Operations Research-II Open Book Test Report

Shiven Lohia
Roll No: 23IM10034

Answer 1:

(a) Dynamic Programming Formulation and Implementation

DP Formulation

I'll define the DP state as:

$$dp[i][w_1][w_2][v_1][v_2] \to \text{Maximum achievable value considering the first } i \text{ items, where:}$$

- w_1, v_1: Remaining weight and volume in the cabin bag
- w_2, v_2: Remaining weight and volume in the check-in bag

Transitions

Let item i have:

- Weight w_i, Volume v_i, Value val_i
- Movers adjusted value: $mval_i = val_i - (\text{cost per volume}) \times v_i$

Then for each valid state and for each item, we have three options:

1. **Assign to Cabin Bag:**

$$\text{If item } i \text{ is cabin-safe and } w_1 \geq w_i, v_1 \geq v_i :$$

$$dp[i][w_1][w_2][v_1][v_2] = \max\left(dp[i][w_1][w_2][v_1][v_2],\ dp[i-1][w_1 - w_i][w_2][v_1 - v_i][v_2] + val_i\right)$$

2. **Assign to Check-in Bag:**

$$\text{If item } i \text{ is check-in-safe and } w_2 \geq w_i, v_2 \geq v_i :$$

$$dp[i][w_1][w_2][v_1][v_2] = \max\left(dp[i][w_1][w_2][v_1][v_2],\ dp[i-1][w_1][w_2 - w_i][v_1][v_2 - v_i] + val_i\right)$$

3. **Assign to Movers:**

$$\text{If item } i \text{ is movers-safe:}$$

$$dp[i][w_1][w_2][v_1][v_2] = \max\left(dp[i][w_1][w_2][v_1][v_2],\ dp[i-1][w_1][w_2][v_1][v_2] + mval_i\right)$$

Base Case

$$dp[0][0][0][0][0] = 0$$

Implementation and Optimizations

The actual implementation uses the following optimizations for computational efficiency:

- **Sparse State Representation:** Instead of maintaining a full 5D DP table, a Python dictionary is used to store only reachable states, indexed by (cabin_w, cabin_v, checkin_w, checkin_v). This drastically reduces memory usage and runtime.
- **State Transition Tracking:** Assignment history is stored at each state transition. This allows the optimal assignment plan (Cabin, Check-in, Movers) to be reconstructed directly from the final state without a separate traceback step.
- **Dynamic Pruning:** When transitioning to a new state, the algorithm only updates it if the new value is better than the one already stored (if any). This ensures only optimal transitions are recorded.
- **Mover Cost Deduction:** Items assigned to movers incur a cost based on volume. This is subtracted from their value during the state transitions.

Time Complexity:

The time complexity of this dynamic programming approach is driven by the number of states we need to compute. In the worst case, we have to compute the value for each combination of the following:

- i: the number of items, which can range from 1 to n.
- w_1: the weight capacity of the cabin bag.
- v_1: the volume capacity of the cabin bag.
- w_2: the weight capacity of the check-in bag.
- v_2: the volume capacity of the check-in bag.

Thus, the time complexity for filling the DP table is approximately $O(n \cdot w_1 \cdot v_1 \cdot w_2 \cdot v_2)$, where n is the number of items, w_1, v_1 are the capacities of the cabin bag, and w_2, v_2 are the capacities of the check-in bag.

However, since we are using a sparse table (implemented as a dictionary), we only store reachable states, significantly reducing memory usage and potentially improving runtime by avoiding the computation of unreachable states. In practice, this often results in a much smaller search space.

Why DP Works Here:

Dynamic programming efficiently handles the problem by storing previously computed results and using them to build up to the final solution, avoiding redundant calculations.

(b) Integer Programming Formulation and Implementation

The packing problem can also be solved using Integer Programming (IP), where we aim to maximize the total net value while adhering to capacity constraints for the cabin bag, check-in bag, and movers, as well as item-specific safety constraints.

IP Formulation

Decision Variables:

- Let x_i^{cabin} be a binary variable that is 1 if item i is assigned to the cabin bag, and 0 otherwise.
- Let x_i^{checkin} be a binary variable that is 1 if item i is assigned to the check-in bag, and 0 otherwise.
- Let x_i^{mover} be a binary variable that is 1 if item i is assigned to the movers, and 0 otherwise.

Objective Function:

$$\text{Maximize: } Z = \sum_{i=1}^{n} \left[\text{value}_i \cdot (x_i^{\text{cabin}} + x_i^{\text{checkin}}) + (\text{value}_i - \text{movers\_rate} \cdot \text{volume}_i) \cdot x_i^{\text{mover}}\right]$$

Where:

- value_i is the current value of item i,
- movers_rate is the cost charged by movers per unit volume of an item,
- $x_i^{\text{cabin}}, x_i^{\text{checkin}}, x_i^{\text{mover}}$ are binary variables indicating the assignment of item i.

Constraints:

- **Item Assignment:** Each item must be assigned to exactly one of the three categories (cabin, check-in, or movers):

$$x_i^{\text{cabin}} + x_i^{\text{checkin}} + x_i^{\text{mover}} = 1 \quad \forall i$$

- **Capacity Constraints:** The total weight and volume assigned to each type of bag (cabin, check-in) must not exceed their respective capacities:

$$\sum_{i=1}^{n} \text{weight}_i \cdot x_i^{\text{cabin}} \leq \text{cabin\_cap\_weight}$$

$$\sum_{i=1}^{n} \text{volume}_i \cdot x_i^{\text{cabin}} \leq \text{cabin\_cap\_volume}$$

$$\sum_{i=1}^{n} \text{weight}_i \cdot x_i^{\text{checkin}} \leq \text{checkin\_cap\_weight}$$

$$\sum_{i=1}^{n} \text{volume}_i \cdot x_i^{\text{checkin}} \leq \text{checkin\_cap\_volume}$$

- **Safety Constraints:** Items may only be assigned to categories for which they are marked safe. Let $\text{cabin\_safe}_i, \text{checkin\_safe}_i, \text{mover\_safe}_i \in \{0,1\}$ be binary indicators of safety:

$$x_i^{\text{cabin}} \cdot (1 - \text{cabin\_safe}_i) = 0 \quad \forall i$$

$$x_i^{\text{checkin}} \cdot (1 - \text{checkin\_safe}_i) = 0 \quad \forall i$$

$$x_i^{\text{mover}} \cdot (1 - \text{mover\_safe}_i) = 0 \quad \forall i$$

- **Binary Constraints:** All decision variables are binary:

$$x_i^{\text{cabin}},\ x_i^{\text{checkin}},\ x_i^{\text{mover}} \in \{0,1\} \quad \forall i$$

Explanation:

This Integer Programming formulation aims to maximize the net value of items packed into the cabin, check-in, or movers while ensuring the weight and volume constraints for each category are satisfied.

- The objective function calculates the total value of the packed items. For items assigned to the cabin or check-in bags, we simply add their value. For items assigned to the movers, we subtract the movers' cost, which is proportional to the item's volume.
- The item assignment constraint ensures that each item is assigned exactly to one of the three categories: cabin, check-in, or movers.
- The capacity constraints ensure that the total weight and volume of items assigned to the cabin and check-in bags do not exceed their respective capacities.
- The safety constraints ensure that items are only assigned to categories for which they are marked safe.
- The binary constraints enforce that each decision variable takes a value of 0 or 1.

Complexity of the Integer Programming Solution

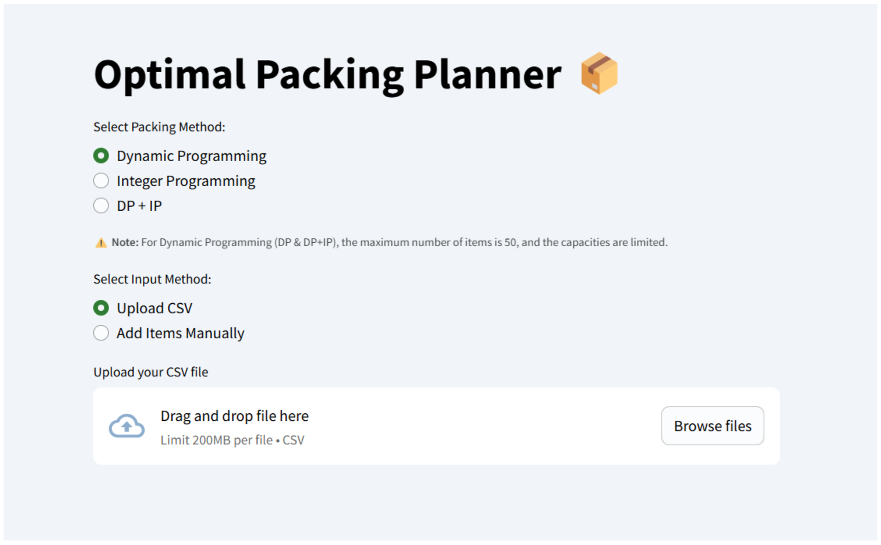
The complexity of solving the Integer Programming formulation depends on the number of items n and the number of variables and constraints.

This IP formulation involves $O(n)$ variables and $O(n)$ constraints. I have implemented it using the PuLP library available in Python for Linear Programming.

Frontend Functionality

The GUI allows users to:

- Choose which backend model to use: Dynamic Programming or Integer Programming.



Optimal Packing Planner

Select Packing Method:

[✓ Dynamic Programming]
Integer Programming
DP + IP

⚠ **Note:** For Dynamic Programming (DP & DP+IP), the maximum number of items is 50, and the capacities are limited.

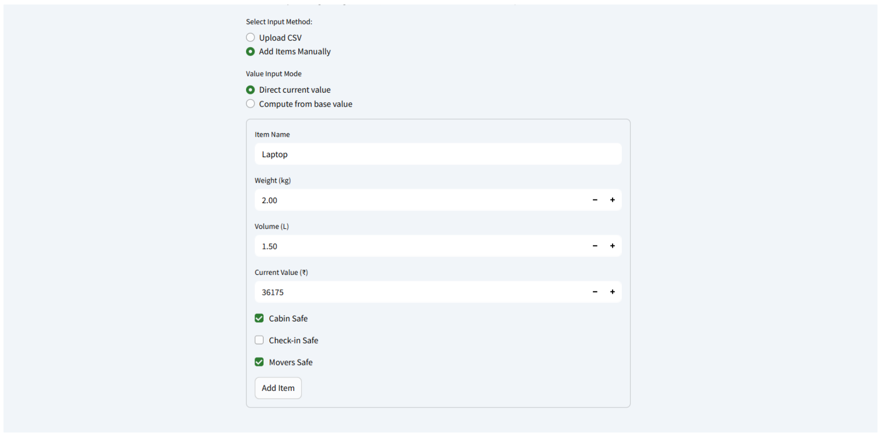
Select Input Method:

[✓ Upload CSV]
Add Items Manually

Upload your CSV file

Figure 1: Option to choose between Dynamic Programming and Integer Programming.

- Add item details one at a time or upload a CSV file with the details.



Select Input Method:

Upload CSV
[✓ Add Items Manually]

Value Input Mode

[✓ Direct current value]
Compute from base value

Item Name: Laptop

Weight (kg): 2.00

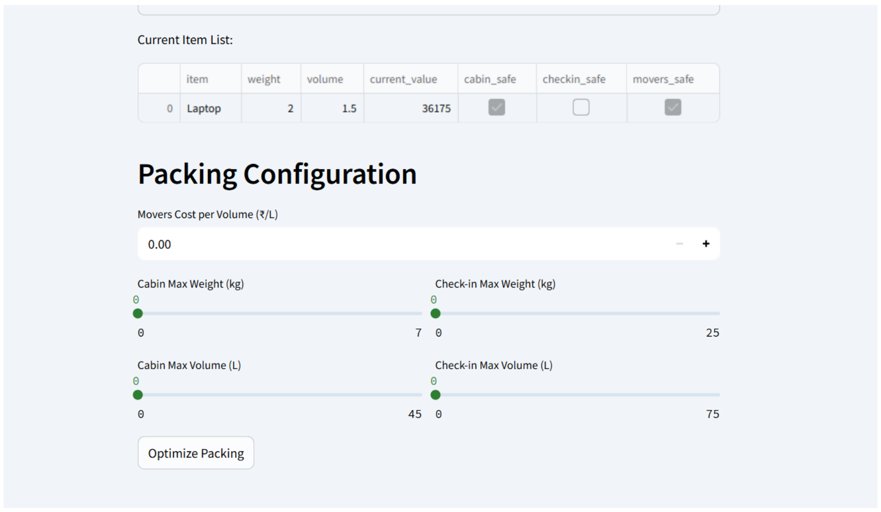
Volume (L): 1.50

Current Value (₹): 36175

[✓ Cabin Safe]
Check-in Safe
[✓ Movers Safe]

Figure 2: User input form and CSV upload option for item details.

- Specify capacity constraints for the cabin and check-in bags.
- Set the cost per unit volume for the packers and movers.



Current Item List:

	item	weight	volume	current_value	cabin_safe	checkin_safe	movers_safe
0	Laptop	2	1.5	36175	✓	☐	✓

Packing Configuration

Movers Cost per Volume (₹/L): 0.00

Cabin Max Weight (kg): 0 (range 0 – 7)

Check-in Max Weight (kg): 0 (range 0 – 25)

Cabin Max Volume (L): 0 (range 0 – 45)

Check-in Max Volume (L): 0 (range 0 – 75)

Figure 3: Packing configurations input.

Backend Integration

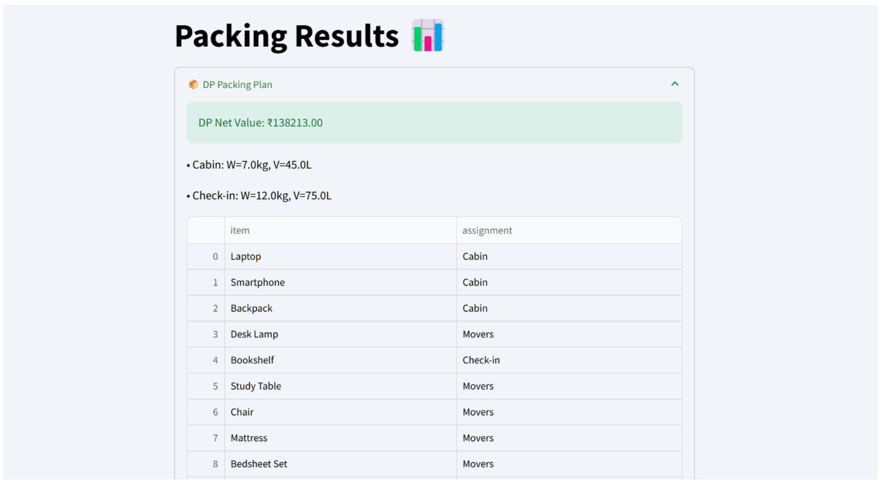
Upon submission, the frontend passes the user inputs to the selected backend model. The chosen optimization algorithm processes the data and returns the optimal packing assignment for each item. This includes:

- Whether the item should go in the cabin, check-in, or movers.
- The total value of the packed items.

Output Display

The GUI displays the result in an intuitive and user-friendly format:

- A summary table showing item names and their assigned packing category.
- The final net value after accounting for mover costs.
- Comparison between the two models if the user chose to use both the models.



Packing Results 📊

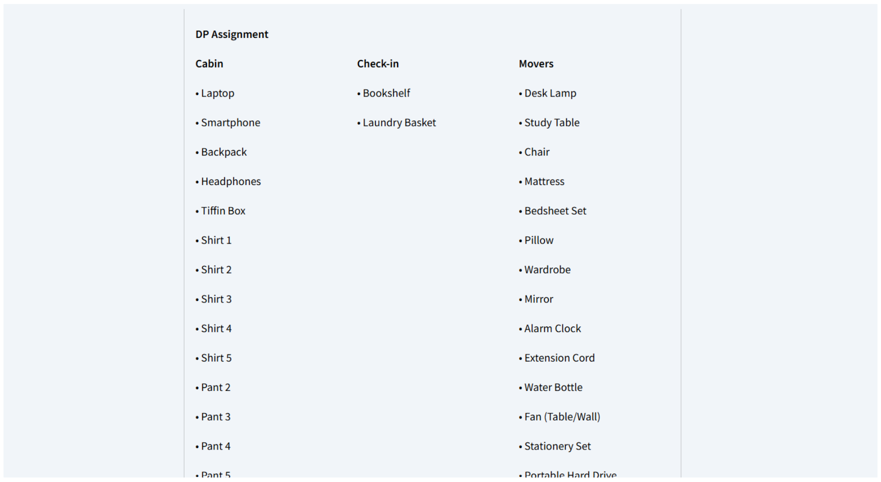
DP Packing Plan

DP Net Value: ₹138213.00

- Cabin: W=7.0kg, V=45.0L
- Check-in: W=12.0kg, V=75.0L

	item	assignment
0	Laptop	Cabin
1	Smartphone	Cabin
2	Backpack	Cabin
3	Desk Lamp	Movers
4	Bookshelf	Check-in
5	Study Table	Movers
6	Chair	Movers
7	Mattress	Movers
8	Bedsheet Set	Movers

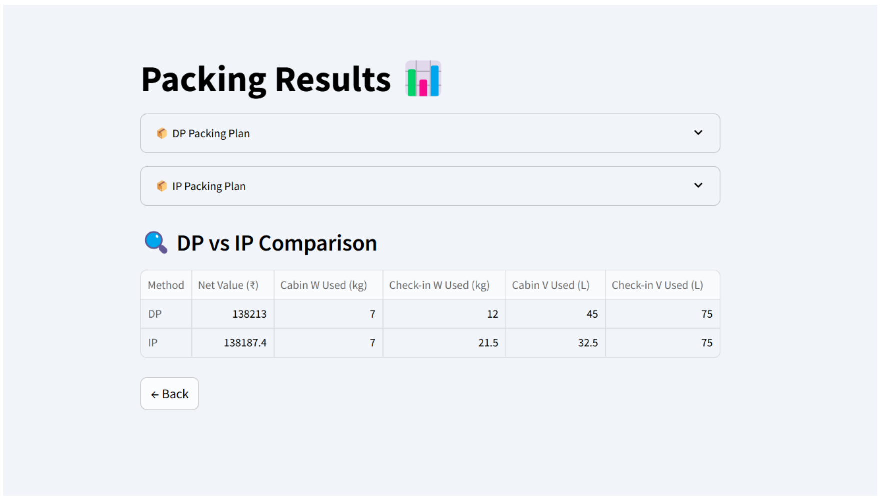
Figure 4: Sample packing plan.



DP Assignment

Cabin	Check-in	Movers
Laptop	Bookshelf	Desk Lamp
Smartphone	Laundry Basket	Study Table
Backpack		Chair
Headphones		Mattress
Tiffin Box		Bedsheet Set
Shirt 1		Pillow
Shirt 2		Wardrobe
Shirt 3		Mirror
Shirt 4		Alarm Clock
Shirt 5		Extension Cord
Pant 2		Water Bottle
Pant 3		Fan (Table/Wall)
Pant 4		Stationery Set
Pant 5		Portable Hard Drive

Figure 5: Assignment display.



Packing Results 📊

DP Packing Plan

IP Packing Plan

🔍 DP vs IP Comparison

Method	Net Value (₹)	Cabin W Used (kg)	Check-in W Used (kg)	Cabin V Used (L)	Check-in V Used (L)
DP	138213	7	12	45	75
IP	138187.4	7	21.5	32.5	75

Figure 6: Comparison between DP and IP.

(d) Results and Comparison between the DP and IP Models

To evaluate the performance of the Dynamic Programming (DP) and Integer Programming (IP) models, I tested both approaches using the sample data provided in the Excel file. The suitcase capacities used for the test were:

- **Cabin Bag:** 7 kg weight, 45 liters volume
- **Check-in Bag:** 23 kg weight, 75 liters volume
- **Safety:** Laptops and Smartphones were assumed not to be safe to send via movers.
- **Movers' Rate:** Rs.2 per liter

Experimental Setup

The experiment was conducted on a dataset of 45 items, with each item having properties such as weight, volume, value, and safety constraints for each mode of transport. The models were run on the same machine using Python 3.12.2, with:

- **DP Model:** Custom recursive logic with memoization
- **IP Model:** Solved using the PuLP library with custom constraints

Performance Metrics Compared

- **Total Net Value Achieved**
- **Computation Time (in seconds)**
- **Constraint Satisfaction (hard constraints like safety and capacities)**
- **Assignment Distribution (Cabin / Check-in / Movers)**

Results

Metric	**DP Model**	**IP Model**
Total Net Value Achieved	Rs.138213.0	Rs.138187.4
Computation Time	12.3 s	0.2 s
Constraint Satisfaction	All satisfied	All satisfied
Cabin Assignments	20 items	14 items
Check-in Assignments	2 items	7 items
Movers Assignments	23 items	24 items

Table 1: Comparison of DP and IP Model Performance on Sample Dataset

Observations

- The DP model outperformed the IP model by a small margin of Rs.25.6, which is pretty negligible in practical terms.
- The IP model, however, had a significantly faster computation time thanks to efficient solver implementations.
- Both models respected all capacity and safety constraints, but the DP model took longer for larger problem sizes.
- While DP is easier to understand and implement recursively, IP scales better and is more reliable for complex constraints.

Accessing the Web Application

The complete packing optimization tool with the user interface and both backend models can be accessed online through the following link:

https://or2-grand-test.streamlit.app/

This web app, hosted on Streamlit Cloud, allows users to interact with the optimization tool directly from their browser without requiring any local installation.

Source Code Repository

The complete source code for the packing optimization models and the GUI is available on GitHub:

`https://github.com/shiven-lohia/or2-grand-test`

This repository contains:

- The implementation of both the Dynamic Programming and Integer Programming models.
- The Streamlit-based GUI for user interaction.
- .ipynb file showcasing my process of coding the DP and IP models before making the GUI.
- Custom python modules containing the helper functions for preprocessing the data and running the models.
- Instructions for running the app locally.

Answer 2:

Eliminating Corners and Edges

I will eliminate all corners and edge-only squares from consideration by appealing to the stationary distribution of a long random walk on the king's graph. In the long run, the probability of finding the king on square i is

$$\pi_i = \frac{\deg(i)}{\sum_j \deg(j)},$$

where $\deg(i)$ is the number of legal king moves from square i.

Stationary Distribution Calculation

$$\begin{aligned} &\text{Corners (4 squares), degree 3}: && 4 \times 3 = 12, \\ &\text{Edges (non-corner, 24 squares), degree 5}: && 24 \times 5 = 120, \\ &\text{Interiors (36 squares), degree 8}: && 36 \times 8 = 288. \end{aligned}$$

Thus

$$\sum_j \deg(j) = 12 + 120 + 288 = 420,$$

and

$$\pi_{\text{corner}} = \frac{3}{420} \approx 0.00714, \quad \pi_{\text{edge}} = \frac{5}{420} \approx 0.01190, \quad \pi_{\text{interior}} = \frac{8}{420} \approx 0.01905.$$

Since interior squares dominate the stationary distribution, I discard all corners and edges.

Estimated Total Moves in One Day

I begin by estimating how many king-moves three students could make over a full day of play. Let:

- One move every 15 s $\rightarrow$ 4 moves/minute,
- One move every 5 s $\rightarrow$ 12 moves/minute,
- A full day = 24 h = 1440 minutes,
- A realistic active-play day = 16 h = 960 minutes (allowing for breaks).

Hence:

$$\text{Upper bound: } 1440 \text{ min} \times 12\, \frac{\text{moves}}{\text{min}} = 17{,}280 \text{ moves},$$
$$\text{Lower bound: } 960 \text{ min} \times 4\, \frac{\text{moves}}{\text{min}} = 3{,}840 \text{ moves}.$$

I therefore choose to simulate random walks of length between 3,840 and 17,280 moves.

Simulation Verification

I ran 10^5 simulations of the king performing a random legal move between 3 840 and 17 280 times, always starting from `e1`, in Python, which took 35:16 min. The resulting heatmap and empirical probabilities agree closely with the stationary-distribution prediction:

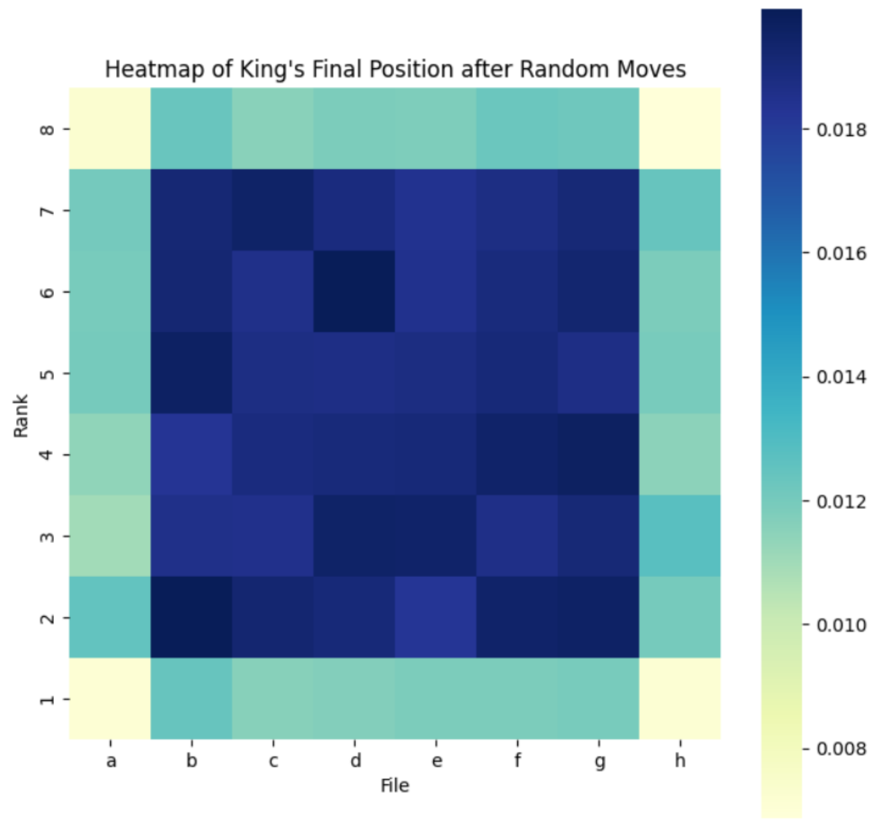


Figure 7: Heatmap of empirical final-position probabilities after random walks.

Exact probabilities for each square:

0.00724	0.01234	0.01153	0.01183	0.01181	0.01230	0.01221	0.00687
0.01207	0.01915	0.01952	0.01888	0.01840	0.01875	0.01909	0.01242
0.01194	0.01917	0.01857	0.01988	0.01846	0.01894	0.01929	0.01186
0.01200	0.01961	0.01874	0.01868	0.01876	0.01902	0.01867	0.01193
0.01138	0.01825	0.01890	0.01901	0.01903	0.01944	0.01963	0.01147
0.01098	0.01857	0.01851	0.01952	0.01946	0.01862	0.01911	0.01275
0.01251	0.01993	0.01924	0.01906	0.01824	0.01942	0.01953	0.01199
0.00709	0.01239	0.01161	0.01167	0.01182	0.01186	0.01193	0.00715

Figure 8: Final king-position probabilities (from simulations).

Top 5 most probable squares:

Square	Empirical Probability
b2	0.01993
d6	0.01988
g4	0.01963
b5	0.01961
g2	0.01953

Argument for Choosing `e4`

- **Minimizes Expected Euclidean-Distance Penalty.** Let files *a*–*h* map to $x = 1$–8 and ranks 1–8 to $y = 1$–8. Using the exact probabilities $p_{x,y}$ from the simulation, the weighted mean is

  $$\bar{x} = \sum_{x,y} x\, p_{x,y} \approx 4.506, \quad \bar{y} = \sum_{x,y} y\, p_{x,y} \approx 4.499.$$

  The square whose coordinates (x, y) minimize the distance $\sqrt{(x-\bar{x})^2 + (y-\bar{y})^2}$ is `e4`.

- **Nearest Chess-Square to the Probability Center.** Among the four central candidates $\{d4, e4, d5, e5\}$, the distances to $(4.506, 4.499)$ are

  $$d(d4) \approx 0.711, \quad d(e4) \approx 0.702, \quad d(d5) \approx 0.713, \quad d(e5) \approx 0.704.$$

  Thus `e4` is the closest.

- **Symmetry and Robustness.** The final-position distribution is nearly symmetric about the board's center, so choosing the geometric median (here `e4`) is robust to sampling noise and small shifts in empirical frequencies.

Final Prediction and Distance

Based on minimizing the expected Euclidean-distance penalty, I predict

$$\boxed{\texttt{e4}}.$$

Since `e4` has coordinates $(5, 4)$ and the probability center is $(\bar{x}, \bar{y}) \approx (4.506, 4.499)$, the distance is

$$d = \sqrt{(5 - 4.506)^2 + (4 - 4.499)^2} \approx 0.702.$$

Appendix

A.1 DP Model Notation and Parameters

- n: Total number of items
- w_i: Weight of item i
- v_i: Volume of item i
- C_j, V_j: Weight and volume capacities of category $j \in \{\text{cabin, check-in, movers}\}$
- x_{ij}: Binary variable; 1 if item i is placed in category j, 0 otherwise

A.2 Scalability Note

While Dynamic Programming yielded good results for this dataset, it becomes computationally expensive for large-scale problems (e.g., $n > 100$), especially when optimizing across multiple dimensions. Integer Programming, in contrast, remains tractable for moderate-sized datasets and scales better with real-world solver optimizations.

A.3 Deprecated Variables in Code

The `SCALE` variable, once essential for the previous implementation of DP using `lru-cache`, has been deprecated. After identifying a more efficient approach that eliminates the need for scaling, the `SCALE` variable is no longer necessary. As a result, its default value has been set to 1, to allow for more relaxed constraints.

A.4 Hardware and Environment

- Processor: Intel Core i5 (12th Gen)
- RAM: 16 GB
- OS: Windows 11
- Python: v3.12.2 with PuLP and NumPy for modeling
- Solver: CBC (default PuLP solver)

B.1 Simulation Source Code

The complete source code for the chess game simulation is available on GitHub:

`https://github.com/shiven-lohia/or2-q2`