P1: UI optimization

@Flora(weierx@andrew.cmu.edu)

Initial formulations

Version-0: Basic Rules

The initial version（V-0) of the UI optimization focuses on enforcing basic constraints to ensure a valid layout. This version does not consider relevance, level of detail (LoD) preferences, or interaction costs but strictly adheres to the fundamental placement rules.

The Basic Rules formulation ensures a structurally valid UI layout without considering any preference-based optimization. It guarantees:

- A maximum of four applications are displayed.
- Each application is only displayed once at a single LoD.
- Applications do not overlap each other.
- Applications remain within grid boundaries.
- Reserved UI elements (question panel, AllApp button, ROI) are not obstructed.

Constraints

1. **Max 4 Elements Placed**

To prevent UI clutter and ensure usability, a maximum of four elements can be placed in the interface. This is enforced through the following constraint:

$$\sum_{app,lod,xIdx,yIdx} x_{app,lod,xIdx,yIdx} \leq 4$$

This ensures that no more than four applications are selected and placed on the interface. **(However in practice, in most cases the system will choose to place the maximum number of components.)**

2. **Each Element Can Only Be Displayed in One LoD**

$$\sum_{lod,xIdx,yIdx} x_{app,lod,xIdx,yIdx} \leq 1, \quad \forall app$$

This prevents an application from appearing in multiple LoDs simultaneously.

3. **Ensure Applications Fit Within Grid Boundaries**

Applications with larger LoD sizes must not extend beyond the screen boundary. This constraint ensures that an app's placement does not exceed the grid's available dimensions:

$$xIdx + \text{width} \leq \text{columns}, \quad yIdx + \text{height} \leq \text{rows}$$

4. **No Overlapping Between Elements**

To ensure that applications do not overlap, I track the occupied grid positions and enforce a constraint that no two applications can occupy the same grid slot:

$$\sum x_{app,lod,xIdx+dx,yIdx+dy} \leq 1, \quad \forall (x, y) \text{ in grid}$$

where dx and dy are determined by the width and height of the element based on its LoD.

5. **Avoid Overlapping the "All Apps" Button and Question Panel**

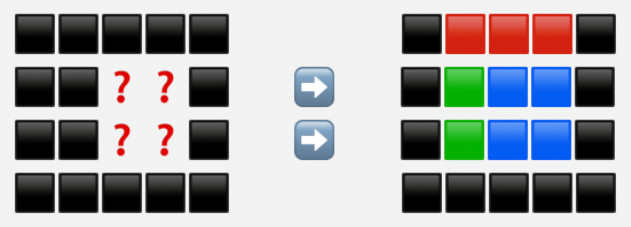
Certain areas of the UI are reserved for buttons and panels, these areas are marked as restricted and applications are not allowed to be placed in these regions.

$$\sum x_{app,lod,xIdx,yIdx} = 0, \quad \forall (xIdx, yIdx) \text{ in restricted areas}$$

Since applications can have different LoDs, their sizes vary. So I d**efine different restricted areas for each LoD to prevent overlap especially with the Question Panel**, the following schematic visually represents the restricted areas for **LoD 0, LoD 1, and LoD 2** around the **Question Panel (Q)**.

💡 Each box represents a **grid cell**, where:

- ❓ **Question signal**: the Question Panel(2×2)
- 🟦 **Blue** cells: restricted area for **LoD 0 & 1 & 2**
- 🟩 **Green** cells: restricted area for **LoD 1 & 2**
- 🟥 **Red** cells: restricted area for **LoD 2**
- ⬛ **Black** cells: unrestricted area



6. **Avoid Overlapping with the Region of Interest (ROI)**

   The UI includes a Region of Interest, which should remain visible and unobstructed. Here I check if their bounding box intersects with the circular ROI area by function `scene_UI.circle_rectangle_overlap` provided in initial code framework in `ui.py`

$$\sum x_{app,lod,xIdx,yIdx} = 0, \quad \forall (xIdx, yIdx) \text{ in ROI}$$

Version-1: Relevance-Based Optimization

In this formulation, the optimization only focuses on maximizing the sum of relevance scores for the selected applications.

$$\max \sum_{\text{app}} \sum_{\text{lod}} \sum_{x=0}^{column-1} \sum_{y=0}^{row-1} R_{\text{app}} \cdot x_{\text{app,lod},x,y}$$

- R_{app} is the relevance score of application app.
- $x_{\text{app,lod},x,y}$ is a binary decision variable, indicating whether application app is placed at position (x,y) with a specific LoD.

Each application's relevance score (get from Scene Json file) is used as the optimization criterion. The objective is to select applications with the highest relevance scores while satisfying spatial constraints. However, LoD is not considered, meaning all levels of detail are treated equally.

Version-2: Relevance with LoD Preference

This formulation extends **Version-1** by adding a **LoD weight** to prioritize higher levels of detail. Higher LoDs provide a richer UI experience but take up more space.

$$\max \sum_{\text{app}} \sum_{\text{lod}} \sum_{x=0}^{column-1} \sum_{y=0}^{row-1} R_{\text{app}} \cdot (1 + 0.5 \cdot \text{lod}) \cdot x_{\text{app,lod},x,y}$$

Assign weights to encourage the selection of higher LoDs when possible.

- **LoD 0 → 1.0** (smallest, lowest detail)
- **LoD 1 → 1.5** (medium size and detail)
- **LoD 2 → 2.0** (largest, highest detail)

Version-3: Relevance, LoD Preference, and Interaction Cost

This version improves **Version-2** by adding an **interaction cost** component, making the UI **more accessible and user-friendly**. Applications should be placed closer to the question panel, and the further an application is placed, the higher its interaction cost.

$$\max \sum_{\text{app}} \sum_{\text{lod}} \sum_{x=0}^{column-1} \sum_{y=0}^{row-1} \frac{R_{\text{app}}}{1 + \lambda \cdot d(x,y)} \cdot (1 + 0.5 \cdot \text{lod}) \cdot x_{\text{app,lod},x,y}$$

where:

- $d(x,y)$ is the squared Euclidean distance from the application position (x,y) to the center of the question panel:

$$d(x,y) = \left(x + \frac{w_{\text{lod}}}{2} - x_q\right)^2 + \left(y + \frac{h_{\text{lod}}}{2} - y_q\right)^2$$

- x_q, y_q are the coordinates of the question panel's center.
- $w_{\text{lod}}, h_{\text{lod}}$are the width and height of the application at its selected LoD.
- λ is the interaction cost penalty weight.

Apps placed closer to the panel receive higher priority, apps placed farther away have their relevance penalized. `lambda_weight = 0.1` controls the penalty intensity, this results in high-relevance apps being placed close to the interaction area. In this version, LoD weighting is still included, so high-LoD apps are preferred when space allows.

User feedback

Experiment Design

To evaluate the effectiveness of each UI optimization version, I conducted a user study involving **4 participants**, each testing **4 different scenes**. Each scene was tested **3 times** to account for UI generation randomness, leading to a total of:

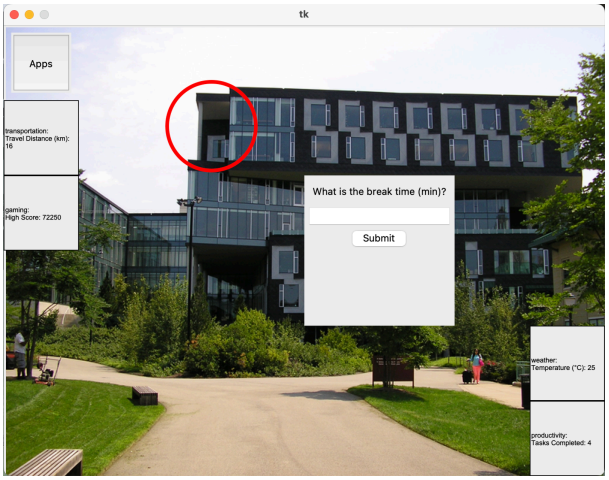
$$4\ (\text{users}) \times 4\ (\text{scenes}) \times 3\ (\text{iterations}) = 48\ \text{rounds/version}$$

During each round, participants completed 10 questions using the generated UI. The results were averaged for **Final Score**, **Average Time Per Question (including penalties)**, **Accuracy**, and **Visual Obstruction**.

Result Analysis

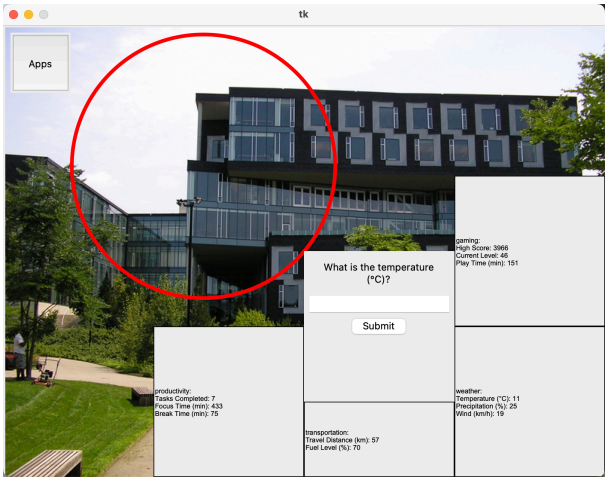
Version	**Final Score** (s)	**Accuracy** (%)	**Obstructions**
Version-1	**9.93**	**96.02**	**0**
Version-2	**8.21**	**100.00**	**0**

Version-3	**7.81**	**99.89**	**0**



For version-1, since this version maximized relevance without considering LoD or interaction cost, **highly relevant applications were sometimes placed far from the question panel, and most of the time the system didn't choose the biggest LoD** so it took users more time to interact with apps, increasing the **average response time**.

By incorporating **LoD preferences**, this version placed high-relevance apps at higher LoD, making them more **visually accessible**. **Users completed tasks faster**, as larger UI elements improved readability and reduced selection time. However we can notice that the layout of the app is spread out, which is not conducive to the user quickly switching the center of attention between the question panel and the app over and over again. So it comes to version3.



This version further optimized UI placement by **minimizing interaction cost**, ensuring **high-relevance apps were placed near the question panel**. **Users found relevant apps faster**, leading to the lowest **average response time**. But despite the many improvements in version 3, there is still room for optimization, such as **font size, and accuracy not reaching 100%**. So I need to optimize it further to get the final version in next chepter.

Summary

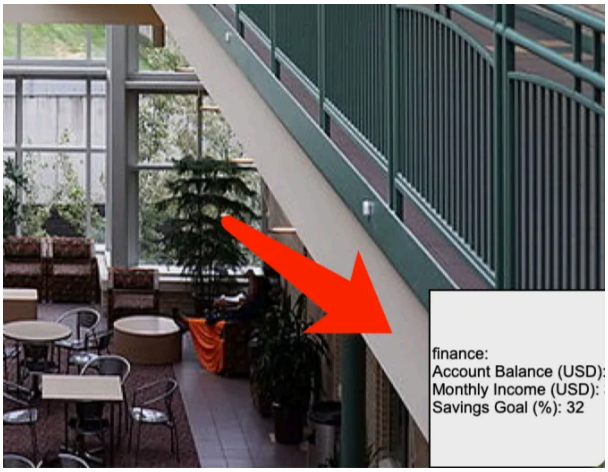
Version	Relevance Considered?	LoD Preference?	Interaction Cost Considered?	Key Focus
Version-1	✅	❌	❌	Places the most relevant apps.
Version-2	✅	✅	❌	Prefers higher-LoD apps when possible.
Version-3	✅	✅	✅	Balances relevance, LoD, and accessibility.

Version-1 had the longest response time, as highly relevant apps were sometimes placed inefficiently. Version-2 improved efficiency but did not fully account for interaction cost, leading to slightly longer completion times compared to Version-3. And Version-3 achieved a balance between relevance, visibility, and accessibility, demonstrating that incorporating interaction cost improves UI usability.

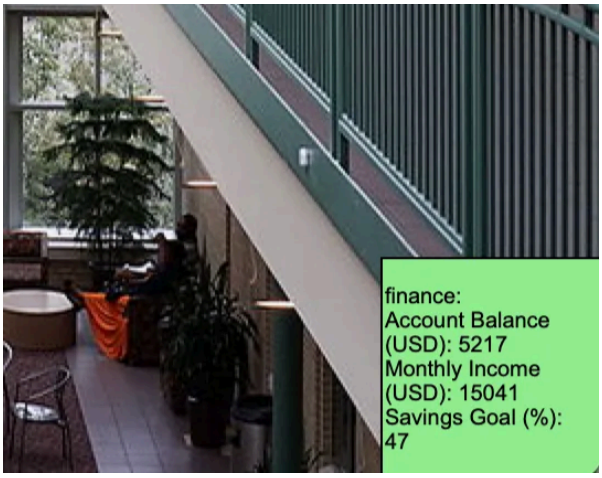
Final formulation

Optimal-1: Fix size regardless of Lod when at rightmost column (in `ui.py`)

For the problem that the correct rate is not 100%, if the app initialization is placed in the right/lower boundary and LoD=1, the size can be unchanged after clicking but will show more details, so I need to dynamically modify the text size to avoid the boundary case for some of the contents of the app, LoD=2 when the information overflowed the boundary of the interface, resulting in the inability to query the answer. (The correct rate in version2 is still 100% because this version prioritizes LoD, so there is no app with LoD=1 during initialization to avoid the bug of not being able to query complete information)



Before



After

Optimal-2: Dynamically define font and color based on LoD(in `ui.py`)

The main idea behind this optimization is to provide visual cues to the user by dynamically adjusting the **color** and **font size** of applications based on their **Level of Detail (LoD)**. This design choice enhances usability by:

1. **Encouraging Interaction:**
    - Applications with **higher LoD (more detailed content)** are displayed with **darker colors**, signaling **more interactive potential**.

- Applications with **lower LoD (minimal content)** are displayed with **lighter colors**, reducing unnecessary attention but still indicating availability.

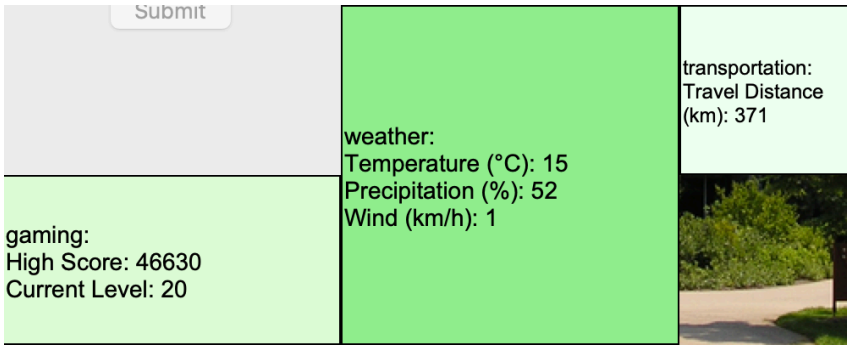
💡 **Font sizes:**

- LoD **0** (Minimal Content): **12px**
- LoD **1** (Moderate Content): **14px**
- LoD **2** (Detailed Content): **14px** *(Same as LoD 1 but benefits from more space)*

Background colors:

- LoD **0** → `#F0FFF0` *(Very light green, minimal emphasis)*
- LoD **1** → `#DFFFD6` *(Light green, moderate emphasis)*
- LoD **2** → `#90EE90` *(Stronger green, highly visible)*

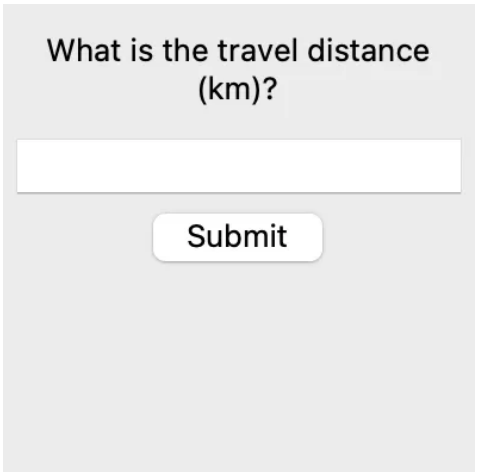
2. **Improving Readability:** As LoD increases, applications occupy more space on the UI, justifying a larger font size for improved readability. The font dynamically adjusts to the application's LoD, making it easier for users to engage with detailed applications.



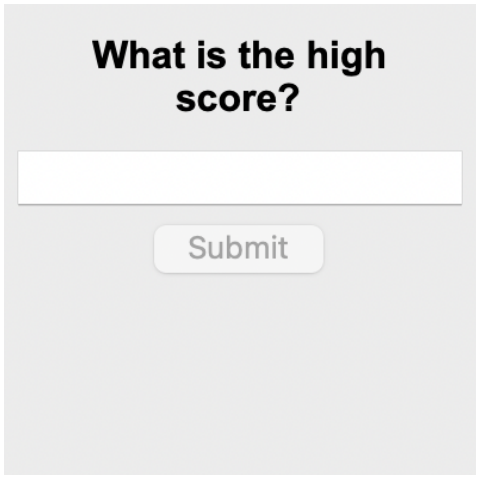
Optimal-3: Optimize question panel(in `ui.py`)

The goal of Optimal-3 is to enhance the **question panel** by:

1. **Improving readability**: Using a larger font size and bold formatting for question text to make it clearer and easier to read.
2. **Ensuring user interaction correctness**: Preventing accidental submissions by disabling the submit button until the user enters input. This helps reduce errors and ensures that each question is answered intentionally.



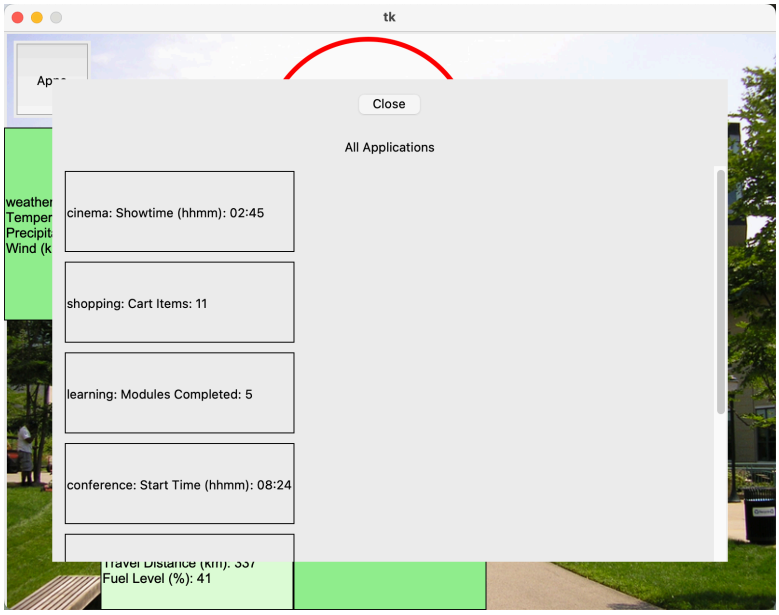
Before



After

Optimal-4: Sorts the remaining apps by relevance score(in `ui.py`)

This optimization mainly **sorts the remaining apps** by **relevance score** in **descending order**. This reduces the amount of time users spend scrolling down the query after clicking the AllApp button and reduces the penalty.

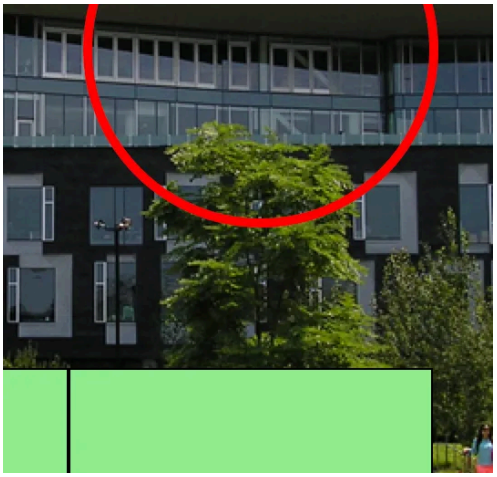


After optimal-4 the apps in list show up in **descending order**

Optimal-5: Define Constraints in Pixels for ROI Overlap(in `main.py`)

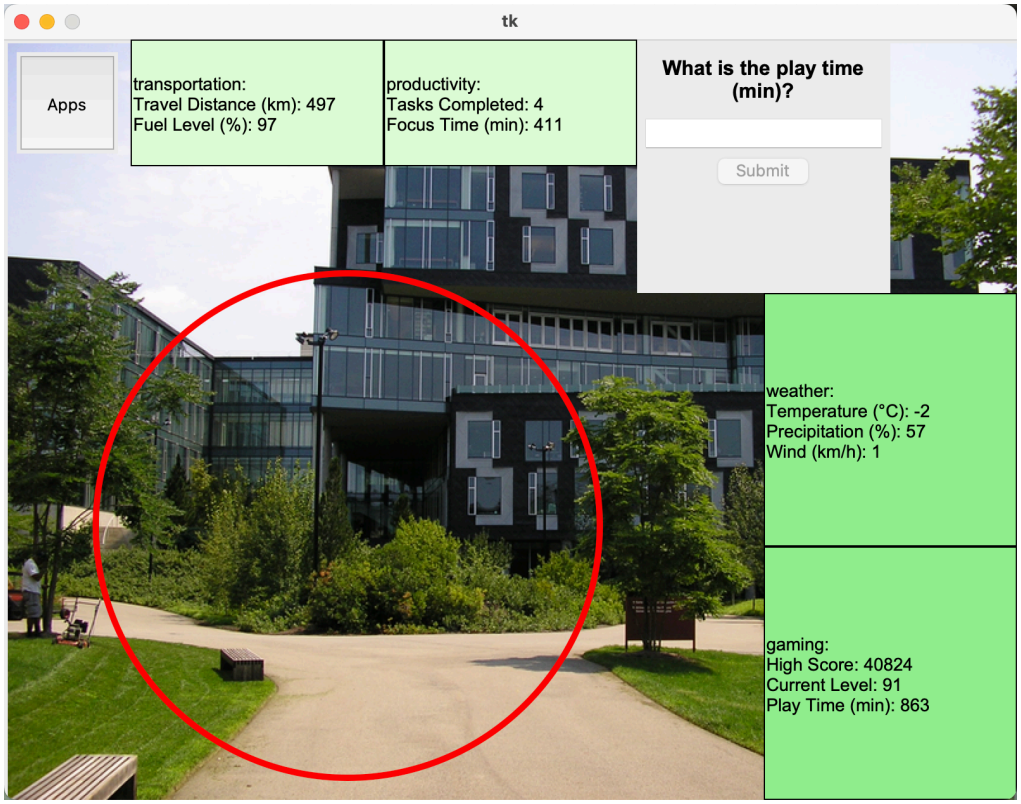
- **Origin Approach (Grid-Based):** Converts pixel coordinates to grid units (`block_size` divisions). Places constraints on the discrete grid positions. ROI and other elements are rounded to nearest grid cells → can cause inaccuracies.
- **New Approach (Pixel-Based):** Keep all constraints in pixels instead of using grid units. Use actual object dimensions (width/height in pixels) for overlap calculations. Define a continuous placement space, allowing for more precise UI alignment.

This will result in a better prevention and control strategy, putting as many apps as possible on the interface and presenting as much information as possible.



Before: quite far away because of Inaccurate grid-based calculations

After: much closer because of more accurate pixel-based calculations



Final Version UI

Bonus

Bonus-1: Automate Relevance Calculation(in `ui.py`)

The initial implementation of relevance values required manual assignment, which could lead to s**uboptimal UI layouts** that might not prioritize frequently used or critical applications, and **additional effort** in defining relevance for each scene manually.

Thus, I implemented automate relevance calculation by these two steps:

1. **Counts occurrences**:
   - A dictionary `question_counts` is created, initializing all apps' counts to zero.
   - For each question, it checks which app is involved and increments its count.
2. **Normalizes relevance values**:
   - The maximum occurrence count is found. Each app's relevance is scaled between 0 and 1, ensuring consistency.

By this method, the system can achieve:

1. Dynamically updates based on actual usage patterns rather than predefined values.
2. No need to manually set relevance scores for every new scene.
3. Apps that appear more frequently in scene questions are given

Bonus-2: Multi-Stage Generation(in `multiStage.py`)

The current implementation optimizes application selection, LoD, and placement within a single formulation. This contrasts with the approach by Lindlbauer et al., who adopted a multi-step method. Here, I attempt to implement the interface optimization process as a multi-step approach and compare the advantages and disadvantages of both methods.

The process is divided into two main stages:

1. **Stage 1:** Determine **visibility and level of detail** (app type and LoD).
2. **Stage 2:** Optimize **placement** based on the selected apps and LoDs.

This approach improves upon previous methods by mitigating early-stage decision biases commonly seen in multi-stage processes, allowing for more flexible and dynamic MR interface placement.

Specific Steps

Step 1: LoD Pre-Selection (With Soft Constraints)

Instead of strictly maximizing LoD=2, use a **soft constraint** to avoid selecting too many large LoD=2 apps. Maximize relevance while penalizing excessive LoD=2 selections by constraining app variety and lod selection based on available space by:

1. Convert Grid Dimensions to Pixels: Using pixels ensures better accuracy than grid units alone.
2. Calculate Available Space: Unlike static constraints, it considers actual available space in real-time.

   $$\text{available\_space\_px} = \text{total\_ui\_area\_px} - (\text{btn\_all\_area\_px} + \text{questions\_area\_px} + \text{roi\_area\_px})$$

   A scaling factor (`bies_for_better_placement`) accounts for additional constraints, adjusting available space dynamically.
3. **LoD Selection Constraint:** By factoring in occupied areas (e.g., the ROI), it avoids excessive LoD selection that might later lead to placement failures.

$$\sum_{\text{app,lod}} y_{\text{app,lod}} \times \text{area}(\text{lod}) \leq \text{available\_space}$$

This refined feasibility constraint helps ensure better application distribution in the subsequent placement optimization stage while balancing LoD selection against spatial constraints effectively.

Step 2: Initial Placement

Solve the placement problem with selected apps and LoDs. If placement fails due to space constraints, move to Step 3.

Step 3: Adaptive LoD Reduction

Find the highest LoD in current selection, filter apps that are at this max LoD and find the least relevant app among them, reduce its LoD by one level. Re-run placement optimization, repeat until placement succeeds.

Result Analysis

The multi-stage optimization approach ensures the system does not overcommit to high-LoD applications too soon. It also enhances flexibility in constrained environments by dynamically reducing LoD only when necessary, allowing for better handling of tight spaces. Moreover, it is highly adaptable to dynamic changes, enabling adjustments without requiring a complete re-optimization of the entire layout. However, these benefits come with trade-offs: the approach is slower due to multiple iterations, increasing computation time. It also introduces additional complexity, requiring extra logic to manage LoD adjustments and re-optimization effectively. Furthermore, if space constraints are too restrictive, frequent LoD downgrades may occur, potentially diminishing interface richness by pushing multiple applications to the lowest LoD level.

```
gaming: 1.0
productivity: 0.6666666666666666
transportation: 0.3333333333333333
shopping: 0.0
cinema: 0.3333333333333333
learning: 0.0
conference: 0.0
library: 0.0
weather: 1.0
time: 0.0
travel: 0.0
--------STAGE-1-------
...
selected_apps:  [('gaming', 2), ('productivity', 2), ('transportation', 2), ('weather', 2)]
--------STAGE-1 END-------
--------STAGE-2: Tryout- 0  --------
....
--------STAGE-2 Tryout- 0  END--------
Placement failed!
Reduced LoD of transportation to 1
--------STAGE-2: Tryout- 1  --------
....
--------STAGE-2 Tryout- 1  END--------
Placement failed!
Reduced LoD of productivity to 1
--------STAGE-2: Tryout- 2  --------
```

```
...
--------STAGE-2 Tryout- 2  END--------
Placement failed!
Reduced LoD of gaming to 1
--------STAGE-2: Tryout- 3  --------
...
--------STAGE-2 Tryout- 3  END--------
Placement failed!
Reduced LoD of weather to 1
--------STAGE-2: Tryout- 4  --------
...
--------STAGE-2 Tryout- 4  END--------
Success: All 4 apps placed.
```

Criteria	Single-Stage	Multi-Stage
Grid Size	Small, simple layouts	Large, complex layouts
Available Space	Sufficient for most LoD choices	Constrained, requiring adaptive LoD
Computation Time	Faster (solves in one step)	Slower (requires iterative adjustments)
Adaptability	Less adaptive to changes	More adaptive to constraints and dynamic layouts
Risk of Early-Stage Bias	High	Low (adaptive LoD reduction)
Best Use Cases	Static UI layouts, fast optimization needs	AR/MR environments, dynamic UI changes

Conclusion

Version	Final Score (s)	Accuracy (%)	Obstructions
Version-3	**7.81**	**99.89**	**0**
Final-Version	**7.23**	**100.00**	**0**
Multi-Stage	**7.19**	**100.00**	**0**

We can see that the result of **multi-stage was not much better than single-stage and even worse**, that's because **single-stage optimization is suitable for simpler UI layouts, static applications, and performance-critical tasks where a quick, optimal result is preferred**(which is exactly the type of my system). While **multi-stage optimization is better for dynamic, constrained environments where flexibility is required to balance LoD, placement, and relevance while handling space constraints iteratively**.

For highly interactive UIs (AR/MR), where placement flexibility is crucial, multi-stage optimization is generally the better choice because it avoids early-stage bias and adapts better to space constraints.