Undecidability and computability for 2-D SFTs

Ronnie Pavlov

University of Denver
www.math.du.edu/∼rpavlov

RTNS 2016
January 27, 2016

2-D SFTs

- Today, we'll work with 2-dimensional shifts of finite type

2-D SFTs

- Today, we'll work with 2-dimensional shifts of finite type
- As before, can assume WLOG nearest-neighbor

2-D SFTs

- Today, we'll work with 2-dimensional shifts of finite type
- As before, can assume WLOG nearest-neighbor
- For most examples today, letters are unit squares with labelings on edges

2-D SFTs

- Today, we'll work with 2-dimensional shifts of finite type
- As before, can assume WLOG nearest-neighbor
- For most examples today, letters are unit squares with labelings on edges
    - Tiles may be adjacent if labels on edges match

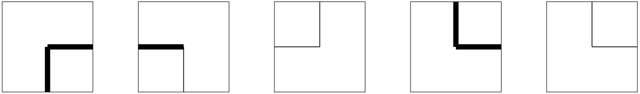
2-D SFTs

- Today, we'll work with 2-dimensional shifts of finite type
- As before, can assume WLOG nearest-neighbor
- For most examples today, letters are unit squares with labelings on edges
  - Tiles may be adjacent if labels on edges match
- Specific type of 2-D SFT called **Wang tiling**

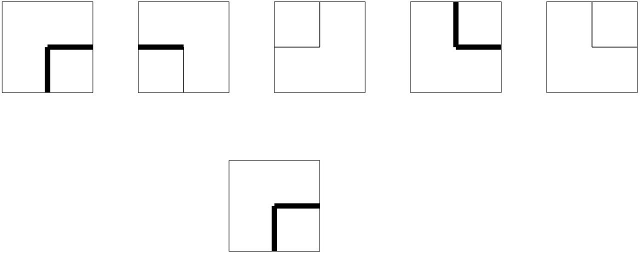
2-D SFTs

- Today, we'll work with 2-dimensional shifts of finite type
- As before, can assume WLOG nearest-neighbor
- For most examples today, letters are unit squares with labelings on edges
    - Tiles may be adjacent if labels on edges match
- Specific type of 2-D SFT called **Wang tiling**
    - In fact 2-D SFT can be assumed Wang tiling WLOG as well, but we won't prove

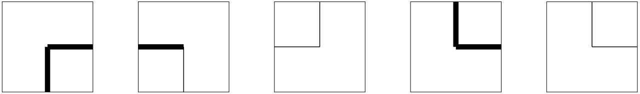
Example

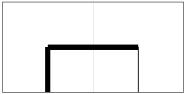


Example

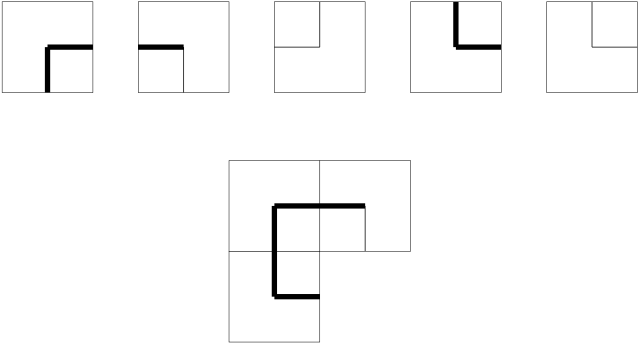


Example

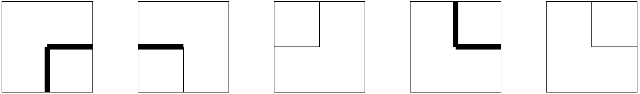


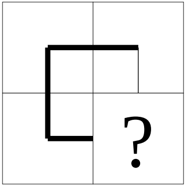


Example

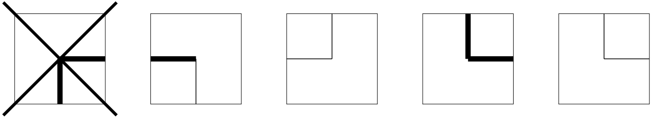


Example

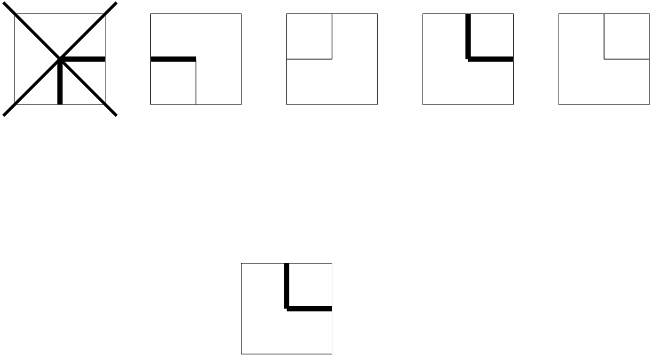




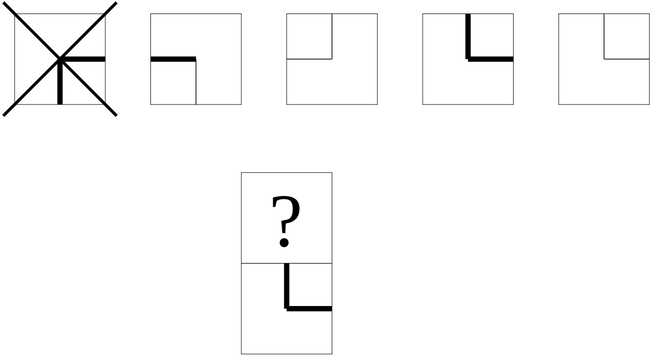
Example



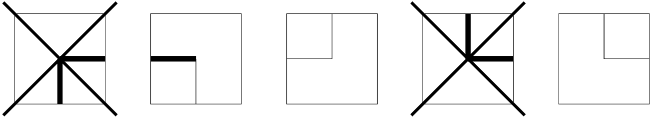
Example



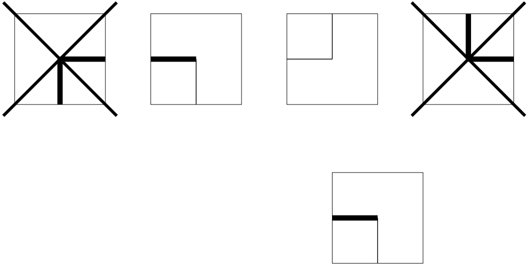
Example



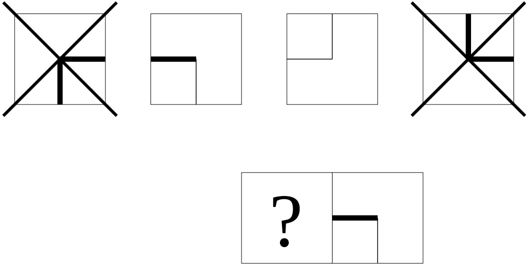
Example



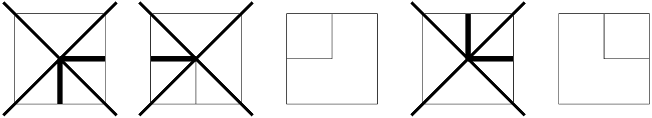
Example



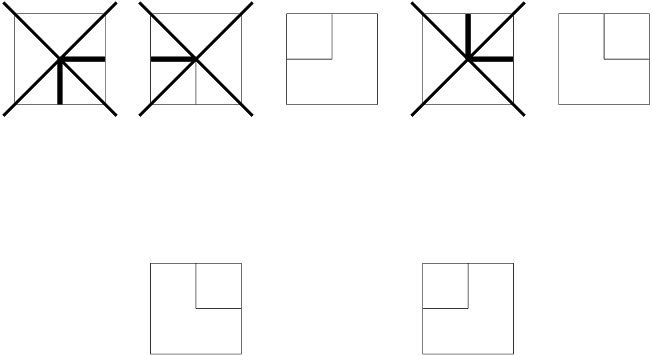
Example



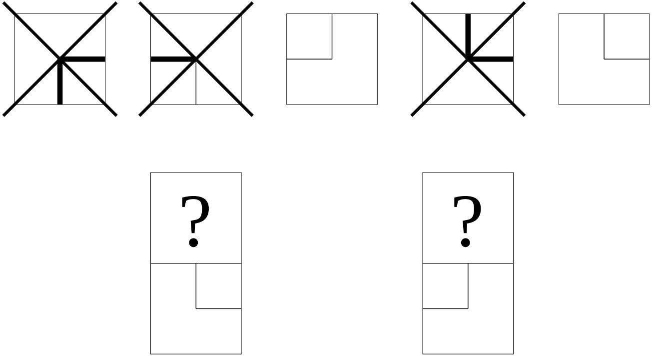
Example



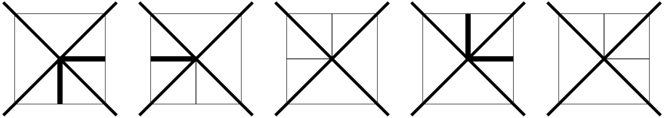
Example



Example



Example



How to decide if a 2-D n.n. SFT X is nonempty

- Basic question: given A, $\mathcal{F}$, is $X(\mathcal{F}) \neq \varnothing$?

How to decide if a 2-D n.n. SFT X is nonempty

- Basic question: given A, $\mathcal{F}$, is $X(\mathcal{F}) \neq \varnothing$?
- Easy to demonstrate that X is empty; show that for some n, every $n \times n$ pattern on A contains forbidden adjacency

How to decide if a 2-D n.n. SFT X is nonempty

- Basic question: given A, $\mathcal{F}$, is $X(\mathcal{F}) \neq \varnothing$?
- Easy to demonstrate that X is empty; show that for some n, every $n \times n$ pattern on A contains forbidden adjacency
- To demonstrate that X is nonempty requires an INFINITE array; impossible to do in finite time

How to decide if a 2-D n.n. SFT X is nonempty

- Basic question: given A, $\mathcal{F}$, is $X(\mathcal{F}) \neq \varnothing$?
- Easy to demonstrate that X is empty; show that for some n, every $n \times n$ pattern on A contains forbidden adjacency
- To demonstrate that X is nonempty requires an INFINITE array; impossible to do in finite time
- Idea: use periodic configurations; existence can be demonstrated via one finite pattern

How to decide if a 2-D n.n. SFT X is nonempty

- Basic question: given A, $\mathcal{F}$, is $X(\mathcal{F}) \neq \varnothing$?
- Easy to demonstrate that X is empty; show that for some n, every $n \times n$ pattern on A contains forbidden adjacency
- To demonstrate that X is nonempty requires an INFINITE array; impossible to do in finite time
- Idea: use periodic configurations; existence can be demonstrated via one finite pattern
- In 1-D, this is simple. If $a \dots a$ is legal, then can make periodic point $\dots a \dots a \dots a \dots$ in X.

Checking nonemptiness for $d = 1$

- Gives algorithm for $d = 1$:

Checking nonemptiness for $d = 1$

- Gives algorithm for $d = 1$:
  - Try to construct a valid string of length $|A| + 1$

Checking nonemptiness for $d = 1$

- Gives algorithm for $d = 1$:
  - Try to construct a valid string of length $|A| + 1$
  - If you can't, clearly there are no infinite configurations $\rightarrow$ empty

Checking nonemptiness for $d = 1$

- Gives algorithm for $d = 1$:
  - Try to construct a valid string of length $|A| + 1$
  - If you can't, clearly there are no infinite configurations → empty
  - If you can, by Pigeonhole Principle some tile repeats; you could use it to create an infinite configuration → nonempty

Checking nonemptiness for $d = 1$

- Gives algorithm for $d = 1$:
  - Try to construct a valid string of length $|A| + 1$
  - If you can't, clearly there are no infinite configurations → empty
  - If you can, by Pigeonhole Principle some tile repeats; you could use it to create an infinite configuration → nonempty
- Clearly decides nonemptiness in finite time!

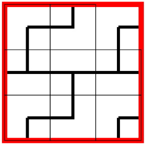
Checking nonemptiness for $d = 1$

- Gives algorithm for $d = 1$:
  - Try to construct a valid string of length $|A| + 1$
  - If you can't, clearly there are no infinite configurations → empty
  - If you can, by Pigeonhole Principle some tile repeats; you could use it to create an infinite configuration → nonempty
- Clearly decides nonemptiness in finite time!
- What about $d = 2$? More complicated, tiles can "interfere" in more complicated ways

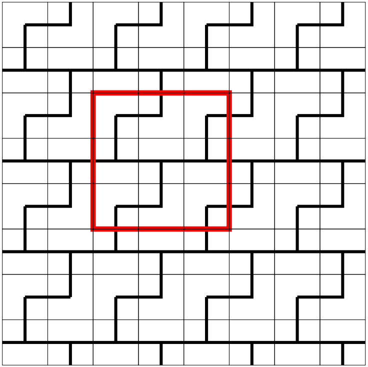
Checking nonemptiness for $d = 1$

- Gives algorithm for $d = 1$:
    - Try to construct a valid string of length $|A| + 1$
    - If you can't, clearly there are no infinite configurations → empty
    - If you can, by Pigeonhole Principle some tile repeats; you could use it to create an infinite configuration → nonempty
- Clearly decides nonemptiness in finite time!
- What about $d = 2$? More complicated, tiles can "interfere" in more complicated ways
- (Totally) periodic configurations still come from finite patterns

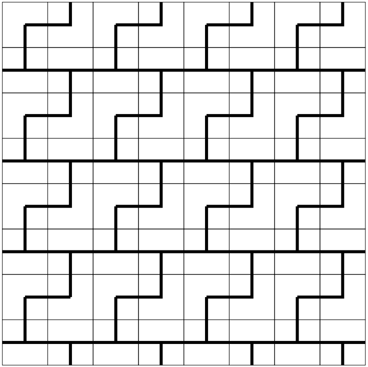
Periodic tiling



Periodic tiling



Periodic tiling



Checking nonemptiness

- Idea (Hao Wang): Assume that every nonempty 2-D n.n. SFT has a (totally) periodic configuration

Checking nonemptiness

- Idea (Hao Wang): Assume that every nonempty 2-D n.n. SFT has a (totally) periodic configuration
    - Algorithm: For parameter n (start with $n = 2$), construct all $n \times n$ patterns.

Checking nonemptiness

- Idea (Hao Wang): Assume that every nonempty 2-D n.n. SFT has a (totally) periodic configuration
  - Algorithm: For parameter n (start with $n = 2$), construct all $n \times n$ patterns.
  - If there are no legal $n \times n$ patterns, clearly there are no infinite configurations → empty

Checking nonemptiness

- Idea (Hao Wang): Assume that every nonempty 2-D n.n. SFT has a (totally) periodic configuration
  - Algorithm: For parameter n (start with $n = 2$), construct all $n \times n$ patterns.
  - If there are no legal $n \times n$ patterns, clearly there are no infinite configurations → empty
  - If there is a legal $n \times n$ pattern with identical left and right edges and identical top and bottom edges, then you could use this to create an infinite configuration → nonempty

Checking nonemptiness

- Idea (Hao Wang): Assume that every nonempty 2-D n.n. SFT has a (totally) periodic configuration
    - Algorithm: For parameter n (start with $n = 2$), construct all $n \times n$ patterns.
    - If there are no legal $n \times n$ patterns, clearly there are no infinite configurations → empty
    - If there is a legal $n \times n$ pattern with identical left and right edges and identical top and bottom edges, then you could use this to create an infinite configuration → nonempty
    - If neither is true, move to next n

Checking nonemptiness

- Idea (Hao Wang): Assume that every nonempty 2-D n.n. SFT has a (totally) periodic configuration
  - Algorithm: For parameter n (start with $n = 2$), construct all $n \times n$ patterns.
  - If there are no legal $n \times n$ patterns, clearly there are no infinite configurations → empty
  - If there is a legal $n \times n$ pattern with identical left and right edges and identical top and bottom edges, then you could use this to create an infinite configuration → nonempty
  - If neither is true, move to next n
- By assumption, at some point algorithm will terminate (but you don't know when!)

Checking nonemptiness

- Conjecture: (Wang, 1961) Every nonempty n.n. SFT has (totally) periodic configurations

Checking nonemptiness

- Conjecture: (Wang, 1961) Every nonempty n.n. SFT has (totally) periodic configurations
    - If true, shows that one can decide nonemptiness in finite time

Checking nonemptiness

- Conjecture: (Wang, 1961) Every nonempty n.n. SFT has (totally) periodic configurations
    - If true, shows that one can decide nonemptiness in finite time
- Theorem: (Berger, 1966) There exists a nonempty 2-D n.n. SFT without (totally) periodic configurations!

Checking nonemptiness

- Conjecture: (Wang, 1961) Every nonempty n.n. SFT has (totally) periodic configurations
    - If true, shows that one can decide nonemptiness in finite time
- Theorem: (Berger, 1966) There exists a nonempty 2-D n.n. SFT without (totally) periodic configurations!
- Berger's original tiling had $20,426$ tiles

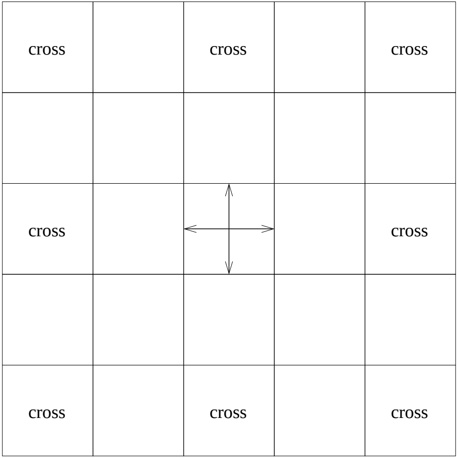
Checking nonemptiness

- Conjecture: (Wang, 1961) Every nonempty n.n. SFT has (totally) periodic configurations
  - If true, shows that one can decide nonemptiness in finite time
- Theorem: (Berger, 1966) There exists a nonempty 2-D n.n. SFT without (totally) periodic configurations!
- Berger's original tiling had $20,426$ tiles
- We'll use a later example of Robinson with only 56 tiles

Robinson tiling

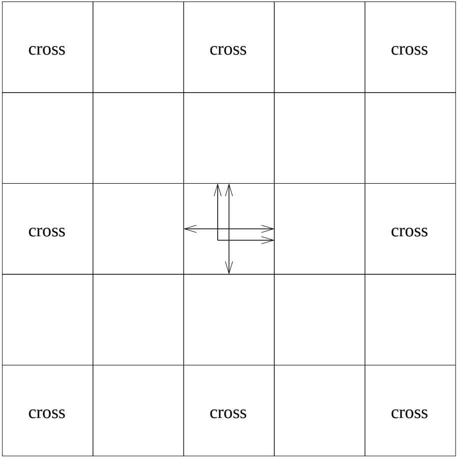
cross		cross		cross
cross		cross		cross
cross		cross		cross

Robinson tiling



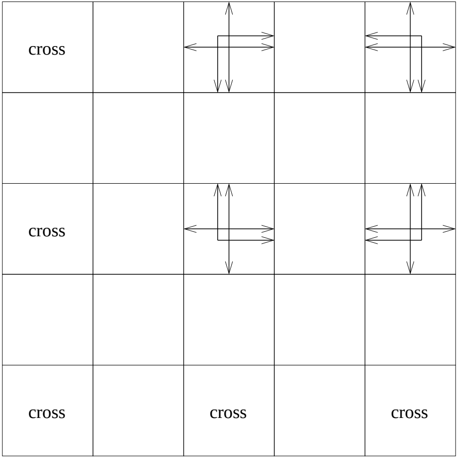
cross		cross		cross
cross				cross
cross		cross		cross

Robinson tiling

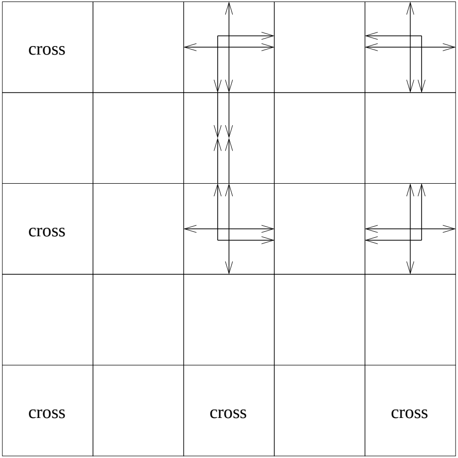


cross		cross		cross
cross				cross
cross		cross		cross

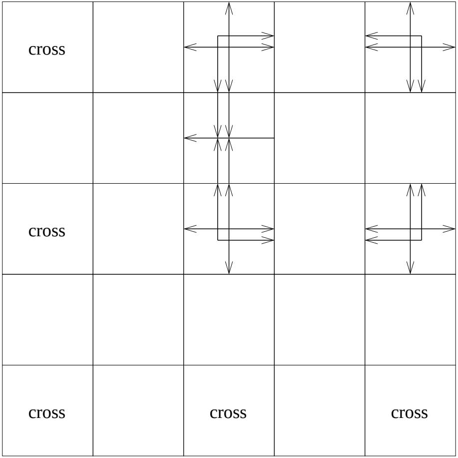
Robinson tiling



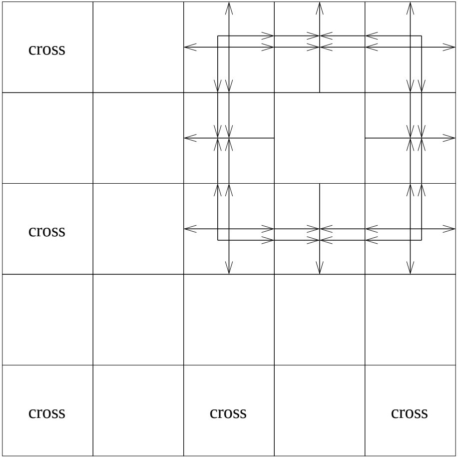
Robinson tiling



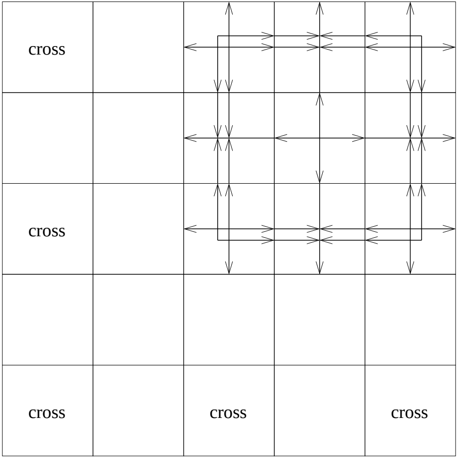
Robinson tiling



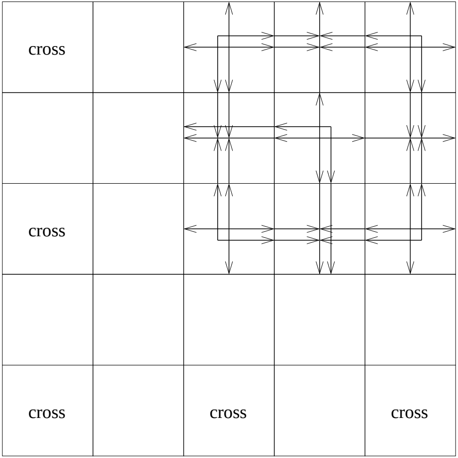
Robinson tiling



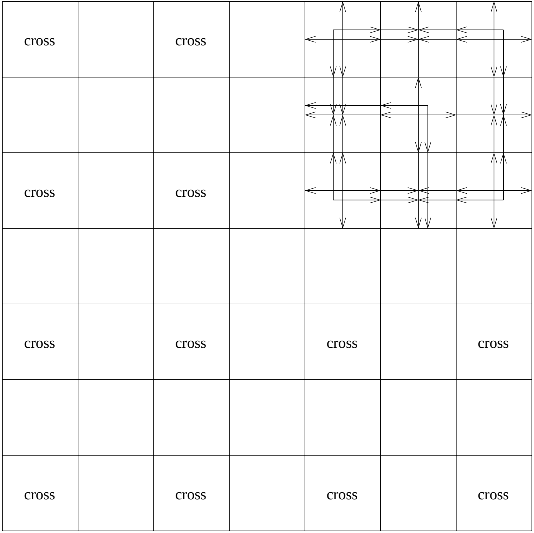
Robinson tiling



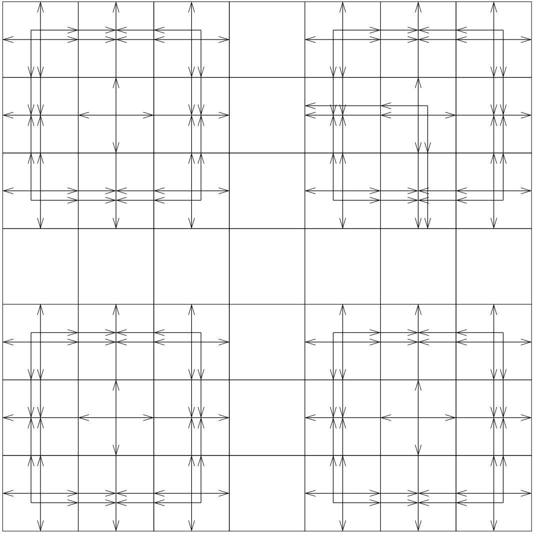
Robinson tiling



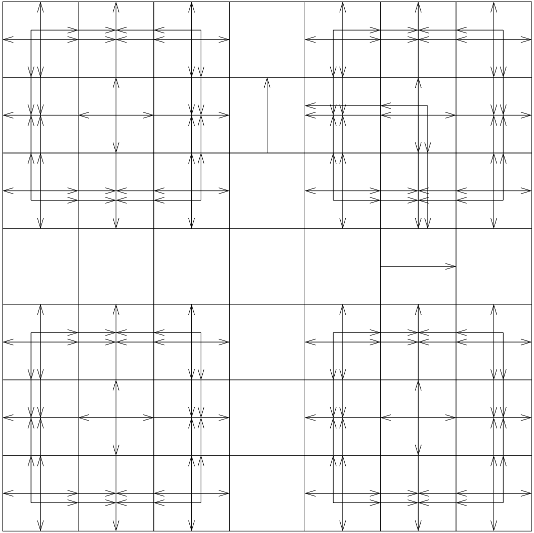
Robinson tiling



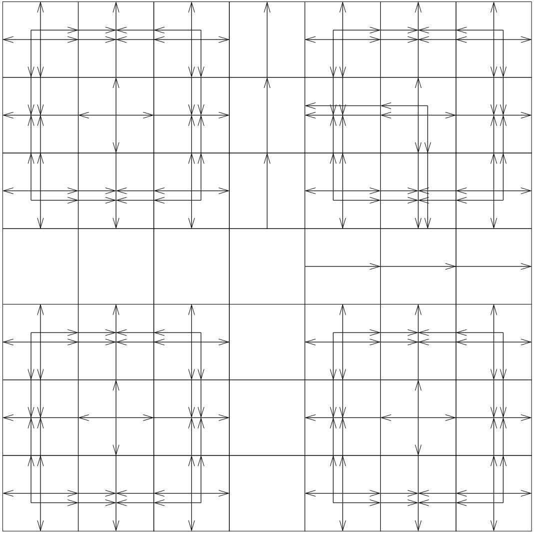
Robinson tiling



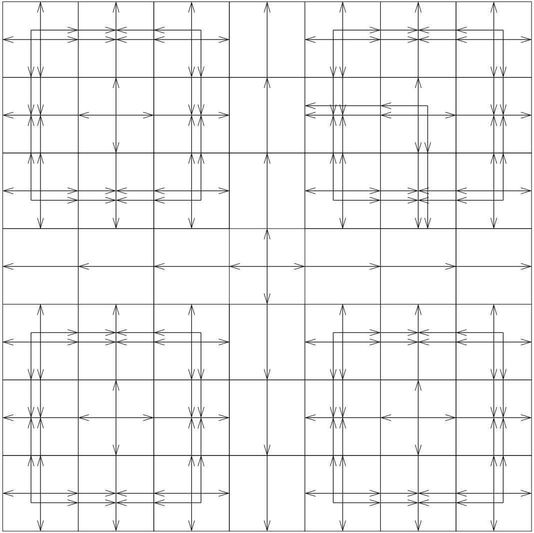
Robinson tiling



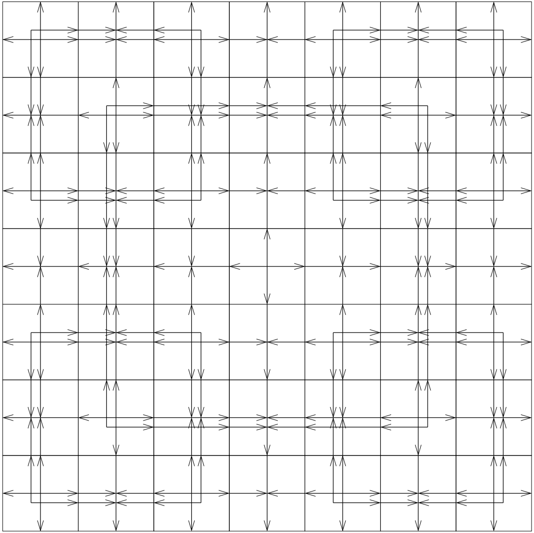
Robinson tiling



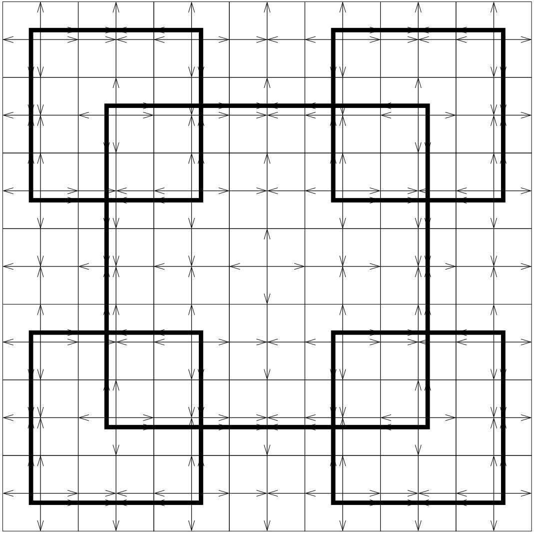
Robinson tiling



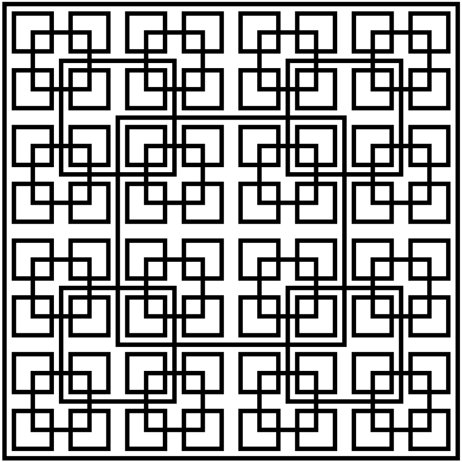
Robinson tiling



Robinson tiling



Robinson tiling



Robinson tiling

- Robinson SFT is nonempty; can continue forever

Robinson tiling

- Robinson SFT is nonempty; can continue forever
- But points have a forced hierarchical structure; no periodic points

Checking nonemptiness

- By itself, this only means that Wang's proposed algorithm won't always work

Checking nonemptiness

- By itself, this only means that Wang's proposed algorithm won't always work
- But amazingly, the technique of the counterexample can show more:

Checking nonemptiness

- By itself, this only means that Wang's proposed algorithm won't always work
- But amazingly, the technique of the counterexample can show more:
- Theorem: (Berger, 1966) The problem of deciding nonemptiness of a 2-D n.n. SFT is undecidable; there CANNOT exist an algorithm which will, on input $A, \mathcal{F}$, decide if it is nonempty

The Halting Problem

- Shift gears for now to computer science

The Halting Problem

- Shift gears for now to computer science
- Any computer program/algorithm will either halt at some point

The Halting Problem

- Shift gears for now to computer science
- Any computer program/algorithm will either halt at some point

```
10 PRINT "HELLO WORLD"
```

The Halting Problem

- Shift gears for now to computer science
- Any computer program/algorithm will either halt at some point

```
10 PRINT “HELLO WORLD”
20 END
```

The Halting Problem

- Shift gears for now to computer science
- Any computer program/algorithm will either halt at some point

  ```
  10 PRINT "HELLO WORLD"
  20 END
  ```

- Or will run forever

The Halting Problem

- Shift gears for now to computer science
- Any computer program/algorithm will either halt at some point

```
10 PRINT "HELLO WORLD"
20 END
```

- Or will run forever

```
10 PRINT "HELLO WORLD"
```

The Halting Problem

- Shift gears for now to computer science
- Any computer program/algorithm will either halt at some point

  ```
  10 PRINT “HELLO WORLD”
  20 END
  ```

- Or will run forever

  ```
  10 PRINT “HELLO WORLD”
  20 GOTO 10
  ```

The Halting Problem

- Shift gears for now to computer science
- Any computer program/algorithm will either halt at some point

```
10 PRINT "HELLO WORLD"
20 END
```

- Or will run forever

```
10 PRINT "HELLO WORLD"
20 GOTO 10
```

- We will say that a **halting oracle** is a computer program/algorithm which, when given the code of an arbitrary computer program P, decides whether P halts or runs forever

The Halting Problem

- Shift gears for now to computer science
- Any computer program/algorithm will either halt at some point

```
10 PRINT "HELLO WORLD"
20 END
```

- Or will run forever

```
10 PRINT "HELLO WORLD"
20 GOTO 10
```

- We will say that a **halting oracle** is a computer program/algorithm which, when given the code of an arbitrary computer program P, decides whether P halts or runs forever
- Can a halting oracle exist?

The Halting Problem

- Suppose a halting oracle exists, call it H

The Halting Problem

- Suppose a halting oracle exists, call it H
- Create a new program R, called a **halting reverser**, whose input is the code of an arbitrary computer program P:

The Halting Problem

- Suppose a halting oracle exists, call it H
- Create a new program R, called a **halting reverser**, whose input is the code of an arbitrary computer program P:
  - First R runs the halting oracle H to decide whether P halts or runs forever

The Halting Problem

- Suppose a halting oracle exists, call it H
- Create a new program R, called a **halting reverser**, whose input is the code of an arbitrary computer program P:
    - First R runs the halting oracle H to decide whether P halts or runs forever
    - If P halts, R begins an infinite loop, thus running forever

The Halting Problem

- Suppose a halting oracle exists, call it H
- Create a new program R, called a **halting reverser**, whose input is the code of an arbitrary computer program P:
  - First R runs the halting oracle H to decide whether P halts or runs forever
  - If P halts, R begins an infinite loop, thus running forever
  - If P runs forever, R halts immediately

The Halting Problem

- Suppose a halting oracle exists, call it H
- Create a new program R, called a **halting reverser**, whose input is the code of an arbitrary computer program P:
  - First R runs the halting oracle H to decide whether P halts or runs forever
  - If P halts, R begins an infinite loop, thus running forever
  - If P runs forever, R halts immediately
- R, on input P, exhibits halting behavior which is the OPPOSITE of P

The Halting Problem

- Suppose a halting oracle exists, call it H
- Create a new program R, called a **halting reverser**, whose input is the code of an arbitrary computer program P:
  - First R runs the halting oracle H to decide whether P halts or runs forever
  - If P halts, R begins an infinite loop, thus running forever
  - If P runs forever, R halts immediately
- R, on input P, exhibits halting behavior which is the OPPOSITE of P
- Contradiction: try running R with input R!

The Halting Problem

- Suppose a halting oracle exists, call it H
- Create a new program R, called a **halting reverser**, whose input is the code of an arbitrary computer program P:
    - First R runs the halting oracle H to decide whether P halts or runs forever
    - If P halts, R begins an infinite loop, thus running forever
    - If P runs forever, R halts immediately
- R, on input P, exhibits halting behavior which is the OPPOSITE of P
- Contradiction: try running R with input R!
- If R halts, then R runs forever; if R runs forever, then R halts

The Halting Problem

- Suppose a halting oracle exists, call it H
- Create a new program R, called a **halting reverser**, whose input is the code of an arbitrary computer program P:
    - First R runs the halting oracle H to decide whether P halts or runs forever
    - If P halts, R begins an infinite loop, thus running forever
    - If P runs forever, R halts immediately
- R, on input P, exhibits halting behavior which is the OPPOSITE of P
- Contradiction: try running R with input R!
- If R halts, then R runs forever; if R runs forever, then R halts
- A halting oracle cannot exist!

The Halting Problem

- Suppose a halting oracle exists, call it H
- Create a new program R, called a **halting reverser**, whose input is the code of an arbitrary computer program P:
  - First R runs the halting oracle H to decide whether P halts or runs forever
  - If P halts, R begins an infinite loop, thus running forever
  - If P runs forever, R halts immediately
- R, on input P, exhibits halting behavior which is the OPPOSITE of P
- Contradiction: try running R with input R!
- If R halts, then R runs forever; if R runs forever, then R halts
- A halting oracle cannot exist!
  - Similar to Russell's paradox, Gödel's Incompleteness Theorem

Turing machines

- Simplistic model of computing

Turing machines

- Simplistic model of computing
- A head moves back and forth on a tape, moving, erasing, and copying symbols dependent on its “internal state” (not written down) and the symbol it sees on the tape

Turing machines

- Simplistic model of computing
- A head moves back and forth on a tape, moving, erasing, and copying symbols dependent on its “internal state” (not written down) and the symbol it sees on the tape
- Some internal states are “halting” states; when the machine reaches those, it will not do further computation

Turing machines

- Simplistic model of computing
- A head moves back and forth on a tape, moving, erasing, and copying symbols dependent on its “internal state” (not written down) and the symbol it sees on the tape
- Some internal states are “halting” states; when the machine reaches those, it will not do further computation
- Can define a n.n. SFT which implements a Turing machine as a space-time diagram; rows show successive steps in computation

Implementation tiles

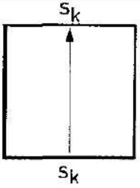


Fig. 12. Alphabet tile

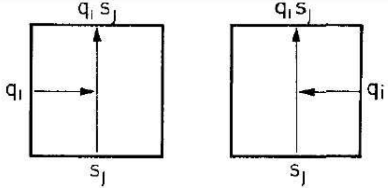


Fig. 13. Merging tiles

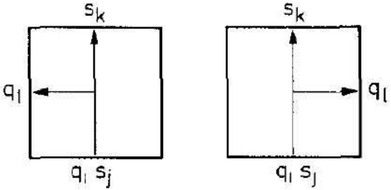


Fig. 14. Action tiles

Turing machines

- Problem: can't force the head to appear! There will always be points consisting of unchanging tape with no head

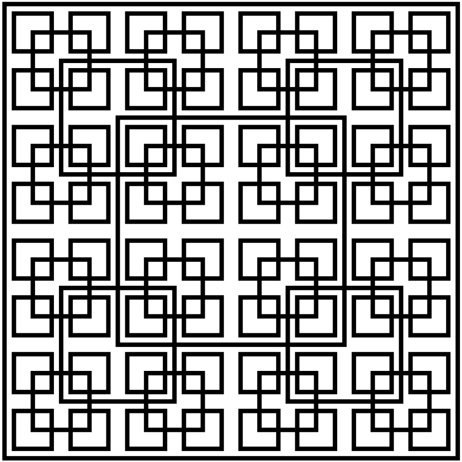
Turing machines

- Problem: can’t force the head to appear! There will always be points consisting of unchanging tape with no head
- Solution: Robinson SFT!

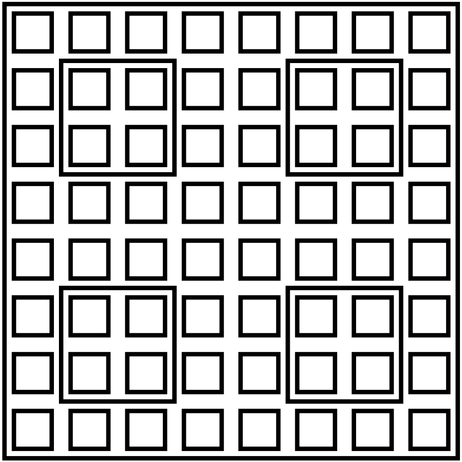
Turing machines

- Problem: can't force the head to appear! There will always be points consisting of unchanging tape with no head
- Solution: Robinson SFT!
- Points of Robinson SFT can separate the plane into disjoint "boards"

Robinson tiling



Robinson tiling



Turing machines

- Initialize computations on center of lower edge of each "board"

Turing machines

- Initialize computations on center of lower edge of each “board”
- Use “transmission signals” to run computation in larger board without intersecting smaller boards inside it

Turing machines

- Initialize computations on center of lower edge of each "board"
- Use "transmission signals" to run computation in larger board without intersecting smaller boards inside it
- Since there exist boards of arbitrary size, if Turing machine halts, X will eventually not be able to fill a board, so X will be empty

Turing machines

- Initialize computations on center of lower edge of each “board”
- Use “transmission signals” to run computation in larger board without intersecting smaller boards inside it
- Since there exist boards of arbitrary size, if Turing machine halts, X will eventually not be able to fill a board, so X will be empty
- If the Turing machine runs forever, then all boards can be filled, and so X will be nonempty

Turing machines

- Initialize computations on center of lower edge of each "board"
- Use "transmission signals" to run computation in larger board without intersecting smaller boards inside it
- Since there exist boards of arbitrary size, if Turing machine halts, X will eventually not be able to fill a board, so X will be empty
- If the Turing machine runs forever, then all boards can be filled, and so X will be nonempty
- If the nonemptiness problem was decidable, the halting problem would be decidable!

Turing machines

- Initialize computations on center of lower edge of each "board"
- Use "transmission signals" to run computation in larger board without intersecting smaller boards inside it
- Since there exist boards of arbitrary size, if Turing machine halts, X will eventually not be able to fill a board, so X will be empty
- If the Turing machine runs forever, then all boards can be filled, and so X will be nonempty
- If the nonemptiness problem was decidable, the halting problem would be decidable!
- So, nonemptiness of a 2-D n.n. SFT is not decidable