

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.*

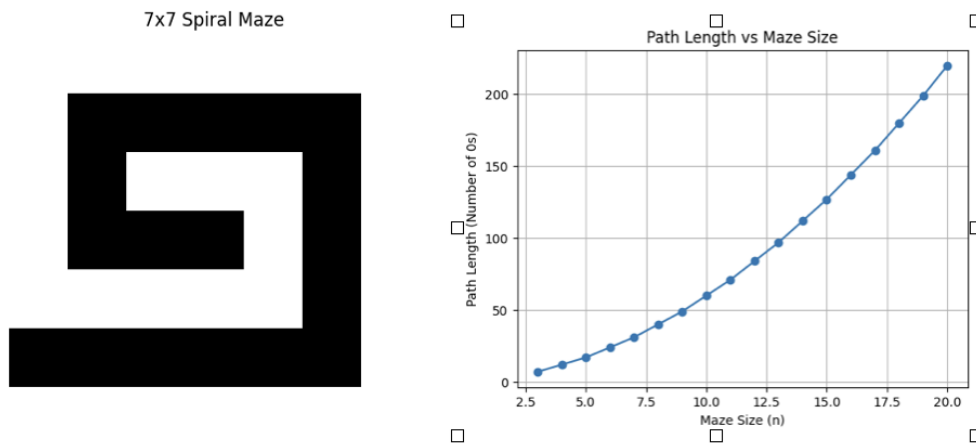


Figure 5: Consider a maze with a spiral wall - the white cells indicate traversable floor, and black indicate intraversable wall. Simulating these environments shows that the maximal length of path (white cells) in the environment grows as $\approx \frac{1}{2}n^2$.

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).

864 This yields

$$\begin{aligned}
 865 \quad n + \sum_{i=0}^{\lfloor n/2 \rfloor - 1} 2(n - 2i - 1) &= \frac{n}{2} 2n - 4 \frac{(n/2)(n/2 + 1)}{2} + O(n) \\
 866 \quad & \\
 867 \quad & \\
 868 \quad & \\
 869 \quad & \\
 870 \quad & \\
 871 \quad & \\
 872 \quad & \\
 873 \quad & \\
 874 \quad & \\
 875 \quad & \\
 876 \quad & \\
 877 \quad & \\
 878 \quad & \\
 879 \quad & \\
 880 \quad & \\
 881 \quad & \\
 882 \quad & \\
 883 \quad & \\
 884 \quad & \\
 885 \quad & \\
 886 \quad & \\
 887 \quad & \\
 888 \quad & \\
 889 \quad & \\
 890 \quad & \\
 891 \quad & \\
 892 \quad & \\
 893 \quad & \\
 894 \quad & \\
 895 \quad & \\
 896 \quad & \\
 897 \quad & \\
 898 \quad & \\
 899 \quad & \\
 900 \quad & \\
 901 \quad & \\
 902 \quad & \\
 903 \quad & \\
 904 \quad & \\
 905 \quad & \\
 906 \quad & \\
 907 \quad & \\
 908 \quad & \\
 909 \quad & \\
 910 \quad & \\
 911 \quad & \\
 912 \quad & \\
 913 \quad & \\
 914 \quad & \\
 915 \quad & \\
 916 \quad & \\
 917 \quad & \\
 \end{aligned} \tag{3}$$

□

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.

□

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

922
923 **Improving expected cost upper bounds** Substantial improvements to the worst-case cost bound
924 in Theorem 3 are easy to obtain when the proof is applied to expected cost by e.g. noting that the
925 depth-first search visits at least one new cell every two steps, meaning that there is clearly at least
926 a 1/4 chance of finding the exit after n^2 steps, or by noting that the true number of “vertices” is
927 reduced by walls. These improvements seem to apply equally to the modular and global optimal
928 policies, and probably do not affect our constants much.

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

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

935
936 *Proof.* In the worst case, the optimal algorithm must explore each fragment, and since there is only
937 one entrance to each fragment it is not possible to gain any advantage by exiting a fragment before
938 it has been fully explored. □

939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971

972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1010
1011
1012
1013
1014
1015
1016
1017
1018
1019
1020
1021
1022
1023
1024
1025

C EXPERIMENTAL ENVIRONMENTS

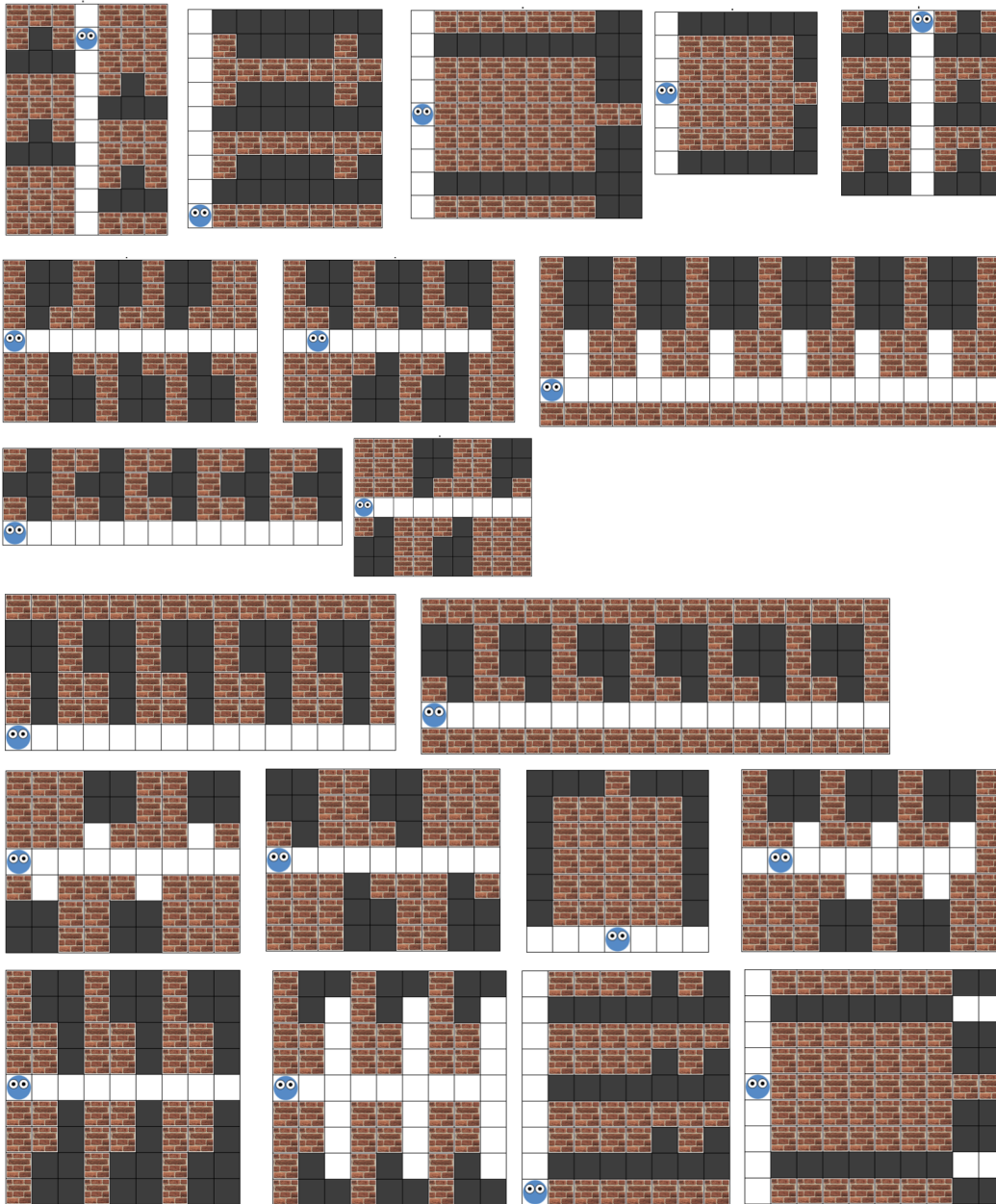


Figure 6: Environments used in Behavioral Experiment 1.

1026
1027
1028
1029
1030
1031
1032
1033
1034
1035
1036
1037
1038
1039
1040
1041
1042
1043
1044
1045
1046
1047
1048
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1059
1060
1061
1062
1063
1064
1065
1066
1067
1068
1069
1070
1071
1072
1073
1074
1075
1076
1077
1078
1079

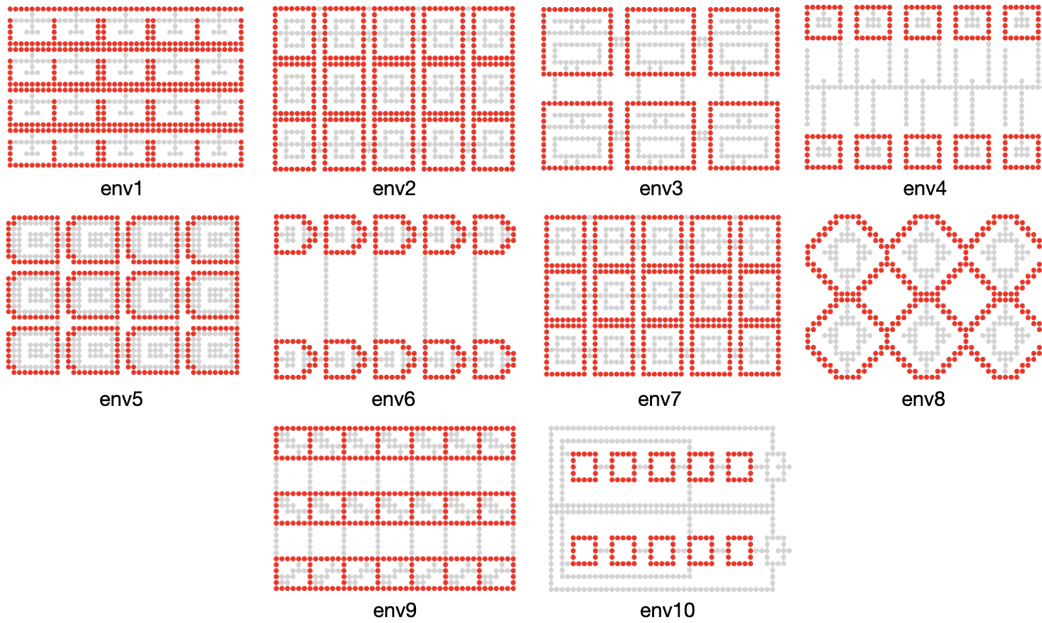


Figure 7: Environments used in Simulation Experiment 2.

D ADDITIONAL RESULTS - SIMULATION EXPERIMENT

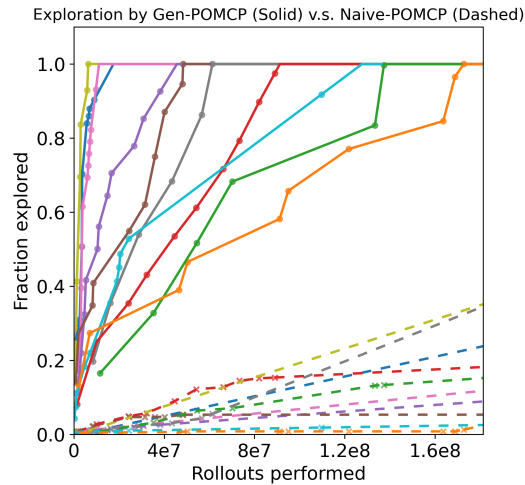


Figure 8: The fractions of each environment searched by Gen-POMCP and Naive-POMCP given identical computational budget. Gen-POMCP requires fewer rollouts and saves computing costs. Each environment is shown in a different color (see also Figure 4.)

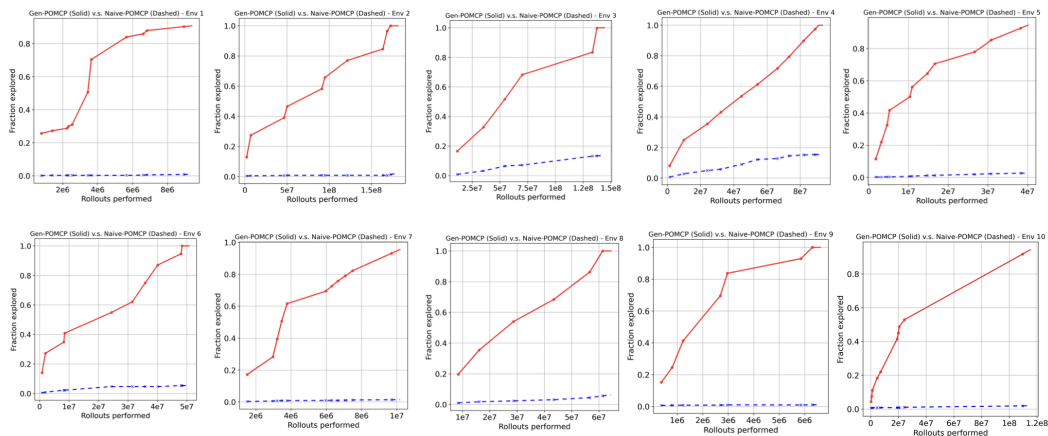


Figure 9: The fractions of each environment searched by Gen-POMCP and Naive-POMCP given identical computational budget. In each individual environment Gen-POMCP requires fewer rollouts and saves computing costs.)

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

1134 planning, as opposed to global search considered in previous work, we use a population-level model
 1135 with high reconstruction accuracy and low planning costs. This leads the model to plan within single
 1136 units intended by design (rather than grouping them) and maximizes the amount of discriminating
 1137 decisions between structure-based and global planning.
 1138

1139 **F GENERALIZING TO ACROSS LLM ARCHITECTURES**
 1140

1141 In the paper, we used GPT-4 to build a proof-of-concept implementation for the GMM, originally
 1142 chosen due to its strong code-generation abilities. However, we clarify that the choice of LLM model
 1143 and prompting strategy are not critical to our framework’s results. The primary contribution of our
 1144 work is showing that human planning in structured environments relies on integrating two cogni-
 1145 tive principles – (1) compressed cognitive maps that leverage redundant structure (implemented in
 1146 GMM) and (2) policy reuse (implemented in SBP).

1147 Below we show that GMM can be implemented with different LLM architectures. To do this, we
 1148 show experimental results producing similar reconstructions by using different LLMs as a backend:
 1149 GPT-4, Gemini-2.5-flash, Llama-3.3-70B, and Kimi-K2-Instruct-0905. Furthermore, we present
 1150 results from two different prompting strategies (one-step prompt and multi-step prompts), showing
 1151 that the exact prompt wording is not critical to producing the given results.
 1152

1153 **F.1 SINGLE PROMPT**
 1154

1155 **Input map**



1163
1164
1165 **Top scoring unit candidate**

1166 GPT-4, Gemini-2.5-flash, Llama-3.3-70B Kimi-K2-Instruct-0905



1176
1177 **Reconstructed map**

1178 GPT-4, Gemini-2.5-flash, Llama-3.3-70B Kimi-K2-Instruct-0905

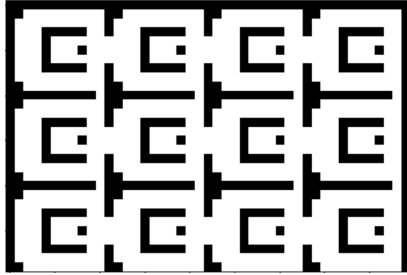


1185
1186
1187

1188 F.2 MULTI PROMPT
1189

1190 **Input map**

1191 Complete input map
1192

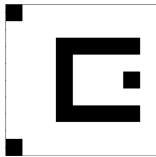


1193 Partial input map



1203 **Top scoring unit candidate**

1204 GPT-4, Gemini-2.5-flash, Llama-3.3-70B

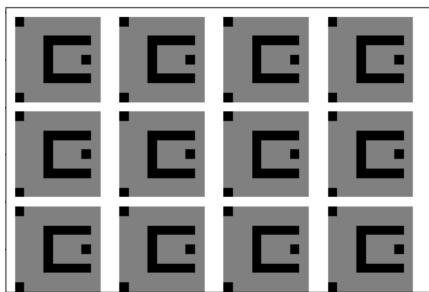


1205 Kimi-K2-Instruct-0905

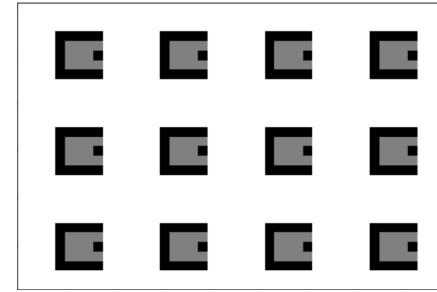


1213 **Reconstructed map**

1214 GPT-4, Gemini-2.5-flash, Llama-3.3-70B



1215 Kimi-K2-Instruct-0905



1226 G PROMPTS

1227 G.1 ONE-STEP PROMPT FOR GMM
1228

1229 The Single Prompt GMM identifies the unit along with the reconstruction program using one prompt
1230 to the LLM. The prompt describes the task as a two-step procedure: first identify the repeating unit,
1231 then complete and return a runnable Python program that contains both the unit as a 2D array and
1232 the reconstruction function.
1233
1234

1235 System prompt:

1236 You are a designer's assistant, skilled in noticing patterns,
1237 combining fragments into a patterns, and extrapolating them. You
1238 are skilled in identifying the underlying structure of a pattern
1239 and generating new fragments that fit the pattern. You are also
1240 skilled at writing Python code.
1241

User prompt:

1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295

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}

```

1296 Given your unit and partition, the user will attempt to
1297 reconstruct the input using the following function:
1298
1299 def construct_copy(unit, reflect, rotations):
1300     if reflect:
1301         unit = np.flip(unit, 1)
1302         unit = np.rot90(unit, rotations)
1303     return unit
1304
1305 def regenerate_pattern(unit, copies, input_dims):
1306     pattern = -1 * np.ones(shape=input_dims)
1307     for copy in copies:
1308         transformed = construct_copy(
1309             unit,
1310             copy["reflect"],
1311             copy["rotations"],
1312         )
1313         height, width = transformed.shape
1314         tli, tlj = copy["top left"]
1315         try:
1316             pattern[tli:tli+height, tlj:tlj+width] = transformed
1317         except:
1318             pass
1319     return pattern
1320
1321 We can test the success of this regeneration with
1322
1323 input_map = np.array(input_map)
1324 output = regenerate_pattern(
1325     unit,
1326     partition(),
1327     input_map.shape,
1328 )
1329
1330 Now is your turn. Propose a unit that can be used to reconstruct
1331 the given input. Respond by completing the following Python code:
1332
1333 input_map = {input map}
1334 # make sure to define all arrays as numpy arrays
1335 import numpy as np
1336 input_map = np.array(input_map)
1337
1338 unit = [ ... ]
1339 unit = np.array(unit)
1340
1341 def partition():
1342     copies = []
1343     # Place your code here. Let's think step by step
1344     return copies
1345
1346 Please include only code in you response, no text.'''

```

1346 G.2 MULTI-PROMPT GMM

1347
1348 To improve GMM scalability on large maps, we introduce a two-step approach. The two steps are
1349 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

1350 asks the LLM to infer a reconstruction program for the complete map given the previously identified
1351 unit.

1352
1353 **System prompt:**

1354 You are a designer's assistant, skilled in noticing patterns,
1355 combining fragments into a patterns, and extrapolating them. You
1356 are skilled in identifying the underlying structure of a pattern
1357 and generating new fragments that fit the pattern. You are also
1358 skilled at writing Python code.

1359
1360 **Unit Identification Prompt:**

1361 You will be given an input (a map) and asked to identify its
1362 constituent units. The input is a matrix, elements of which can
1363 take values 1 and 0. Your task is to identify a repeating unit in
1364 this input, and ONLY output the unit as a 2D python array. DO NOT
1365 include anything else in the completion.

1366 To be considered a repeating unit, the unit does not have to tile
1367 the space exactly, but it must appear at least twice. The unit
1368 instances may be flipped horizontally or vertically, translated
1369 horizontally or vertically, and rotated by multiples of 90 degrees
1370 (i.e. 0, 90, 180, 270).

1371
1372 **IMPORTANT:**

- 1373 1. Instances of the unit must NOT overlap in the original input.
1374 2. The height and width of the unit need not be equal

1375
1376 **Example 1.**

1377 Given input: {example input 1}
1378 The repeating unit is: {example unit 1}

1379
1380 **Example 2.**

1381 Given input: {example input 2}
1382 The repeating unit is: {example unit 2}

1383
1384 **Example 3.**

1385 Given input: {example input 3}
1386 The repeating unit is: {example unit 3}

1387 The input you are working with is the following map: {input map}

1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403

1404 Reconstruction Program Prompt:
1405

1406 You will write a function that attempts to identify all
 1407 non-overlapping occurrences of the unit in the input. Return a
 1408 list containing the indexical locations of the top left corner
 1409 for each copy, along with whether to reflect the copy horizontally
 1410 and the number of 90 degree counter-clockwise rotations (these
 1411 operations together generate the dihedral group D_4).

1412 Example 1.

1413 Given input: {example input 1}
 1414 The unit is: {example unit 1}
 1415 A solution is: {example 1 program 1}
 1416 An alternative, more structured solution is: {example 1 program
 1417 2}
 1418

1419 Given input: {example input 2}
 1420 The unit is: {example unit 2}
 1421 A solution is: {example 2 program 1}
 1422 An alternative, more structured solution is: {example 2 program
 1423 2}
 1424

1425 Given your unit and partition, the user will attempt to
 1426 reconstruct the input using the following function:
 1427

```

1428 def construct_copy(unit, reflect, rotations):
1429     if reflect:
1430         unit = np.flip(unit, 1)
1431         unit = np.rot90(unit, rotations)
1432     return unit
1433
1434 def regenerate_pattern(unit, copies, input_dims):
1435     pattern = -1 * np.ones(shape=input_dims)
1436     for copy in copies:
1437         transformed = construct_copy(
1438             unit,
1439             copy["reflect"],
1440             copy["rotations"],
1441         )
1442         height, width = transformed.shape
1443         tl_i, tl_j = copy["top left"]
1444         try:
1445             pattern[tl_i:tl_i+height, tl_j:tl_j+width] = transformed
1446         except:
1447             pass
1448     return pattern
  
```

1449 We can test the success of this regeneration with

```

1450
1451 input_map = np.array(input_map)
1452 output = regenerate_pattern(
1453     unit,
1454     partition(),
1455     input_map.shape,
1456 )
  
```

1457

```
1458 Now is your turn. Respond by completing the following Python
1459 code,
1460 1. include everything that is between START OF CODE and END OF
1461 CODE
1462 2. include the entire input map provided, do not use ... to omit
1463 3. ONLY fill partition(), do not use variables/functions that are
1464 not defined
1465 4. In the returned copies, follow the exact key names in the
1466 examples: 'top left', 'reflect', 'rotations'.
1467 4. DO NOT include anything else in the completion
1468
1469 # START OF CODE, make sure to define all arrays as numpy arrays
1470 import numpy as np
1471
1472 input_map = {input map}
1473 unit = {input unit}
1474 input_map = np.array(input_map)
1475 unit = np.array(unit)
1476
1477 def partition():
1478     copies = []
1479     # Place your code here. Let's think step by step
1480     return copies
1481 result = partition()
1482
1483 # END OF CODE
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
```

1512 G.3 IN-PROMPT EXAMPLES

1513

1514 **Example 1**

1515

1516 **Input map**

Unit

1517

1518

```
1519 [
1520   [1, 0, 1, 0],
1521   [0, 1, 0, 1],
1522   [0, 0, 0, 0],
1523   [1, 0, 1, 0],
1524   [0, 1, 0, 1]
1525 ]
```

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

1524

1525 **Reconstruction program 1:**

1526

```
1527 def partition():
1528     return [
1529         {"top left": (0,0), "reflect": False, "rotations": 0},
1530         {"top left": (0,2), "reflect": False, "rotations": 0},
1531         {"top left": (3,0), "reflect": False, "rotations": 0},
1532         {"top left": (3,2), "reflect": False, "rotations": 0},
1533     ]
```

1533

1534

1535

1536

1537

1538

1539

1540

1541

1542

1543

1544

1545

1546

1547

1548

1549

1550

1551

1552

1553

1554

1555

1556

1557

1558

1559

1560

1561

1562

1563

1564

1565

1566 **Example 2**

1567

1568 **Input map**

1569

1570

1571

1572

1573

1574

1575

1576

1577

1578

1579

1580

1581

1582 **Reconstruction program 1:**

1583

1584 `def partition():`1585 `return [`1586 `{"top left": (0,0), "reflect": False, "rotations": 0},`1587 `{"top left": (0,3), "reflect": False, "rotations": 0},`1588 `{"top left": (5,0), "reflect": True, "rotations": 2},`1589 `{"top left": (5,3), "reflect": True, "rotations": 2},`1590 `]`

1591

1592 **Reconstruction program 2:**

1593

1594 `def partition():`1595 `copies = []`1596 `for tl_i, tl_j in [(0,0), (0,3)]:`1597 `copies.append(`1598 `{"top left": (tl_i, tl_j), "reflect": False, "rotation": 0},`1599 `)`1600 `for tl_i, tl_j in [(5,0), (5,3)]:`1601 `copies.append(`1602 `{"top left": (tl_i, tl_j), "reflect": True, "rotations": 2},`1603 `)`1604 `return corners`

1605

1605 **Example 3**

1606

1607 **Input map**

1608

1609

1610

1611

1612

1613

1614

1615

1616

1617

1618

1619

Unit

[

[1,1,1],

[0,0,1],

[0,0,1],

[1,0,1]

]

[

[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

```

1:  $S' \leftarrow 0$ 
2:  $\lambda \leftarrow ""$ 
3:  $u \leftarrow ""$ 
4: while  $S' < t$  do
5:   Generate a prompt from  $I$ 
6:   Send the prompt and receive  $C$  completions  $(\lambda_1, u_1), \dots, (\lambda_C, u_C)$ 
7:   Extract Python programs  $\{\lambda_1, \lambda_2, \dots, \lambda_C\}$ 
8:   for all  $\lambda_i \in \{\lambda_1, \dots, \lambda_C\}$  do
9:     if  $\lambda_i$  runs successfully then
10:       $S_i \leftarrow S(\lambda_i)$ 
11:     end if
12:   end for
13:    $(S', i) \leftarrow \max_i S_i$  ▷ highest scoring program based on likelihood
14:    $\lambda \leftarrow \lambda_i$ 
15:    $u \leftarrow u_i$ 
16: end while
17: return  $\lambda, u$ 

```

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

```

1:  $S' \leftarrow 0$ 
2:  $\lambda \leftarrow ""$ 
3:  $u \leftarrow ""$ 
4: while  $S' < t$  do
5:   Generate a prompt from  $I_p$ 
6:   Send the prompt and receive  $C$  unit candidates  $u_1, \dots, u_C$ 
7:   for all  $u_i \in \{u_1, \dots, u_C\}$  do
8:     Generate a prompt from  $I_c$  and  $u_i$ 
9:     Send the prompt and receive program  $\lambda_i$ 
10:    if  $\lambda_i$  runs successfully then
11:       $S_i \leftarrow S(\lambda_i)$ 
12:    end if
13:  end for
14:   $(S', i) \leftarrow \max_i S_i$  ▷ highest scoring program based on likelihood
15:   $\lambda \leftarrow \lambda_i$ 
16:   $u \leftarrow u_i$ 
17: end while
18: return  $\lambda, u$ 

```

1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673

1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727

Algorithm 3 Structure-Based Planner

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

- 1: $(r, c) \leftarrow$ Initial agent position
- 2: $C \leftarrow$ Empty set to track fully explored unit copies
- 3: $\pi \leftarrow$ Policy for unit exploration based on location of entrance
- 4: **while** C does not contain all copies **do**
- 5: Run POMCP on I until reaching an unexplored unit C_i
- 6: Identify the current entrance e into C_i
- 7: **if** policy from e found in π **then**
- 8: Explore C_i with policy from π
- 9: Add C_i to C
- 10: **else**
- 11: Run POMCP on C_i explore the unit
- 12: Add new policy (e, π_e) to π
- 13: Add C_i to C
- 14: **end if**
- 15: $o_1, \dots, o_m \leftarrow$ Exit locations of C_i
- 16: **for** o_j in $\{o_1, \dots, o_m\}$ **do**
- 17: $p_j \leftarrow$ compute penalty for escaping C_i from o_j
- 18: **end for**
- 19: Run MCTS on C_i until reaching an exit to escape
- 20: **end while**
- 21: Run POMCP on I to explore the rest of the map
- 22: **return** P

Algorithm 4 POMCP for In-Unit Planning

| | |
|---|--|
| <ol style="list-style-type: none"> 1: procedure SEARCH(h) 2: if $B(h) = \emptyset$ then 3: return 4: else 5: repeat 6: $s_{\text{exit}} \sim B(h)$ 7: Simulate($s_{\text{exit}}, h, 0$) 8: until Timeout 9: return $\arg \max_a V(ha)$ 10: end if 11: end procedure 12: 13: procedure ROLLOUT($s_{\text{exit}}, h, \text{depth}$) 14: if $\text{depth} > \text{depth limit}$ then 15: return 0 16: end if 17: $a \sim \pi_{\text{random}}$ 18: $(o, r) \sim \mathcal{G}(h, a)$ 19: if o contains s_{exit} then 20: return r 21: else 22: return $r + \text{Rollout}(s_{\text{exit}}, ha, \text{depth} + 1)$ 23: end if 24: end procedure | <ol style="list-style-type: none"> 1: procedure SIMULATE($s_{\text{exit}}, h, \text{depth}$) 2: $N(h) \leftarrow N(h) + 1$ 3: if $\text{depth} > \text{depth limit}$ then 4: return 0 5: end if 6: if $h \notin T$ then 7: for $a \in \{\text{up, right, bottom, left}\}$ do 8: $T(ha) \leftarrow (N_{\text{init}}(ha), V_{\text{init}}(ha), \emptyset)$ 9: end for 10: return Rollout($s_{\text{exit}}, h, \text{depth}$) 11: end if 12: $a \leftarrow \arg \max_a \left[V(ha) + c \sqrt{\frac{\log N(h)}{N(ha)}} \right]$ 13: $(o, r) \sim \mathcal{G}(h, a)$ 14: if o contains s_{exit} then 15: $N(ha) \leftarrow N(ha) + 1$ 16: else 17: $r \leftarrow r + \text{Simulate}(s_{\text{exit}}, ha, \text{depth} + 1)$ 18: end if 19: $V(ha) \leftarrow V(ha) + \frac{r - V(ha)}{N(ha)}$ 20: return r 21: 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:

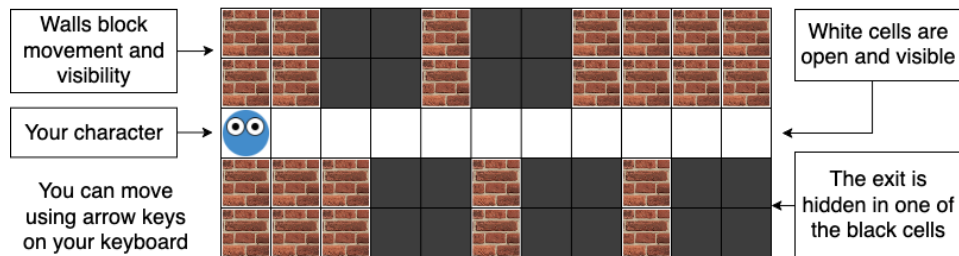


Figure 11: Instructions.

1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835

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.

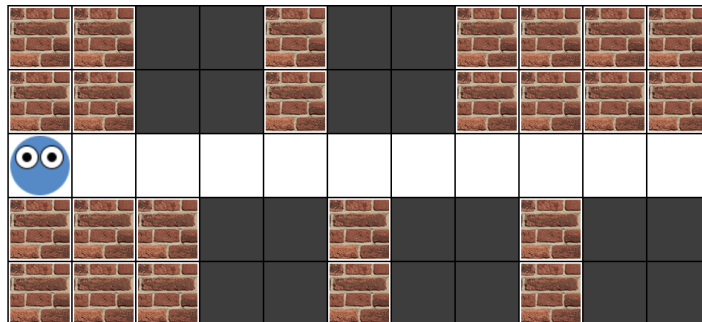


Figure 12: Practice (there are 5 practice trials).

1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889

Please answer the quiz to move on.

Question 1: My task is to ..

- visit every square in the maze
- there is no specific task
- find the exit in each maze
- click as fast as possible

Question 2: Exits are always placed ...

- in the bottom left corner
- in one of the black cells
- some mazes have no exit
- there may be multiple exits

Question 3: Which image correctly shows unseen parts of the maze?

Image A Image B

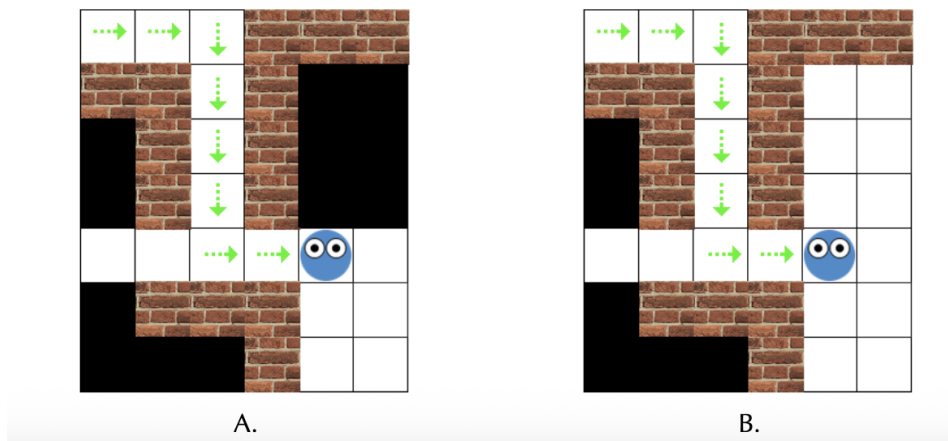


Figure 13: Comprehension Quiz.

1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943

Maze 1 of 21

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

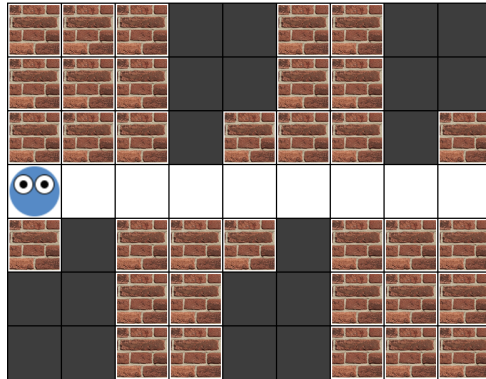


Figure 14: Experiment view.