A CODE AVAILABILITY

The Gen-POMCP implementation is available here: https://anonymous.4open.science/r/GenPlan-FBCD/README.md

B PERFORMANCE BOUNDS FOR STRUCTURE-BASED PLANNING

In structured environments Gen-POMCP can explore the environment faster than Naive-POMCP (using fewer rollouts and in less time) by taking advantage of limited resources. However, it simplifies the planning problem by entirely exploring each fragment it enters before moving to the next. This heuristic can result in longer overall paths taken to search the environments. It is reasonable to ask by how much the global Naive-POMCP can actually improve on the path length taken by Gen-POMCP (and specifically the Structure-Based Planner).

Below we sketch a proof that considers the limit in which each planner fully optimizes its respective objective: Naive-POMCP follows the Bayes-optimal plan in each fragment and Gen-POMCP follows the Bayes-optimal global policy for the maze. We bound the cost difference according to the worst-case cost in steps.

Expected and worst-case The expected number of steps it takes for a policy to explore a maze is the average over the length of path this policy takes to reach uniformly sampled exit locations. The worst-case number of steps is the largest number of steps that the policy could take for some exit position. This is bounded below by the number of steps required to fully explore the maze.

Lemma 1. *There exists a fragment of size $n \times n$ which takes $O(n^2)$ steps to search in expectation, and to explore fully.*

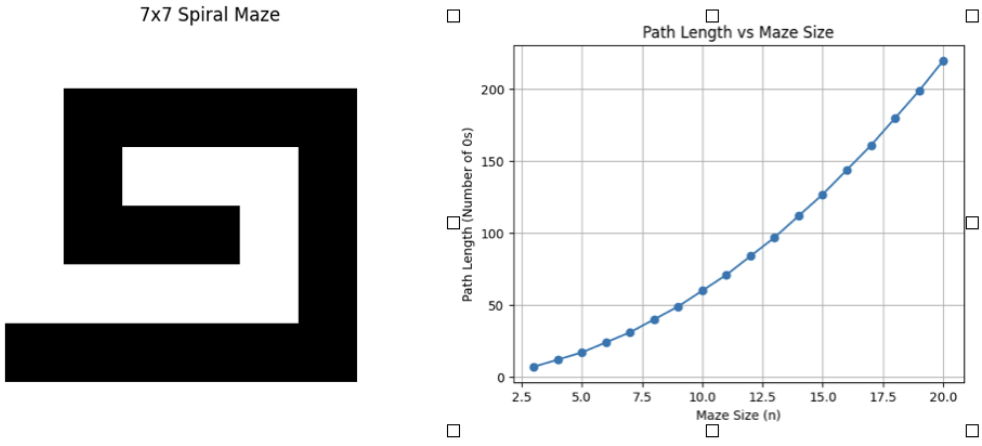


Figure 5: Consider a maze with a spiral wall - the white cells indicate traversable floor, and black indicate intraversable wall. Simulating these environments shows that the maximal length of path (white cells) in the environment grows as $\approx \frac{1}{2}n^2$

.

Proof. Consider a fragment with the maximum spiral path (e.g. Figure 5). The length of this path scales quadratically with n. In particular, following a spiral path takes a series of four legs at each depth, and the length of every other leg reduces by two (one for the wall and one for the path itself).

This yields

$$n + \sum_{i=0}^{\lfloor n/2 \rfloor - 1} 2(n - 2i - 1) = \frac{n}{2} 2n - 4\frac{(n/2)(n/2+1)}{2} + O(n) \\ = \frac{1}{2}n^2 + O(n) \tag{3}$$

□

Theorem 2. *In the an $n \times n$ maze, the expected number of steps taken by SBP may exceed the expected number of steps of an optimal policy by $\Omega(n^2)$.*

Proof. Build a fragment by adjoining an empty room and a spiral by a single door at a corner. Now connect the two fragments by adding a door between the empty rooms in the opposite corner. Assume the size of the empty rooms is such that the optimal algorithm can find the exit with probability $1/2$ by checking each empty room, but the SBP algorithm must explore entirely the first fragment that it enters. With probability $3/4$, the exit is not in the first empty room, so it must explore the spiral, which takes time $\Omega(n^2)$ to fully explore by Lemma 1. The spiral also must be exited, so around n^2 steps are spent when the exit is in the other empty room (in this case the optimal planner finds immediately by checking each room). Since the optimal planner takes only a constant number of steps to check each empty room, and then behaves identically to the SBP, the expected cost when the exit is in any other location is asymptotically the same, so the expected cost difference is roughly $\frac{1}{4}n^2 = \Omega(n^2)$. □

Theorem 3. *The number of steps to fully explore a maze is $O(n^2)$.*

Proof. Consider a v-vertex connected graph. The maximum width (roughly achievable by the spiral) is v, leading to a naive bound of $O(v^2) = O(n)$. This can be improved to $O(v)$ by running a depth-first search. Since there are 4 movement directions the degree of this graph is 4 meaning the maximum number of backtracks *to* a vertex is 3, which immediately gives $4v$. However, in a depth first search there is only one backtrack *from* each vertex is 1, which leads to an easy inductive proof that the bound is $O(2v - 1)$ regardless of degree, yielding $2n^2 - 1 = O(n^2)$. Note that further improvements should be possible by considering the number of walls required to induce the worst-case topology. □

This implies that the Bayes-optimal policy has $O(n^2)$ expected cost (since its *expected* cost must be at least as good as the *expected* cost of exhaustive search), regardless of the maze. Together, Lemma 1 and Theorem 3 demonstrate that the SBP heuristic does not damage the (asymptotic) expected cost in the worst maze.

Theorem 4. *Assume that an $n \times n$ maze is fragmented in such a way that any time a fragment is entered, it can be fully explored before exiting, into c^2 square $(n/c) \times (n/2)$ fragments. The asymptotic expected cost is $\Theta(n^2)$ in the worst such maze for the modular optimal and globally optimal policies.*

Proof. First, consider the global optimal policy. The additional requirements placed on the maze cannot make the $O(n^2)$ bound in Theorem 3 worse, and we can get a matching lower bound by simply adjoining multiple spiral examples as in Lemma 1 and adding doors between them.

Now consider the modular optimal policy. It is clear that the globally optimal policy has an expected cost as least as low as the modular optimal policy (even in their respective worst mazes), by definition, so the $\Omega(n^2)$ lower bound automatically carries over to the modular optimal policy. We assumed that the modular optimal policy takes the Bayes-optimal paths between fragments. This must be at least as good as the following strategy: mimic the global optimal policy, but any time a new fragment is entered, first explore it completely and return to the entrance. By Theorem 3, each such "extra" exploration detour takes at most $2(\frac{n}{c})^2 - 1$ steps, and the return takes at most $(\frac{n}{c})^2$ steps. The total is $3(\frac{n}{c})^2 - 1$. There are exactly c^2 such detours, for $4n^2 - c^2 = \Theta(n^2)$ extra steps. The global optimal policy also takes $\Theta(n^2)$ steps.

□

Therefore, in the worst case the modular algorithm is inferior by at least a constant factor of the total search time in expectation. Examining the proof of Theorem 4 yields a factor of 2.5 over our upper bound in Theorem 3, but presumably this can be improved substantially since a lot of exploration is being redone after the detours.

Improving expected cost upper bounds Substantial improvements to the worst-case cost bound in Theorem 3 are easy to obtain when the proof is applied to expected cost by e.g. noting that the depth-first search visits at least one new cell every two steps, meaning that there is clearly at least a $1/4$ chance of finding the exit after n^2 steps, or by noting that the true number of "vertices" is reduced by walls. These improvements seem to apply equally to the modular and global optimal policies, and probably do not affect our constants much.

For worst-case cost, the situation is similar. However, the worst-case cost analysis simplifies significantly with the additional assumption that transitions between fragments are negligible (say, if they all branch off from a central room). This observation is trivial but worth stating explicitly:

Theorem 5. *When the cost to transition between fragments is negligible, each has one entrance, and there is no line-of-sight across fragments, the modular algorithm has the same worst-case step count as the optimal algorithm.*

Proof. In the worst case, the optimal algorithm must explore each fragment, and since there is only one entrance to each fragment it is not possible to gain any advantage by exiting a fragment before it has been fully explored. □

C EXPERIMENTAL ENVIRONMENTS

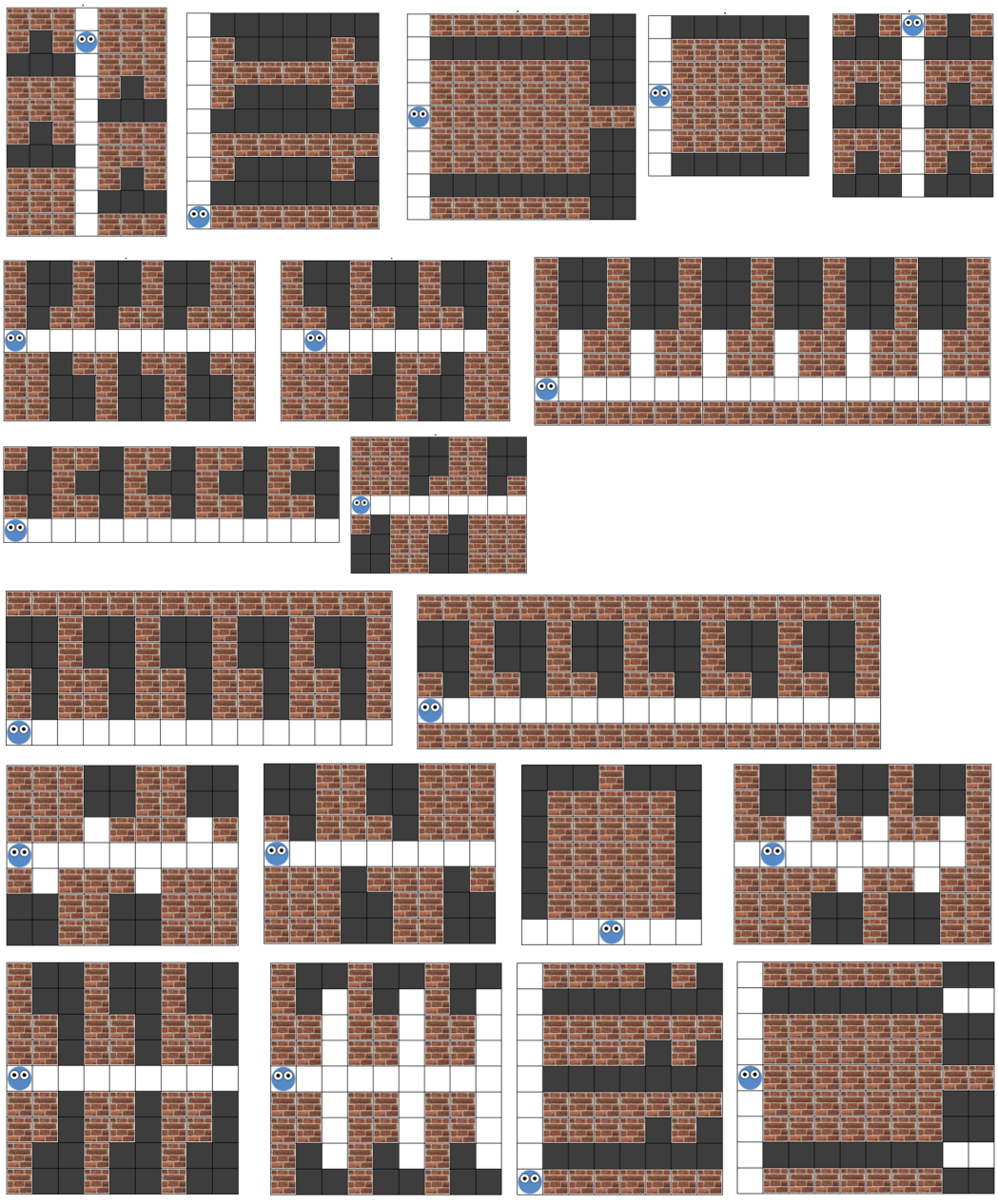


Figure 6: Environments used in Behavioral Experiment 1.

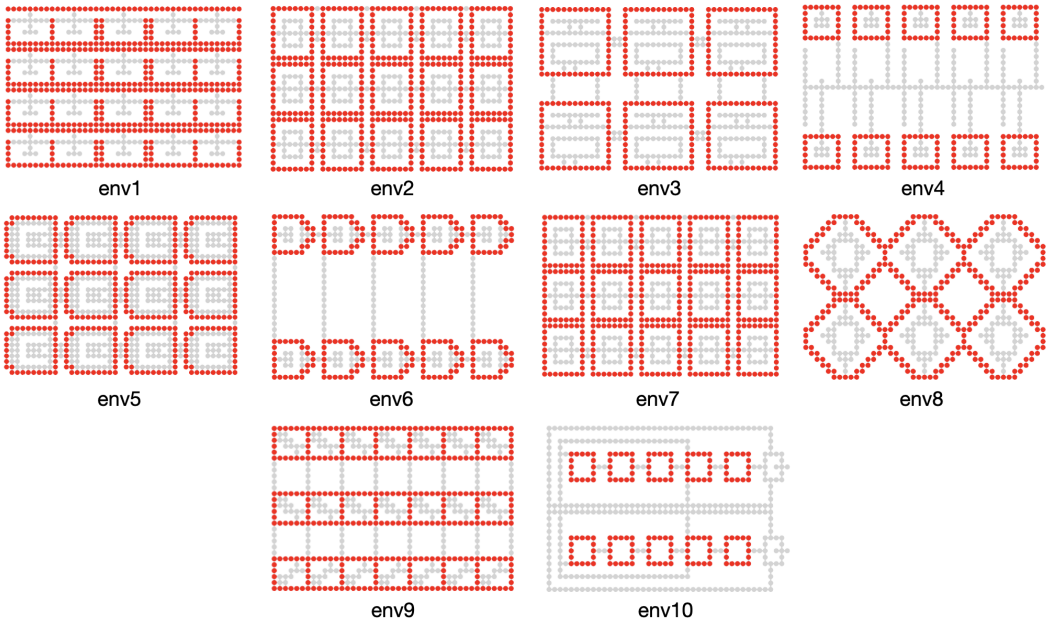


Figure 7: Environments used in Simulation Experiment 2.

D ADDITIONAL RESULTS - SIMULATION EXPERIMENT

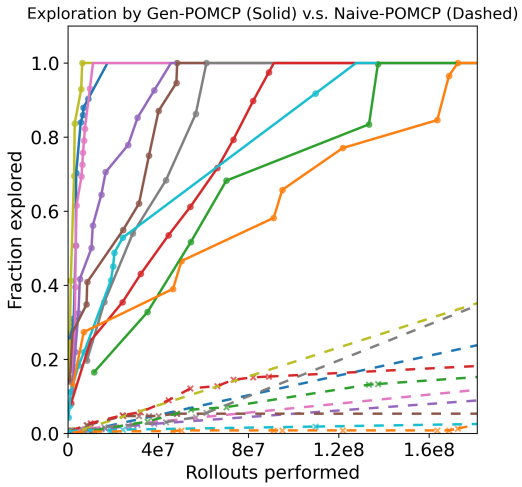


Figure 8: The fractions of each environment searched by Gen-POMCP and Naive-POMCP given identical computational budget. Gen-POMCP requires fewer rollouts and saves computing costs. Each environment is shown in a different color (see also Figure 4.)

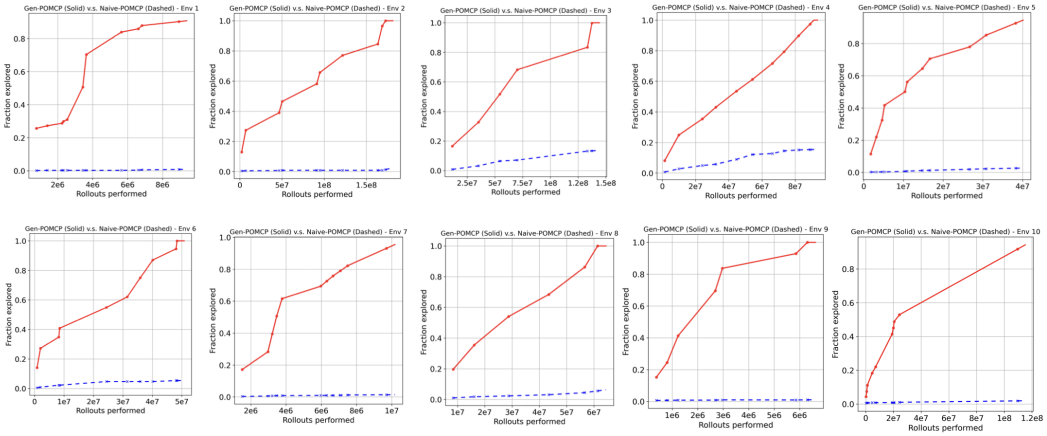


Figure 9: The fractions of each environment searched by Gen-POMCP and Naive-POMCP given identical computational budget. In each individual environment Gen-POMCP requires fewer rollouts and saves computing costs.)

E THE EFFECT OF FREE PARAMETERS ON RESULTS

Reconstruction accuracy. As map accuracy decreases, the amount of online heuristic planning increases, and the amount of structure-based planning decreases. We implement this heuristic based prior work with Maze Search (Kryven et al 2024). A zero reconstruction accuracy entails a fully heuristic planning, regardless of planning cost.

Planning cost. As planning cost increases the units become smaller, leading to more localized search. A negligible planning cost paired with a high reconstruction accuracy reduces the model to a global planner. A high accuracy and high planning cost leads to a fully structure-based planning (the population level model used in the paper)

Dissociating between these parameters in a human experiment requires a complex targeted design, beyond the scope of the current work. As our goal is to test whether people use structure-based

planning, as opposed to global search considered in previous work, we use a population-level model with high reconstruction accuracy and low planning costs. This leads the model to plan within single units intended by design (rather than grouping them) and maximizes the amount of discriminating decisions between structure-based and global planning.

F GENERALIZING TO ACROSS LLM ARCHITECTURES

In the paper, we used GPT-4 to build a proof-of-concept implementation for the GMM, originally chosen due to its strong code-generation abilities. However, we clarify that the choice of LLM model and prompting strategy are not critical to our framework's results. The primary contribution of our work is showing that human planning in structured environments relies on integrating two cognitive principles – (1) compressed cognitive maps that leverage redundant structure (implemented in GMM) and (2) policy reuse (implemented in SBP).

Below we show that GMM can be implemented with different LLM architectures. To do this, we show experimental results producing similar reconstructions by using different LLMs as a backend: GPT-4, Gemini-2.5-flash, Llama-3.3-70B, and Kimi-K2-Instruct-0905. Furthermore, we present results from two different prompting strategies (one-step prompt and multi-step prompts), showing that the exact prompt wording is not critical to producing the given results.

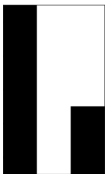
F.1 SINGLE PROMPT

Input map

Top scoring unit candidate

GPT-4, Gemini-2.5-flash, Llama-3.3-70B

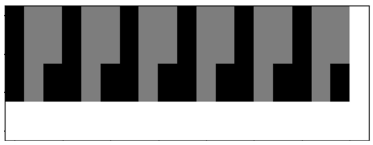
Kimi-K2-Instruct-0905



Reconstructed map

GPT-4, Gemini-2.5-flash, Llama-3.3-70B

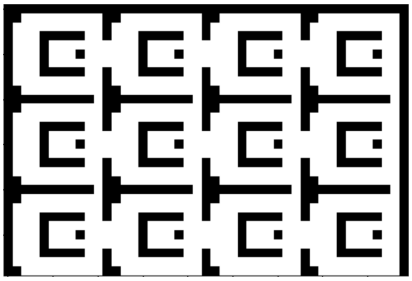
Kimi-K2-Instruct-0905



F.2 MULTI PROMPT

Input map

Complete input map



Partial input map

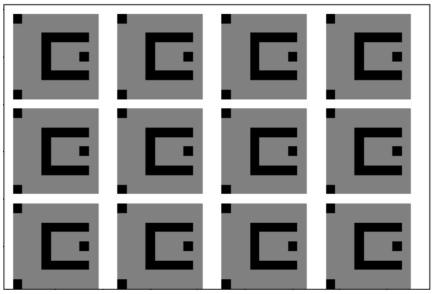
Top scoring unit candidate

GPT-4, Gemini-2.5-flash, Llama-3.3-70B

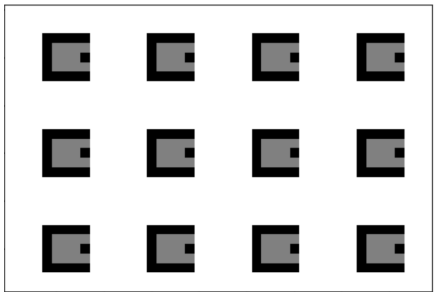
Kimi-K2-Instruct-0905

Reconstructed map

GPT-4, Gemini-2.5-flash, Llama-3.3-70B



Kimi-K2-Instruct-0905



G PROMPTS

G.1 ONE-STEP PROMPT FOR GMM

The Single Prompt GMM identifies the unit along with the reconstruction program using one prompt to the LLM. The prompt describes the task as a two-step procedure: first identify the repeating unit, then complete and return a runnable Python program that contains both the unit as a 2D array and the reconstruction function.

System prompt:

```
You are a designer's assistant, skilled in noticing patterns,
combining fragmets into a patterns, and extrapolating them.  You
are skilled in identifying the underlying structure of a pattern
and generating new fragments that fit the pattern.  You are also
skilled at writing Python code.
```

User prompt:

```
There are two steps to this task.  In Step 1, you will be given
an input (a map) and asked to identify its constituent units.
The input is a matrix, elements of which can take values 1 and
0.  Your task is to identify a repeating unit in this input.

To be considered a repeating unit, the unit does not have to tile
the space exactly, but it must appear at least twice.  The unit
instances may be flipped horizontally or vertically, translated
horizontally or vertically, and rotated by multiples of 90 degrees
(i.e.  0, 90, 180, 270).

IMPORTANT:
1.  Instances of the unit must NOT overlap in the original input.
2.  The height and width of the unit need not be equal

Example 1.
Given input:  {example input 1}
The repeating unit is:  {example unit 1}

Example 2.
Given input:  {example input 2}
The repeating unit is:  {example unit 2}

Example 3.
Given input:  {example input 3}
The repeating unit is:  {example unit 3}

In Step 2, you will write a function that attempts to identify all
occurrences of the unit in the input.  Return a list containing
the indexical locations of the top left corner for each copy,
along with whether to reflect the copy horizontally and the
number of 90 degree counter-clockwise rotations (these operations
together generate the dihedral group D4).

For instance, in the examples above, possible solutions include

Example 1.
Solution 1:  {example 1 program 1}
Solution 2.  {example 1 program 2}

Example 2.
Solution 1:  {example 2 program 1}
Solution 2:  {example 2 program 2}
```

```
Given your unit and partition, the user will attempt to
reconstruct the input using the following function:

def construct_copy(unit, reflect, rotations):
    if reflect:
      unit = np.flip(unit, 1)
   unit = np.rot90(unit, rotations)
   return unit

def regenerate_pattern(unit, copies, input_dims):
   pattern = -1 * np.ones(shape=input_dims)
   for copy in copies:
      transformed = construct_copy(
         unit,
         copy["reflect"],
         copy["rotations"],
      )
      height, width = transformed.shape
      tl_i, tl_j = copy["top left"]
      try:
         pattern[tl_i:tl_i+height, tl_j:tl_j+width] = transformed
      except:
         pass
   return pattern


We can test the success of this regeneration with

input_map = np.array(input_map)
output = regenerate_pattern(
   unit,
   partition(),
   input_map.shape,
)

Now is your turn.  Propose a unit that can be used to reconstruct
the given input.  Respond by completing the following Python code:

input_map = {input map}
# make sure to define all arrays as numpy arrays
import numpy as np
input_map = np.array(input_map)

unit = [ ...  ]
unit = np.array(unit)

def partition():
   copies = []
   # Place your code here.  Let's think step by step
   return copies

Please include only code in you response, no text.'''
```

G.2 MULTI-PROMPT GMM

To improve GMM scalability on large maps, we introduce a two-step approach. The two steps are implemented in two separate prompts, which adapt the strategy described in the previous section. The first prompt provides a part of the map and asks to identify a repeating unit. The second prompt

asks the LLM to infer a reconstruction program for the complete map given the previously identified unit.

System prompt:

```
You are a designer's assistant, skilled in noticing patterns,
combining fragments into a patterns, and extrapolating them.  You
are skilled in identifying the underlying structure of a pattern
and generating new fragments that fit the pattern.  You are also
skilled at writing Python code.
```

Unit Identification Prompt:

```
You will be given an input (a map) and asked to identify its
constituent units.  The input is a matrix, elements of which can
take values 1 and 0.  Your task is to identify a repeating unit in
this input, and ONLY output the unit as a 2D python array.  DO NOT
include anything else in the completion.

To be considered a repeating unit, the unit does not have to tile
the space exactly, but it must appear at least twice.  The unit
instances may be flipped horizontally or vertically, translated
horizontally or vertically, and rotated by multiples of 90 degrees
(i.e.  0, 90, 180, 270).

IMPORTANT:
1.  Instances of the unit must NOT overlap in the original input.
2.  The height and width of the unit need not be equal

Example 1.
Given input:  {example input 1}
The repeating unit is:  {example unit 1}

Example 2.
Given input:  {example input 2}
The repeating unit is:  {example unit 2}

Example 3.
Given input:  {example input 3}
The repeating unit is:  {example unit 3}

The input you are working with is the following map:  {input map}
```

Reconstruction Program Prompt:

```
You will write a function that attempts to identify all
non-overlapping occurrences of the unit in the input.  Return a
list containing the indexical locations of the top left corner
for each copy, along with whether to reflect the copy horizontally
and the number of 90 degree counter-clockwise rotations (these
operations together generate the dihedral group D4).


Example 1.
Given input:  {example input 1}
The unit is:  {example unit 1}
A solution is:  {example 1 program 1}
An alternative, more structured solution is:  {example 1 program
2}


Given input:  {example input 2}
The unit is:  {example unit 2}
A solution is:  {example 2 program 1}
An alternative, more structured solution is:  {example 2 program
2}


Given your unit and partition, the user will attempt to
reconstruct the input using the following function:

def construct_copy(unit, reflect, rotations):
    if reflect:
      unit = np.flip(unit, 1)
   unit = np.rot90(unit, rotations)                                    )
   return unit

def regenerate_pattern(unit, copies, input_dims):
   pattern = -1 * np.ones(shape=input_dims)
   for copy in copies:
      transformed = construct_copy(
         unit,
         copy["reflect"],
         copy["rotations"],
      )
      height, width = transformed.shape
      tl_i, tl_j = copy["top left"]
      try:
         pattern[tl_i:tl_i+height, tl_j:tl_j+width] = transformed
      except:
         pass
   return pattern


We can test the success of this regeneration with

input_map = np.array(input_map)
output = regenerate_pattern(
   unit,
   partition(),
   input_map.shape,
)
```

```
Now is your turn.  Respond by completing the following Python
code,
1.  include everything that is between START OF CODE and END OF
CODE
2.  include the entire input map provided, do not use ...  to omit
3.  ONLY fill partition(), do not use variables/functions that are
not defined
4.  In the returned copies, follow the exact key names in the
examples:  'top left', 'reflect', 'rotations'.
4.  DO NOT include anything else in the completion

# START OF CODE, make sure to define all arrays as numpy arrays
import numpy as np

input_map = {input map}
unit = {input unit}
input_map = np.array(input_map)
unit = np.array(unit)

def partition():
   copies = []
   # Place your code here.  Let's think step by step
   return copies
result = partition()

# END OF CODE
```

G.3 IN-PROMPT EXAMPLES

Example 1

Input map

```
[
  [1, 0, 1, 0],
  [0, 1, 0, 1],
  [0, 0, 0, 0],
  [1, 0, 1, 0],
  [0, 1, 0, 1]
]
```

Unit

```
[
  [1, 0],
  [0, 1]
]
```

Reconstruction program 1:

```
def partition():
  return [
    {"top left": (0,0), "reflect": False, "rotations": 0},
    {"top left": (0,2), "reflect": False, "rotations": 0},
    {"top left": (3,0), "reflect": False, "rotations": 0},
    {"top left": (3,2), "reflect": False, "rotations": 0},
  ]
```

Reconstruction program 2:

```
def partition():
   copies = []
   for tl_i, tl_j in [(0,0),(0,2),(3,0),(3,2)]:
      copies.append(
         {"top left": (tl_i, tl_j), "reflect": False, "rotations": 0},
      )
   return copies
```

Example 2

Input map

```
[
  [1,1,1,1,1,1],
  [0,0,1,0,0,1],
  [0,0,1,0,0,1],
  [1,0,1,1,0,1],
  [0,0,0,0,0,0],
  [1,1,1,1,1,1],
  [1,0,1,1,0,1],
  [0,0,1,0,0,1],
  [0,0,1,0,0,1]
]
```

Unit

```
[
  [1,1,1],
  [0,0,1],
  [0,0,1],
  [1,0,1]
]
```

Reconstruction program 1:

```
def partition():
   return [
      {"top left": (0,0), "reflect": False, "rotations": 0},
      {"top left": (0,3), "reflect": False, "rotations": 0},
      {"top left": (5,0), "reflect": True, "rotations": 2},
      {"top left": (5,3), "reflect": True, "rotations": 2},
   ]
```

Reconstruction program 2:

```
def partition():
   copies = []
   for tl_i, tl_j in [(0,0),(0,3)]:
      copies.append(
         {"top left": (tl_i, tl_j), "reflect": False, "rotation": 0},
      )
   for tl_i, tl_j in [(5,0),(5,3)]:
      copies.append(
         {"top left": (tl_i, tl_j), "reflect": True, "rotations": 2},
      )
   return corners
```

Example 3

Input map

```
[
  [1,1,1,1,1,1],
  [1,0,1,0,1,0],
  [1,0,1,0,1,0],
  [0,0,0,0,0,0],
  [0,1,0,1,0,1],
  [0,1,0,1,0,1],
  [1,1,1,1,1,1]
]
```

Unit

```
[
  [1,1],
  [1,0],
  [1,0]
]
```

H PSEUDOCODE

Algorithm 1 Single-prompt Generative Map Module

Require: I : Input map, t : Threshold, C : Number of completions, S : Likelihood function
Ensure: λ : Generative program, u : Unit

$S' \leftarrow 0$
$\lambda \leftarrow$ ""
$u \leftarrow$ ""
while $S' < t$ **do**
  Generate a prompt from I
  Send the prompt and receive C completions $(\lambda_1, u_1), \ldots, (\lambda_C, u_C)$
  Extract Python programs $\{\lambda_1, \lambda_2, \ldots, \lambda_C\}$
  for all $\lambda_i \in \{\lambda_1, \ldots, \lambda_C\}$ **do**
    if λ_i runs successfully **then**
      $S_i \leftarrow S(\lambda_i)$
    end if
  end for
  $(S', i) \leftarrow \max_i S_i$ ▷ highest scoring program based on likelihood
  $\lambda \leftarrow \lambda_i$
  $u \leftarrow u_i$
end while
return λ, u

Algorithm 2 Multi-prompt Generative Map Module

Require: I_c : Input map, I_p : Partial map, t : Threshold, C : Number of completions, S : Likelihood function
Ensure: λ : Generative program, u : unit

$S' \leftarrow 0$
$\lambda \leftarrow$ ""
$u \leftarrow$ ""
while $S' < t$ **do**
  Generate a prompt from I_p
  Send the prompt and receive C unit candidates $u_1, \ldots, u_C$
  for all $u_i \in \{u_1, \ldots, u_C\}$ **do**
    Generate a prompt from I_c and u_i
    Send the prompt and receive program λ_i
    if λ_i runs successfully **then**
      $S_i \leftarrow S(\lambda_i)$
    end if
  end for
  $(S', i) \leftarrow \max_i S_i$ ▷ highest scoring program based on likelihood
  $\lambda \leftarrow \lambda_i$
  $u \leftarrow u_i$
end while
return λ, u

Algorithm 3 Structure-Based Planner

Require: I : Input map, u : Unit, C_i : Unit copies
Ensure: P : Agent path

$(r, c) \leftarrow$ Initial agent position
$C \leftarrow$ Empty set to track fully explored unit copies
$\pi \leftarrow$ Policy for unit exploration based on location of entrance
while C does not contain all copies **do**
    Run POMCP on I until reaching an unexplored unit C_i

    Identify the current entrance e into C_i
    if policy from e found in π **then**
        Explore C_i with policy from π
        Add C_i to C
    else
        Run POMCP on C_i explore the unit
        Add new policy (e, π_e) to π
        Add C_i to C
    end if

    $o_1, \ldots, o_m \leftarrow$ Exit locations of C_i
    for o_j in $\{o_1, \ldots, o_m\}$ **do**
        $p_j \leftarrow$ compute penalty for escaping C_i from o_j
    end for
    Run MCTS on C_i until reaching an exit to escape
end while

Run POMCP on I to explore the rest of the map
return P

Algorithm 4 POMCP for In-Unit Planning

procedure SEARCH(h)
    if $B(h) = \emptyset$ **then**
        return
    else
        repeat
            $s_{\text{exit}} \sim B(h)$
            Simulate($s_{\text{exit}}, h, 0$)
        until Timeout
        return $\arg\max_a V(ha)$
    end if
end procedure

procedure ROLLOUT(s_{exit}, h, depth)
    if depth $>$ depth limit **then**
        return 0
    end if
    $a \sim \pi_{\text{random}}$
    $(o, r) \sim \mathcal{G}(h, a)$
    if o contains s_{exit} **then**
        return r
    else
        return r+Rollout(s_{exit}, ha, depth$+1$)
    end if
end procedure

procedure SIMULATE(s_{exit}, h, depth)
    $N(h) \leftarrow N(h) + 1$
    if depth $>$ depth limit **then**
        return 0
    end if
    if $h \notin T$ **then**
        for $a \in \{\text{up}, \text{right}, \text{bottom}, \text{left}\}$ **do**
            $T(ha) \leftarrow (N_{\text{init}}(ha), V_{\text{init}}(ha), \emptyset)$
        end for
        return Rollout(s_{exit}, h, depth)
    end if
    $a \leftarrow \arg\max_a \left[V(ha) + c\sqrt{\frac{\log N(h)}{N(ha)}} \right]$
    $(o, r) \sim \mathcal{G}(h, a)$
    if o contains s_{exit} **then**
        $N(ha) \leftarrow N(ha) + 1$
    else
        $r \leftarrow r +$ Simulate(s_{exit}, ha, depth$+1$)
    end if
    $V(ha) \leftarrow V(ha) + \frac{r - V(ha)}{N(ha)}$
    return r
end procedure

I HUMAN EXPERIMENT - MAZE SEARCH TASK

This study runs best on a desktop/laptop.

The study will **NOT** run on Safari, or a mobile device.

In this study you will look for an exit in a maze.

After this, you will be asked to provide demographic information.

The study is expected to take about 10 minutes.

Thanks for participating!

Figure 10: Introductory screen.

INSTRUCTIONS (PLEASE READ CAREFULLY)

Your task is to exit the maze by reaching the red square, which is initially hidden.

You can move one square at a time by clicking on the white squares next to your character.

You cannot see through the walls. The squares you cannot see yet are black.

The exit could be behind any of the black squares.

A maze looks like this:

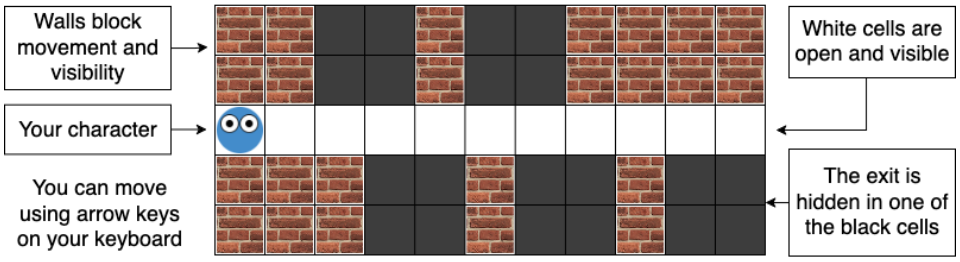


Figure 11: Instructions.

Practice Maze 1 of 5

Let's look at this map. There are some black squares, a brick wall, and your character.

The exit could be behind any of the black cells.

You can move your character by clicking one of adjacent white cells.

Find the exit and step on it to exit the maze.

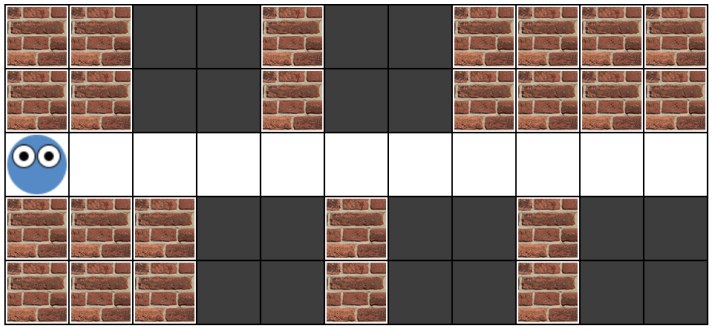


Figure 12: Practice (there are 5 practice trials).

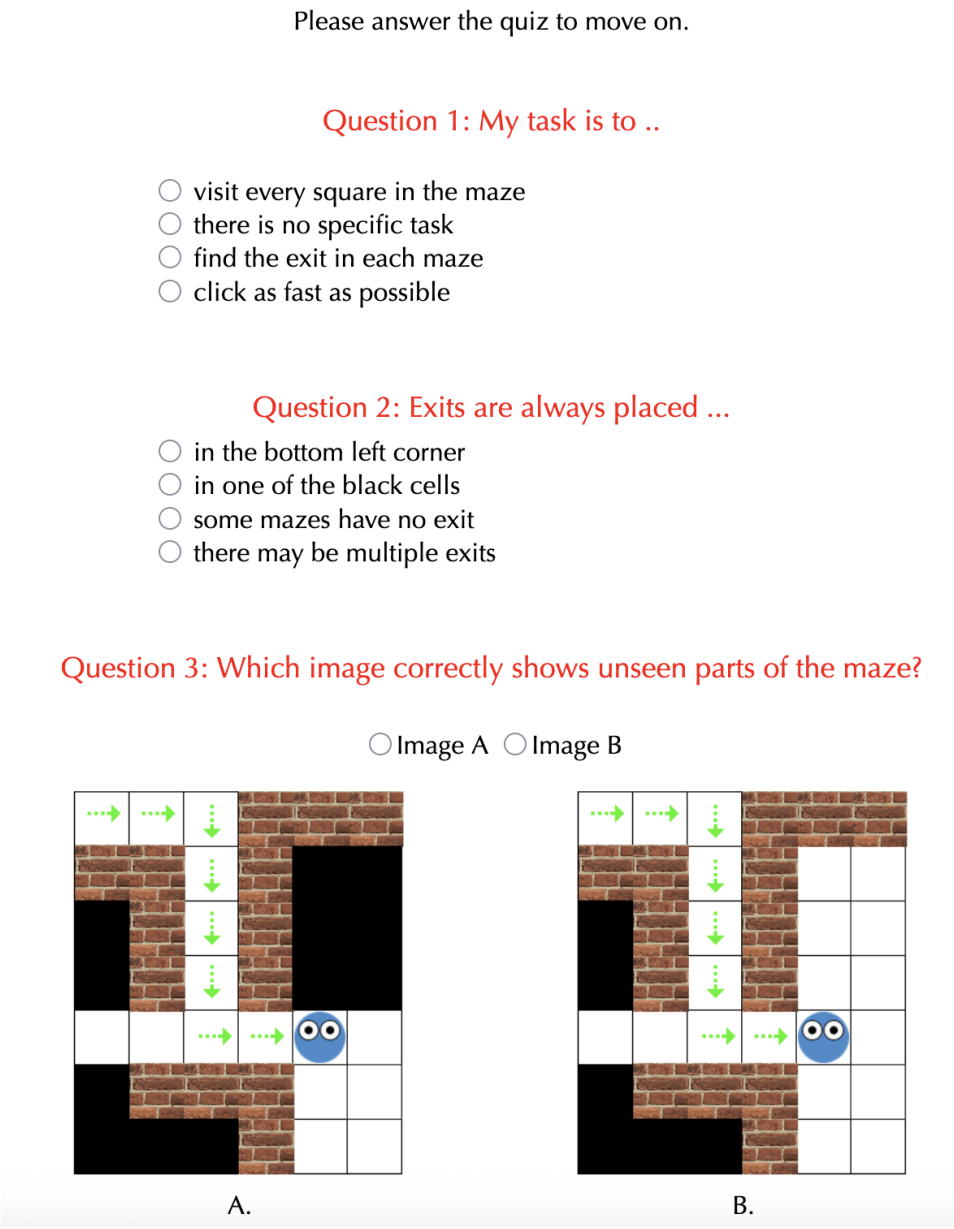


Figure 13: Comprehension Quiz.

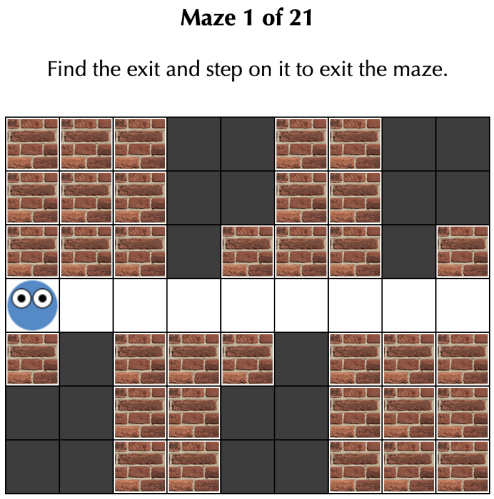


Figure 14: Experiment view.