COMP26120 Lab 5 Report

Junle Yu

24 February 2023

1 What makes a problem hard for Dynamic Programming?

1.1 Hypothesis

In hypothesis, the capacity of the knapsack is an aspect that will make a knapsack problem hard for dynamic programming. More specifically, if the number of items is constant and each item's weight and value also remain constant, as the capacity of the knapsack increases, the amount of time for computing an optimal combination of items in the knapsack using dynamic programming will increase. The reason for this hypothesis is that dynamic programming uses a matrix of size (n+1) * (w+1) to store the solutions to the subproblems, where each cell represents a subproblem. To fill in all (n+1) * (w+1) cells in the matrix, and each cell requires constant time to compute, the time complexity for dynamic programming is O(n*w) where w is the knapsack capacity, and n is the number of items.

1.2 Design

The experiment involves the following steps:

1. We generate ten .txt files using kp_generate.py (in Appendix B.1), each with 1000 items. When generating these ten files, we set the upper bound and the knapsack capacity to 100 and 200, respectively, and each item's weight and value are randomly generated. We save the ten .txt files with knapsack capacity set to 200 in a folder named 200_capacity_input_tests.
2. We make another nine copies of this folder, with knapsack capacity change to 400, 600, 800, 1000, 1200, 1400, 1600, 1800, and 2000, respectively (in Appendix A.1).
3. We run the ten files in the folder named 200_capacity_input_tests using a shell script called custom_test.sh (in Appendix B.2). We record the time taken to run each file in a CSV file (in Appendix A.2), which is measured using the UNIX time command. We repeat the third step for the folders with knapsack capacity set to 400, 600, 800, 1000, 1200, 1400, 1600, 1800, and 2000 respectively. We record all data in the same CSV file.
4. We use Pandas to read the CSV file and NumPy to calculate the total time to run all ten files with knapsack capacity = 200. The total time is divided by 10 to find the average time to run each of the ten files. It represents the time to compute an optimal combination of items in the knapsack given knapsack capacity = 200. We repeat the fourth step to calculate the time taken to compute an optimal combination of items in the knapsack given knapsack capacity = 400, 600, 800, 1000, 1200, 1400, 1600, 1800, and 2000, respectively.
5. We use pyplot to plot a resulting graph for all average time values with their corresponding knapsack capacity, with knapsack capacity on the x-axis and average time (measured in milliseconds) on the y-axis. A line of best fit is generated in the form of y = m*x + c using NumPy's polyfit function, in which m is

the gradient of the line, and c is a constant value. Consequently, in this experiment, the only independent variable is the knapsack capacity, the controlled variable is the number of items and the weight and value of each item, and the dependent variable is the time taken to compute an optimal combination of items in the knapsack.

We generate the weight and the value of each item randomly to remove bias because an uneven distribution of weight and value for each item in this experiment might cause bias. We use a large sample size by generating ten files with 1000 items in each file for a specific knapsack capacity to reduce uncertainty and obtain more precise results estimates.

1.3 Results

We use Matplotlib and NumPy to generate the resulting graph in Figure 1. The graph shows how capacity affects the difficulty of the knapsack problem for dynamic programming, measured by the time taken (y-axis) to find a solution for 1000 items with a specific knapsack capacity (x-axis). NumPy's polyfit function generates the line of best fit of all data points in the graph with the formula: y = 0.24*x+9.73, where 0.24 is the line gradient, and 9.73 is a constant value. The line of best fit with gradient value = 0.24 clearly shows a linear relationship between the knapsack capacity and the time to find a solution. More specifically, given the same number of items and for each item, its weight and value remain constant; as the capacity of the knapsack increases, the time taken for computing an optimal combination of items in the knapsack using dynamic programming will increase.

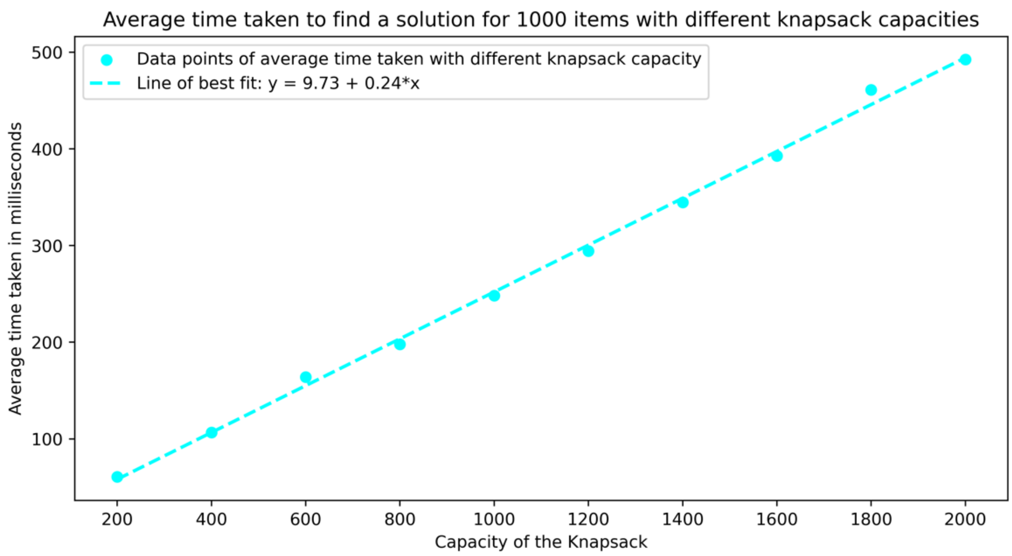

Figure 1: Time taken to find a solution for 1000 items with different knapsack capacities.

1.4 Discussion

The result in the graph confirms the hypothesis that knapsack capacity is an aspect that will make a knapsack problem hard for dynamic programming. There are three perspectives to discuss the correctness and truthfulness of the hypothesis:

1. The line of best fit with gradient = 0.24 shows the linear relationship between knapsack capacity and the time taken to compute an optimal combination of items in the knapsack using dynamic programming. Given that the number of items stays constant, and their weight and value remain constant for each item, the time taken will increase as the knapsack capacity increases.

2. All data points are the actual measurements of average time taken with different knapsack capacities, and no data point deviates from the line of the best fit. Thus, the actual measurements also support the hypothesis. Furthermore, when performing the actual measurements, bias and uncertainty are eliminated, which means the actual measurements have high accuracy that supports the truthfulness of the hypothesis.

3. Theoretically, the time complexity of dynamic programming in solving the knapsack problem is O(n*w). Since in this experiment, n (number of items) is constant, the time complexity of dynamic programming should only be affected by w (knapsack capacity), which should be O(w). It represents a linear relationship between the time to compute an optimal solution and the knapsack capacity.

In conclusion, the accurate actual measurements, the line of best fit generated by the actual measurements, and the time complexity theory of dynamic programming all confirm that the hypothesis is correct.

1.5 Data Statement

In terms of the data statements,

1. All the test files generated by kp_generate.py that are with knapsack capacity set to 200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000 respectively are shown in Appendix A.1
2. The raw data for time taken to run 10 different sets of items with the same knapsack capacity. This is repeated 10 times, with capacity set to 200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000 respectively is shown in Appendix A.2.
3. The raw data of the average time taken to compute an optimal solution with knapsack capacity set to 200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000 respectively is shown in Appendix A.3.
4. The script of kp_generate.py is shown in Appendix B.1.
5. The shell script of custom_test.sh is shown in Appendix B.2.
6. The command lines to generating 10 sets of items, each set has 1000 items are shown in Appendix B.3.
7. The command line to run all tests files using custom_test.sh is shown in Appendix B.4.

Appendix

A Raw Data for the experiment

A.1 All test files generated by kp.generate.py with knapsack capacity set to 200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000 respectively

10 test files of 1000 items with knapsack capacity = 200, in each test file the weight and value are randomly generated for each item

- 200_capacity_input_tests
  - test1.txt
  - test2.txt
  - test3.txt
  - test4.txt
  - test5.txt
  - test6.txt
  - test7.txt
  - test8.txt
  - test9.txt
  - test10.txt

10 test files of 1000 items with knapsack capacity = 400, in each test file the weight and value are randomly generated for each item

- 400_capacity_input_tests
  - test1.txt
  - test2.txt
  - test3.txt
  - test4.txt
  - test5.txt
  - test6.txt
  - test7.txt
  - test8.txt
  - test9.txt
  - test10.txt

10 test files of 1000 items with knapsack capacity = 600, in each test file the weight and value are randomly generated for each item

- 600_capacity_input_tests
  - test1.txt
  - test2.txt
  - test3.txt
  - test4.txt
  - test5.txt
  - test6.txt
  - test7.txt
  - test8.txt
  - test9.txt
  - test10.txt

10 test files of 1000 items with knapsack capacity = 800, in each test file the weight and value are randomly generated for each item

- 800_capacity_input_tests
  - test1.txt
  - test2.txt
  - test3.txt
  - test4.txt
  - test5.txt
  - test6.txt
  - test7.txt
  - test8.txt
  - test9.txt
  - test10.txt

10 test files of 1000 items with knapsack capacity = 1000, in each test file the weight and value are randomly generated for each item

- 1000_capacity_input_tests
  - test1.txt
  - test2.txt
  - test3.txt
  - test4.txt
  - test5.txt
  - test6.txt
  - test7.txt
  - test8.txt
  - test9.txt
  - test10.txt

10 test files of 1000 items with knapsack capacity = 1200, in each test file the weight and value are randomly generated for each item

- 1200_capacity_input_tests
  - test1.txt
  - test2.txt
  - test3.txt
  - test4.txt
  - test5.txt
  - test6.txt
  - test7.txt
  - test8.txt
  - test9.txt
  - test10.txt

10 test files of 1000 items with knapsack capacity = 1400, in each test file the weight and value are randomly generated for each item

- 1400_capacity_input_tests
  - test1.txt
  - test2.txt
  - test3.txt
  - test4.txt
  - test5.txt
  - test6.txt
  - test7.txt
  - test8.txt
  - test9.txt
  - test10.txt

10 test files of 1000 items with knapsack capacity = 1600, in each test file the weight and value are randomly generated for each item

- 1600_capacity_input_tests
  - test1.txt
  - test2.txt
  - test3.txt
  - test4.txt
  - test5.txt
  - test6.txt
  - test7.txt
  - test8.txt
  - test9.txt
  - test10.txt

10 test files of 1000 items with knapsack capacity = 1800, in each test file the weight and value are randomly generated for each item

- 1800_capacity_input_tests
  - test1.txt
  - test2.txt
  - test3.txt
  - test4.txt
  - test5.txt
  - test6.txt
  - test7.txt
  - test8.txt
  - test9.txt
  - test10.txt

10 test files of 1000 items with knapsack capacity = 2000, in each test file the weight and value are randomly generated for each item

- 2000_capacity_input_tests
  - test1.txt
  - test2.txt
  - test3.txt
  - test4.txt
  - test5.txt
  - test6.txt
  - test7.txt
  - test8.txt
  - test9.txt
  - test10.txt

A.2 Time taken to run 10 different sets of items with the same knapsack capacity. This is repeated 10 times, with capacity set to 200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000 respectively.

Capacity of knapsack	Time in milliseconds
200	0.093
200	0.056
200	0.058
200	0.056
200	0.057
200	0.056
200	0.056
200	0.061
200	0.058
200	0.056
400	0.105
400	0.105
400	0.108
400	0.105
400	0.107
400	0.106
400	0.106
400	0.105
400	0.111
400	0.109
600	0.163
600	0.157
600	0.166
600	0.25
600	0.157
600	0.151
600	0.15
600	0.149

600	0.148
600	0.149
800	0.196
800	0.195
800	0.199
800	0.2
800	0.197
800	0.197
800	0.196
800	0.2
800	0.201
800	0.198
1000	0.248
1000	0.246
1000	0.249
1000	0.251
1000	0.255
1000	0.253
1000	0.25
1000	0.244
1000	0.242
1000	0.243
1200	0.291
1200	0.29
1200	0.293
1200	0.295
1200	0.3
1200	0.292
1200	0.295
1200	0.295
1200	0.298
1200	0.295
1400	0.347

1400	0.347
1400	0.347
1400	0.341
1400	0.344
1400	0.344
1400	0.343
1400	0.343
1400	0.349
1400	0.341
1600	0.391
1600	0.396
1600	0.389
1600	0.394
1600	0.393
1600	0.389
1600	0.401
1600	0.391
1600	0.391
1600	0.392
1800	0.441
1800	0.446
1800	0.439
1800	0.443
1800	0.455
1800	0.521
1800	0.487
1800	0.475
1800	0.449
1800	0.455
2000	0.5
2000	0.501
2000	0.503
2000	0.489

2000	0.485
2000	0.489
2000	0.486
2000	0.498
2000	0.483
2000	0.49

A.3 Average time taken to compute an optimal solution with knapsack capacity set to 200, 400, 600, 800, 1000, 1200, 1400, 1600, 1800, 2000 respectively.

Capacity of the Knapsack	Time in milliseconds
200.0	0.0607
400.0	0.1067
600.0	0.1640
800.0	0.1979
1000.0	0.2481
1200.0	0.2944
1400.0	0.3446
1600.0	0.3927
1800.0	0.4611
2000.0	0.4924

B Shell Scripts for this experiment

B.1 kp_generate.py

```
import random
import sys

n = int(sys.argv[1])
c = sys.argv[2]
b = int(sys.argv[3])
name = sys.argv[4]

fileName = name
fileObj = open(fileName,'w')
fileObj.write(str(n) + "\n")

for i in range(1, n + 1):
    weight = random.randint(1, b)
    profit = random.randint(1, b)
    fileObj.write(str(i) + " " + str(profit) + " " + str(weight) + "\n")
fileObj.write(str(c))
```

B.2 custom_test.sh

```
function timeout() { perl -e 'alarm shift; exec @ARGV' "$@"; }
```

```
function for_c {
  echo "c/$1"
}
function for_java {
  echo "java -cp java comp26120.$1_kp"
}
function for_python {
  echo "python3 python/$1_kp.py"
}
function get {
  if [[ $1 == "c" ]]; then for_c $2; fi
  if [[ $1 == "java" ]]; then for_java $2; fi
  if [[ $1 == "python" ]]; then for_python $2; fi
}


lang="c"
if [ $# -eq 1 ]
then
  lang=$1
fi
if [[ "$lang" != "c"  &&  "$lang" != "java"  &&  "$lang" != "python" ]]
then
  echo "Supply either c, java, or python as the language"
  exit
fi

algs=(dp)
inputs=( 'test1.txt' 'test2.txt' 'test3.txt' 'test4.txt' 'test5.txt'
'test6.txt' 'test7.txt' 'test8.txt' 'test9.txt' 'test10.txt')
declare -A times=(
  ['test1.txt']=20
  ['test2.txt']=20
  ['test3.txt']=20
  ['test4.txt']=20
  ['test5.txt']=20
  ['test6.txt']=20
  ['test7.txt']=20
  ['test8.txt']=20
  ['test9.txt']=20
  ['test10.txt']=20
)
Sizes=( 200 400 600 800 1000 1200 1400 1600 1800 2000 )
  for SIZE in ${Sizes[@]}
  do
    for FILE in ${inputs[@]}
      do
      LIMIT="${times[$FILE]}"
      echo "Running on $FILE for $LIMIT seconds"
      echo
      echo "\begin{tabular}{|l|l|l|l|} \hline"
      echo "Algorithm & Optimal Value & Time Taken & Result \\\\ \hline"
        for alg in ${algs[@]}
        do
        RUN=$(get $lang $alg)
        TIME=$({ time timeout ${LIMIT}s ${RUN}
data/${SIZE}_capacity_input_tests/$FILE > ${alg}_${FILE}_out ; } 2>&1 |
grep real | grep -o '[0-9].*')
        LAST=$(grep -o '\(Current best solution\|value\)=[0-9]*'
${alg}_${FILE}_out | tail -1)
        VALUE=$(echo $LAST | sed -n -e 's/.*=//g' -e 'p')
```

```
  #
        minutes=${TIME%m*}    # remove everything after "m" (including "m")
        seconds=${TIME#*m}
        seconds=${seconds%*s}  # remove everything after "s" (including
"s")
        minutes=$(($minutes*60))
        total_seconds=$(echo "$minutes + $seconds" | bc)

        printf -v alg %-10.10s $alg
        printf -v VALUE %-25.25s "$VALUE"
        echo "$alg & $VALUE & ${TIME} & $CORRECT  \\\\"
        echo $SIZE, $total_seconds>> data/data_python.csv
  done
  done
  done
  echo "\hline \end{tabular}"
  echo
  echo
```

B.3 Generating 10 sets of items, each set has 1000 items

Command Line

python3 kp_generate.py 1000 200 100 test1.txt

python3 kp_generate.py 1000 200 100 test2.txt

python3 kp_generate.py 1000 200 100 test3.txt

python3 kp_generate.py 1000 200 100 test4.txt

python3 kp_generate.py 1000 200 100 test5.txt

python3 kp_generate.py 1000 200 100 test6.txt

python3 kp_generate.py 1000 200 100 test7.txt

python3 kp_generate.py 1000 200 100 test8.txt

python3 kp_generate.py 1000 200 100 test9.txt

python3 kp_generate.py 1000 200 100 test10.txt

B.4 Run all tests files using custom_test.sh

Command Line

bash custom_test.sh python