Solving Exact Cover Problems

Alexander Neuschotz

UC Davis

August 12, 2021

Tiling Problems

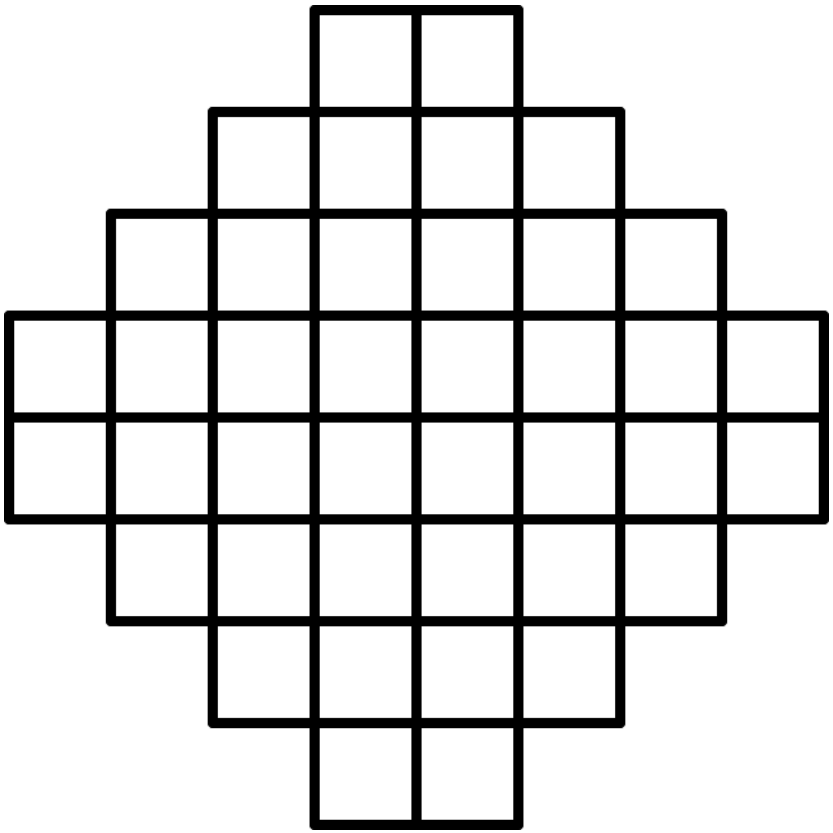


Figure: Aztec Diamond of Order 4

Tiling Problems

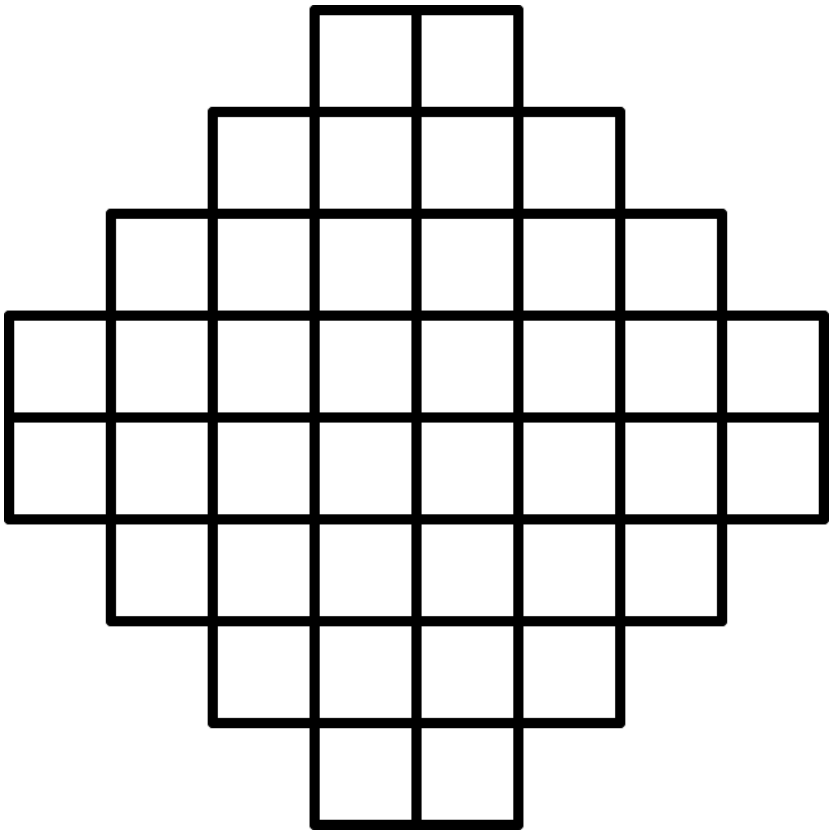


Figure: Aztec Diamond of Order 4

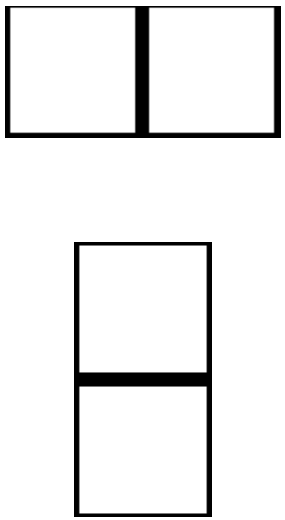


Figure: Tile Types

Set Covers

- Start with a set $U = \{1, \ldots, n\}$ for some $n \in \mathbb{Z}^+$.

Set Covers

- Start with a set $U = \{1, \ldots, n\}$ for some $n \in \mathbb{Z}^+$.
- Consider a family of subsets $S \subseteq U$.

Set Covers

- Start with a set $U = \{1, \ldots, n\}$ for some $n \in \mathbb{Z}^+$.
- Consider a family of subsets $S \subseteq U$.
- Given (U, S), a set cover is a subfamily $S' \subseteq S$ such that $\bigcup_{s' \in S'} s = U$.

Set Cover Problems

- There are a few types of problems one might consider with set covers given (U, S).

Set Cover Problems

- There are a few types of problems one might consider with set covers given (U, S).
  1. Does a non-trivial S' exist?

Set Cover Problems

- There are a few types of problems one might consider with set covers given (U, S).
  1. Does a non-trivial S' exist?
  2. How many set covers exist?

Set Cover Problems

- There are a few types of problems one might consider with set covers given (U, S).
  1. Does a non-trivial S' exist?
  2. How many set covers exist?
  3. Does a pair-wise disjoint set cover exist?

Set Cover Problems

- There are a few types of problems one might consider with set covers given (U, S).
  1. Does a non-trivial S' exist?
  2. How many set covers exist?
  3. Does a pair-wise disjoint set cover exist?
- An exact set cover (or simply exact cover) is a pairwise disjoint set cover.

Set Cover Problems

- There are a few types of problems one might consider with set covers given (U, S).
  1. Does a non-trivial S' exist?
  2. How many set covers exist?
  3. Does a pair-wise disjoint set cover exist?
- An exact set cover (or simply exact cover) is a pairwise disjoint set cover.
- Here, we are interested in determining how many exact covers there are of a given (U, S).

Tiling Problems As Exact Cover Problems

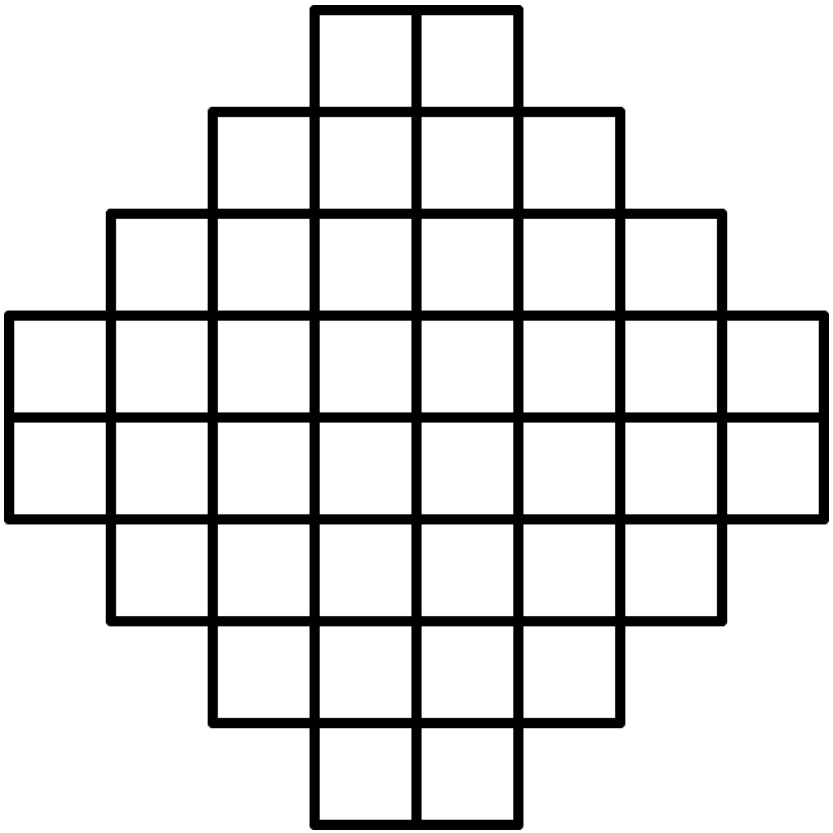


Figure: Aztec Diamond of Order 4

Tiling Problems As Exact Cover Problems

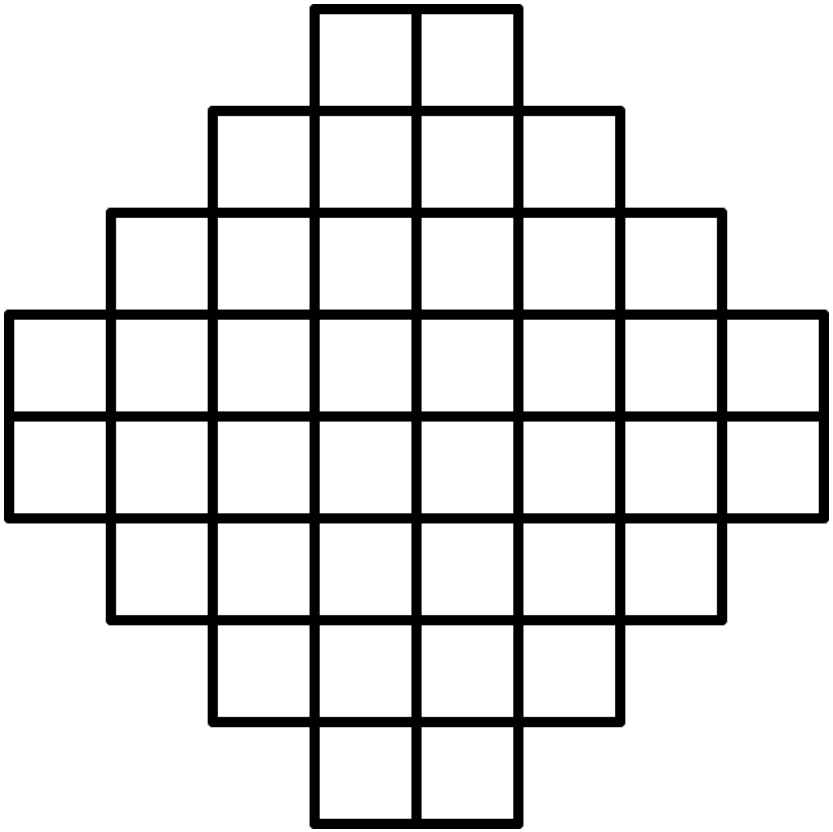


Figure: Aztec Diamond of Order 4

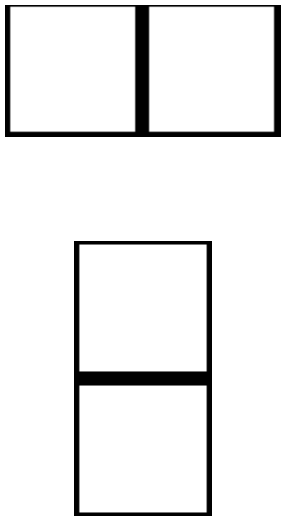


Figure: Tile Types

Tiling Problems As Exact Cover Problems

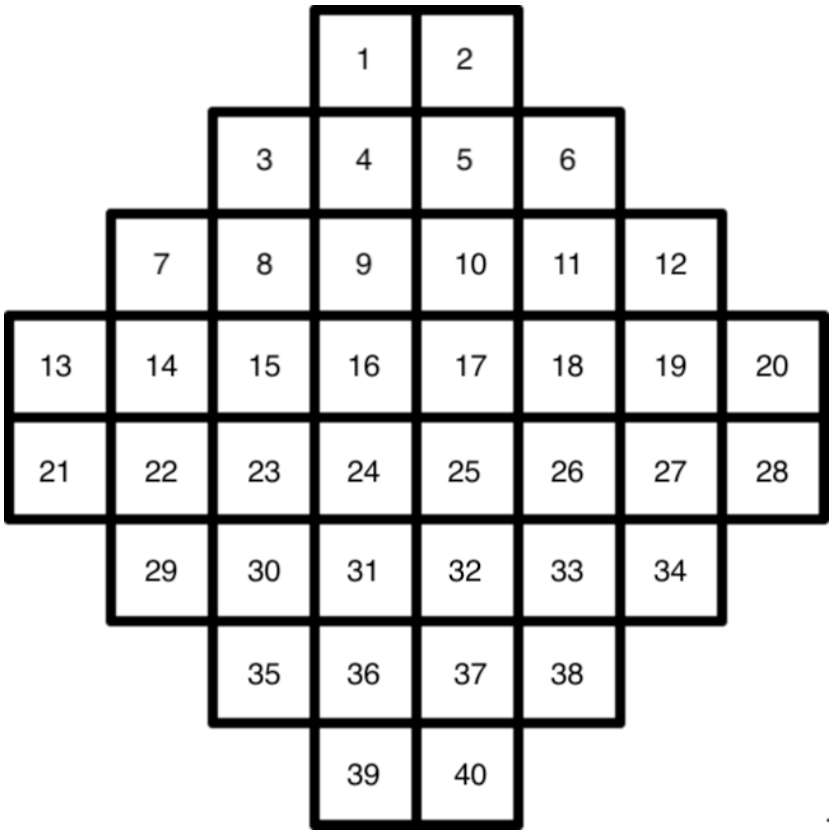


Figure: Enumerated Region

$$\text{Tiles} = \{(1,2),(1,4),(2,5),(3,4),...\}$$

Tiling Problems As Exact Cover Problems

- The set of all numbers used to enumerate the region is our universe U and the set of tiles is our family S.

Tiling Problems As Exact Cover Problems

- The set of all numbers used to enumerate the region is our universe U and the set of tiles is our family S.
- The question of finding an exact cover then becomes isomorphic to the question of how to perfectly tile our region with non-overlapping tiles.

Tiling Problems As Exact Cover Problems

- The set of all numbers used to enumerate the region is our universe U and the set of tiles is our family S.
- The question of finding an exact cover then becomes isomorphic to the question of how to perfectly tile our region with non-overlapping tiles.
- So, given a region that we can enumerate and tile types that we can use to find tiles, solving the exact cover problem that arises will tell us exactly how many perfect tilings of that region are possible with those given tile types.

Ordering Tile Placement

- Regardless of method, it is important to pick an ordering for the placement of tiles.

Ordering Tile Placement

- Regardless of method, it is important to pick an ordering for the placement of tiles.

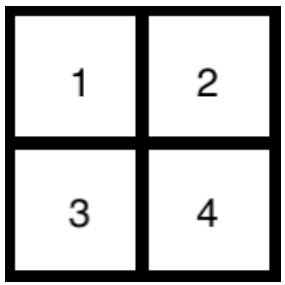


Figure: Aztec Diamond of Order 1

- Suppose we are tiling the above region with the tiles $(1, 2)$ and $(3, 4)$. We don't want $(1, 2) \cup (3, 4)$ and $(3, 4) \cup (1, 2)$ to count as two different perfect tilings since they are equal to the same cover set: $\{(1, 2), (3, 4)\}$

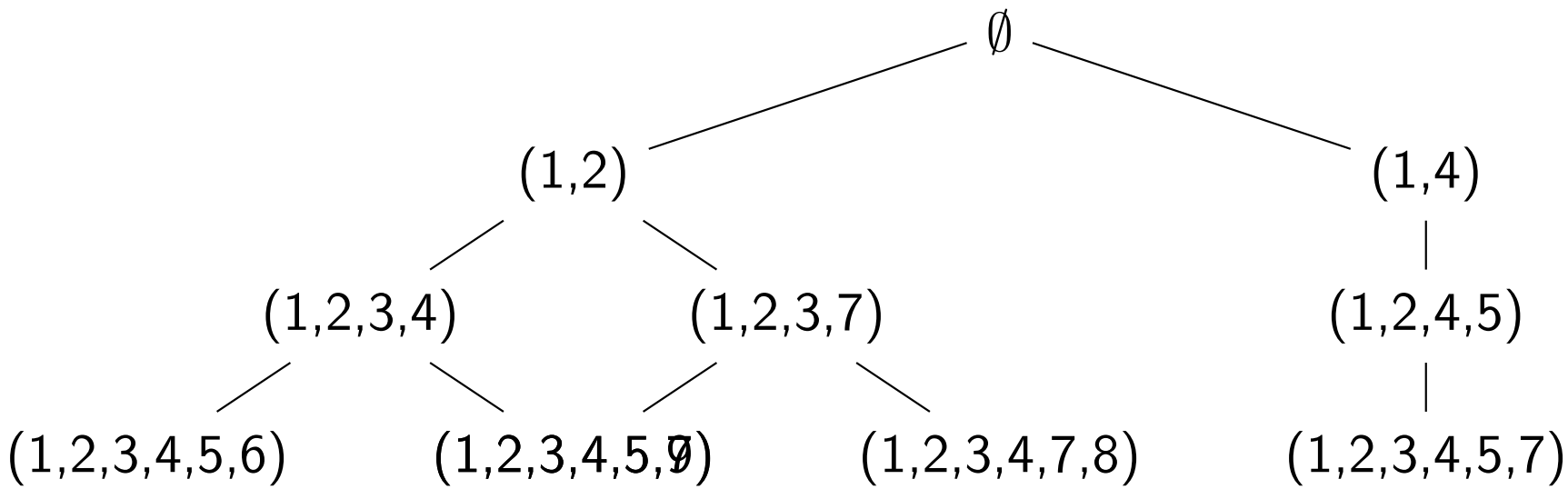
Ordering Tile Placement

- To avoid this issue, we impose the following system: given a partial tiling, we look fro the smallest integer that is not covered and we cover that next.

Ordering Tile Placement

- To avoid this issue, we impose the following system: given a partial tiling, we look fro the smallest integer that is not covered and we cover that next.
- With this system, we would have required $(1, 2)$ to be placed before $(3, 4)$.

The Naïve Approach



The Naïve Approach

- If you were to go complete this tree and count the number of paths to U, you would solve the exact cover problem.

The Naïve Approach

- If you were to go complete this tree and count the number of paths to U, you would solve the exact cover problem.
- The main problem is that this blows up really quickly. In fact, this method has a time cost that is exponential in the area of the region being tiled.

The Naïve Approach

- If you were to go complete this tree and count the number of paths to U, you would solve the exact cover problem.
- The main problem is that this blows up really quickly. In fact, this method has a time cost that is exponential in the area of the region being tiled.
- Solving exact cover problems is always going to be exponential, but we can do better than this approach with a method called dynamic programming.

Dynamic Programming

- Suppose we have a partial tiling of a region that can be achieved in multiple (m) ways ways:

Dynamic Programming

- Suppose we have a partial tiling of a region that can be achieved in multiple (m) ways ways:
- For each way of achieving that partial tiling, there is a branch of n options for the next tile.

Dynamic Programming

- Suppose we have a partial tiling of a region that can be achieved in multiple (m) ways ways:
- For each way of achieving that partial tiling, there is a branch of n options for the next tile.

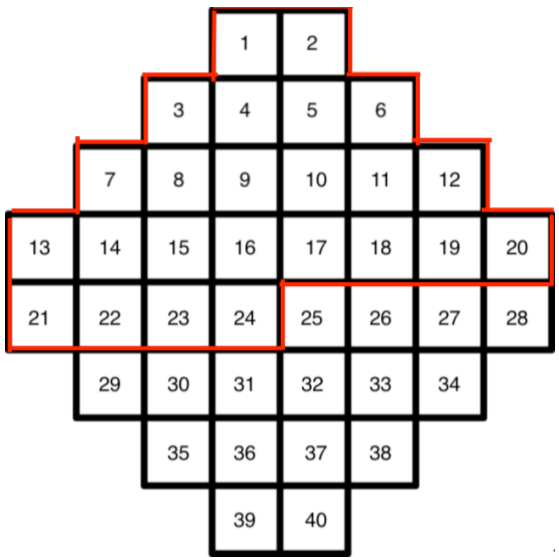


Figure: Partial Tiling of an Aztec Diamond of Order 4

Dynamic Programming

- Suppose we have a partial tiling of a region that can be achieved in multiple (m) ways ways:
- For each way of achieving that partial tiling, there is a branch of n options for the next tile.

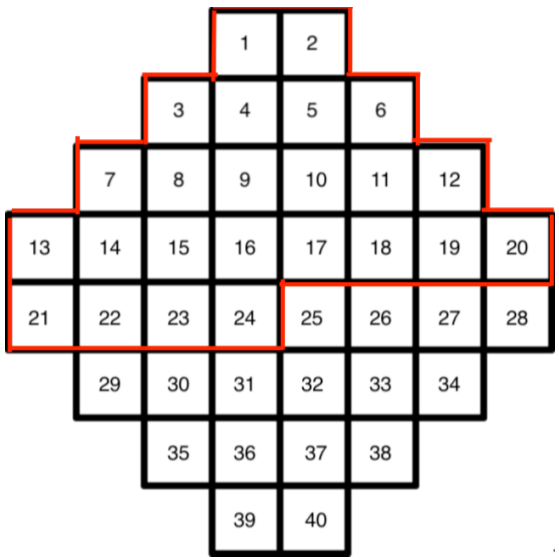


Figure: Partial Tiling of an Aztec Diamond of Order 4

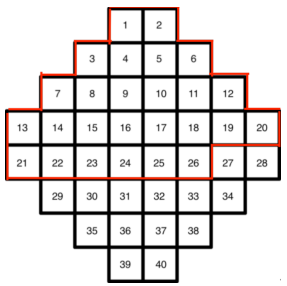


Figure: Option 1

Dynamic Programming

- Suppose we have a partial tiling of a region that can be achieved in multiple (m) ways ways:
- For each way of achieving that partial tiling, there is a branch of n options for the next tile.

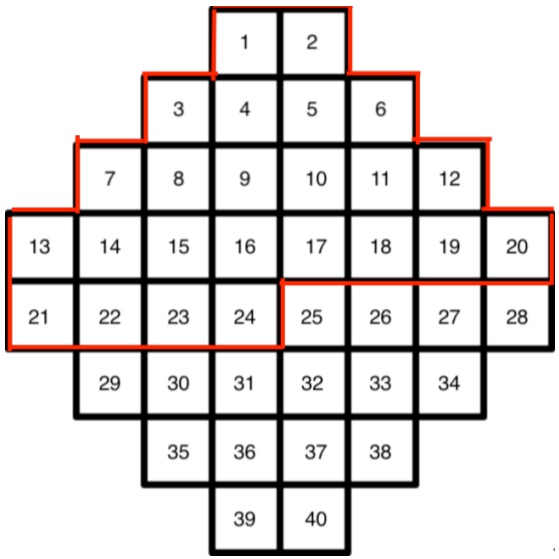


Figure: Partial Tiling of an Aztec Diamond of Order 4

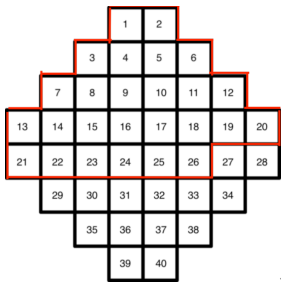


Figure: Option 1

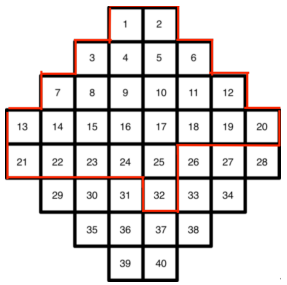


Figure: Option 2

Dynamic Programming

- Rather than branching off m times, we can keep track of the fact that there are m ways to achieve the partial tiling and note that that means there are m ways to achieve each of the n next partial tilings.

Dynamic Programming

- Rather than branching off m times, we can keep track of the fact that there are m ways to achieve the partial tiling and note that that means there are m ways to achieve each of the n next partial tilings.
- In this way, we have "merged" the m paths together.

Dynamic Programming

- Rather than branching off m times, we can keep track of the fact that there are m ways to achieve the partial tiling and note that that means there are m ways to achieve each of the n next partial tilings.
- In this way, we have "merged" the m paths together.
- If we start merging, not at the 24th step, but from the very beginning, we can minimize the amount of repeat work.

Dynamic Programming

- Rather than branching off m times, we can keep track of the fact that there are m ways to achieve the partial tiling and note that that means there are m ways to achieve each of the n next partial tilings.
- In this way, we have "merged" the m paths together.
- If we start merging, not at the 24th step, but from the very beginning, we can minimize the amount of repeat work.
- This method is called dynamic programming and it has a time cost that is exponential in the *width* of the region, which is a lot better than in the area.

Dynamic Programming

- Rather than branching off m times, we can keep track of the fact that there are m ways to achieve the partial tiling and note that that means there are m ways to achieve each of the n next partial tilings.
- In this way, we have "merged" the m paths together.
- If we start merging, not at the 24th step, but from the very beginning, we can minimize the amount of repeat work.
- This method is called dynamic programming and it has a time cost that is exponential in the *width* of the region, which is a lot better than in the area.
- Of course, there is a space trade-off, so this is supposing that time is more valuable than space.

Dynamic Programming

- Rather than branching off m times, we can keep track of the fact that there are m ways to achieve the partial tiling and note that that means there are m ways to achieve each of the n next partial tilings.
- In this way, we have "merged" the m paths together.
- If we start merging, not at the 24th step, but from the very beginning, we can minimize the amount of repeat work.
- This method is called dynamic programming and it has a time cost that is exponential in the *width* of the region, which is a lot better than in the area.
- Of course, there is a space trade-off, so this is supposing that time is more valuable than space.
- This will be central to my program.

Motivation

- A mathematician named David desJardins developed a program that can tile a region given tile types and has been working with a mathematician named Jim Propp who studies tiling problems.

Motivation

- A mathematician named David desJardins developed a program that can tile a region given tile types and has been working with a mathematician named Jim Propp who studies tiling problems.
- David's code is maximally efficient, but there are a couple of problems:

Motivation

- A mathematician named David desJardins developed a program that can tile a region given tile types and has been working with a mathematician named Jim Propp who studies tiling problems.
- David's code is maximally efficient, but there are a couple of problems:
  1. It is not generalized to solve exact cover problems in general.

Motivation

- A mathematician named David desJardins developed a program that can tile a region given tile types and has been working with a mathematician named Jim Propp who studies tiling problems.
- David's code is maximally efficient, but there are a couple of problems:
  1. It is not generalized to solve exact cover problems in general.
  2. It can be made to be more user-friendly. (e.g. It doesn't have a cover script.)

Motivation

- He gave me access to his code and I worked on a program, which I call the Exact Cover Problem Solver (ECPS), which takes the dynamic programming aspect of his program and effectively does two things:

Motivation

- He gave me access to his code and I worked on a program, which I call the Exact Cover Problem Solver (ECPS), which takes the dynamic programming aspect of his program and effectively does two things:
  1. It converts a tiling problem to an exact cover problem.

Motivation

- He gave me access to his code and I worked on a program, which I call the Exact Cover Problem Solver (ECPS), which takes the dynamic programming aspect of his program and effectively does two things:
  1. It converts a tiling problem to an exact cover problem.
  2. It solves exact cover problems.

Motivation

- He gave me access to his code and I worked on a program, which I call the Exact Cover Problem Solver (ECPS), which takes the dynamic programming aspect of his program and effectively does two things:
  1. It converts a tiling problem to an exact cover problem.
  2. It solves exact cover problems.
- The idea was to write out the program in Python and to swap out the computationally intensive part (the dynamic programming) for David's generalized exact cover solver that he would write in $C++$ for efficiency.

Motivation

- He gave me access to his code and I worked on a program, which I call the Exact Cover Problem Solver (ECPS), which takes the dynamic programming aspect of his program and effectively does two things:
  1. It converts a tiling problem to an exact cover problem.
  2. It solves exact cover problems.
- The idea was to write out the program in Python and to swap out the computationally intensive part (the dynamic programming) for David's generalized exact cover solver that he would write in $C++$ for efficiency.
- The program could then be assembled in Cython.

Step 0: User Input

- The ECPS takes in input from the user in the form of ASCII art of regions and tile types with hashtags for tileable units and anything else for non-tileable units.

```
...##...
..####..
.######.
########
########
.######.
..####..
...##...

##

#
#
```

Figure: Input For The ECPS

Step 1: Enumeration

- The region gets converted into a list of lists, (As do each of the tiles, but separately).

Step 1: Enumeration

- The region gets converted into a list of lists, (As do each of the tiles, but separately).
- This list of lists provides a coordinate system whereby a symbol in the region is at (x, y) if it is x down steps and y steps right from the top left symbol.

Step 1: Enumeration

- The region gets converted into a list of lists, (As do each of the tiles, but separately).
- This list of lists provides a coordinate system whereby a symbol in the region is at (x, y) if it is x down steps and y steps right from the top left symbol.
- The region gets sent to a function that assigns a number to each of the hashtags, starting from 1 and not skipping numbers, and returns a coordinate dictionary, or cdict, whose keys are coordinates that contain hashtags and whose values are the numbers associated with those hashtags.

Step 1: Enumeration

- For example, numbering up-to-down, left-to-right, numbering the region from step 0 would return to $\{(0,4):1,(0,5):2,(1,3):3,(1,4):4,(1,5):5,(1,6):6,\ldots\}$.

Step 1: Enumeration

- For example, numbering up-to-down, left-to-right, numbering the region from step 0 would return to $\{(0,4):1,(0,5):2,(1,3):3,(1,4):4,(1,5):5,(1,6):6,\ldots\}$.
- This gives us our universe for the exact cover problem: the set of values of this cdict.

Step 1: Enumeration

- For example, numbering up-to-down, left-to-right, numbering the region from step 0 would return to $\{(0,4):1,(0,5):2,(1,3):3,(1,4):4,(1,5):5,(1,6):6,\ldots\}$.
- This gives us our universe for the exact cover problem: the set of values of this cdict.
- The decision to go from up-to-down, left-to-right works for this particular region, but is not necessarily the best option for regions in general. Right now, the code only numbers in this way, but I aim to fix that. I may address this at the end if there is time.

Step 2: Finding Tiles

- Each of the tile types is similarly numbered, so they have their own cdicts.

Step 2: Finding Tiles

- Each of the tile types is similarly numbered, so they have their own cdicts.
- Choosing one tile type in particular, we look at the keys in its cdict, which are the coordinates $(x_1, y_1), ..., (x_n, y_n)$.

Step 2: Finding Tiles

- Each of the tile types is similarly numbered, so they have their own cdicts.
- Choosing one tile type in particular, we look at the keys in its cdict, which are the coordinates $(x_1, y_1), ..., (x_n, y_n)$.
- Since the numbering started from the top left for both the region and the tile types, there will be a tile in the top left of the region iff each (x_k, y_k) in the region is a hashtag.

Step 2: Finding Tiles

- Each of the tile types is similarly numbered, so they have their own cdicts.
- Choosing one tile type in particular, we look at the keys in its cdict, which are the coordinates $(x_1, y_1), ..., (x_n, y_n)$.
- Since the numbering started from the top left for both the region and the tile types, there will be a tile in the top left of the region iff each (x_k, y_k) in the region is a hashtag.
- In fact, we can find all possible tiles by shifting the tile type coordinates down and to the right together and checking whether they are all positions of hashtags in the region.

Step 2: Finding Tiles

- Taking the inputted region and the first tile type, we can illustrate this process:

Step 2: Finding Tiles

- Taking the inputted region and the first tile type, we can illustrate this process:

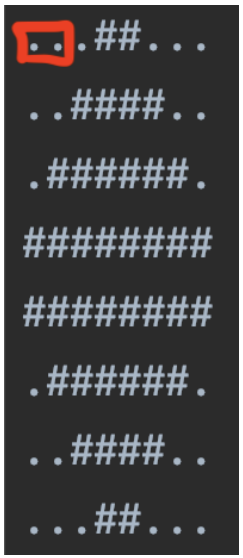


Figure: No Tile

Step 2: Finding Tiles

- Taking the inputted region and the first tile type, we can illustrate this process:

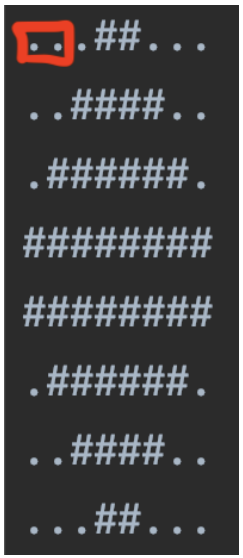


Figure: No Tile

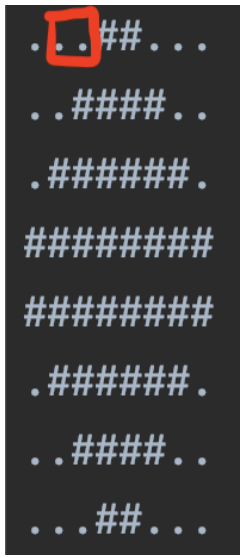


Figure: No Tile

Step 2: Finding Tiles

- Taking the inputted region and the first tile type, we can illustrate this process:

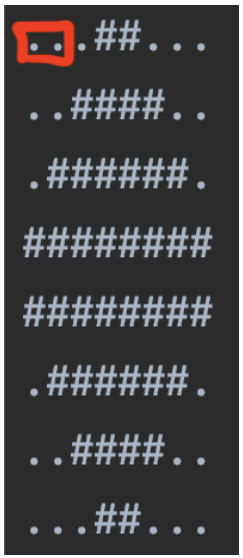


Figure: No Tile

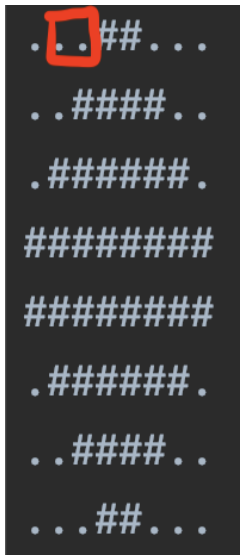


Figure: No Tile

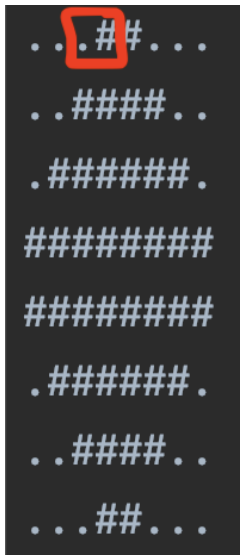


Figure: No Tile

Step 2: Finding Tiles

- Taking the inputted region and the first tile type, we can illustrate this process:

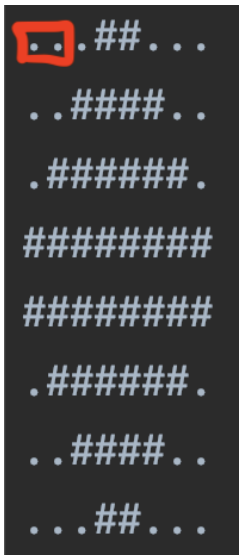


Figure: No Tile

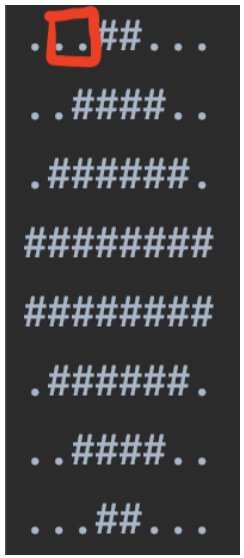


Figure: No Tile

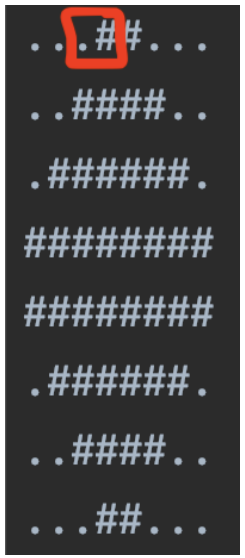


Figure: No Tile

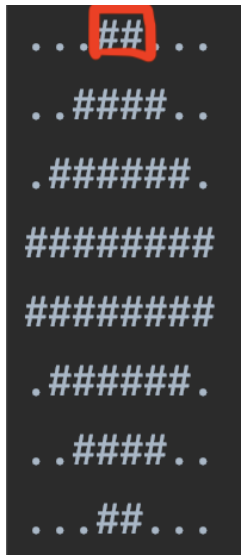


Figure: Tile

Step 2: Finding Tiles

- Taking the inputted region and the first tile type, we can illustrate this process:

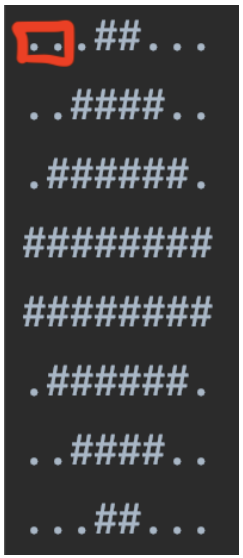


Figure: No Tile

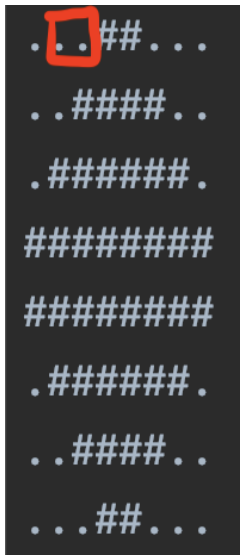


Figure: No Tile

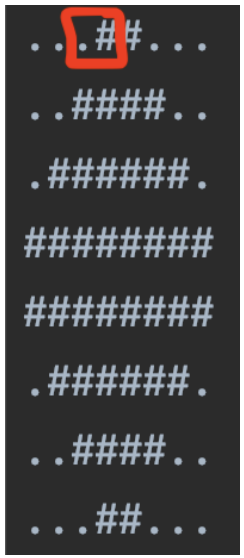


Figure: No Tile

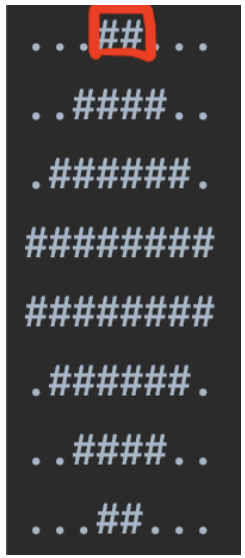


Figure: Tile

- If a potential tile passes the test, the tile is saved as a tuple containing the numbers associated with the hashtags it covers. So, the tile above would be $(1, 2)$.

Solving the Cover set problem

- At this point, we have our universe U (the set of numbers used to number the region) and our family of subsets S (the tiles). We have successfully converted our tiling problem to an exact cover problem.

Solving the Cover set problem

- At this point, we have our universe U (the set of numbers used to number the region) and our family of subsets S (the tiles). We have successfully converted our tiling problem to an exact cover problem.
- From this point on, a "tile" is just another word for a subset in the family S and "tiles" is another word for S itself. We retain the terminology for consistency with the first part of the ECPS, but this part can, importantly, solve exact cover problems in general, not just tiling problems.

Step 2.5: Sorting

- Each of our tiles is a now in the form of a tuple that contains the numbers it covers. We can represent the same information in a binary sequence of length n, where n is the dimension of U and the i'th position from the left is a 1 iff that tile covers i.

Step 2.5: Sorting

- Each of our tiles is a now in the form of a tuple that contains the numbers it covers. We can represent the same information in a binary sequence of length n, where n is the dimension of U and the i'th position from the left is a 1 iff that tile covers i.
- The binary representation of a given tile also represents a unique non-negative integer in base 10, so we can associate the tile with this integer, which we call its key representation, or key rep.

Step 2.5: Sorting

- Each of our tiles is a now in the form of a tuple that contains the numbers it covers. We can represent the same information in a binary sequence of length n, where n is the dimension of U and the i'th position from the left is a 1 iff that tile covers i.
- The binary representation of a given tile also represents a unique non-negative integer in base 10, so we can associate the tile with this integer, which we call its key representation, or key rep.
- For example, when $n = 40$ as above, $(39, 40)$ would be represented by 000...11, which is 3 in base 10.

Step 2.5: Sorting

- Each of our tiles is a now in the form of a tuple that contains the numbers it covers. We can represent the same information in a binary sequence of length n, where n is the dimension of U and the i'th position from the left is a 1 iff that tile covers i.
- The binary representation of a given tile also represents a unique non-negative integer in base 10, so we can associate the tile with this integer, which we call its key representation, or key rep.
- For example, when $n = 40$ as above, $(39, 40)$ would be represented by 000...11, which is 3 in base 10.
- The ECPS creates a list of all key reps of tiles, called a key chain.

Step 2.5: Sorting

- The ECPS creates a list of n lists, called "q-sets", where the i'th list, or q-set contains all of the key reps that have a 1 in the i'th place.

Step 2.5: Sorting

- The ECPS creates a list of n lists, called "q-sets", where the i'th list, or q-set contains all of the key reps that have a 1 in the i'th place.
- The program then creates a list called counts = $\{0, \{0:1\}, \{\}, ...\}$.

Step 2.5: Sorting

- The ECPS creates a list of n lists, called "q-sets", where the i'th list, or q-set contains all of the key reps that have a 1 in the i'th place.
- The program then creates a list called counts $= \{0, \{0 : 1\}, \{\}, ...\}$.
- When we start having partial tilings, the keys of the i'th dictionary in counts will be the key reps for the the partial tilings whose first 0 from the left is in the i'th position and its values will be how many times the partial tiling has been achieved. The integer keeps track of perfect tilings.

Step 2.5: Sorting

- The ECPS creates a list of n lists, called "q-sets", where the i'th list, or q-set contains all of the key reps that have a 1 in the i'th place.
- The program then creates a list called counts = $\{0, \{0 : 1\}, \{\}, ...\}$.
- When we start having partial tilings, the keys of the i'th dictionary in counts will be the key reps for the the partial tilings whose first 0 from the left is in the i'th position and its values will be how many times the partial tiling has been achieved. The integer keeps track of perfect tilings.
- At first, our only partial tiling is the trivial one, represented by 0, of which there is 1.

Step 3: Solving the Problem

- for i in range(n):

Step 3: Solving the Problem

- for i in range(n):
- for key rep in counts[i]: (i.e. for partial tiling that needs i to be covered:)

Step 3: Solving the Problem

- for i in range(n):
- for key rep in counts[i]: (i.e. for partial tiling that needs i to be covered:)
- for key rep of tile in q-sets[i]: (i.e. for tile that can cover i:)

Step 3: Solving the Problem

- for i in range(n):
- for key rep in counts[i]: (i.e. for partial tiling that needs i to be covered:)
- for key rep of tile in q-sets[i]: (i.e. for tile that can cover i:)
- if the tile is compatible with the partial tiling, sum them to get a new partial tiling.

Step 3: Solving the Problem

- for i in range(n):
- for key rep in counts[i]: (i.e. for partial tiling that needs i to be covered:)
- for key rep of tile in q-sets[i]: (i.e. for tile that can cover i:)
- if the tile is compatible with the partial tiling, sum them to get a new partial tiling.
- Update the value of the new partial tiling.
  - If it is a full tiling, update counts[0].
  - If it has a hole, send it to the appropriate dict in counts as a key with the updated value.

Future Work

1. Write a module that will determine the most efficient numbering of a region.
2. David has not gotten back to Greg about the code for the computationally intensive part, so I plan to write that up in Cython myself.

Thank You!

- Thank you to Greg for you guidance!
- Thank you to David for your code!
- Thank you to all of you for listening!