Performance Impact Factors for the Knapsack Algorithm Based on Dynamic Programming

David Kuhlen[1] and Andreas Speck[2]

[1]Mönkhofer Weg 239, 23562 Lübeck, Germany, david.kuhlen@th-luebeck.de
[2]Hermann-Rodewald-Straße 3, 24118 Kiel, Germany, aspe@informatik.uni-kiel.de

Abstract

The knapsack problem is a classical optimization problem in computer science that can be efficiently solved using an algorithm based on dynamic programming. This study investigates the impact of technical factors such as the number of elements, the size of the knapsack, or the sorting of elements on the performance of the algorithm. Performance refers to the speed of the algorithm implementation in solving the problem. The analysis focuses on so-called Performance Impact Factors, which are the selected influences that can affect the speed of the algorithm. While the size of the considered knapsack represents a parameterization/configuration, the number of items is directly related to the problem itself; therefore, the term Performance Impact Factors is generally used. The investigation is based on a simulation experiment. The analysis of the simulation experiment data shows that as the total number of elements and the knapsack capacity increase, the algorithm's runtime also increases. In this context, an increase in knapsack capacity has a more pronounced negative effect on the algorithm's runtime. An examination of sorting strategies further reveals that a prior sorting of elements by value in descending order can lead to significant deteriorations in runtime.

Keywords

Dynamic Programming, Knapsack Problem, Performance

1. Introduction

The knapsack problem is a classical problem in computer science, categorized as an optimization problem. In the knapsack problem, a list of items with known values and weights is considered, and the goal is to select an optimal subset of items with the highest possible total value without exceeding a specified weight limit [7, p. 213]. To solve this problem, an algorithm exists that can be implemented using either divide-and-conquer or dynamic programming techniques [7, p. 213]. The range of tasks requiring the application of the knapsack algorithm is broad. Possible areas of application include the optimization of machine utilization, weight-optimized product manufacturing, or solving classical selection problems.

Solving the knapsack problem using dynamic programming offers significant performance advantages. The focus of this study is the implementation of an algorithm for solving the knapsack problem using dynamic programming. This algorithm will hereafter be referred to in simplified terms as the Knapsack Algorithm (KSA).

It should also be noted that, in addition to implementing the algorithm in recursive form and using dynamic programming, other implementation variants are conceivable but are not considered in this study. This study focuses on the 0/1 knapsack algorithm. In further

David C. Wyld et al. (Eds): CSML, AISCA, DNLP, SOEA, NET, BDHI, SIPO – 2026
pp. 137-150, 2026. CS & IT - CSCP 2026 DOI: 10.5121/csit.2026.160211

research, the analysis could be extended to other knapsack variants or alternative algorithm implementation approaches.

This paper investigates factors that influence the performance of the Knapsack Algorithm. The following research questions are examined:

- **RQ1:** What influence does the sorting of items by value or weight have on the performance of the KSA?
- **RQ2:** What influence does the total number of items have on the performance of the KSA?
- **RQ3:** What influence does the size of the knapsack have on the performance of the KSA?

This study does not aim to provide exact statements regarding the speed of the KSA. The precise performance depends on many factors, such as the programming language used for implementation and the specifications of the computer system on which the KSA is executed. Instead, the objective is to demonstrate the extent to which the aforementioned technical factors affect the algorithm's performance.

The following chapter presents related work from the current state of research. Chapter three explains the methodology. This is followed by a presentation of the results that address the above-mentioned research questions. The paper concludes with a summary and an outlook.

2. Related Work

Toth describes algorithms that solve the 0-1 knapsack problem using dynamic programming and compares them [12]. This comparison focuses on the speed of the algorithms when solving the problem for different quantities of items ($n = 50...200$) [12, p. 44]. In this way, Toth identifies the DPT1 and DPT2 versions of the algorithm as advantageous for solving "hard problems" [12, p. 44].

A comprehensive overview of various algorithms that solve the knapsack problem can be found in [7]. According to Martello and Toth, the application of dynamic programming for implementing the KSA results in an exact solution to the problem [7, p. 213]. According to Martello and Toth, the runtime complexity of the KSA usually ranges from $O(n)$ to $O(n \log n)$ [7, p. 216]. Many KSA algorithms require a preliminary sorting based on the ratio of value to weight in descending order, as stated by Martello and Toth [7, p. 219]. Although solving the knapsack problem using dynamic programming does not require sorting, sorting by the aforementioned value-to-weight ratio can positively influence speed [7, p. 227].

Similar to the approach taken in this study, Martello and Toth also test the algorithm's performance through an experiment based on random test data [7, p. 213]. In this experiment, the total number of items ranges from 50 to 10,000 [7, p. 228]. Martello and Toth test the runtime of the algorithms and compare different algorithm versions [7, p. 229 ff.]. While the performance of the KSA based on dynamic programming surpasses that of *branch-and-bound algorithms*, Martello and Toth report that dynamic programming could not be considered for large problems due to the associated memory consumption [7, p. 230].

Numerous other studies examine the performance of algorithms for solving the knapsack problem. Bertsimas and Demir investigate solutions for multidimensional knapsack problems [1]. Rong et al. analyze the performance of dynamically programmed KSAs in solving the "discounted 0-1 knapsack problems" [9]. Clautiaux et al. examine the runtime behavior

of a solution to the "temporal knapsack problem" [2]. Furthermore, reference should be made to the research conducted by [11, 5, 6].

3. Analysis

This study addresses the 0-1 knapsack problem, in which an item is either entirely selected or not selected at all [7, p. 214]. According to Martello and Toth, the problem is sometimes interpreted to include an upper limit on the value of knapsacks [7, p. 213]. Such a value constraint is not considered in the following. Furthermore, this study does not address the "multiple knapsack problem", where the aim is to optimize the total value across several knapsacks [7, p. 242].

```
private Knapsack buildOptimalKnapsack(int c) {
   int n = allItems.size() - 1;
   int i = n;
   int z = 1;
   for (int rc = 0; rc <= c; rc++) {
      if (rc < g(i))
         f[n][rc] = new Knapsack();
      else {
         f[n][rc] = new Knapsack();
         f[n][rc].add(o(i));
      }
   }
   for (i = n - 1; i >= z; i--) {
      for (int rc = 0; rc <= c; rc++) {
         if (rc < g(i)) {
            f[i][rc] = f[i + 1][rc];
         } else {
            if (f[i + 1][rc].sumValue() >
                f[i + 1][(rc - g(i))].sumValue() + w(i)) {
               f[i][rc] = f[i + 1][rc];
            } else {
               Knapsack tR = f[i + 1][(rc - g(i))].clone();
               tR.add(o(i));
               f[i][rc] = tR;
            }
         }
      }
   }
   if (c < g(i))
      return f[z][c];
   if (f[z][c].sumValue() > f[z][(c - g(i))].sumValue() + w(i))
      return f[z][c];

   Knapsack r2 = f[z][(c - g(i))].clone();
   r2.add(o(i));
   return r2;
}
```

Figure 1: Implementation of the KSA using dynamic programming based on the explanations by [10, p. 234 f.].

The knapsack problem can be solved using either recursive programming or dynamic programming. In the algorithm textbook by Saake and Sattler, both the recursive and the dynamic programming versions of the knapsack algorithm are described [10, pp. 233–235]. The code in 1 presents the implementation of the KSA using dynamic programming, based on the algorithm described by Saake and Sattler [10, p. 234 f.]. This corresponds to the implementation of the `buildOptimalKnapsack` method in the `DynamicProgrammingSolver` class, see Figure 4.

Solving the knapsack problem using dynamic programming is similar to the recursive version of the algorithm. Therefore, a brief discussion of the recursive solution, illustrated in Figure 2, is useful. The recursive version of the algorithm is based on the description in [10, p. 233]. The recursion terminates once the final item in the list has been processed. During execution, the algorithm evaluates for each item whether adding it would result in a higher total value than excluding it. The activity diagram in Figure 2 illustrates the flow of the recursive algorithm.

The implementation of the KSA using dynamic programming relies on the maintenance of an intermediate table. This is a matrix in which the optimal combination of items is

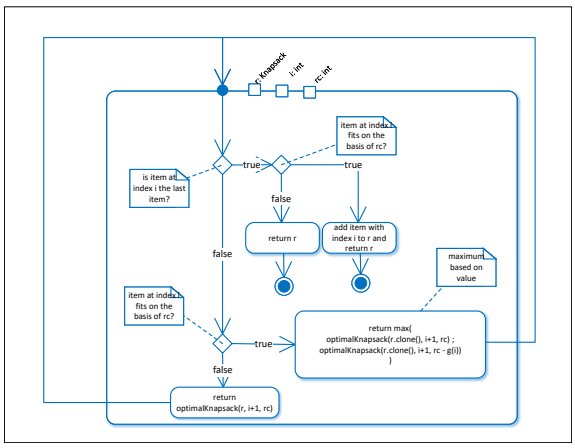

Figure 2: Depiction of the procedure of a recursive algorithm for solving the knapsack problem. Own illustration, created with Microsoft© Visio© [4]

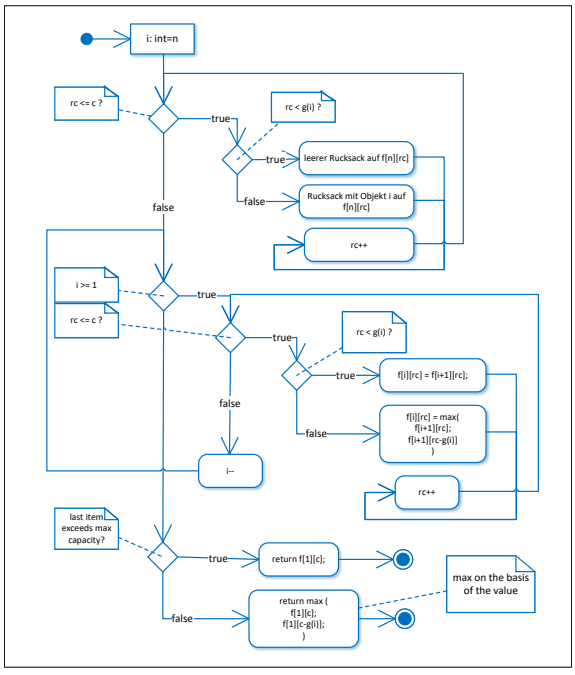

Figure 3: Depiction of the KSA procedure as an activity diagram, implemented using dynamic programming. Own illustration, created with Microsoft© Visio© [4]

stored for various possible remaining capacities of the knapsack. The algorithm iterates through the entire list of items row by row. During execution, it checks whether adding the current item to an optimal knapsack, into which it would still fit, would yield a higher total value than leaving the knapsack unchanged. The diagram in 3 illustrates the process as an activity diagram, based on the algorithm description from [10, p. 234 f.].

Before implementing the KSA in an object-oriented programming language, it is important to note that Java typically uses *call-by-reference*. As a result, separate processing of object lists can lead to side effects. To address this, a `clone()` method was created for a `Knapsack`. This method creates a new knapsack with the same capacity as the original and a copy of all items contained in the original.

4. Method

To address research questions RQ1 through RQ3, a simulation experiment was conducted. For this purpose, a simulation software application was developed in the Java programming language. The program created will hereinafter be referred to as KSApp (Knapsack Solving App). Figure 4 presents the structure of the experimental software in the form of a class diagram. For simplicity, the diagram does not display `Exceptions` or test classes containing `JUnit` test cases.

The execution entry point of KSApp is the `KnapsackSimulationApp` class. This class creates a controller (`ExperimentController`) and transfers control to it. In collaboration with the `View`, the controller implements a textual interface (console application). Through this interface, users can configure and start a simulation experiment. During configuration, information is provided such as the name and path of the CSV results file, along with domain-specific settings concerning the experiment's procedure.

To address research questions RQ1–RQ3, the experiment classes `VaryCapacityCPExp--eriment` (VC) and `VaryNumberOfElementsCPExperiment` (VE) were created. These extend the abstract `Experiment` class and implement the `execute` method. This method defines the execution flow of the experiment. In the `VaryCapacityCPExperiment`, the knapsack's capacity is incrementally varied, while all other parameters remain unchanged from the base configuration (=`KnapsackConfiguration`) (ceteris paribus). In the `VaryNumberOf--ElementsCPExperiment`, the total number of elements is varied step by step, while all other parameters remain constant relative to the base configuration (ceteris paribus). Figure 5 illustrates the process flow of the VC experiment, which addresses RQ3. The process flow of the VE experiment, which addresses RQ2, is shown in Figure 7.

Table 1: Basic Configuration

knapsackCapacity	100
numberOfObjects	2000
maxWeight	15
maxValue	15

Each experiment refers to the base settings, which are presented in Table 1. These base settings are used unless the respective experiment specifies a variation. The property `knapsackCapacity` defines the standard knapsack size, `numberOfObjects` indicates the usual number of items considered, `maxWeight` defines the maximum weight per item, and `maxValue` the maximum value per item.

Within the `execute` methods, the `solve` method is called. This method performs one run of the experiment and generates a result dataset. To address research questions RQ1–RQ3, the cycle time (CT) is measured. Figure 6 illustrates the procedure used to generate the result dataset. It shows that the system time is recorded before and after executing the `solve` method of the selected strategy (`ISolveKnapsackProblem`) via `System.currentTimeMillis()`. The algorithm's runtime is calculated as the difference between end time and start time in milliseconds. Moreover, the code in Figure 6 clarifies that the `init` method is executed before the start time is recorded, meaning that the runtime of the `init` method is not included in the performance measurement.

As shown in Figures 6, 7, and 5, the solution to the knapsack problem is based on an implementation of `ISolveKnapsackProblem`. Figure 4 illustrates that this interface is implemented by a `Solver`, which provides base functionality to both the `DynamicProgramming--Solver` and the `RecursiveSolver`, the latter implementing the actual algorithms for determining an optimal knapsack. This study does not further examine the `RecursiveSolver`. To address research question RQ1, subclasses of the `KnapsackDynamicProgrammingSolver` were developed. These subclasses—`DynamicProgrammingSolverSortByValueASC`, `Dynamic--ProgrammingSolverSortByValueDESC`, `DynamicProgrammingSolverSortByWeightASC`, and `DynamicProgrammingSolverSortByWeightDESC` - differ in the implementation of the `init` method. As shown in the source code in Figure 6, this `init` method is executed before the actual knapsack algorithm is applied. Consequently, the `init` method performs preparatory tasks, such as sorting the items. An implementation of the `init` method is included as a note in Figure 4.

The experiment was executed on a *Windows Server 2012 R2* with an *Intel Xeon CPU E3-1225 v5* running at 3.3GHz and equipped with 8 GB RAM, using Oracle JDK 17.0.2 [8]. It was executed in two variants, each using one of the experiment classes VC and VE. In each experiment, `n=100` was set, meaning that each configuration was executed 99 times, which is why the value $n = 99$ is used in the following for simplicity. In the VC variant, the knapsack capacity was varied from 10 to 500 in increments of 10. In the VE variant, the total number of items was varied from 100 to 10,000 in increments of 100. As shown in Figures 5 and 7, the respective upper limit is not included in the experiment. As a result, 24,500 data records were generated for the evaluation of the VC experiment, and 49,500 data records for the VE experiment.

5. Results

To analyze the results, KSApp generates a separate CSV file for each experiment variant (VC and VE). This CSV file contains data per sorting strategy and experiment run in the

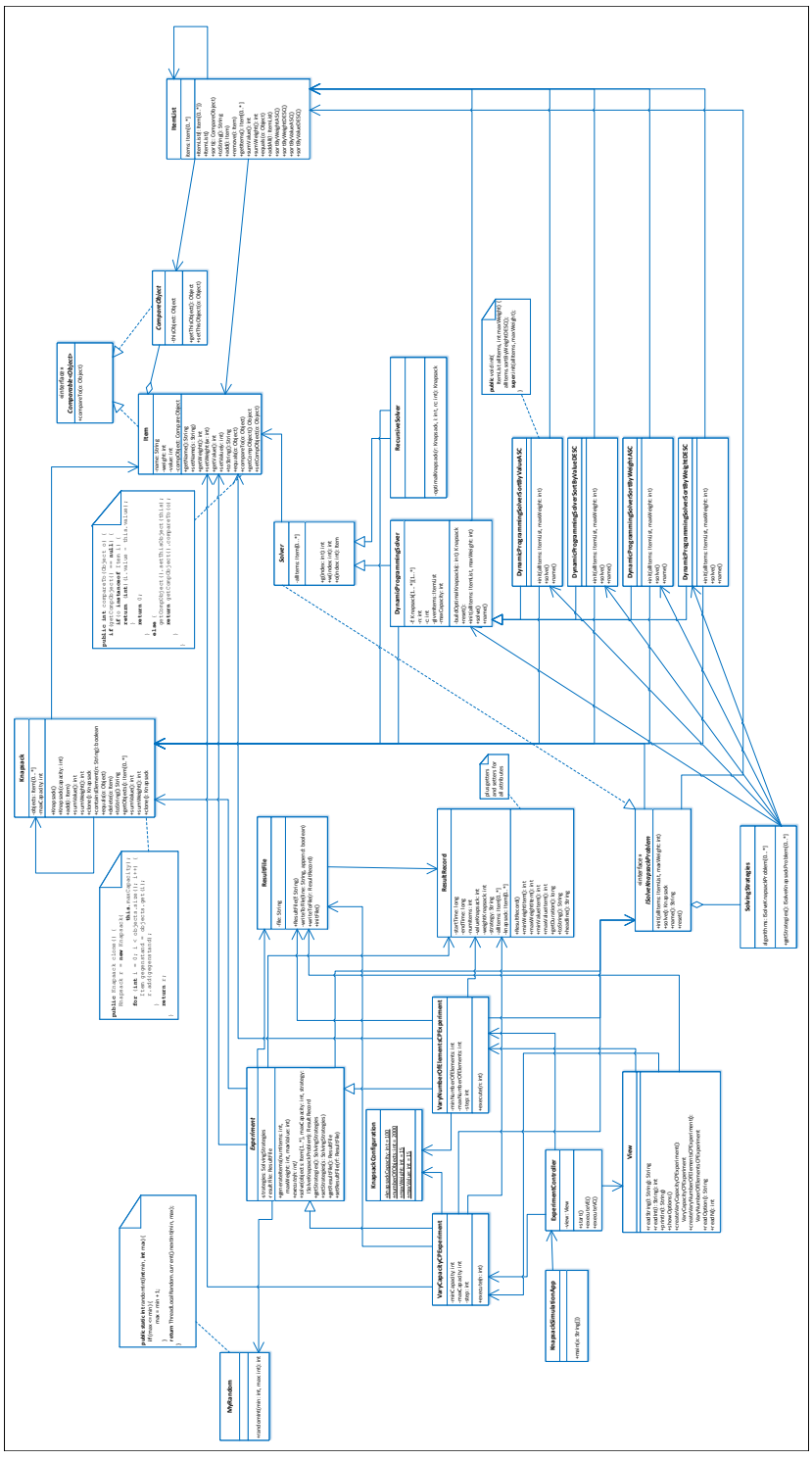

Figure 4: Illustration of the architecture of the KSApp in the form of a class diagram. Own illustration, created with Microsoft© Visio© [4]

```
public void execute(int n) {
 for (int c = minCapacity;
 c < maxCapacity; c = c + step) {
   for (int i = 0; i < n; i++) {
    List<Item> objects = generateItems(
     KnapsackConfiguration.numberOfObjects,
     KnapsackConfiguration.maxWeight,
     KnapsackConfiguration.maxValue);
    for (ISolveKnapsackProblem s:
     getStrategies().getStrategies()) {
      ResultRecord r = solve(objects, c, s);
      this.getResultFile().writeToFile(r);
      }
    }
 }
}
```

Figure 5: Procedure of the VaryCapacityCPExperiment. Own implementation.

```
public ResultRecord solve(List<Item> objects,
 int maxCapacity,
 ISolveKnapsackProblem strategy) {
  ResultRecord r = new ResultRecord();
  r.setNumArtikel(objects.size());
  r.setAllItems(objects);
  r.setMaxRucksackGewicht(maxCapacity);
  ItemList il = new ItemList(objects);
  strategy.init(il, maxCapacity);
  r.setStartTime(System.currentTimeMillis());
  Knapsack rucksack = strategy.solve();
  r.setEndTime(System.currentTimeMillis());
  r.setWertRucksack(rucksack.sumValue());
  r.setGewichtRucksack(rucksack.sumWeight());
  r.setKnapsack(rucksack.getObjects());
  r.setSrategie(strategy.name());
  strategy.reset();
  return r;
}
```

Figure 6: Implementation of the solve method in the Experiment class. Own implementation.

```
public void execute(int n) {
 for (int o = minNumberOfElements;
  o < maxNumberOfElements; o = o + step) {
  for (int i = 0; i < n; i++) {
   List<Item> objects = generateItems(o,
    KnapsackConfiguration.maxWeight,
    KnapsackConfiguration.maxValue);
   for (ISolveKnapsackProblem s:
    getStrategies().getStrategies()) {
     int ck = KnapsackConfiguration.
      knapsackCapacity;
     ResultRecord r = solve(objects, ck, s);
     this.getResultFile().writeToFile(r);
     }
   }
  }
}
```

Figure 7: Procedure of the VaryNumberOfElementsCPExperiment. Own implementation.

following columns: `Strategy; NumberOfItems; MaxBackpackWeight; DurationInMilli--seconds; MinItemWeight; MaxItemWeight; MinItemValue; Max ItemValue; Back--packResultValue; BackpackResultWeight`. The `Strategy` column distinguishes the sorting strategies applied to the full set of items prior to the execution of the KSA (see the explanations regarding the `init` method in Section). The following strategies are therefore differentiated: `DP-Classic` (=no sorting), `DP-Sort-By-Weight-ASC` (=ascending sort by weight), `DP-Sort-By-Weight-DESC` (=descending sort by weight), `DP-Sort-By-Value-ASC` (=ascending sort by value), and `DP-Sort-By-Value-DESC` (=descending sort by value). The `DurationIn Milliseconds` column represents the cycle time, calculated as the difference between the end and start times (see Figure 6).

5.1. Evaluation of the VC Experiment

For the evaluation of the VC experiment, a pivot table was first created using Microsoft© Excel© 2019 ([3]), based on the CSV results dataset. The strategies are represented as columns and the variation in maximum knapsack capacity as rows. The result values are the average cycle times across all corresponding experiment runs ($n = 99$).

During the VC experiment, the maximum allowed knapsack capacity is incrementally varied and its effect on cycle time (CT) is measured. Figure 8 presents the experimental results. The data for `DP-Classic` show the development of cycle time with increasing knapsack capacity and unsorted items. The remaining data show how cycle time evolves with increasing capacity and prior sorting of the complete item set. Here, ASC denotes ascending and DESC descending sorting.

Figure 8 shows a clear increase in cycle time as knapsack capacity rises. When the items are sorted in descending order by value, the increase in cycle time is, on average, the most pronounced. In contrast, sorting the items in ascending order by value or weight leads to a positive effect on cycle time compared to the unsorted variant.

Figure 9 shows the average cycle time observed during the VC experiment, broken down by sorting strategy. Similar to the presentation in Figure 8, it becomes evident that descending sorting by item value leads to the longest average cycle time. Conversely, the shortest aver-

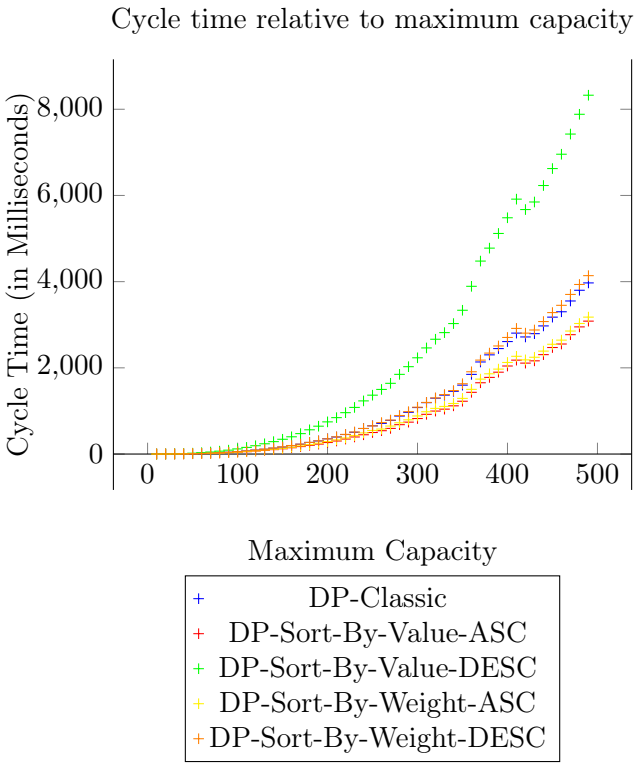

Figure 8: Development of cycle time considering a variation in knapsack capacity. Own illustration, created with TikZ (tikzpicture) being part of TeX Live Version 2024 [13] on the basis of data sets proprocessed with Microsoft Excel 2019 [3]

age cycle time across all experiments is achieved with the strategy `DP-Sort-By-Value-ASC`.

A hypothesis test for two-tailed questions is applied to determine whether the deviations in average cycle time due to prior sorting are statistically significant compared to the unsorted variant (`DP-Classic`). The evaluation is based on the aggregated data from the experiment runs, with a sample size of 49 data points. The significance level is set at $\alpha = 5\%$. The analysis shows that only the data for `DP-Sort-By-Value-DESC` reveal significant differences in cycle time compared to `DP-Classic`.

5.2. Evaluation of the VE Experiment

In the VE experiment, the total number of items is incrementally varied while all other settings remain unchanged (c.p.). As with the VC experiment (see Section), data preparation begins with the creation of a pivot table using Microsoft© Excel© 2019 ([3]). The strategies are displayed as columns and the varied item counts as rows. Once again, the result values are the average cycle times over all experiment runs ($n = 99$).

Figure 10 shows the results of the data analysis. It becomes evident that the cycle time increases with the total number of items. A sharp increase is observed with the `DP-Sort-By-Value-DESC` strategy, which sorts items in descending order by value. In contrast, ascending sorting by weight or value has a positive effect on cycle time.

Figure 11 shows the average cycle time across all experiment runs ($n = 99$), broken down by sorting strategy. As with Figure 9, it is apparent in Figure 10 that descending sorting by item value leads to the highest average cycle time. Again, the shortest average cycle time across all experiment runs is achieved using the strategy `DP-Sort-By-Value-ASC`.

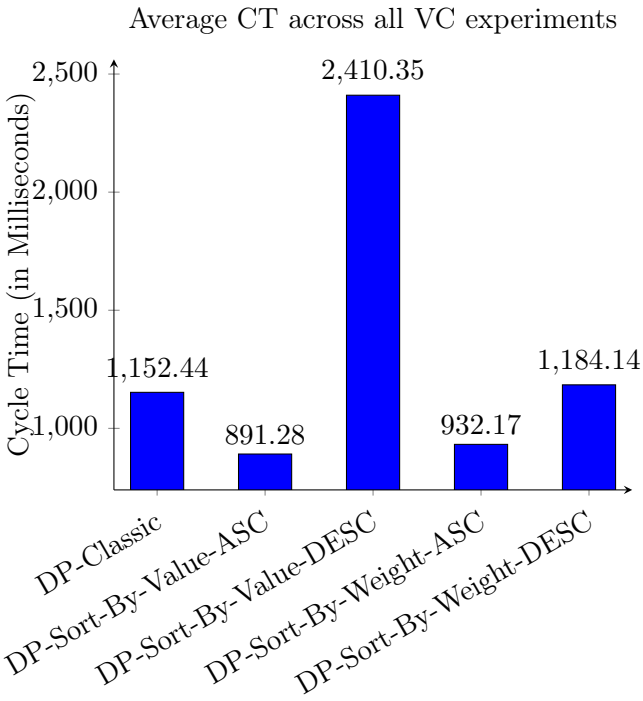

Figure 9: Average cycle time with a variation in knapsack capacity (VC) in the range from 10 to 490. Own illustration, created with TikZ (tikzpicture) being part of TeX Live Version 2024 [13] on the basis of data sets proprocessed with Microsoft Excel 2019 [3]

As part of a significance analysis using a two-tailed hypothesis test, it is evaluated whether deviations in cycle time due to prior sorting result in statistically significant differences compared to the unsorted approach (`DP-Classic`). The significance level is set at $\alpha = 5\%$. As before, the evaluation for the VE experiments is based on the aggregated data from the experiment runs ($n = 99$). It is shown that the differences in average cycle time are significant for the strategies `DP-Sort-By-Value-ASC`, `DP-Sort-By-Value-DESC`, and `DP-Sort-By-Weight-ASC`. However, the application of the `DP-Sort-By-Weight-DESC` strategy did not yield a significant impact on the average cycle time in the VE experiment.

6. Threats to Validity

The findings presented here provide a basis for better estimating the runtime behavior of the KSA. However, it is strongly recommended that these insights be validated through further studies. Given that each experiment included only $n = 99$ runs, the conclusions drawn cannot be considered definitive or universally applicable. In this regard, there is also the limitation that the simulation was conducted on a single server and implemented in one programming language. Different results could occur if the experiment were repeated on various servers and with different programming languages. Additionally, the performance of the server used may have influenced the experimental results. For instance, it is conceivable that background tasks were executed during the experiments, potentially affecting performance.

7. Conclusion

The application of the KSA in two experiments involving variations in knapsack capacity (VC) and the total number of items (VE) revealed partially significant differences in average

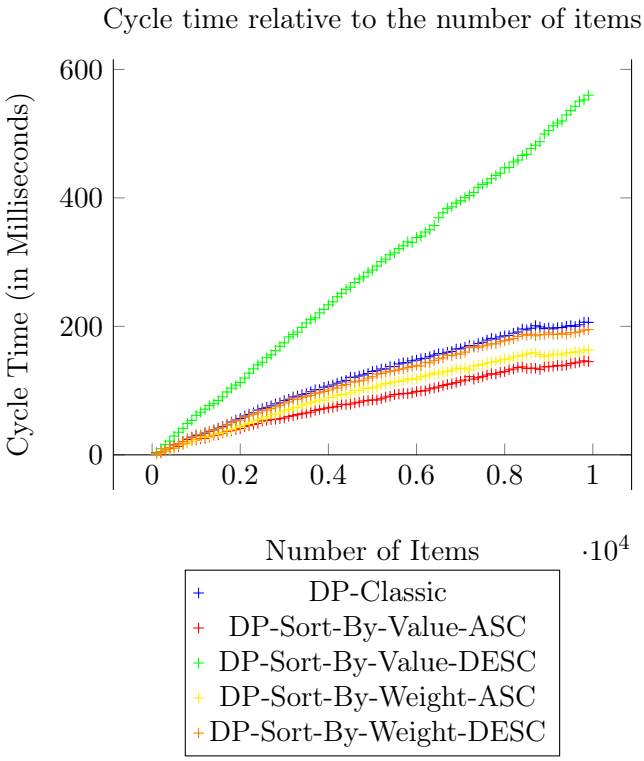

Figure 10: Development of cycle time considering a variation in the total number of items. Own illustration, created with TikZ (tikzpicture) being part of TeX Live Version 2024 [13] on the basis of data sets proprocessed with Microsoft Excel 2019 [3]

cycle times among the investigated strategies. Regarding RQ1, it is evident that prior sorting of items in descending order by value leads to a significant deterioration in cycle time. This effect was confirmed in both the VE and VC experiments. In the VE experiment, where the total number of items increased, sorting items in ascending order by value resulted in a statistically significant improvement in cycle time.

Regarding RQ2, the results of the VE experiment show that the cycle time increases with the total number of items. The same effect was also observed in the VC experiment as the knapsack capacity increased (RQ3). The analysis reveals that increasing the knapsack capacity leads to a more rapid and pronounced rise in cycle time compared to increasing the total number of items.

Future research building on this study should investigate whether the observed effects are influenced by the choice of programming language or the target system. Further studies may also focus on optimizing the KSA to prevent performance degradation, particularly when the knapsack capacity increases.

Overcoming the limitations described in Section represents a valid starting point for future research. Furthermore, it is worth noting that this study exclusively used fictitious, randomly generated data. The use of real-world data could provide valuable additional insights into performance, for example, if real data systematically contain structural similarities or differences that are absent in fictitious data. In addition, extending the analysis to other target variables, such as memory consumption, would also be beneficial.

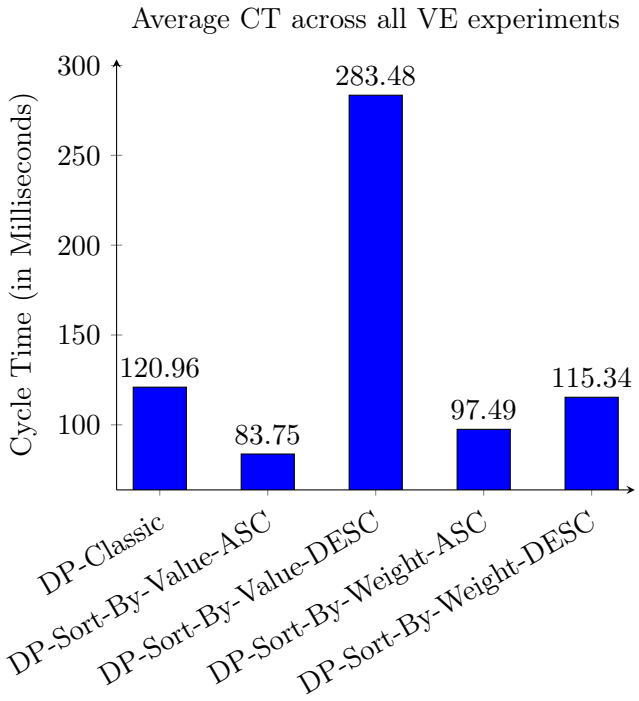

Figure 11: Average cycle time with a variation in the total number of items (VE) in the range from 100 to 9900 items. Own illustration, created with TikZ (tikzpicture) being part of TeX Live Version 2024 [13] on the basis of data sets proprocessed with Microsoft Excel 2019 [3]

8. Acknowledgements

Special thanks are extended to Thomas Zink for providing a LaTeX template that enabled the paper to be formatted in accordance with the present design guidelines. See [14]. We would also like to thank the reviewers for their valuable comments. We would also like to thank the reviewers for their valuable comments.

9. References

[1] Dimitris Bertsimas and Ramazan Demir. An Approximate Dynamic Programming Approach to Multidimensional Knapsack Problems. *Management Science*, 48(4):550 – 565, April 2002. INFORMS.

[2] François Clautiaux, Boris Detienne, and Gaël Guillot. An iterative dynamic programming approach for the temporal knapsack problem. *European Journal of Operational Research*, 293(2), 2021.

[3] Microsoft Corporation. Microsoft© Excel©. Office Home & Business 2019, 2019. Version 2212.

[4] Microsoft Corporation. Microsoft© Visio© 2021 MSO. Visio Standard, 2021. Version 2308 Build 16.0.16731.20052, 64 Bit.

[5] Weiming Feng and Ce Jin. Approximately Counting Knapsack Solutions in Subquadratic Time. In *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 1094–1135. DOI: 10.1137/1.9781611978322.32.

[6] Yosra Ali Hassan and Ibrahim Mahmood Ibrahim. Review on Algorithmic Approaches to Solving Knapsack Problem. *Asian Journal of Research in Computer Science*, 18(3):314–324, February 2025. DOI: 10.9734/ajrcos/2025/v18i3595.

[7] Silvano Martello and Paolo Toth. Algorithms for Knapsack Problems. In Silvano Martello, Gilbert Laporte, Michel Minoux, and Celso Ribeiro, editors, *Annals of Discrete Mathematics. Survey in Combinatorial Optimization*, volume 31 of *North-Holland Mathematics Studies 132*, pages 213 – 257. Elsevier Science Publishers B.V. (North-Holland), Amsterdam, The Netherlands, 1987.

[8] Oracle Corporation. Java™ Platform, Standard Edition Development Kit (JDK™). Java™ SE Runtime Environment. Online available at URL: `https://www.oracle.com/java/technologies/downloads/#javasejdk`, accessed 2022-MAR-07, 2022. Version 17.0.2 2022-01-18 LTS.

[9] Aiying Rong, José Rui Figueira, and Kathrin Klamroth. Dynamic programming based algorithms for the discounted 0–1 knapsack problem. *Applied Mathematics and Computation*, 218(12):6921 – 6933, February 2012. ScienceDirect, Elsevier. DOI: 10.1016/j.amc.2011.12.068.

[10] Gunter Saake and Kai-Uwe Sattler. *Algorithmen und Datenstrukturen.* dpunkt.verlag, 2014. 5. Auflage, Heidelberg.

[11] Tibor Szkaliczki. Solution Methods for the Multiple-Choice Knapsack Problem and Their Applications. *Mathematics*, 13(1097), March 2025. MDPI. DOI: 10.3390/math13071097.

[12] Paolo Toth. Dynamic programming algorithms for the Zero-One Knapsack Problem. 25(1):29–45, 1980.

[13] The TeX Users Group (TUG). TeX Live, 2024. Version 3.141592653-2.6-1.40.26.

[14] Thomas Zink. aircc.cls v 1.1, 2012/09/13, 2012. This is an unofficial Latex class for Authors of AIRCC Papers. Access timestamp: 2016-10-19.