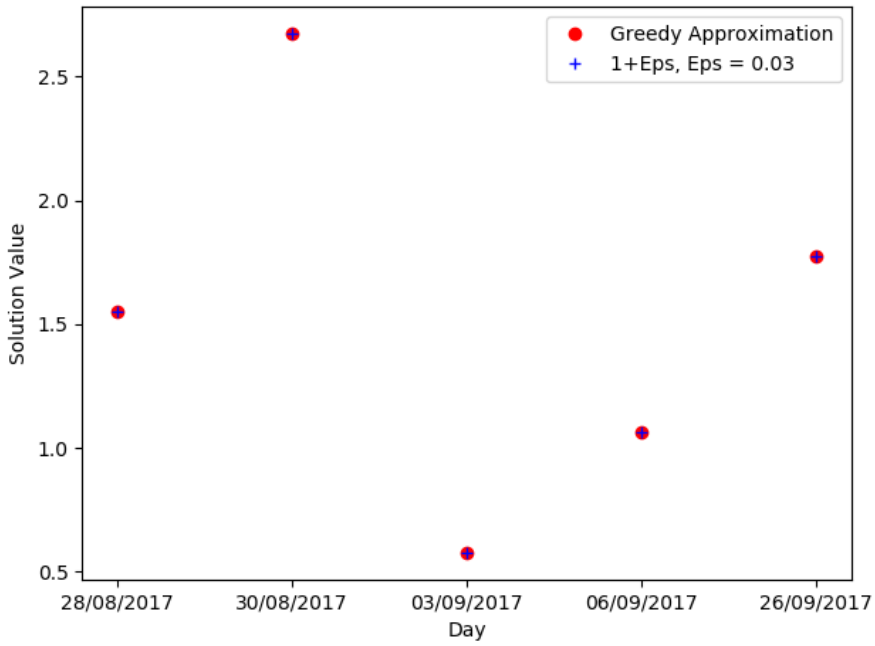


Figure 22: The solution value per day

Here we see that even for $\epsilon = 0.03$, which implies $V_a^C \neq \emptyset$ every day, the greedy approximation algorithm and the $(1+\epsilon)$ approximation algorithm yield the same value. The ϵ can not be decreased further because of the runtime as we saw in the previous graph. This concludes that the $(1+\epsilon)$ approximation algorithm can not be used to increase the profit.

Experiment III

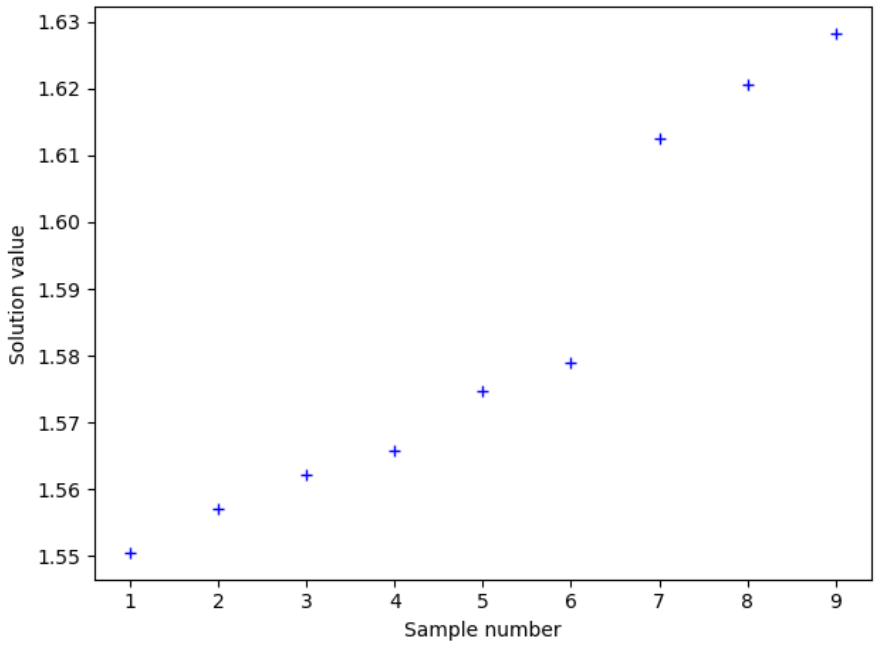


Figure 23: Solution value as a function of the samples in time

The data was sampled on 28/08/2017. Think of the sample number as a time t. Each time we sample, we receive from the API an added and a removed section. At time $t = 1$ the added section contained the whole mempool, and every time $t > 1$ the added section contains transactions that are added to the mempool since the sample at time $t - 1$. The removed section contains transactions that are removed from the mempool since the sample at time $t - 1$. As we mentioned, the removed section is usually empty. In fact this is the case in this data. Hence, as expected, the solution value increases if the sample number increases as it should it be.

6.4 The incremental solution to the greedy approximation algorithm

6.4.1 The incremental solution on mock data

Experiment I

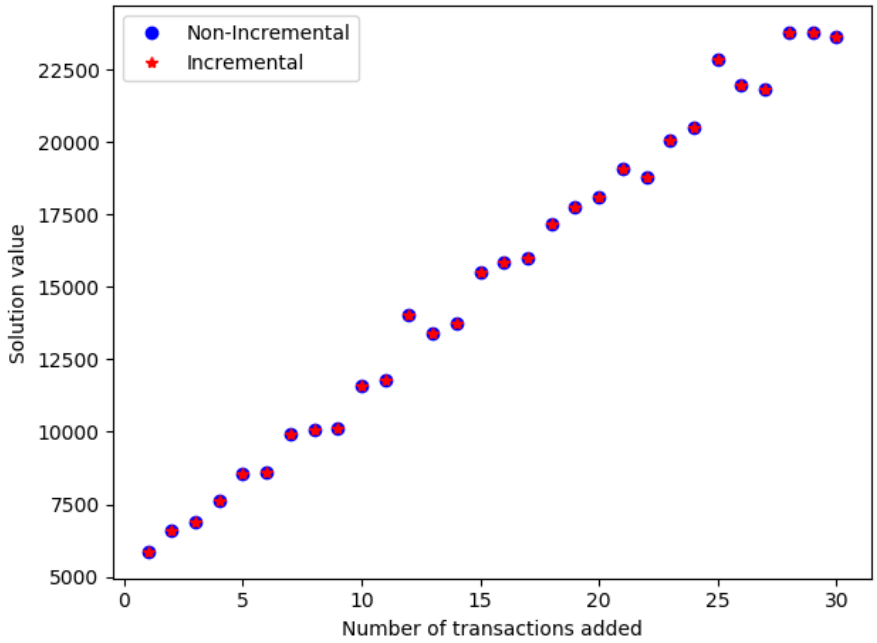


Figure 24: Solution value as a function of the number of transactions added

In this test, for each $1 \leq n \leq 30$, we create a random instance of a dependency knapsack problem with 100 transactions a_i such that $1 \leq s_i, f_i \leq 100$ and $W = 50000$. Then we add n transactions to this instance such that the are no circular dependencies. The data points are created as an average of 25 runs. The figure confirms that the solution values between the incremental version and the non-incremental version of the greedy approximation algorithm are identical.

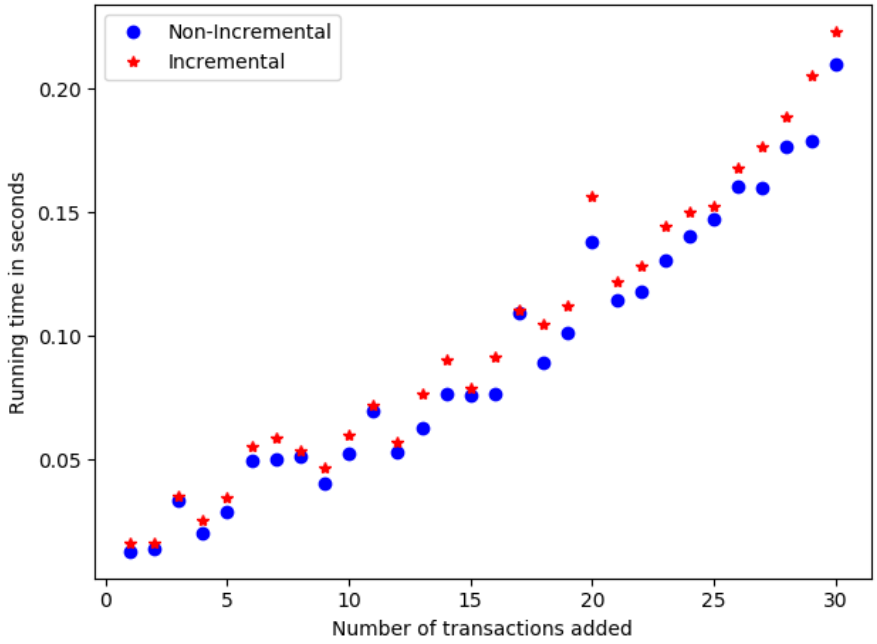


Figure 25: Running time in seconds as a function of the number of transactions added

However, we see that the incremetal version runs a bit longer than the non-incremental version. The reason for lack of improvement in this case is that the amount of $Ancestor(\cdot)$ sets that can be added automatically to the solution (that hold that their fee over size ratio is bigger than α) is very small. We started with a small amount of transactions (100) and added up to 30 new transactions. This is different from the case of real data where we start from a very big amount of transactions and add a small amount. Hence the running time is almost the same with the incremental solution taking a bit more because it checks if sets from the previous solution can be automatically added. We can see that the runtime graph resembles $O(n^3)$ which is the running time of the greedy approximation algorithm.

Experiment II

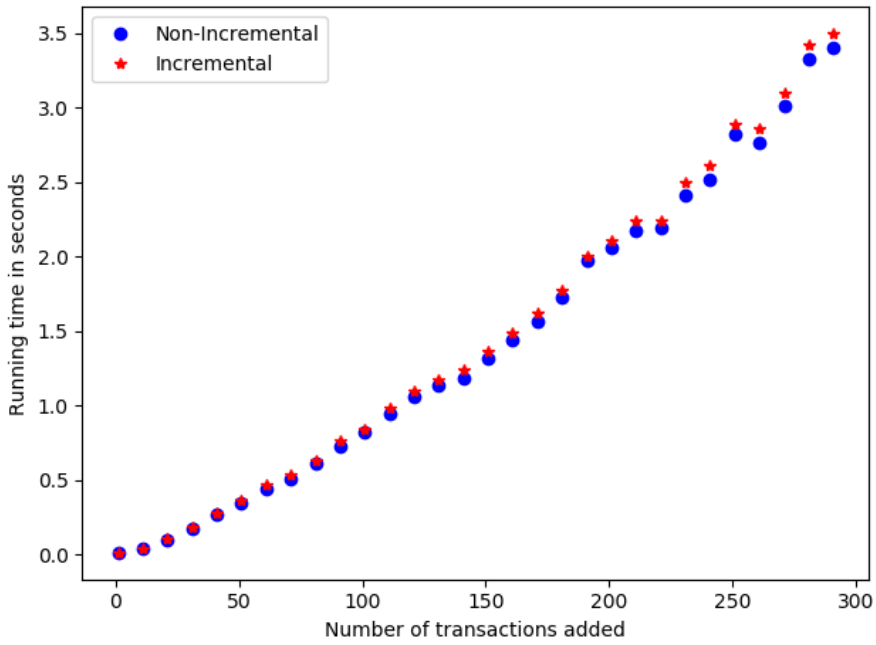


Figure 26: Running time in seconds as a function of the number of transactions added

Now we added up to 300 new transactions to the existing 100, and as such not a lot of the transactions that were in the previous solution will be in the new one. In this case we prefer the non-incremental version of the greedy approximation algorithm. Once again we can pay attention to the runtime that assembles $O(n^3)$.

6.4.2 The incremental solution on real data

The real data provide the constraints needed in order to use the incremental solution, since most of the time there are no removed transaction but only added ones.

Experiment III

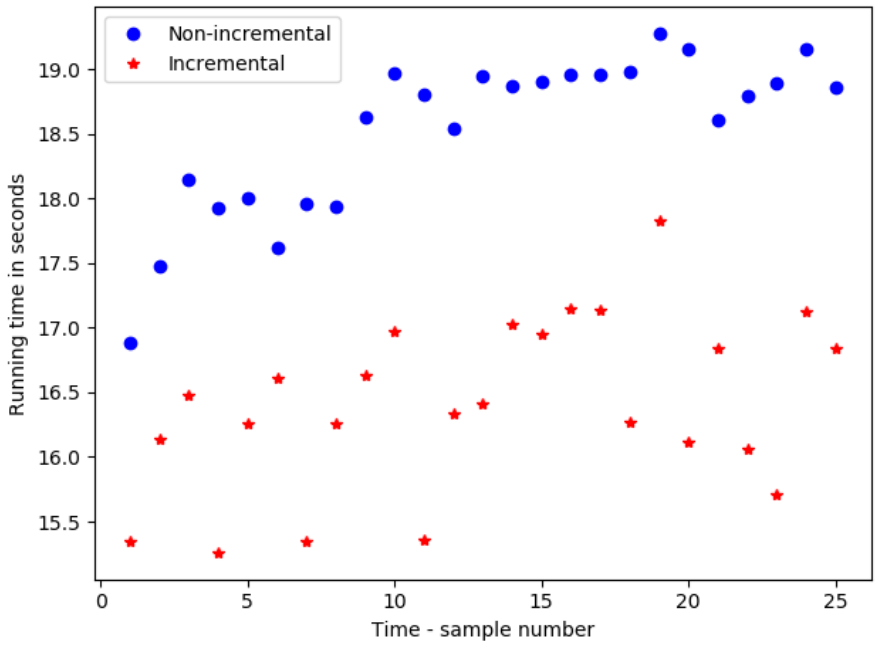


Figure 27: Runtime in seconds as a function of the samples in time

The data was sampled on 28/08/2017. On real data we see that the incremental solution makes a difference in runtime. As mentioned before, in real data the amount of transactions added from one sample to the next (from time t to time $t+1$) is small compared to the amount of transactions in time t. As such, the amount of $Ancestor(\cdot)$ sets that their fee over size ratio is bigger than α is not that small anymore and this can improve the runtime of the incremental solution in comparison to the non-incremetal solution as the graph confirms.

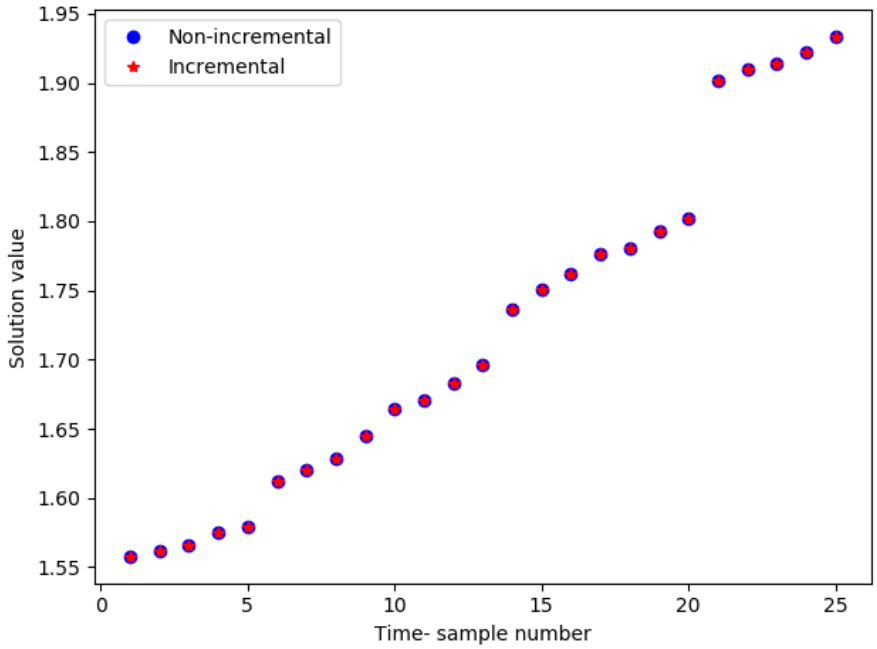


Figure 28: Solution value as a function of the samples in time

The same data as in the previous figure, meaning it was sampled through out 28/08/2017. As expected the solution values between the two versions of the greedy approximation are identical. Pay attention that the solution value increases as the sample number increases. This is because that the removed section is empty.

Experiment IV

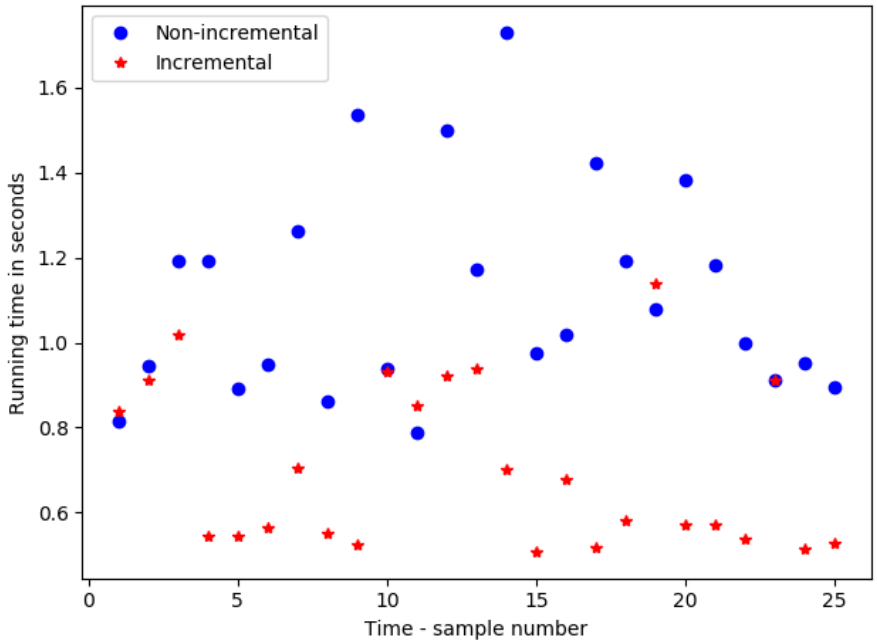


Figure 29: Runtime in seconds as a function of the samples in time

We ran the same tests on another day. This data was sampled on 03/09/2017. Almost always it holds that the incremental version of the greedy approximation algorithm is faster than the non-incremental version of the greedy approximation algorithm.

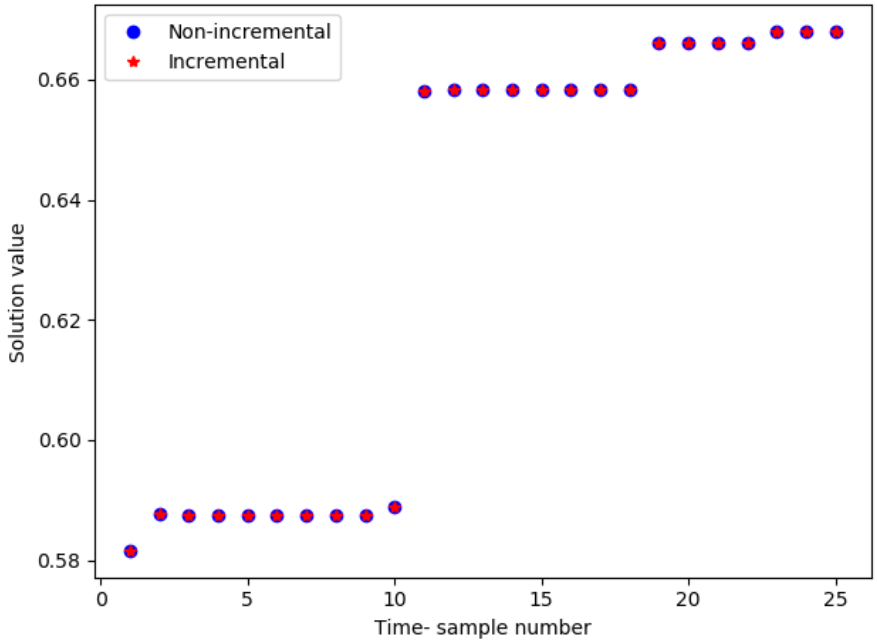


Figure 30: Solution value as a function of the samples in time

The same data as in the previous figure, meaning data sampled on 03/09/2017. As expected the solution is identical. Pay attention that the solution value increases as the sample number increases. This is because that the removed section is empty.

7 Conclusions and future work

7.1 Conclusions

- The exhaustive search algorithm and the dynamic programming algorithms are not feasible for big n and big W.
- The dependency knapsack problem is harder to solve than the knapsack problem as we saw.
- We conclude that the $(1+\epsilon)$ approximation is not feasible in the dependency knapsack because the greedy approximation value $\gg$ fees of transactions. This means that $\epsilon \ll 1$ which means that $\frac{2}{\epsilon}$ is big, hence making the algorithm infesiable in terms of time and memory.
- Also, it holds that the greedy approximation algorithm fills almost fully the block, which makes its approximation very close to the optimal solution.
- We conclude that the incremental solution algorithm on real data gives the same results as the greedy approximation algorithm on real data and is faster most of the time. It is important to remember that the incremental solution can be used only if no transactions were removed.
- The graphs created on real data are sparse. One must take this in consideration in the algorithms as it makes them much faster.

7.2 Future work

- Writing parallel programmable versions of the algorithms might help with the runtime and make the $(1 + \epsilon)$ feasible in time, but the memory problem remains.
- Generalize the incremental solution algorithm, for example in the case transactions are removed.