Rectangle Corner Visiblity Graphs

August Bergquist Ezekiel Jakob Druker Juni DeYoung
Lani Southern

Background: Graphs

$$G := V + E$$

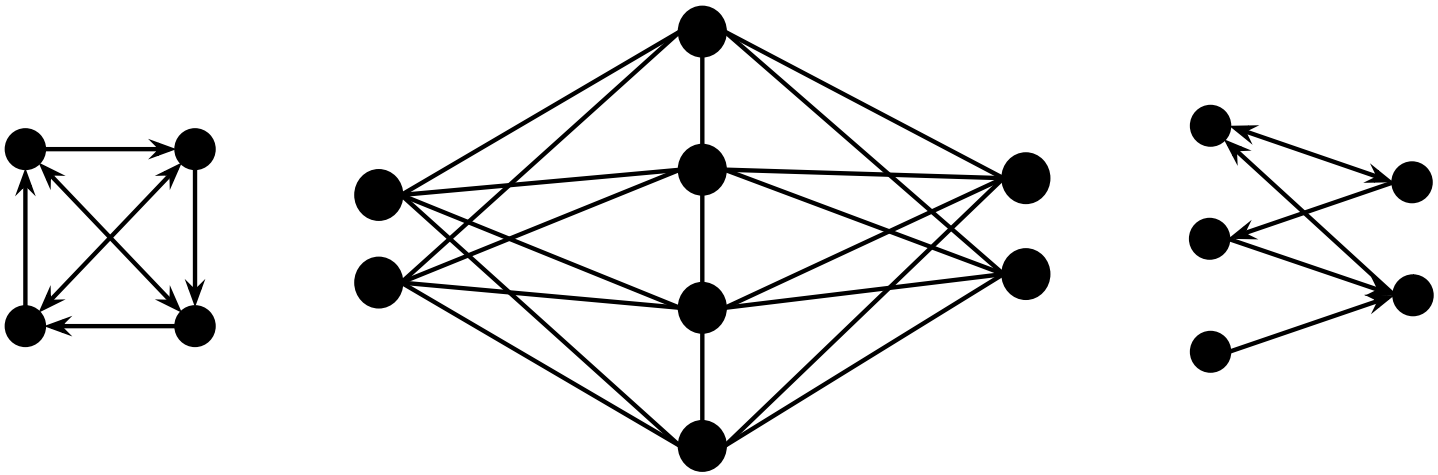
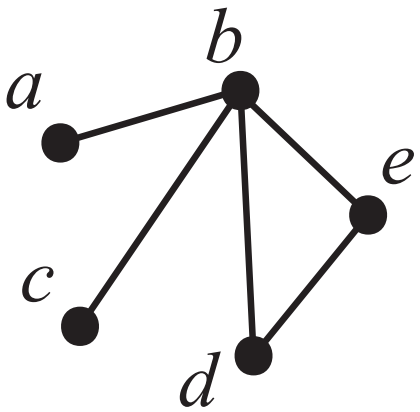
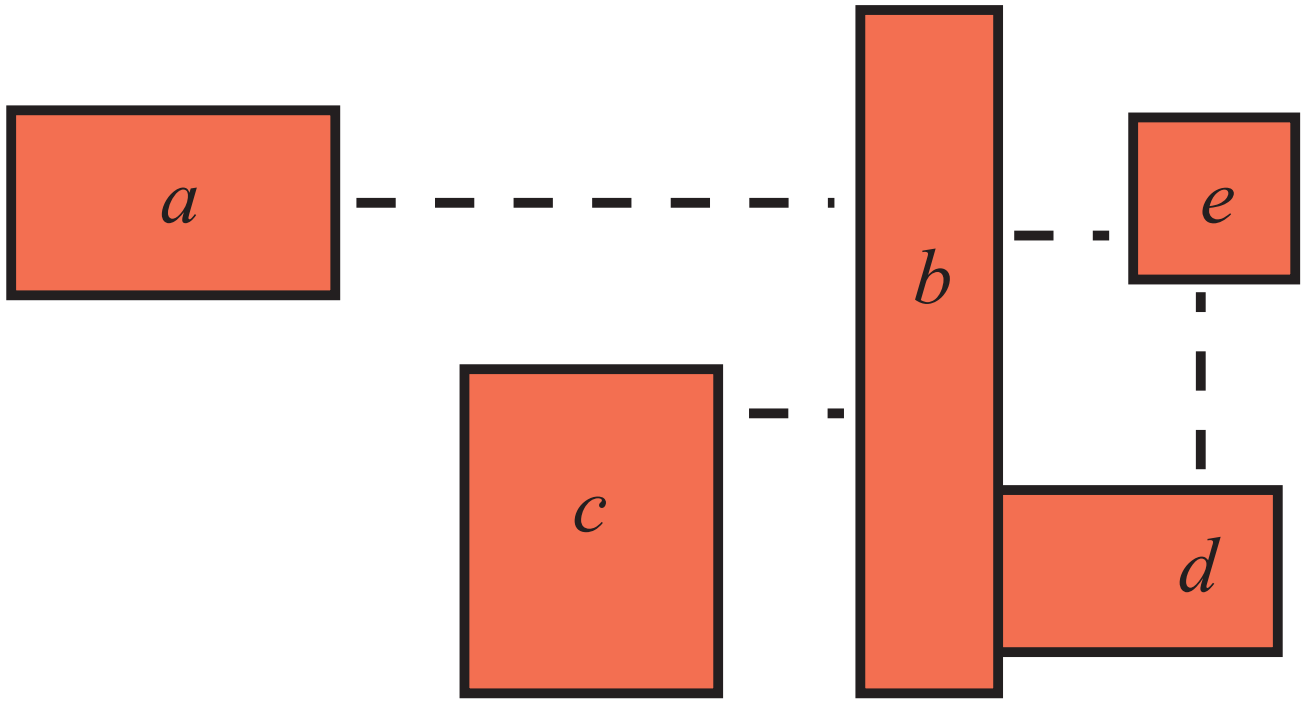


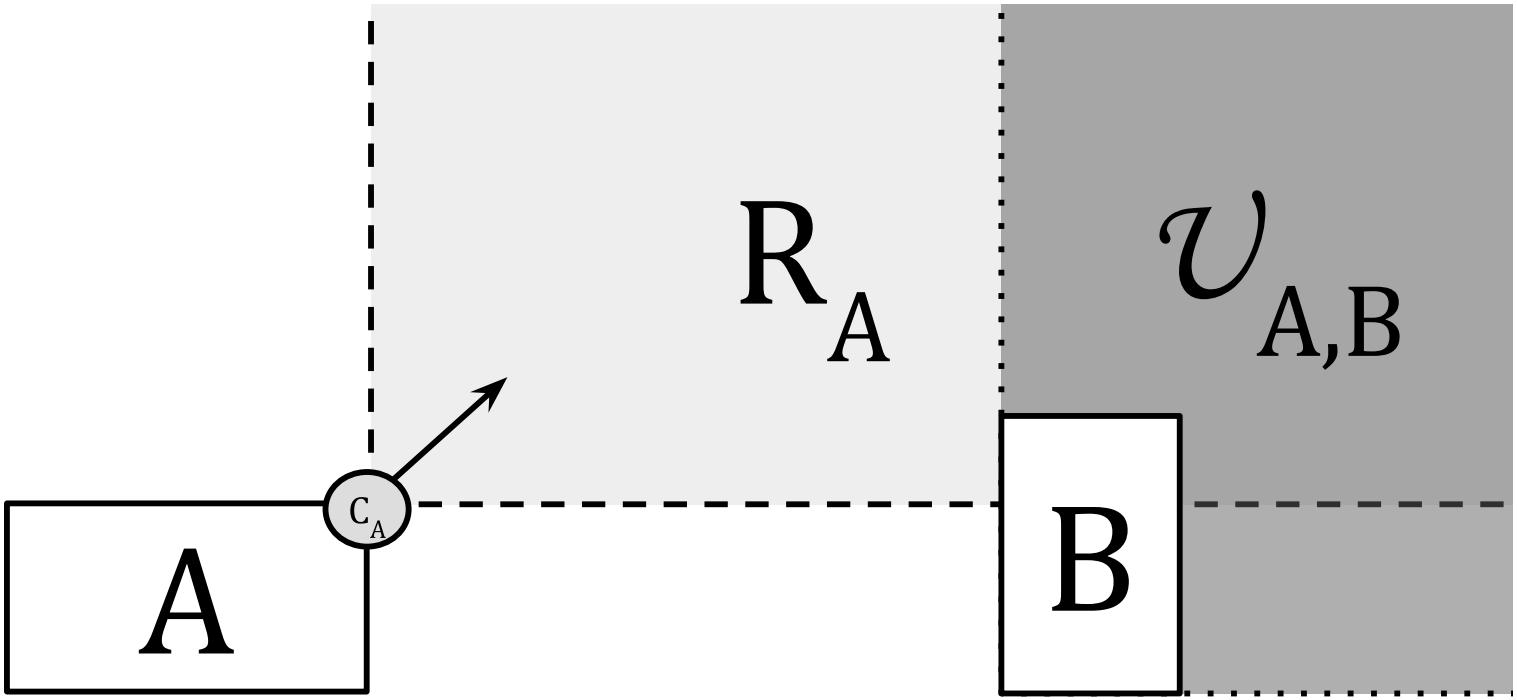
Figure: A complete graph (K_4), a double fan, and a bipartite graph

Background: Rectangle Visibility





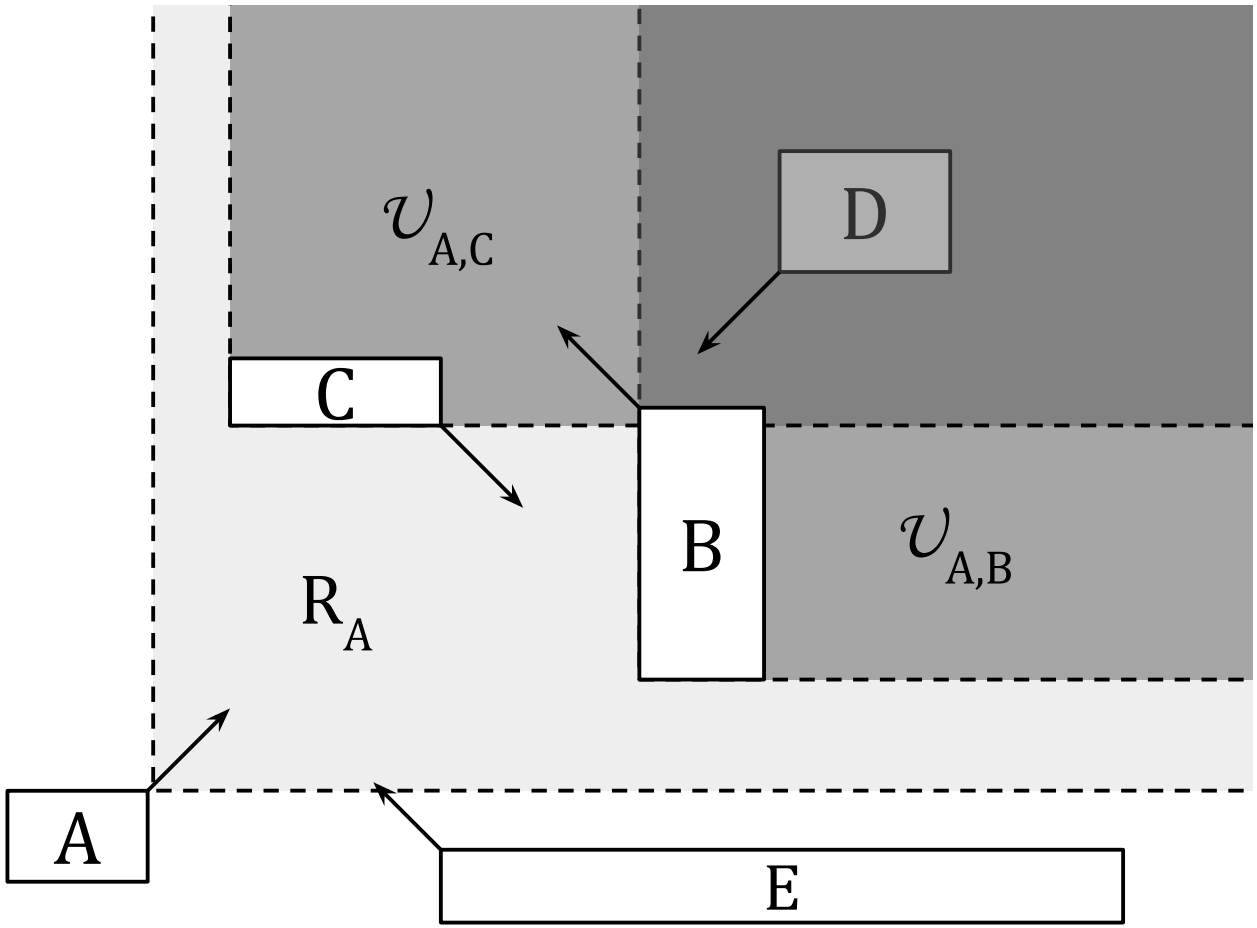
Definitions



A simple example where *A* sees *B*.

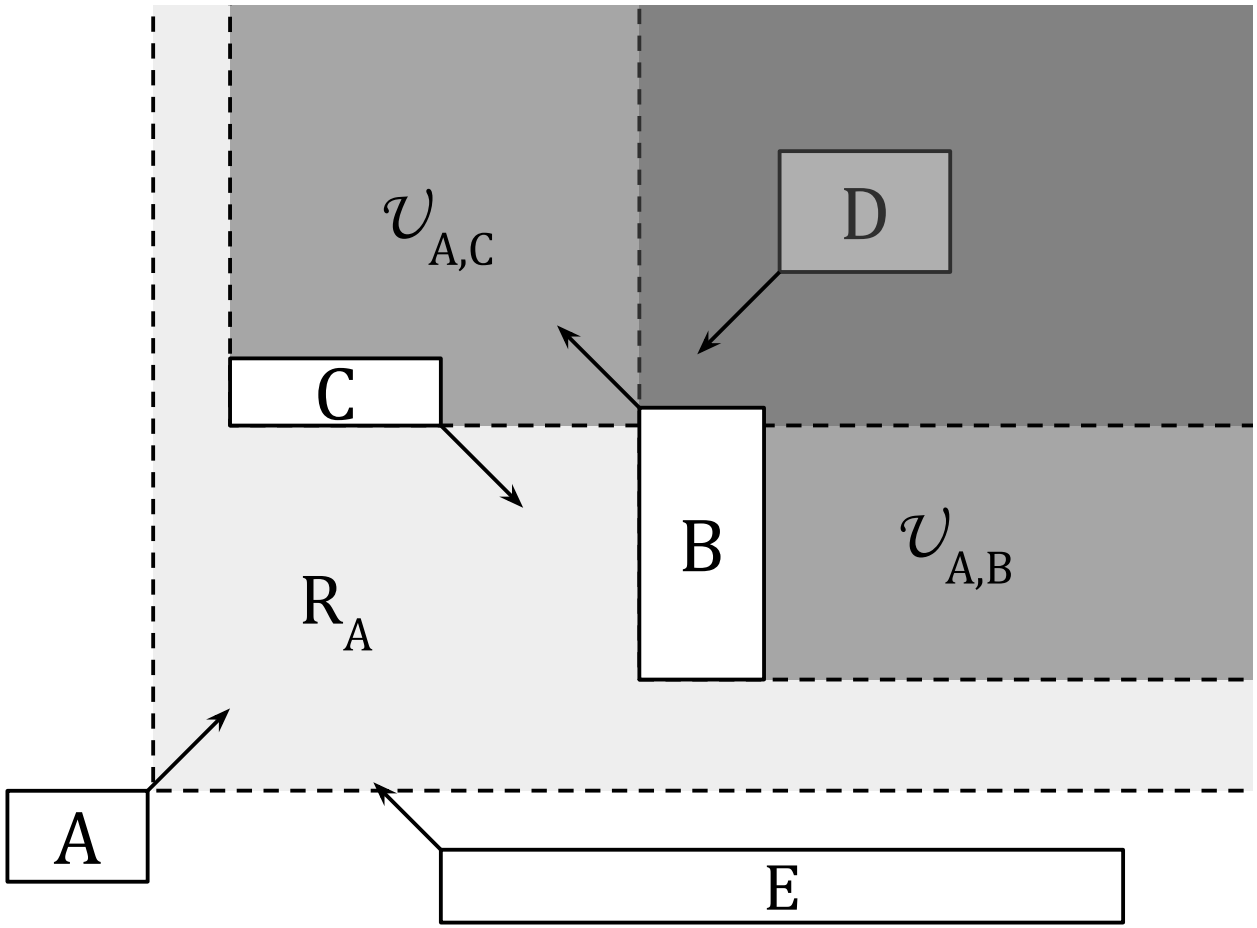
Example 1: $K_5 - 3E$

CRV representation

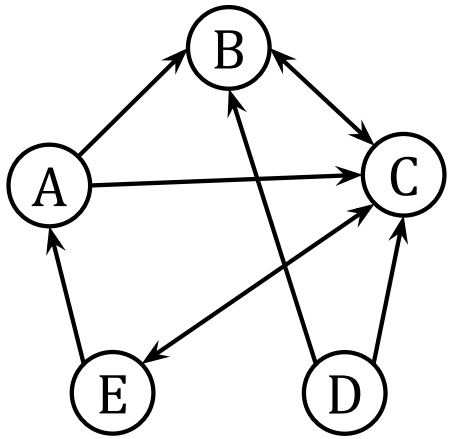


Example 1: $K_5 - 3E$

CRV representation



CRV graph



Example 2: K_4

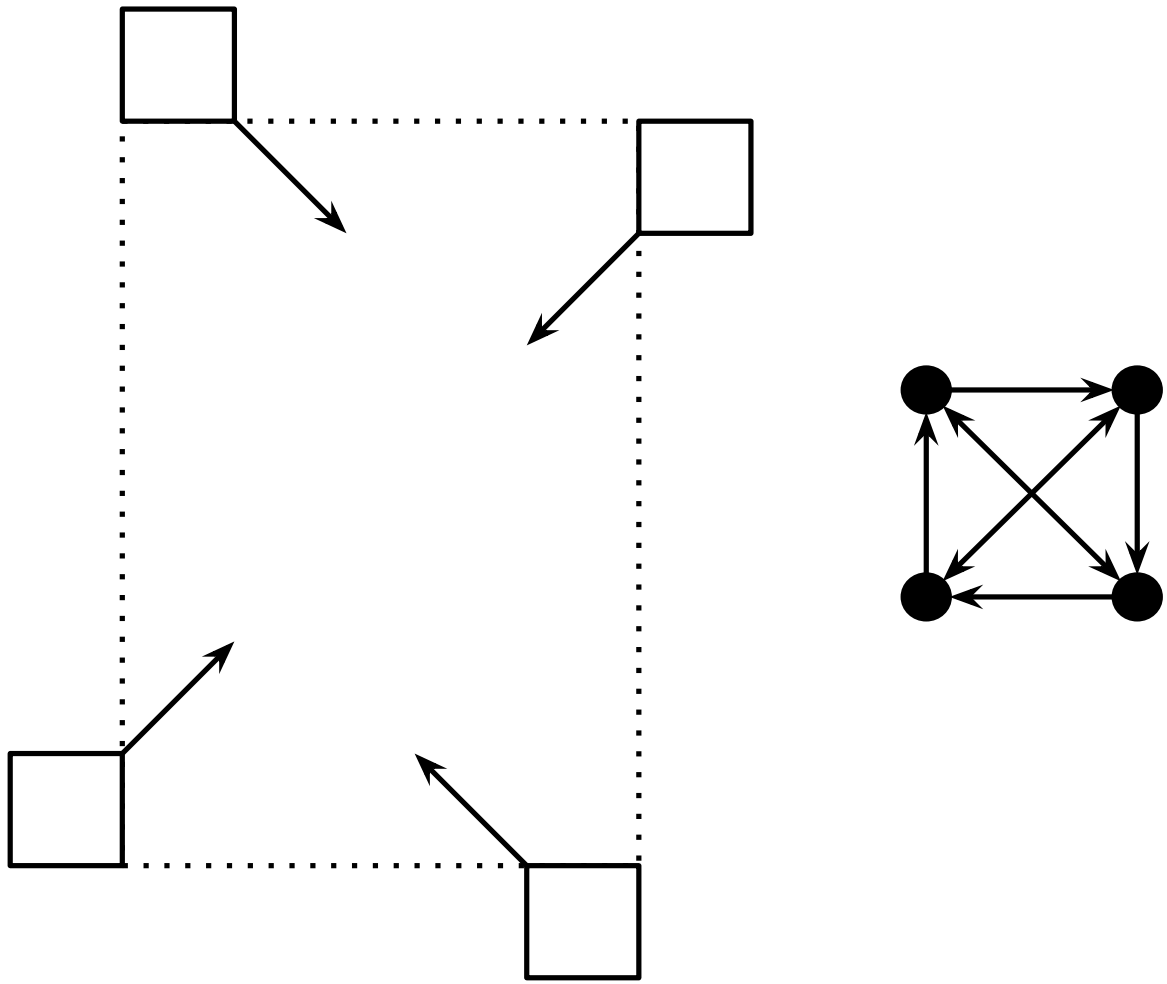


Figure: K_4, a Unit Cyclops Rectangle Visibility Graph

Collapsibility Theorem

Theorem

Any unit square (UCRVG) representation has the same visibility graph as the point representation made by collapsing each square into its upper right corner.

Coordinates

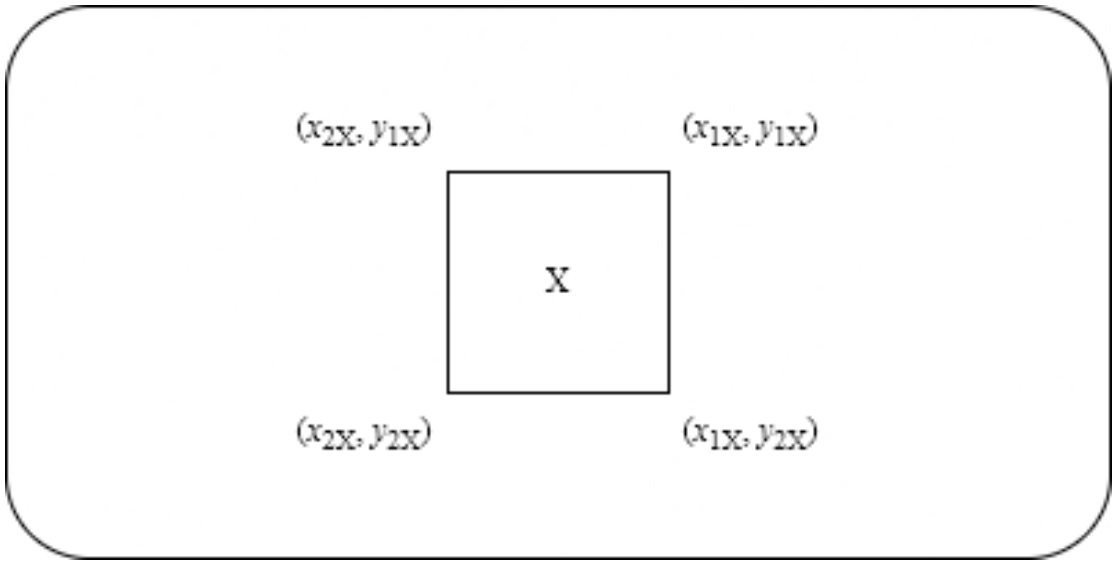


Figure: Squares, and diagram.

Collapsibility Proof

If A, B, and C are unit squares with C in $\mathcal{U}_{A,C}$ and A is looking southwest, then:
$x_{1B} \leq x_{1C} \Leftrightarrow x_2 B \leq x_{2C}$ and
$y_{1B} \leq y_{1C} \Leftrightarrow y_{2B} \leq y_{2C}$.
When collapsed, we have:
$x_{2B} \leq x_{2C}$ and
$y_{2B} \leq y_{2C}$ so B is still in the shadow of C.
By similar means, we can prove the other direction and likewise for both shadow and sight regions.
Intuitively, we can think of "zooming out" on the plane such that the unit squares become points.

Squares?

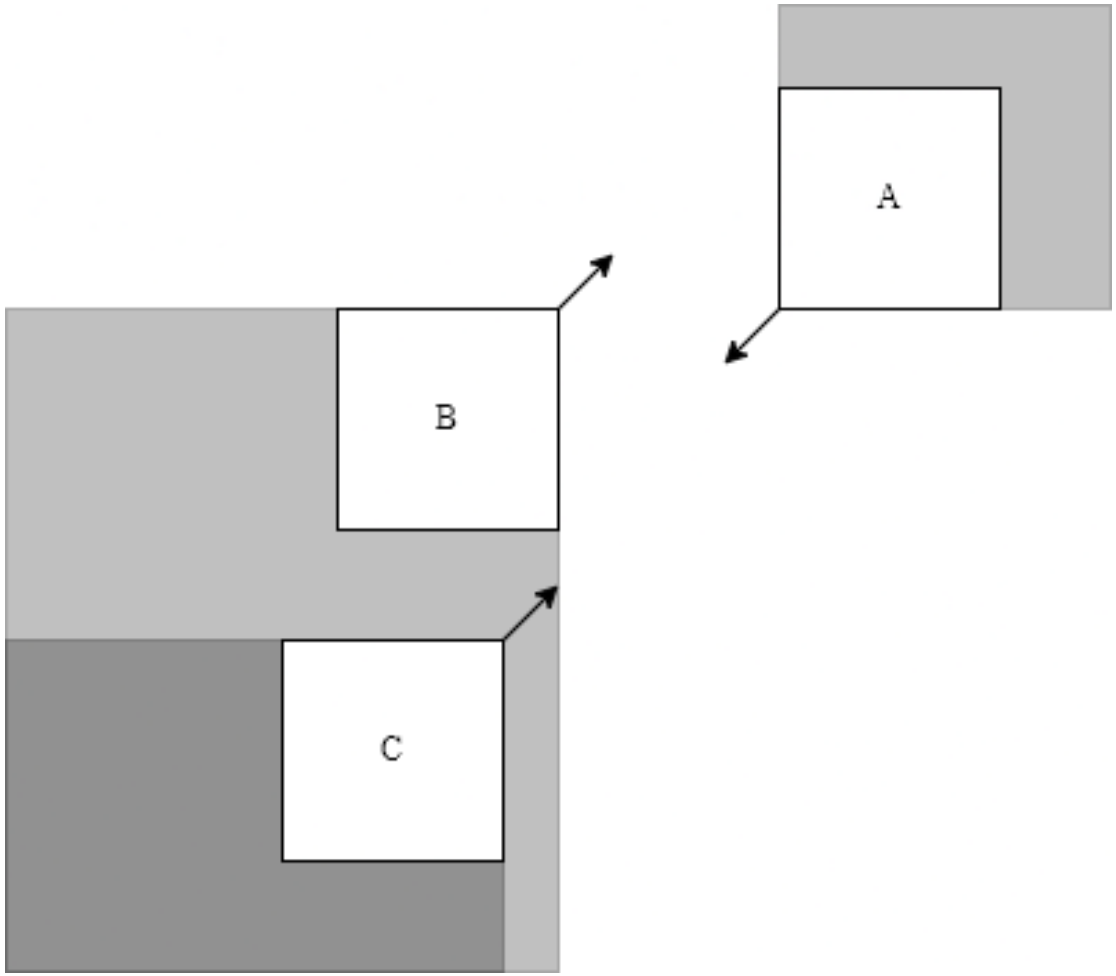


Figure: A UCRVG.

Points!

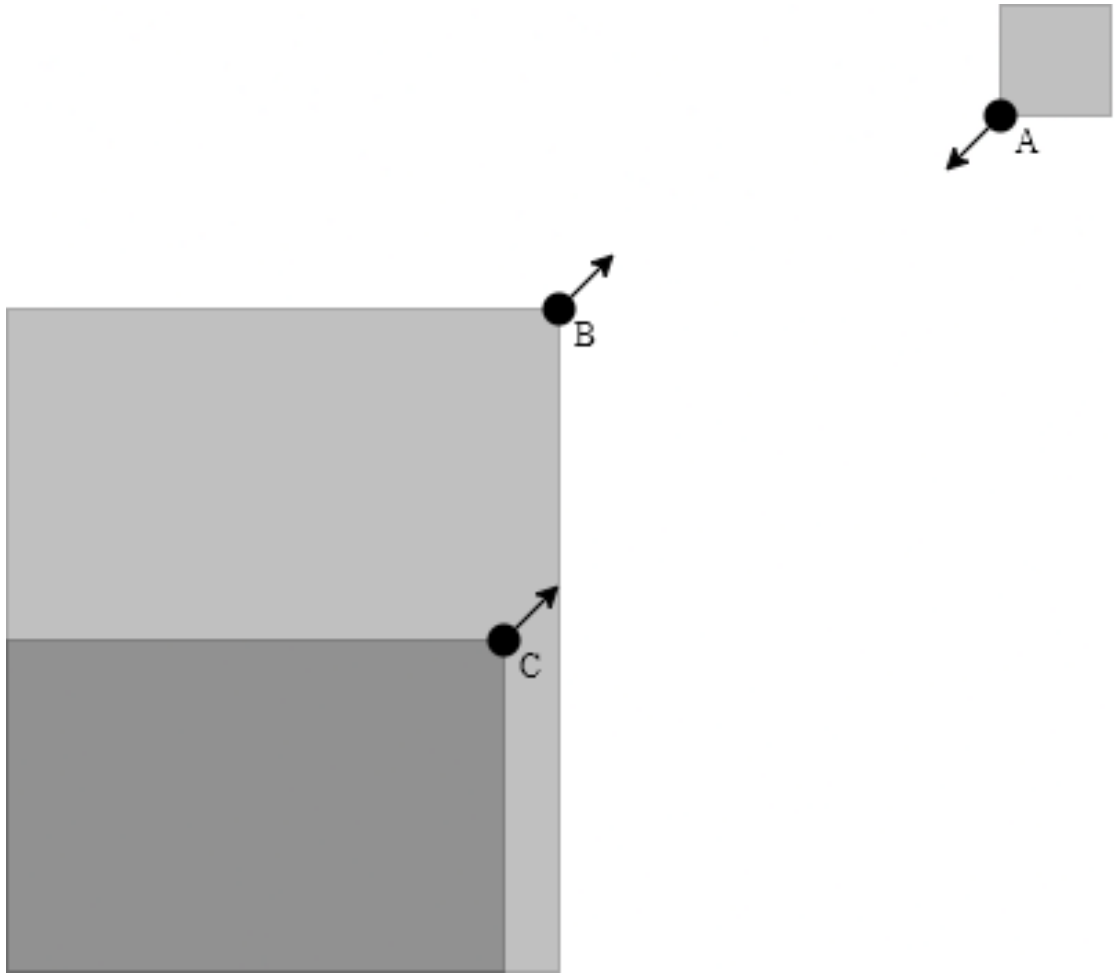


Figure: An equivalent representation of the same graph with only points.

Counterexample

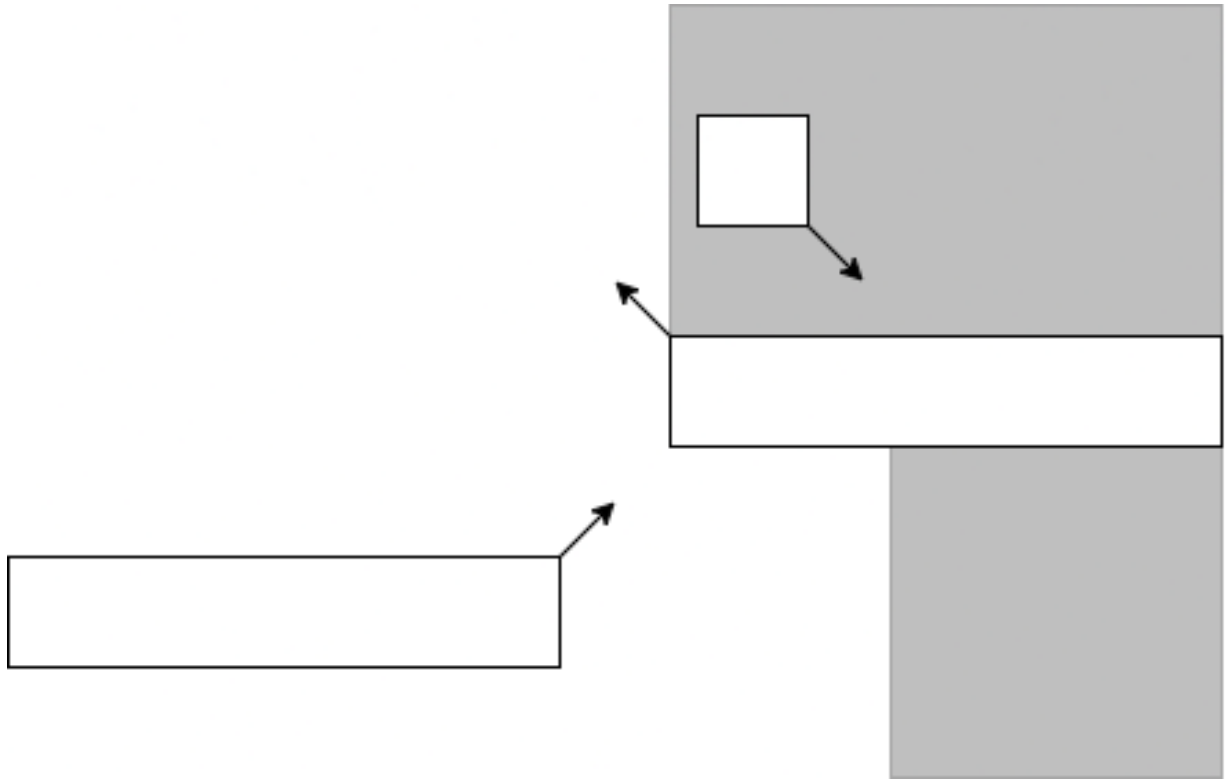
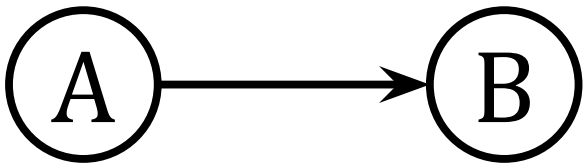
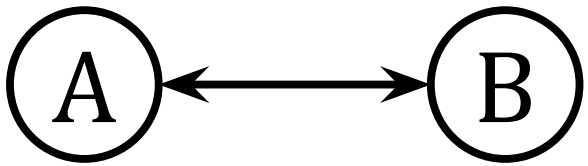


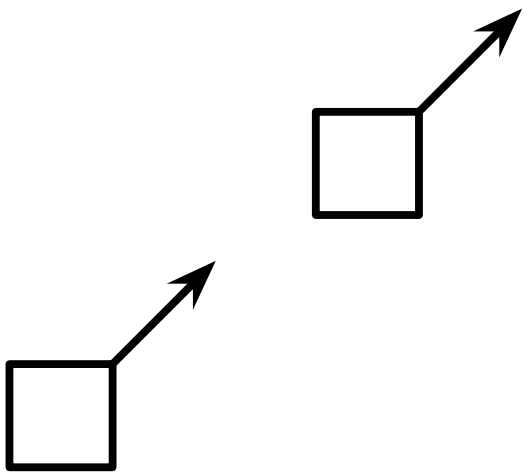
Figure: A CRVG where the theorem wouldn't work.

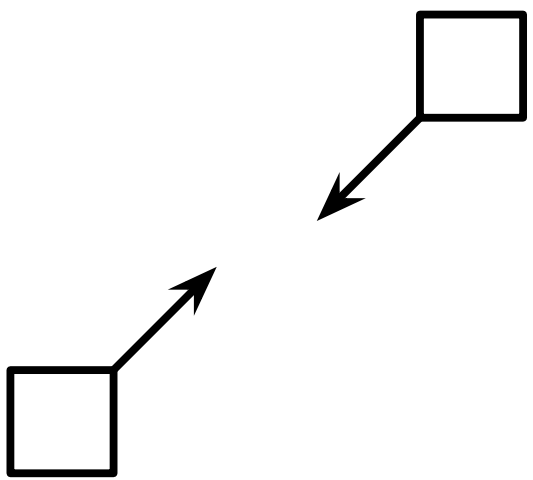
Complete Directed Graphs

What directed variants of complete graphs can we represent with CRV diagrams?



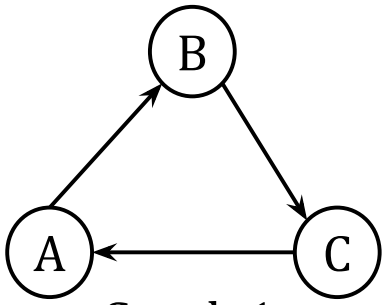




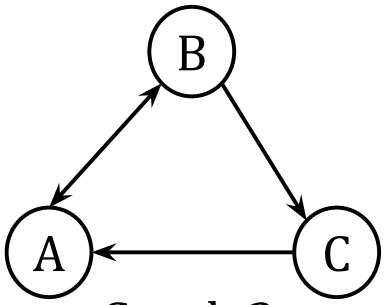


Complete Directed Graphs

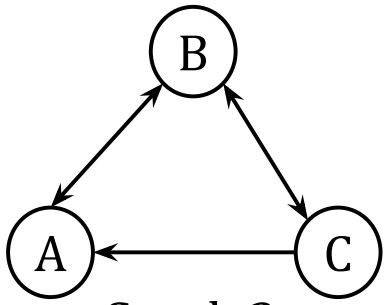
What directed variants of complete graphs can we represent with CRV diagrams?



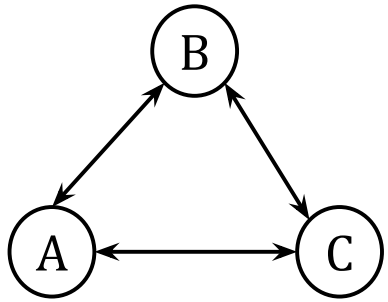
Graph 1.



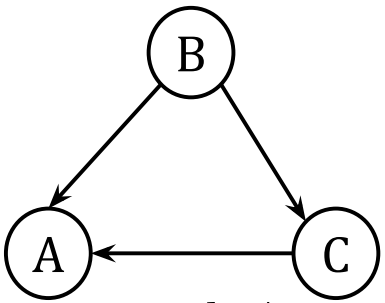
Graph 2.



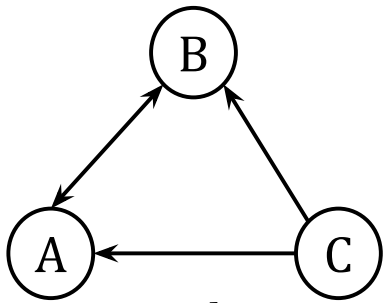
Graph 3.



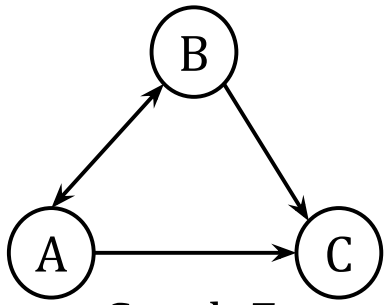
Graph 4.



Graph 5.



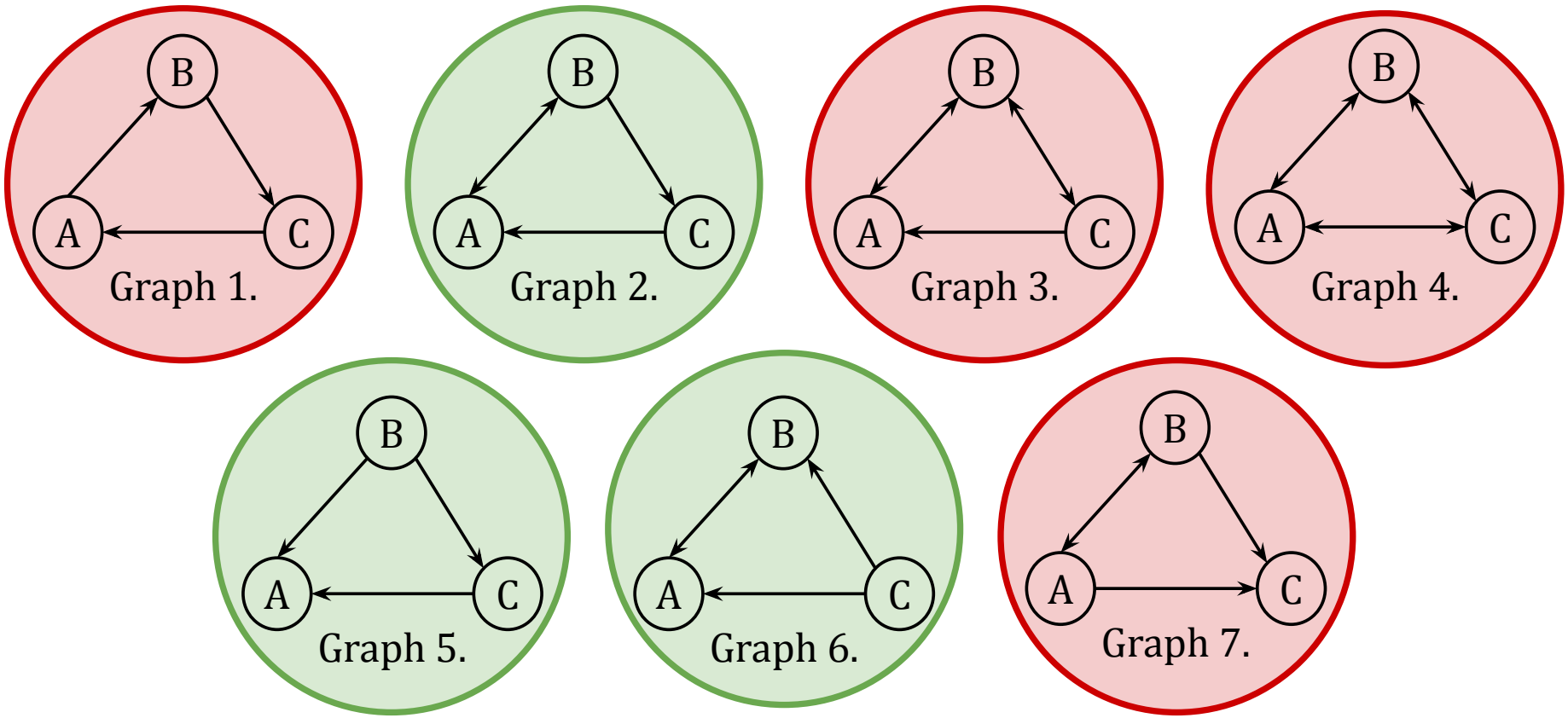
Graph 6.



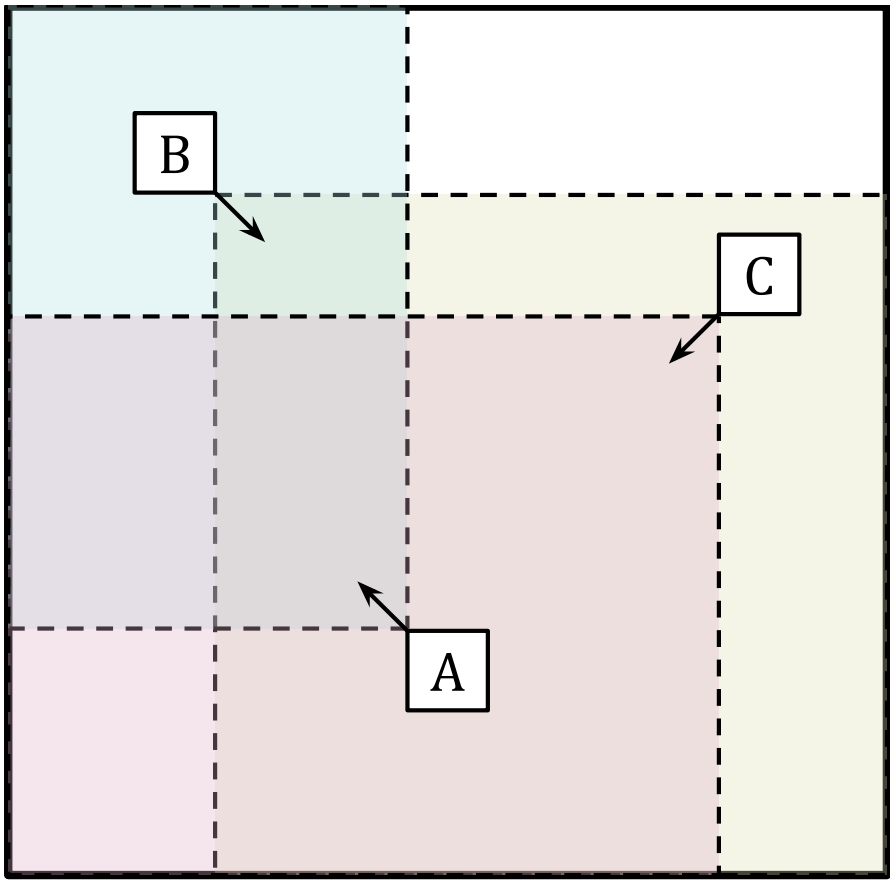
Graph 7.

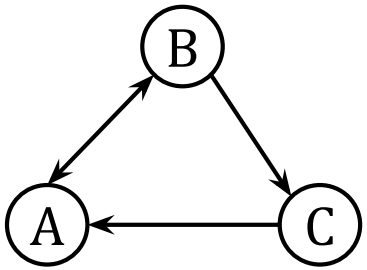
Directed K_3 Representations

Theorem: only the green graphs are representable.



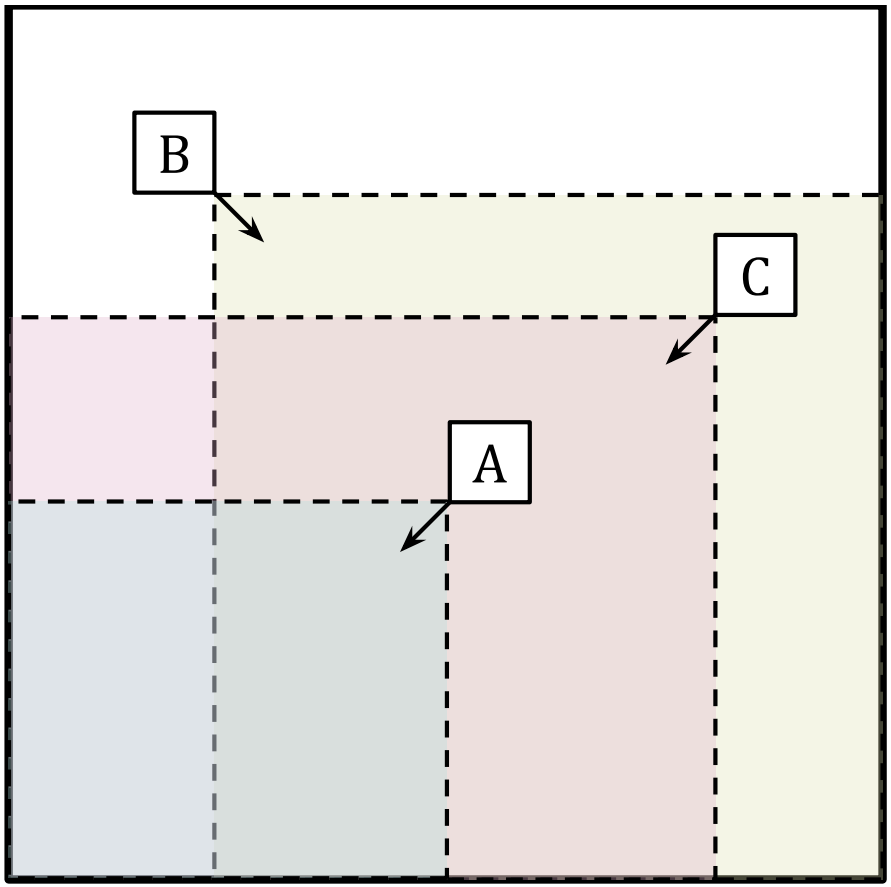
Directed K_3 Representations

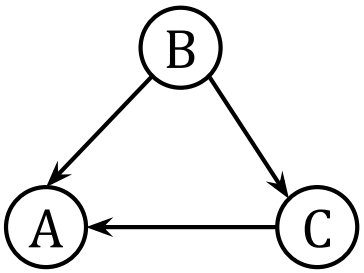




Graph 2.

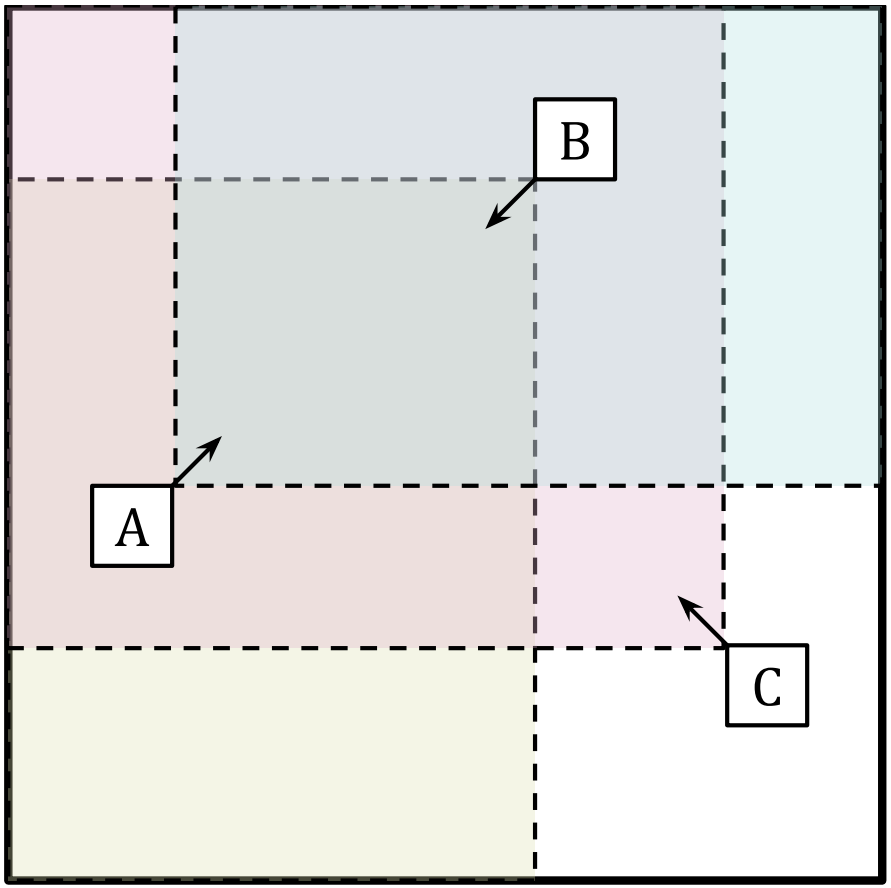
Directed K_3 Representations

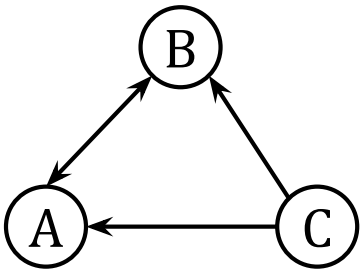




Graph 5.

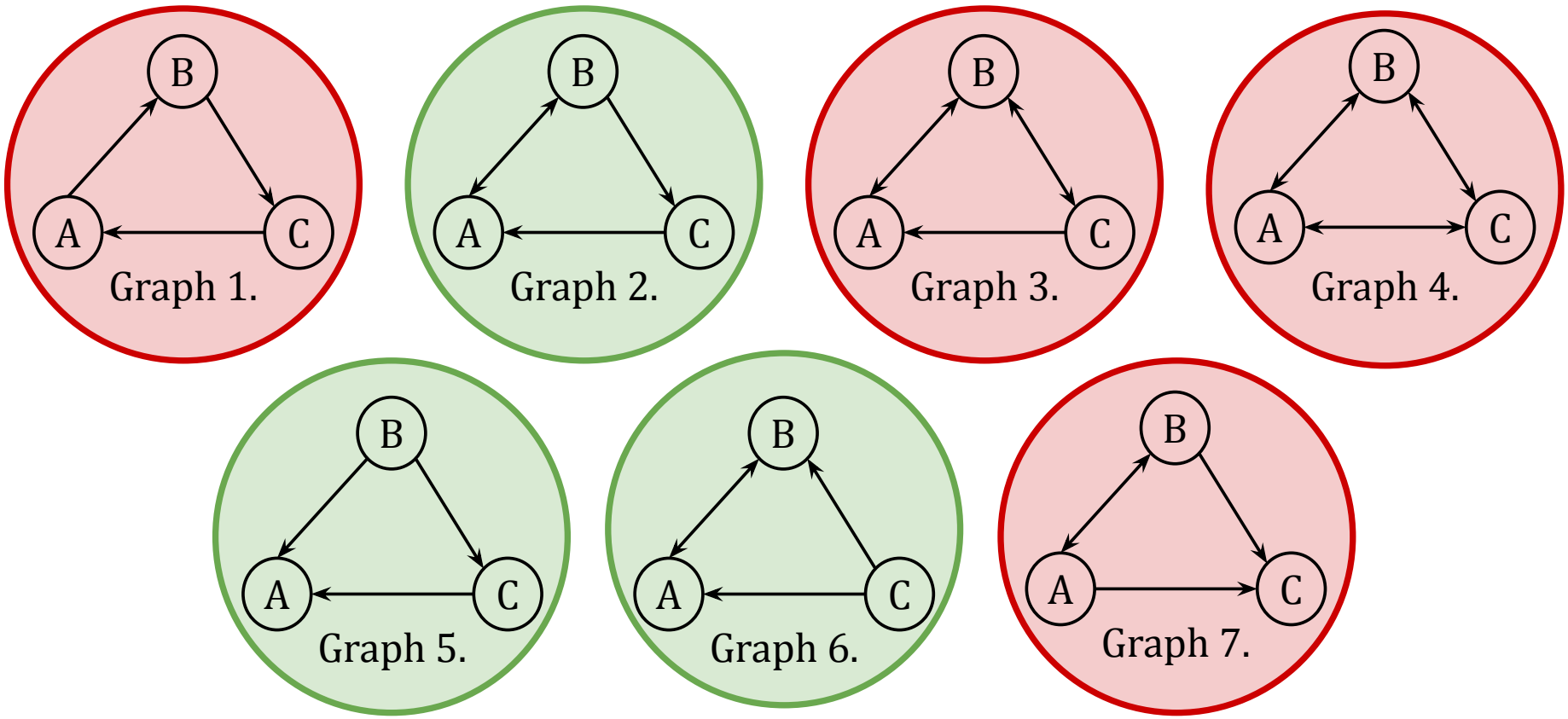
Directed K_3 Representations





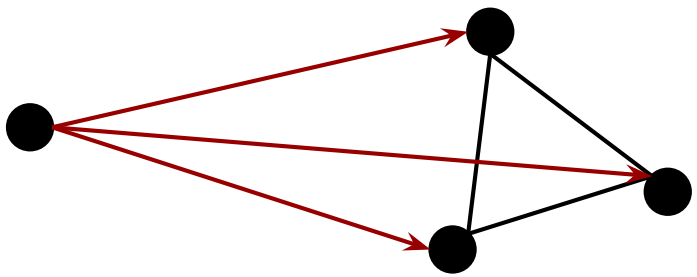
Graph 6.

Directed K_3 Representations

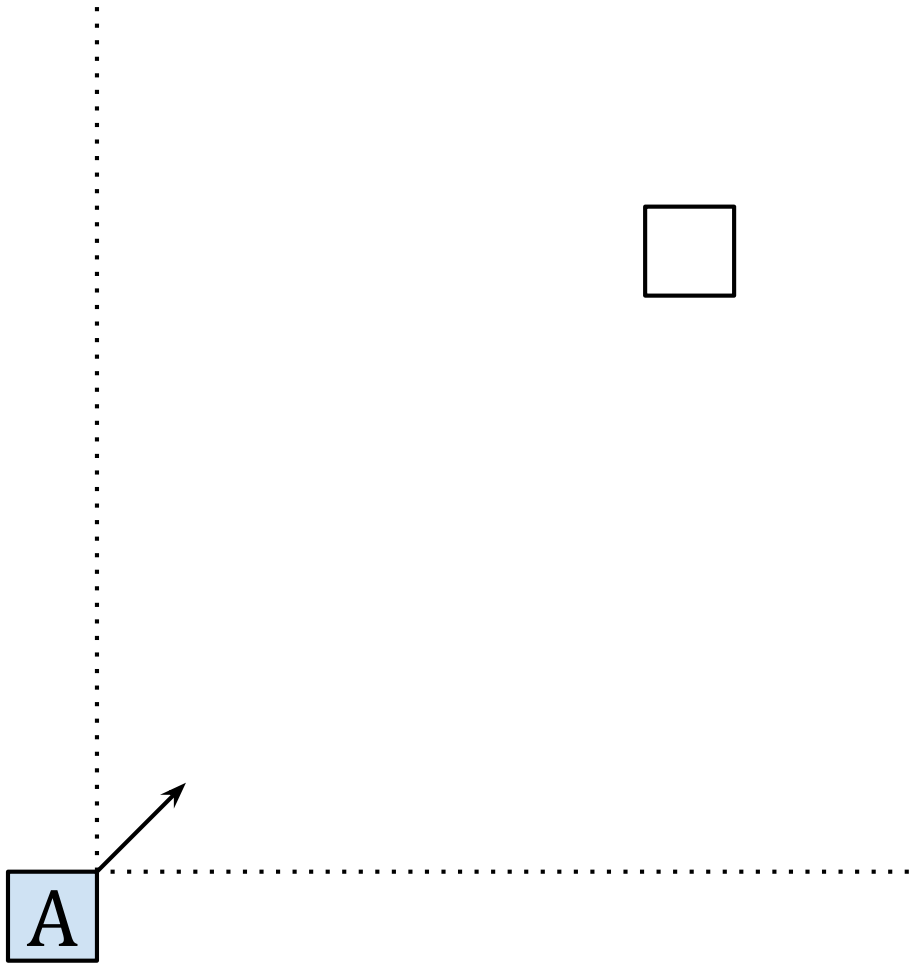


K_6 is not a UCRVG

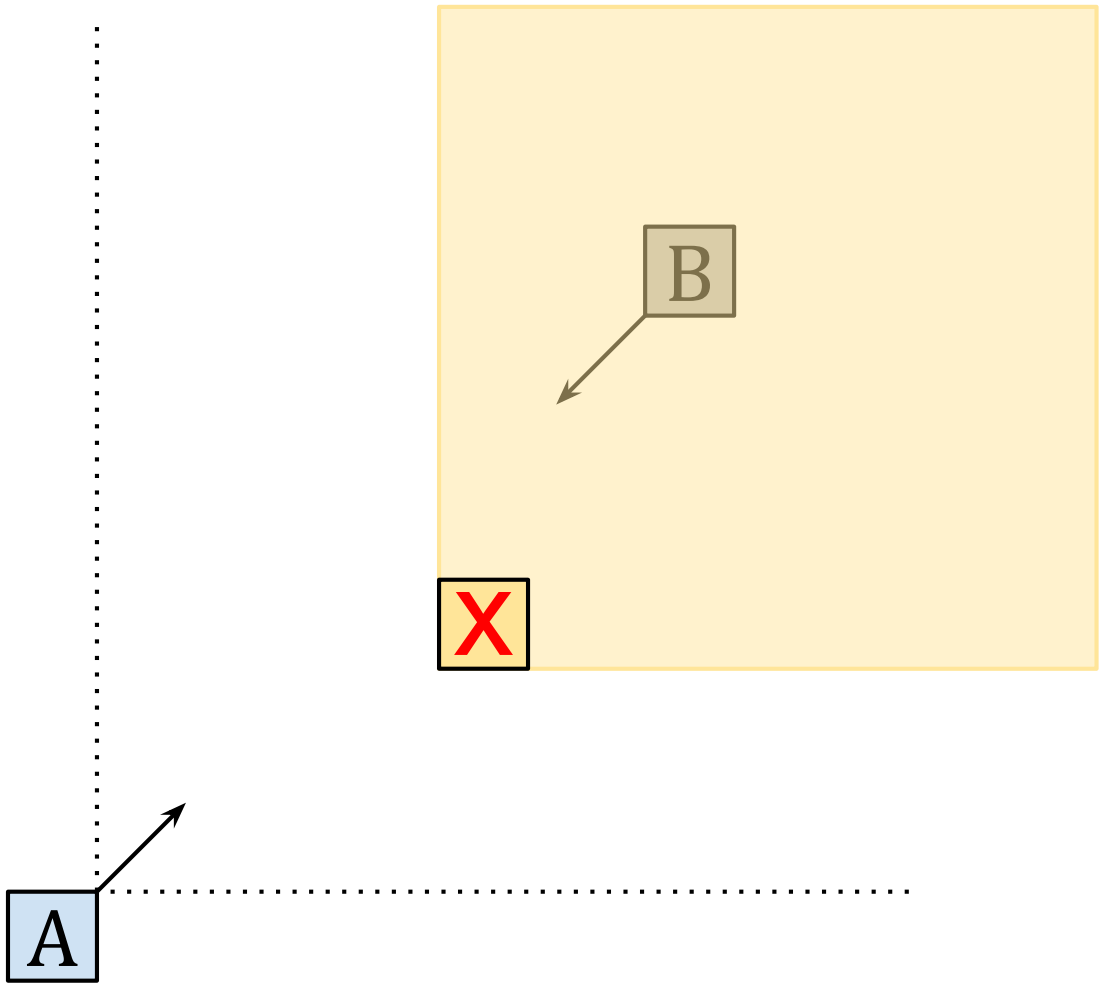
Lemma: A unit square cannot see all three unit squares of a triangle.



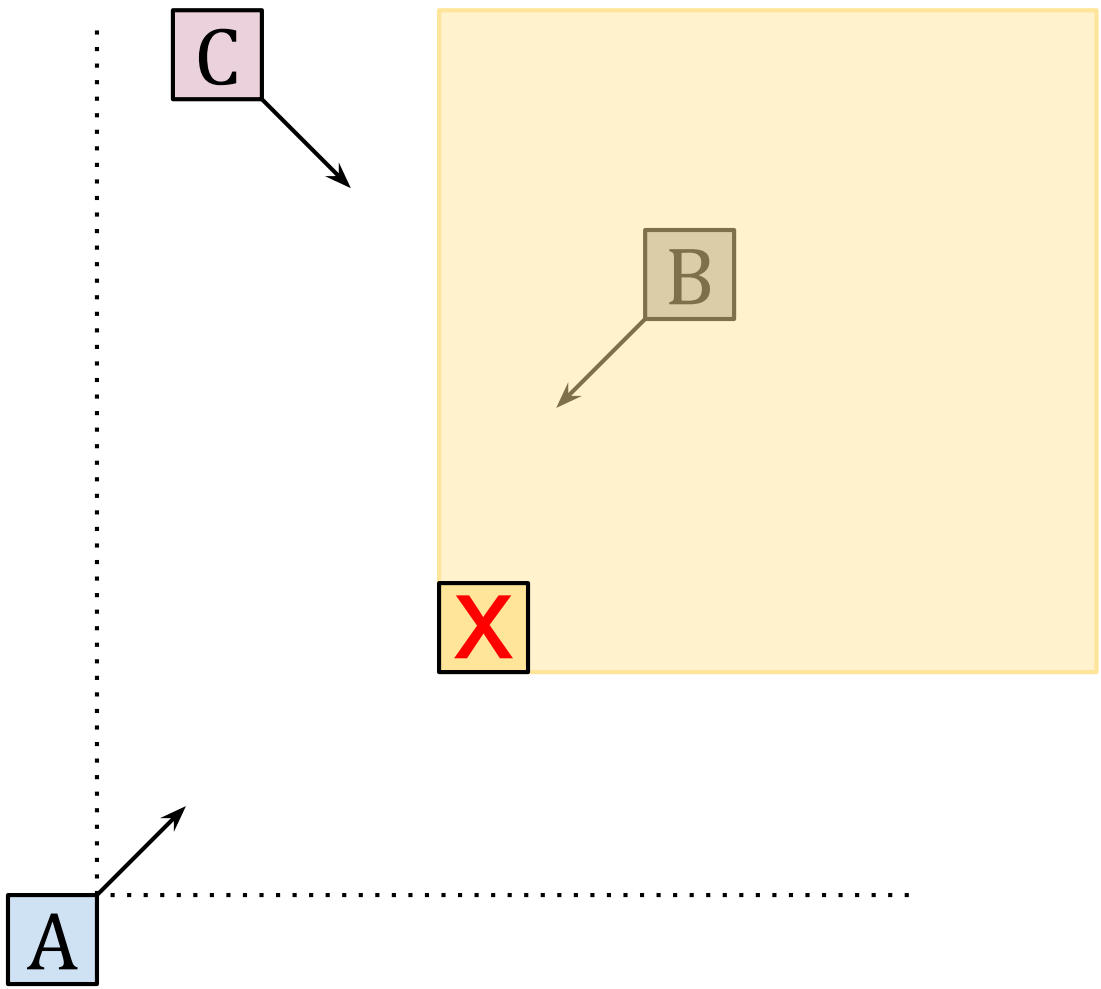
This graph is not representable with unit squares.



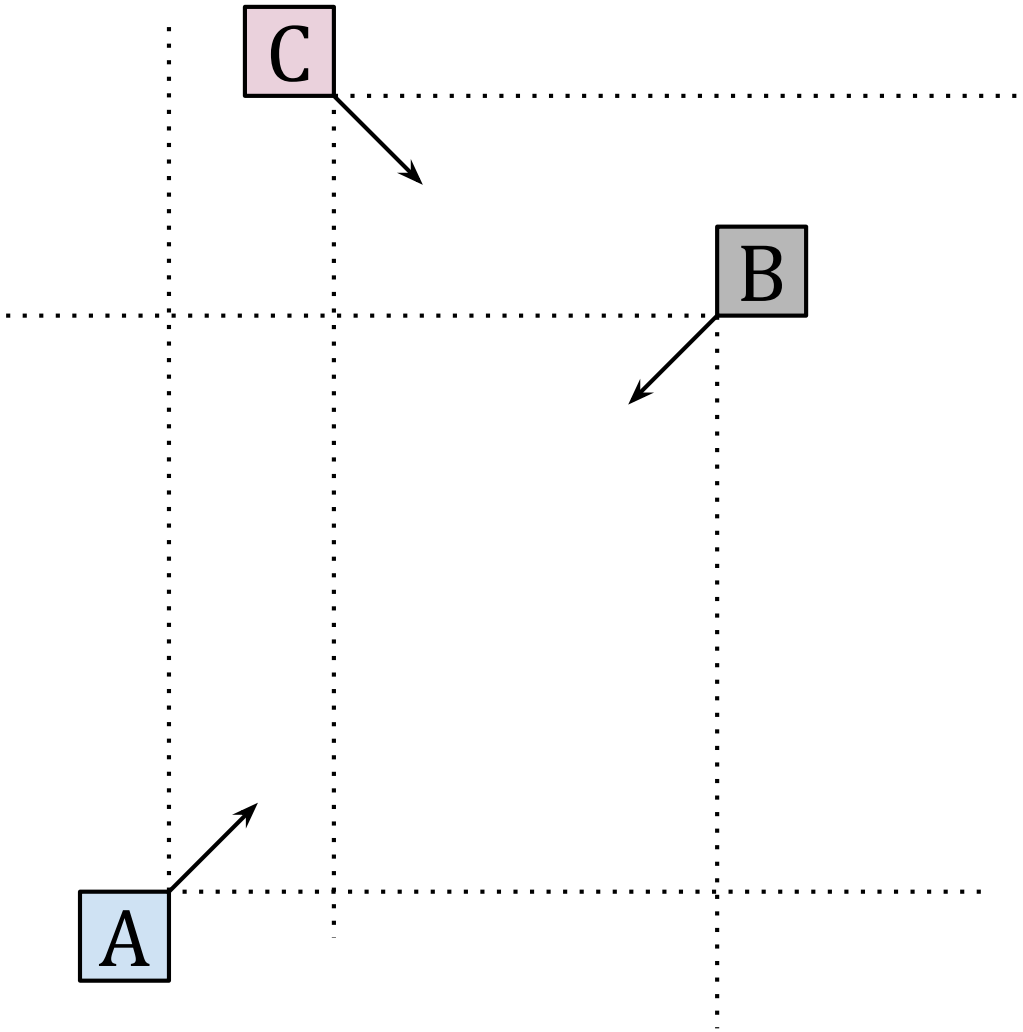
Without loss of generality, we can start like this.



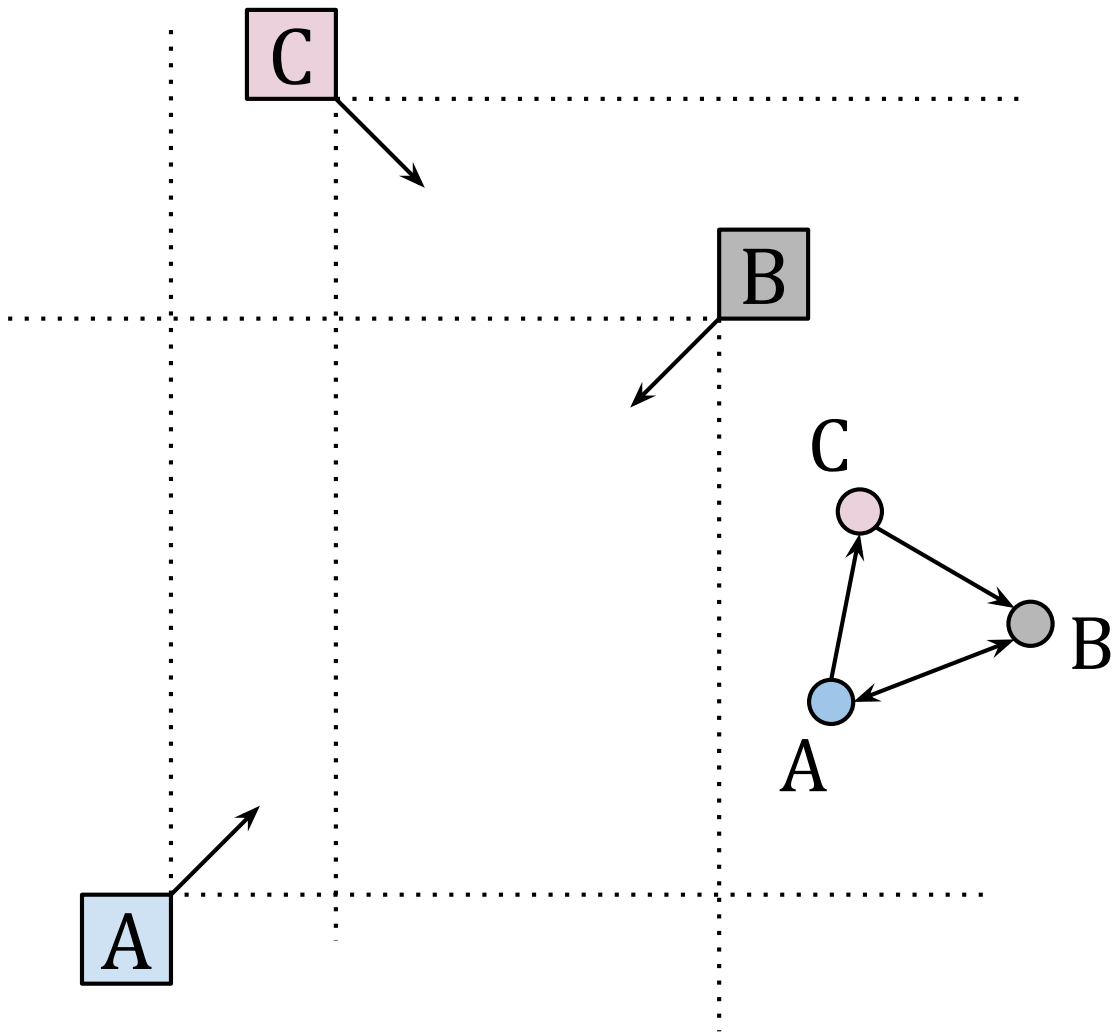
A square here casts a shadow that blocks sight between A and B.



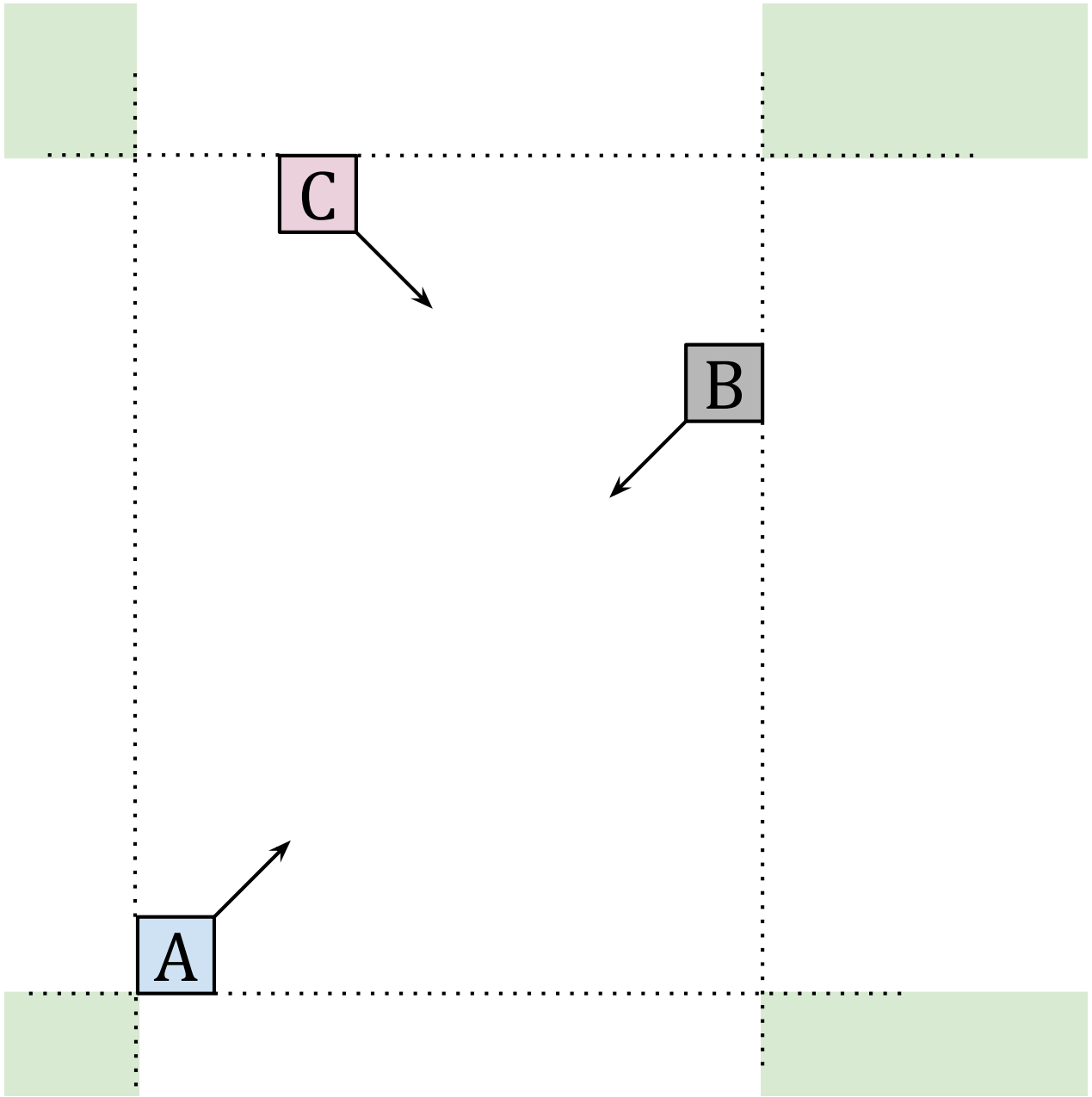
We can put a square like this.



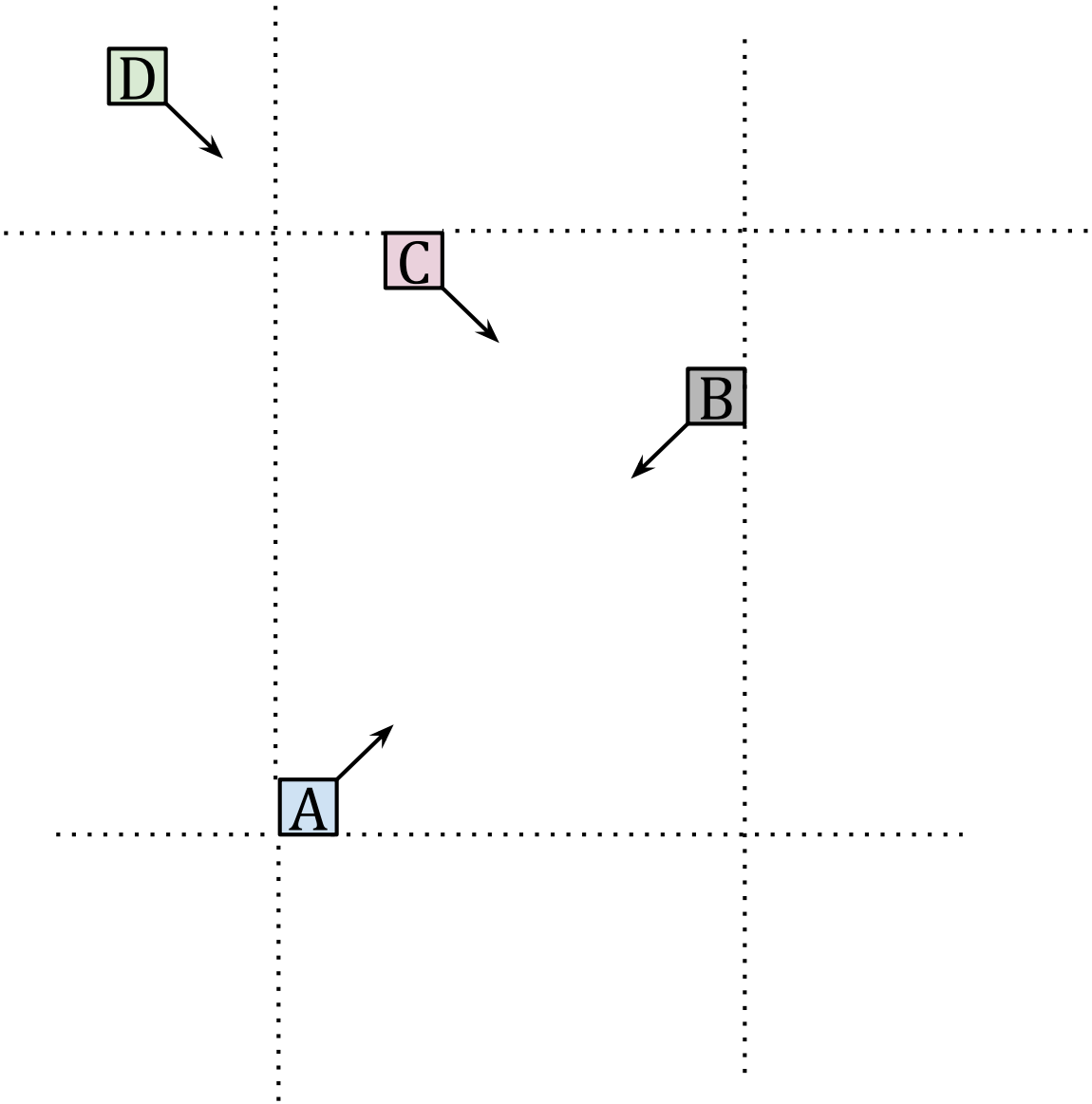
Now we have a triangle.



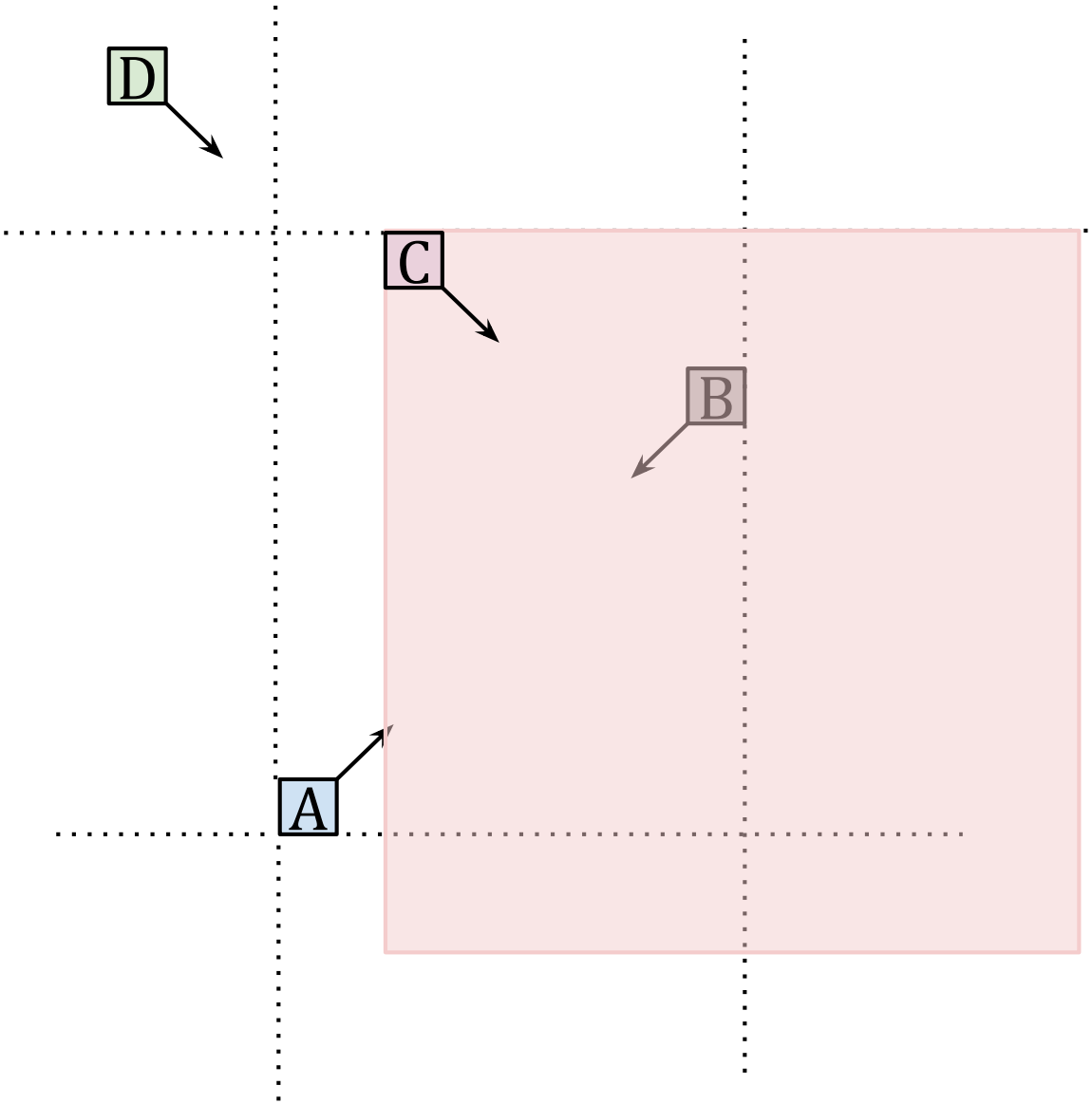
Here is the directed graph.



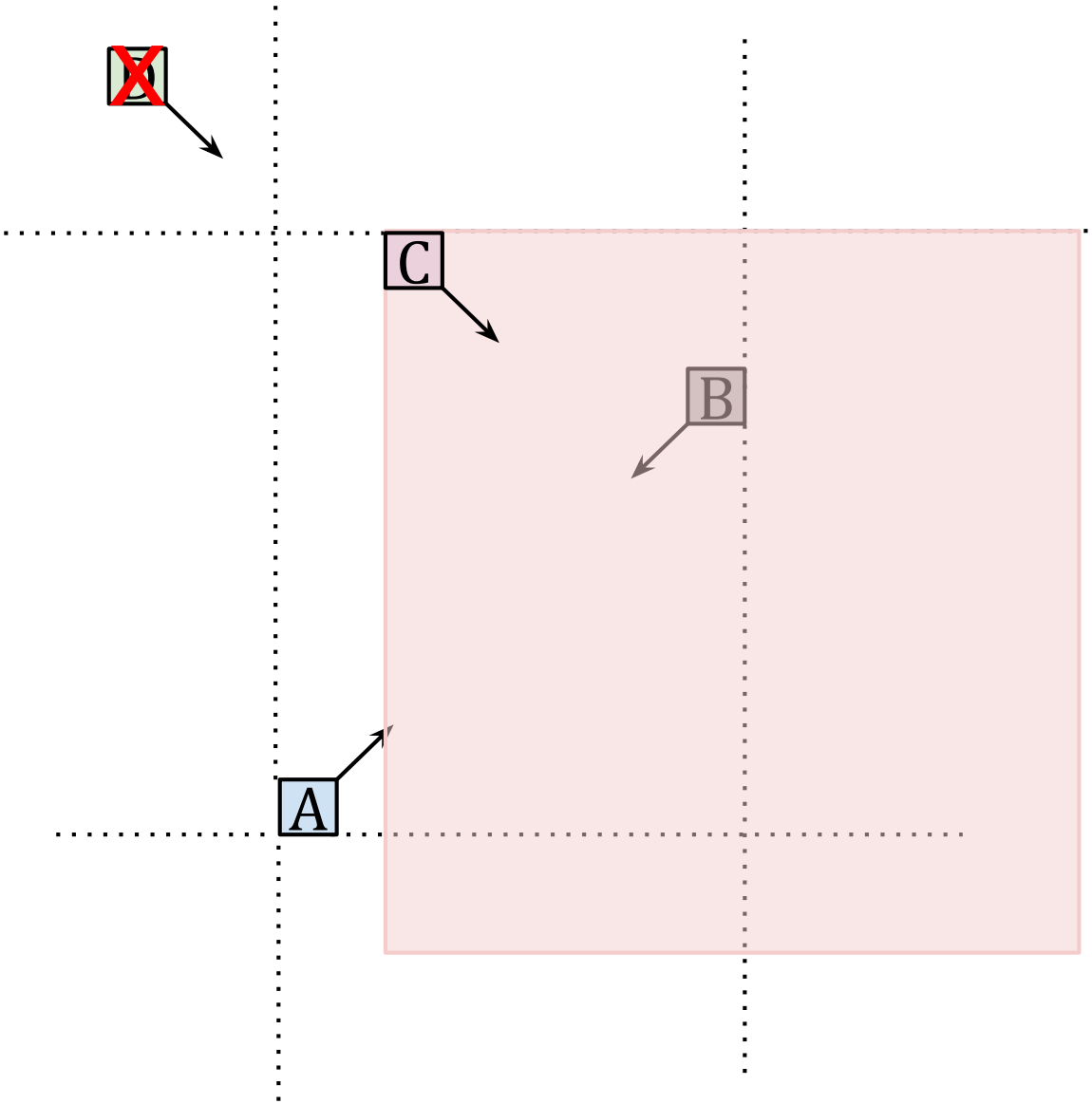
A fourth square must go in one of the green areas to see A, B, and C.



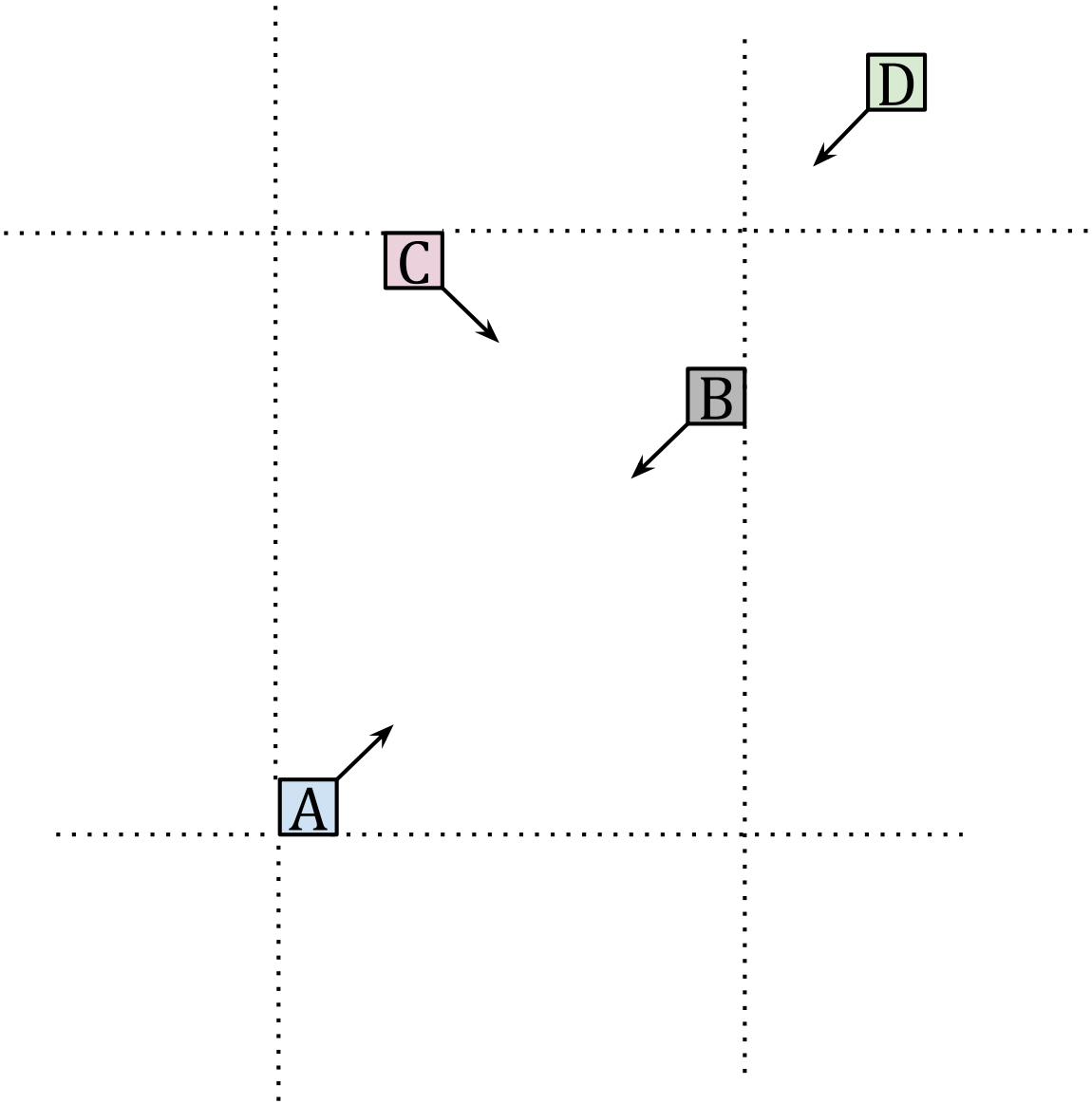
Here?



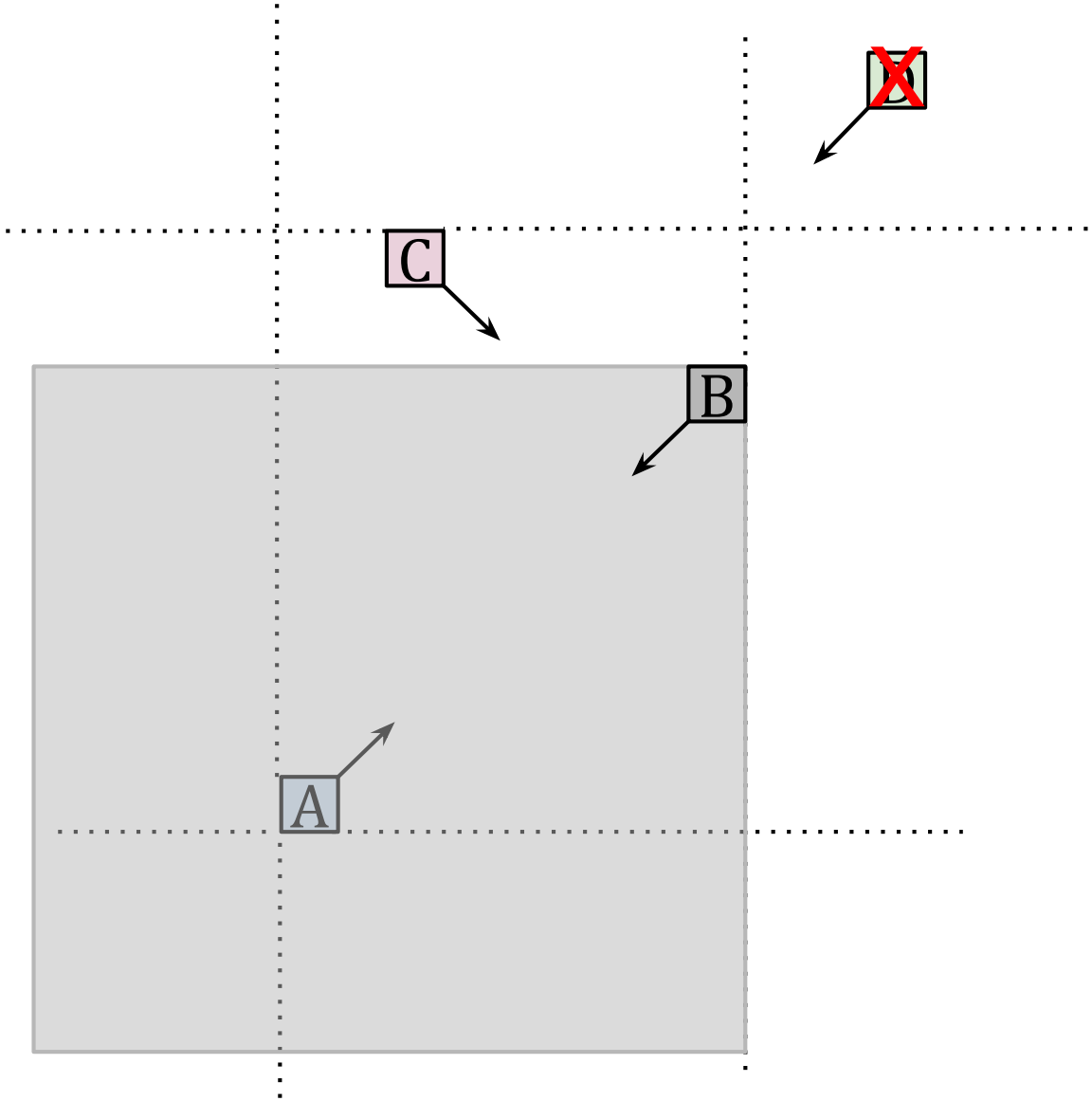
Shadow.



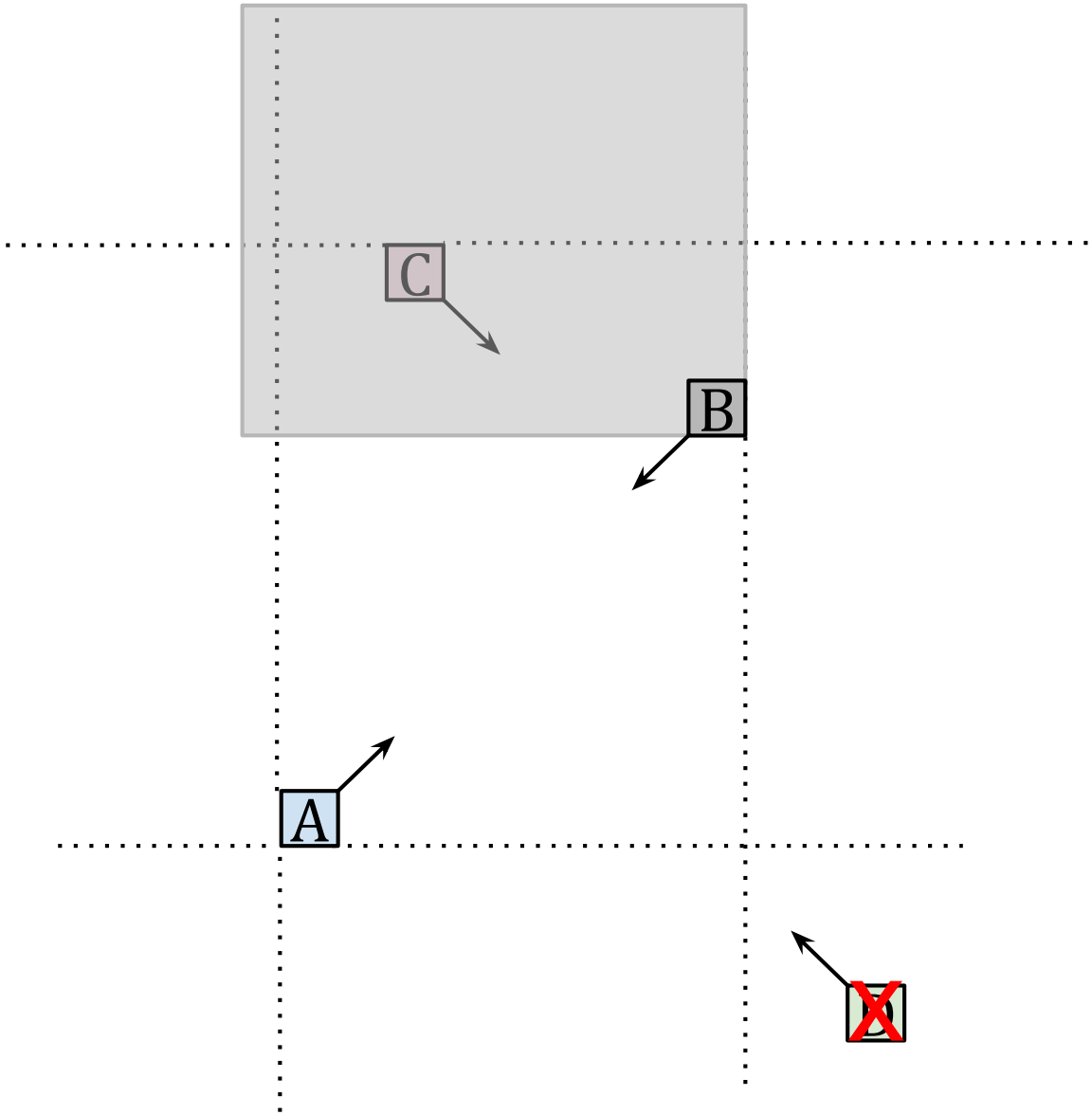
No.



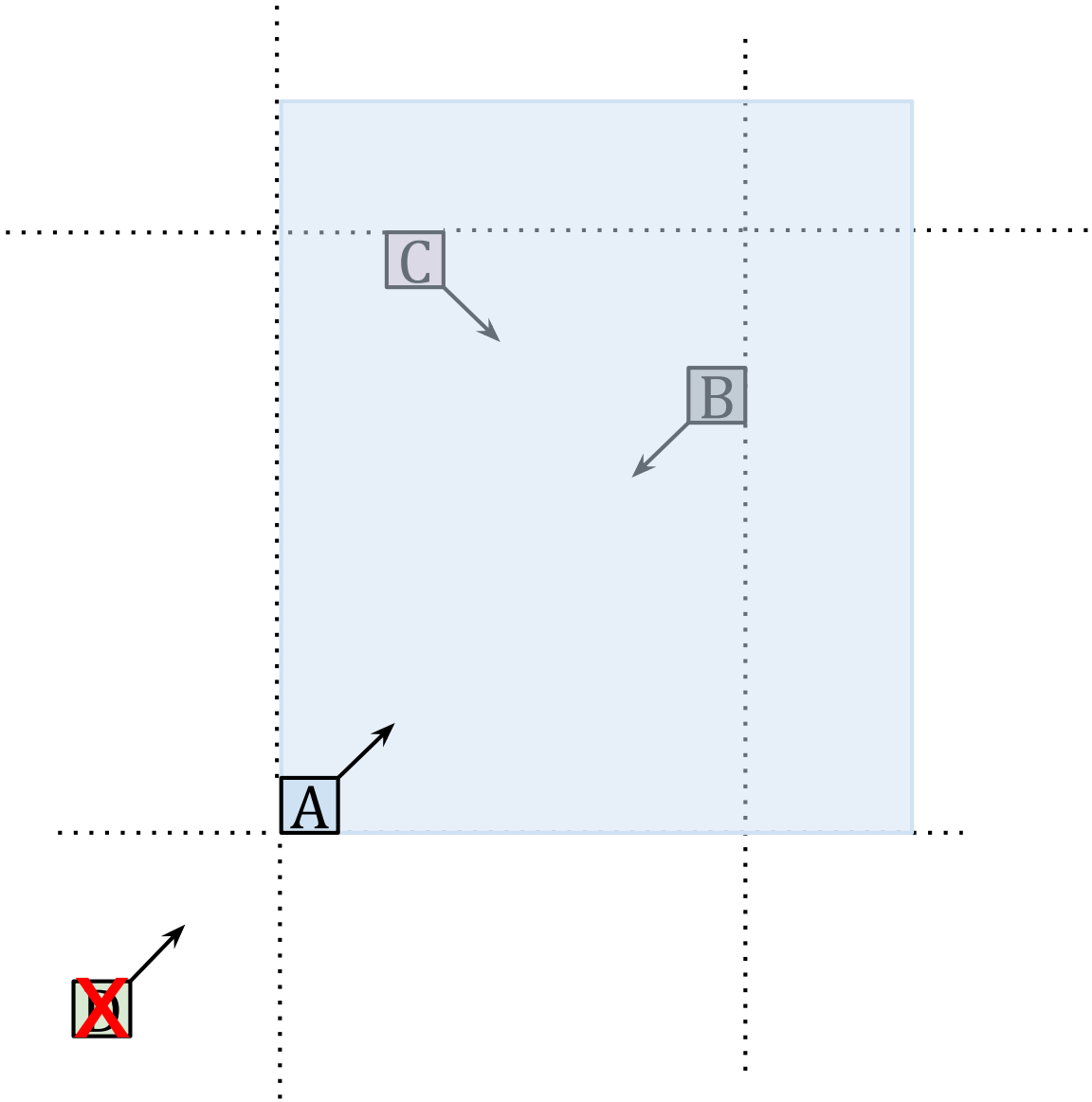
Here?



No.



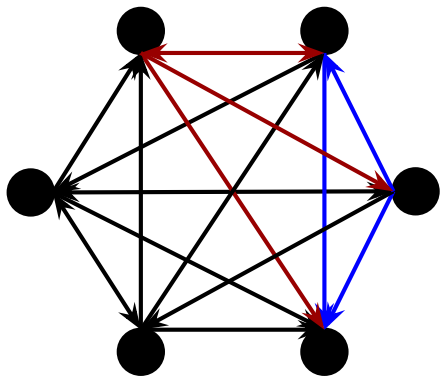
Not here.



Or here.

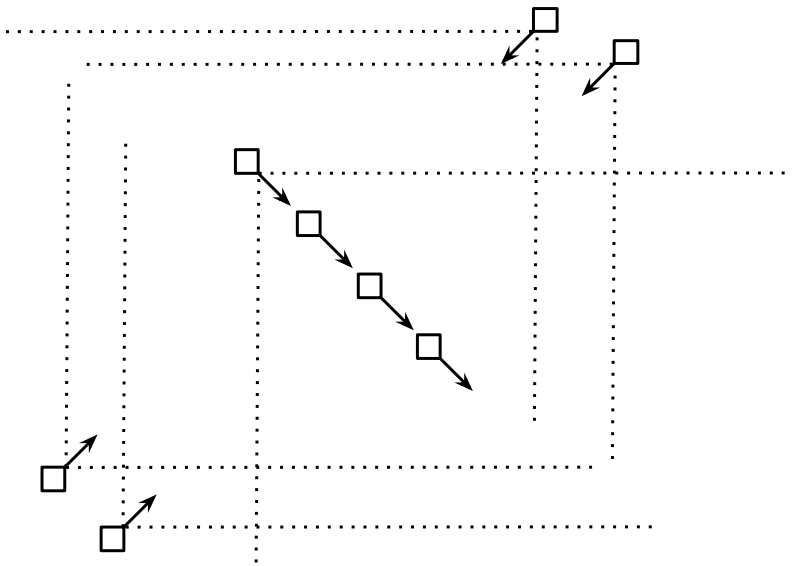
K_6 is not a UCRVG

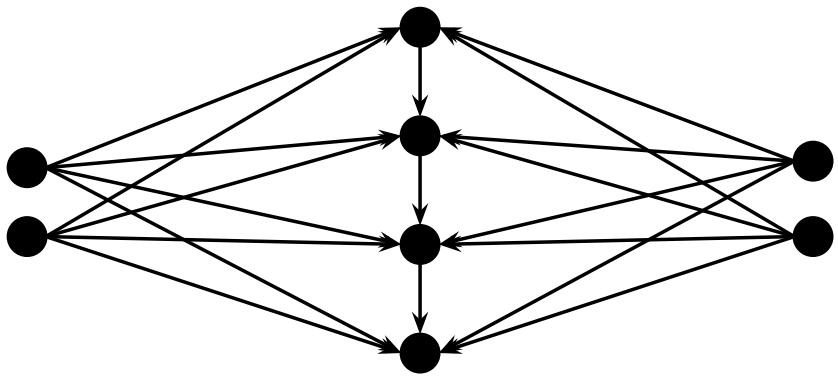
Theorem: K_6 is not a UCRVG.
Proof:
All directed K_6 graphs have a vertex with a directed edge outward toward the three vertices of a triangle. This is and impossible situation to represent with unit squares. Thus K6 is not a UCRVG. ■



Attempts to Maximize Edges

Question: What is the biggest number of edges we can get on a CRVG?
For UCRVGs: Double Fan

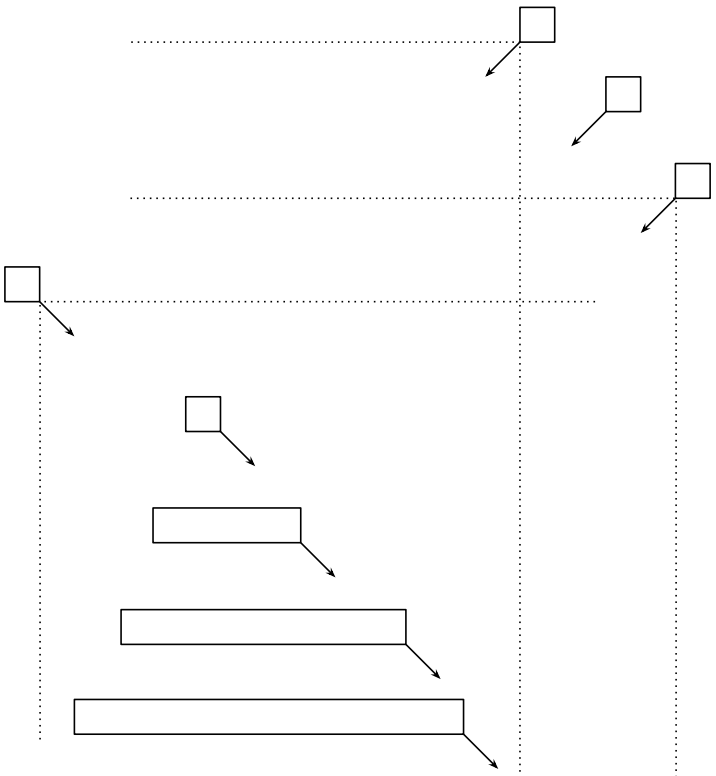


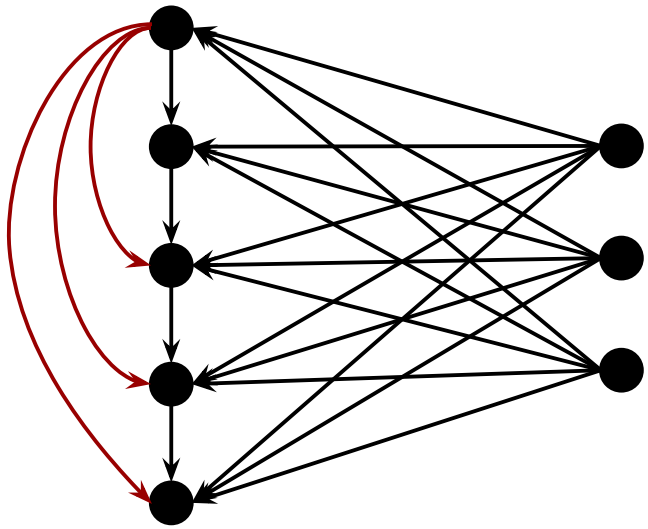


Gives $-\lceil \frac{n-1}{2} \rceil^2 + (n-1)\lceil \frac{n-1}{2} \rceil + (n-1)$ edges on n vertices.

Attempts to Maximize Edges

For non-collapsible CRVGs: Crown

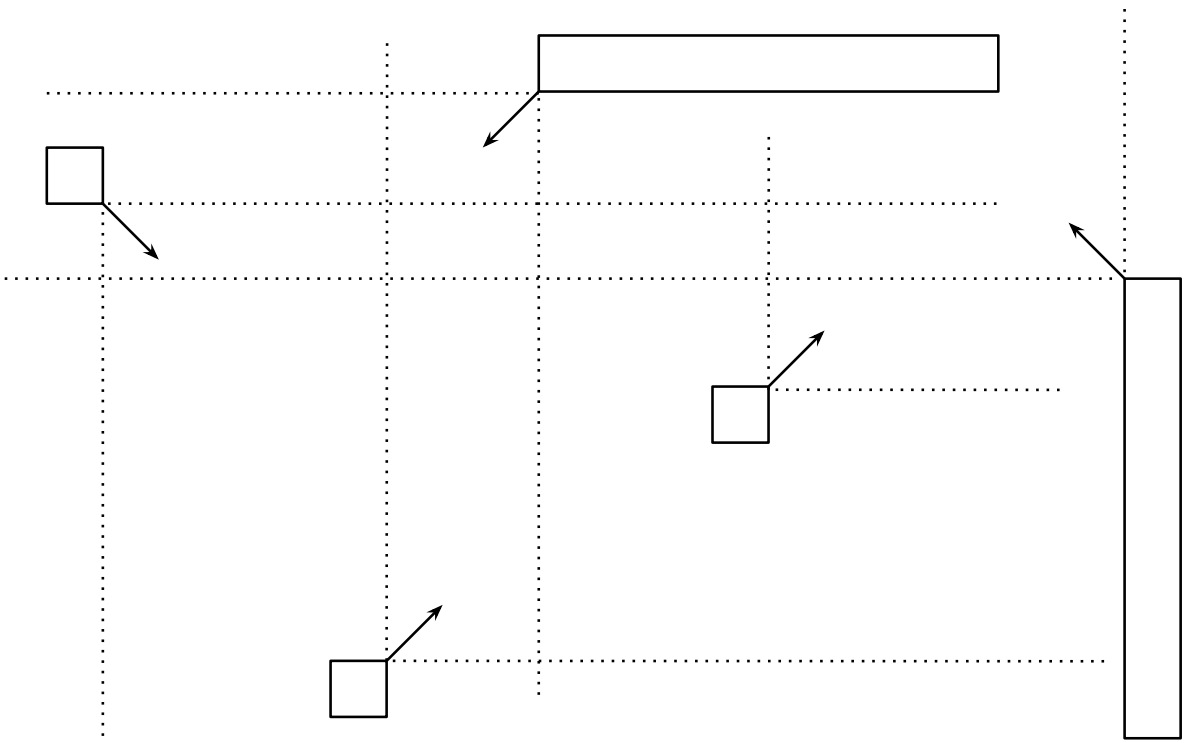




Gives $-\lceil \frac{n+2}{2} \rceil^2 + (n+2)\lceil \frac{n+2}{2} \rceil - 3$ edges on n vertices.

Attempts to Maximize Edges

Also we have this representation that gets $K5$. (Thank you Cin!)



Edge Comparison

n	Edges in DF	In C	In K_n	K_n is a UCRVG	K_n is a CRVG
3	3	3	3	yes	yes
4	5	6	6	yes	yes
5	8	9	10	?	yes
6	11	13	15	no	?
7	15	17	21	no	?
8	19	22	28	no	?
9	24	27	36	no	?
10	29	33	45	no	?

Using Computers in Our Research

Much of the research involved creating large amounts of examples. While sometimes our human intuitions were extremely helpful, there was often need for large amounts of brute force. When it comes to brute force, what could be better than a computer!!
And hence, we made a computer program to:

- Representing unit "squares" as five-tuples
- Taking a list of squares, and creating a directed and undirected graph
- Doing probabilistic searches for potential graphs of maximum edges using n squares.
- Run through exhaustive cases to prove some of our conjectures.

Representing Squares as 5-Tuples

In the program, squares were represented as five tuples.
As an example: (-6,-6,True,True,"z")

- The first two entries specify the x-y coordinates of the square.
- These are followed by two boleans [sp: Booleans — JLD], the first specifying whether or not the square is looking up, the second specifying whether the square is looking right.
- Finally, for the sake of keeping track of things and labeling, the final entry specifies the name of the square.

Analyzing a Situation, and Creating a Graph (too little detail)

The function "analyze_situation" is the most important function in this program. It's kind of like the engine, which makes every other part of the program possible.
In short, it analyzes a situation.

Analyzing a Situation, and Creating a Graph (too much detail)

The engine, so to speak, of the computer program, is a function titled "analyze_situation". This function took up the bulk of the program. Although the details of this function may be a bit much for a single presentation, we will provide a brief outline for what it does:

- Define a list for storing the edges of the graph (directed).
- Loop through each of the squares.
  - Define a list for storing the squares in the squares field of view.
  - Loop through all of the other squares.
    - Decide which squares are in that square's field of view.
    - Determine the distance from that square to the current square in question.

  Sort each of the squares looped for by distance from the square in question.
    - Loop through each of the squares in the field of view, as sorted by distance, starting with the closest.
    - Run the "check_seen" function (described in the next slide).
    - If seen, append to the list of edges!

The "check_seen" and "check_in_shadow" functions

Although the details of these functions are not all that important, they should still be briefly mentioned.

The "check_seen" function takes a square, and a list of other squares, as well as the direction of the shadows cast.

Making Sure Everything Works, and Pretty Colors!!

- Creating the program to analyze a setup of squares was fairly difficult, and the core part of the code totalled around 150 lines (this is only a lot for math majors).
- Testing would be hard work without this nifty program for creating random setups, and visualizing them.
- Luckily, everything works. See the next slide for the graph generated by a situation of 1000 squares!

1000 Squares!

Enjoy this chaotic and beautiful test-run of 1000 squares!

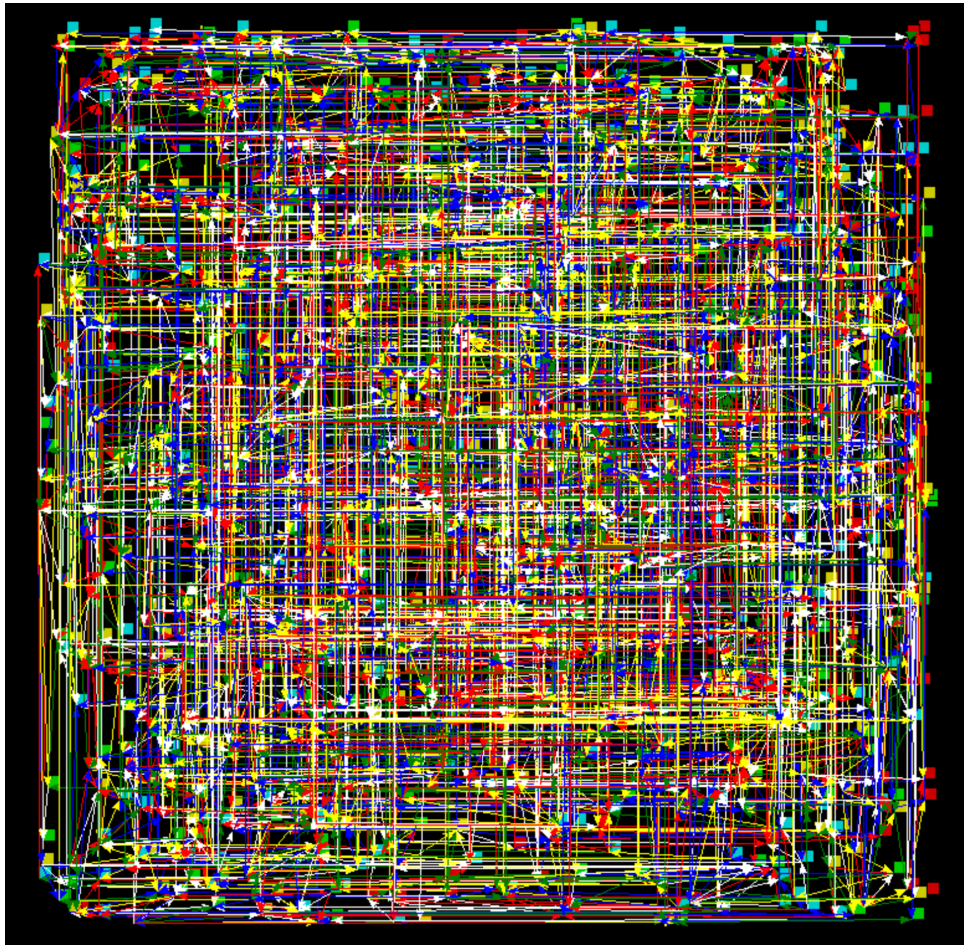


Figure: Caption

Searching For Maximum Edges: "find_maximum_n"

Recall the DF method for finding an large amounts of edges. We wanted to see how effective of an upper bound this was, so we tested it with a computer program!

Number of squares	Theoretical	Computer
3	3	3
4	6	6
5	8	8
6	11(DF?)	11
7	15(DF?)	15
8	Find	19
9	24(DF)	24
10	(bigger, find)	24

Table: Caption

Some pictures of maximal graphs

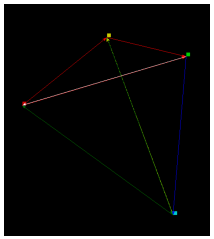
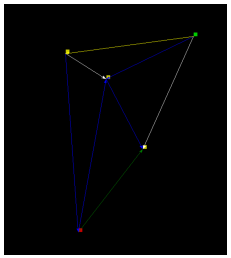
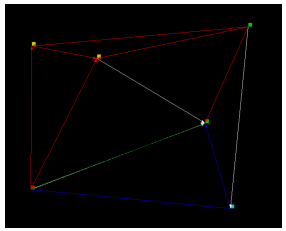
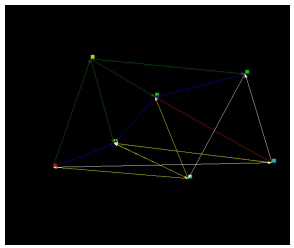
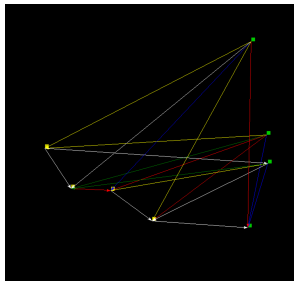
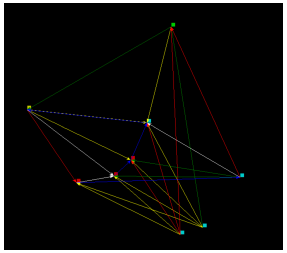
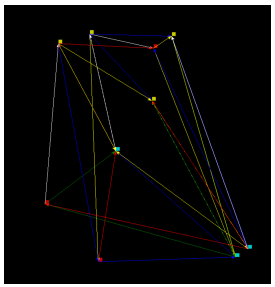
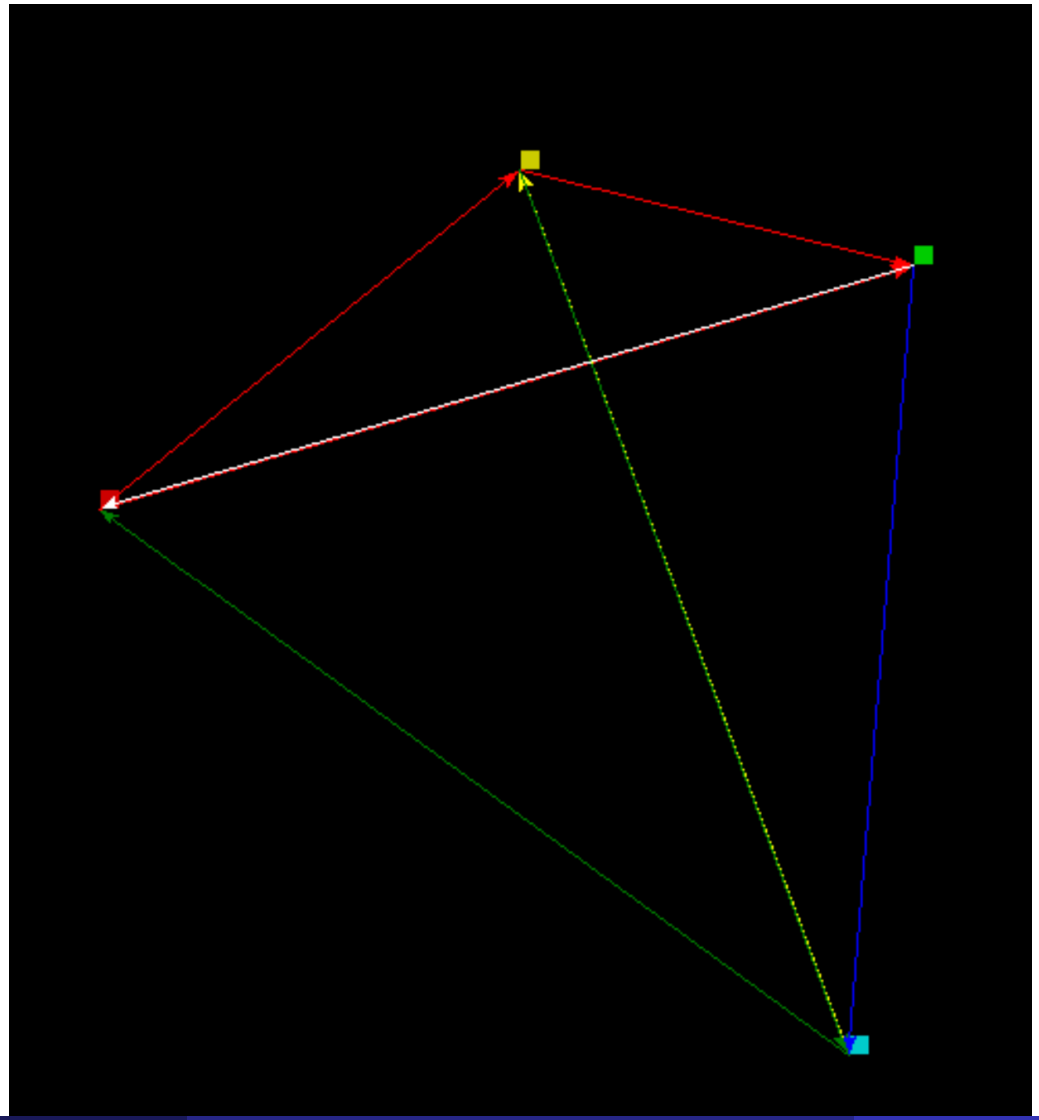
4	
5	
6	
7	
8	
9	
10	

Table: Maximum graphs found via computer search

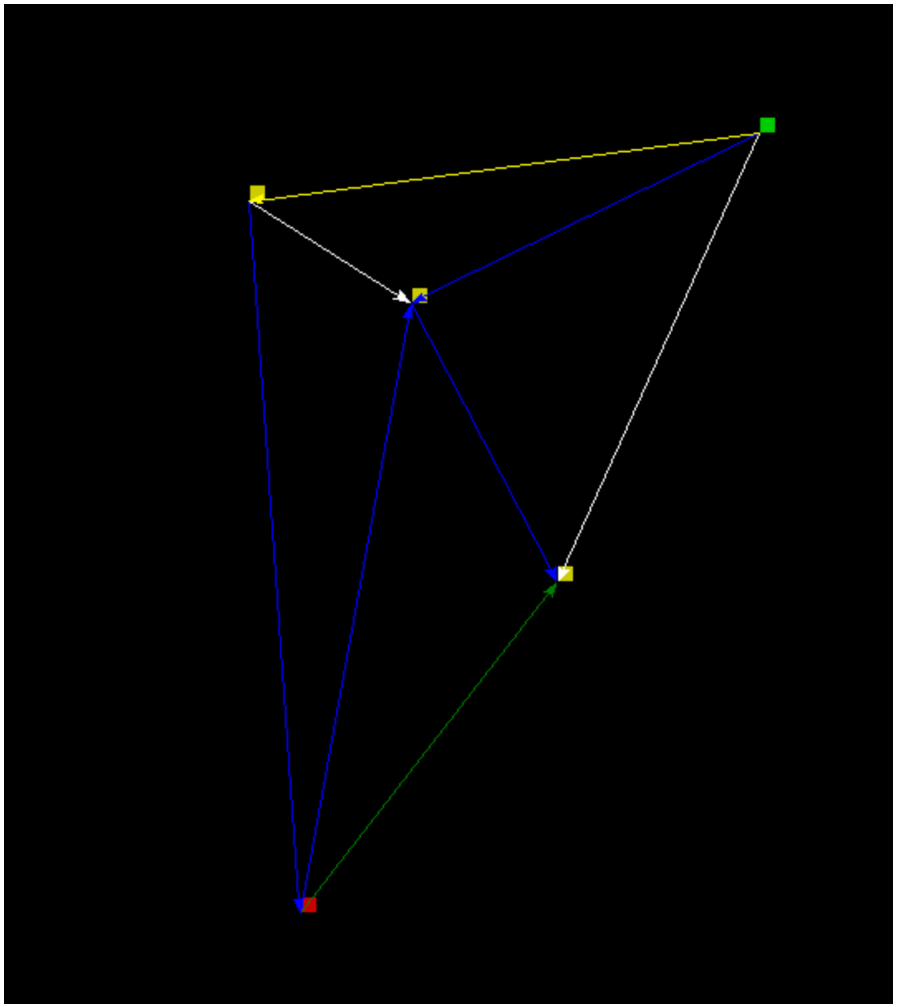
n = 4

Notice that we found K4. Almost every time, the search arrived at this graph.



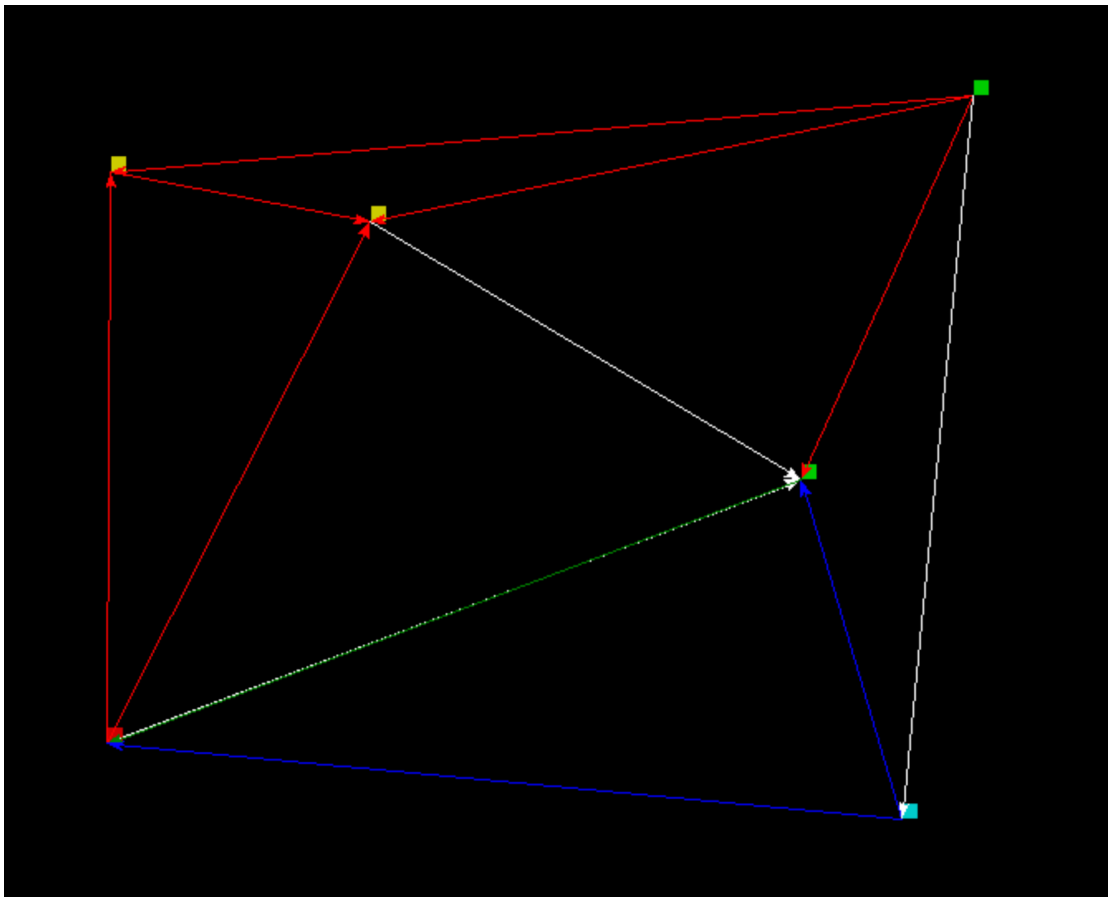
n = 5

The program consistently found K5, missing two edges, consistent with our conjecture that this is the best we can get.



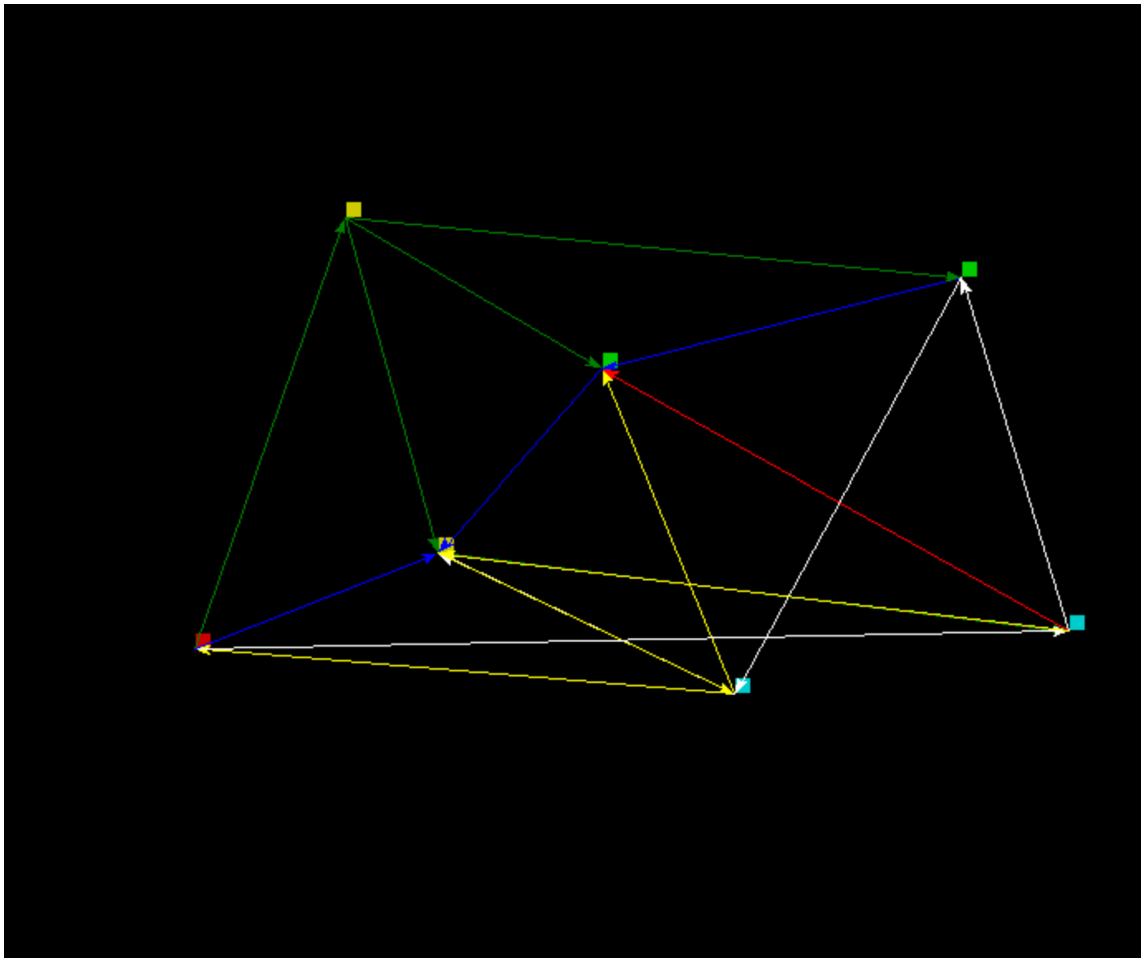
n = 6

Not sure what to think about this one. Tis a pretty picture!



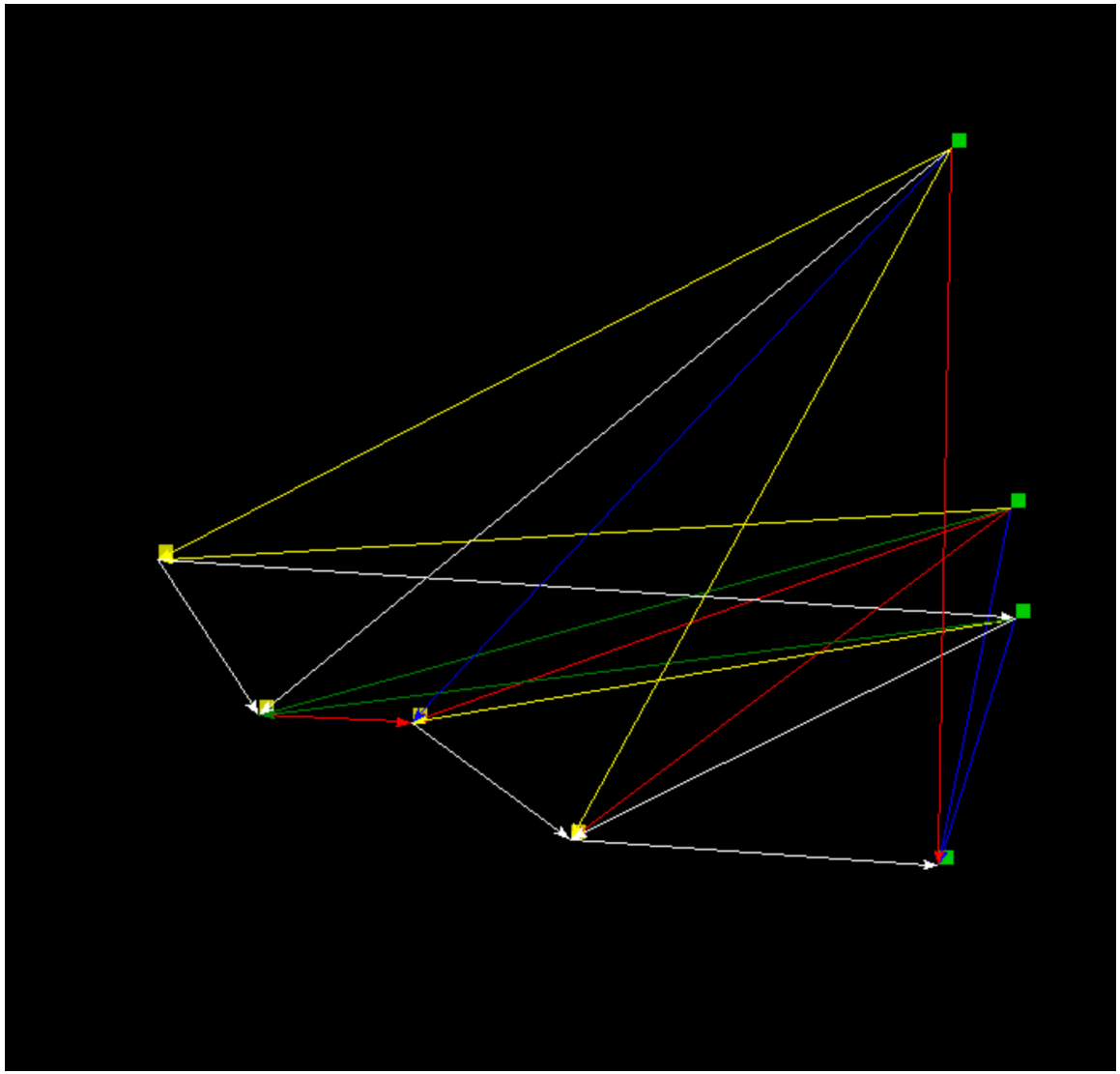
n = 7

Notice that we are starting to get something similar to the double fan conjecture.



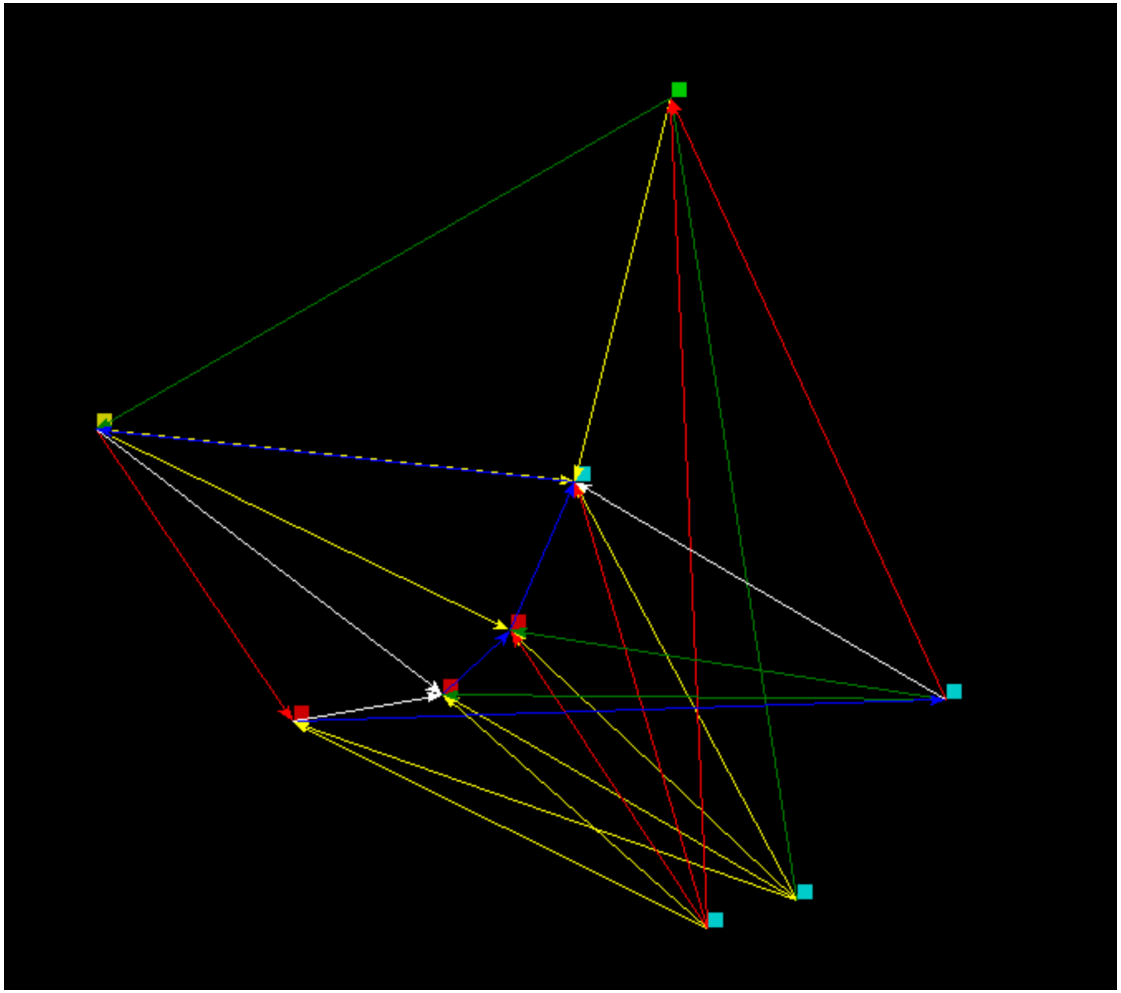
n = 8

Once again, we are starting to get something similar to the double fan conjecture.



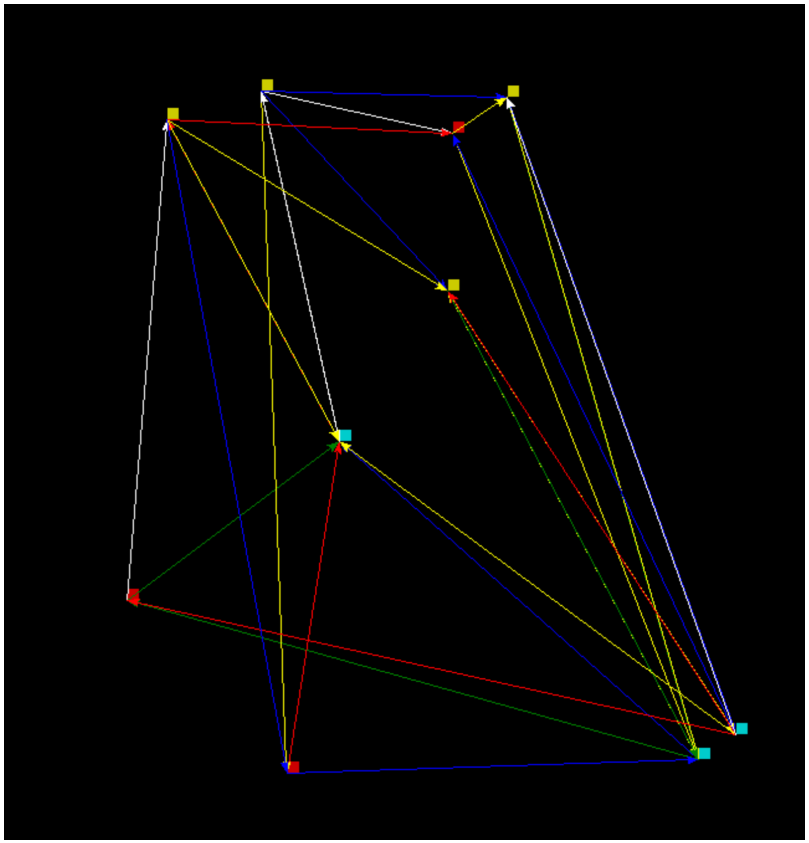
n = 9

Though it took upwards of 200 million trials, for n = 9 we actually found THE double fan!!



n = 10

After 10 vertices, the computer started varying more in its results, indicating that the probability at finding maximum edges was probably far smaller. Nonetheless, every trial came out eerily similar to the double fan, even if not exactly the same.

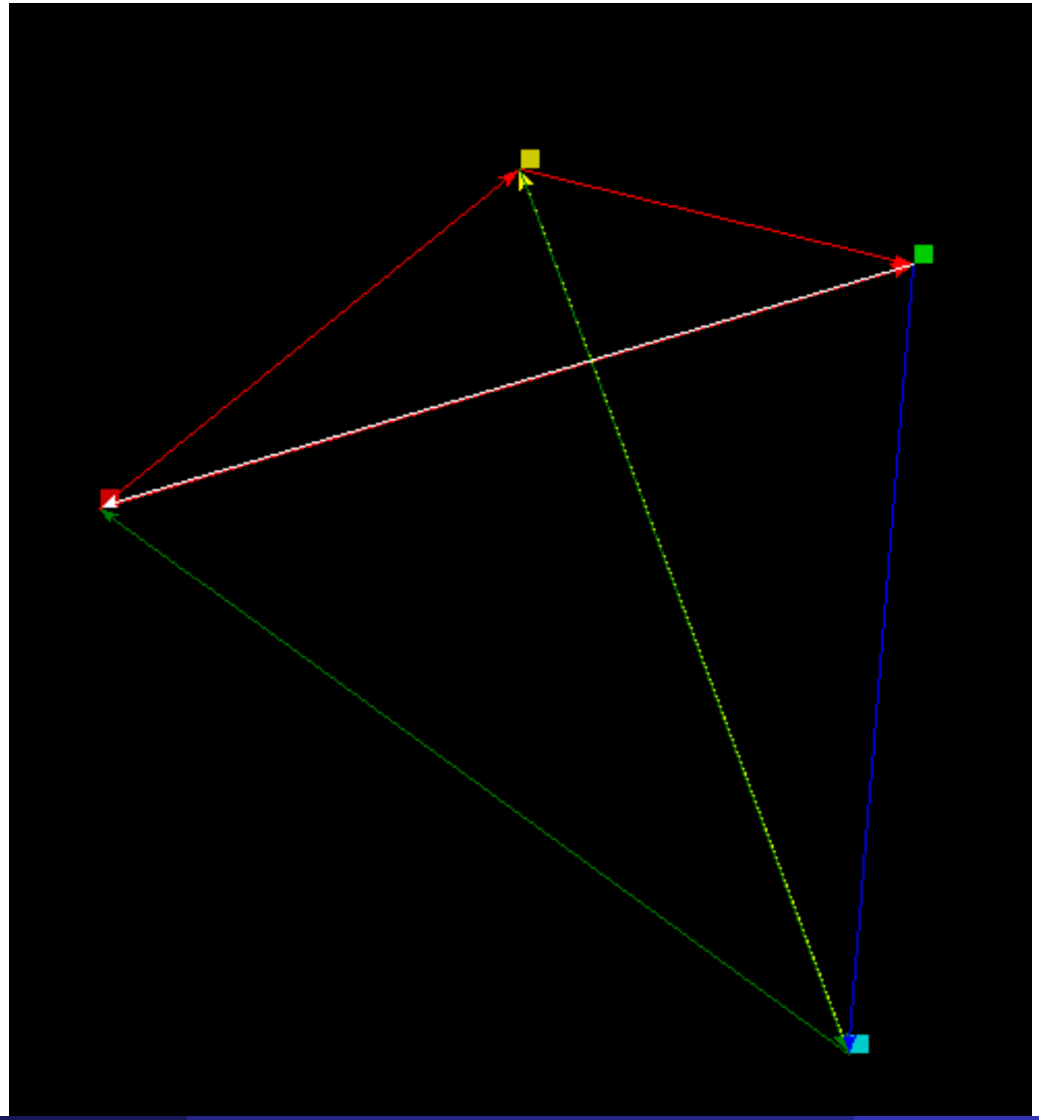


Proving Stuff: analyze_regions.py

- Finally, the program was not simply meant to provide motivation for conjectures and create heuristic arguments: we wanted to actually prove stuff!
- A good test-run was the proof that there is only one directed K4.
- And we might have, pending on whether the algorithm was correct!

The conjecture

The conjecture was simple: that this is the only directed K4 (notice the arrows):



Proving stuff: an outline of the method

First, without loss of generality, place a square somewhere, and choose a direction of sight (why not up-up). Collapse it to a point (as Zeke has shown works), and draw a grid around it.

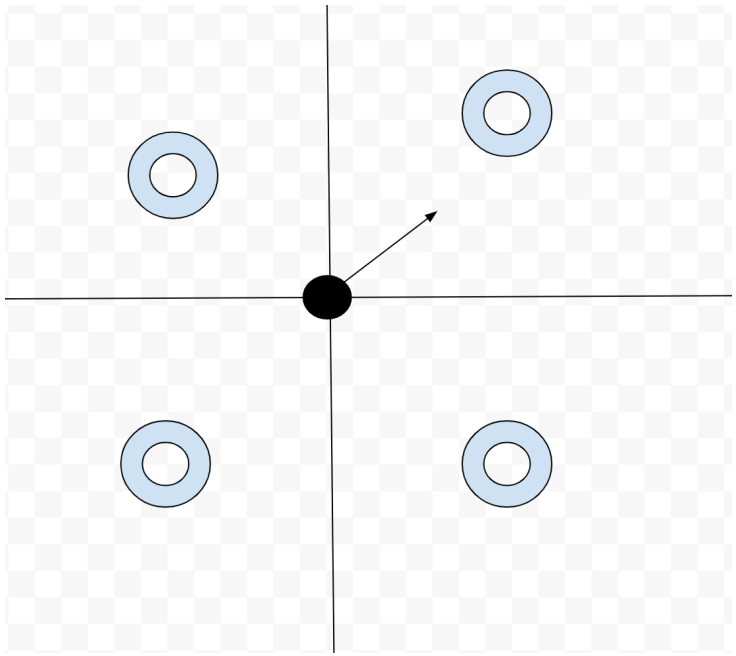


Figure: The grid

Place a square in each region, chose each possible direction for each region

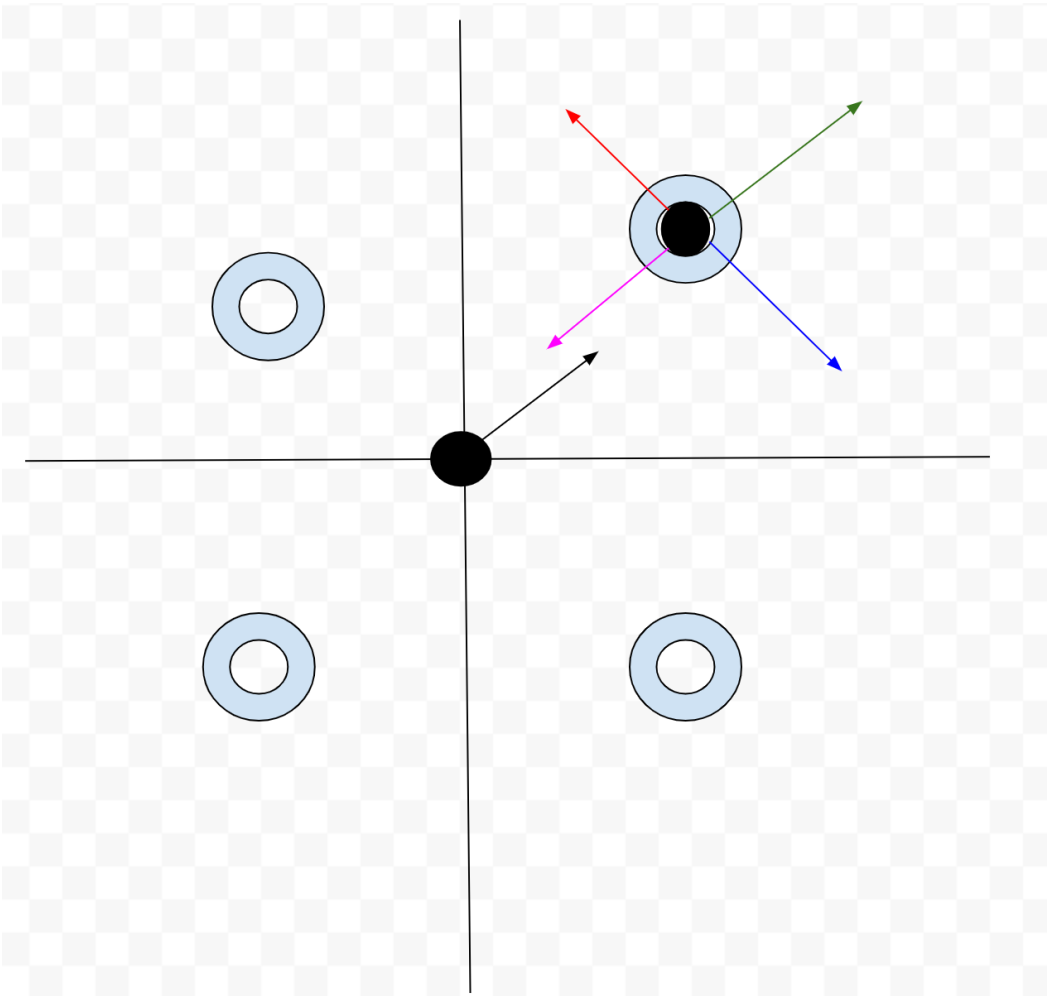


Figure: Place a square

Filter through all of the graphs that do not form a subgraph of K4

Figure: A filter

Originally, I called the ones that made it through the filter "candidates." For here, let's call them "survivors."

For each of the survivors, create a grid around each of the points, and repeat the process

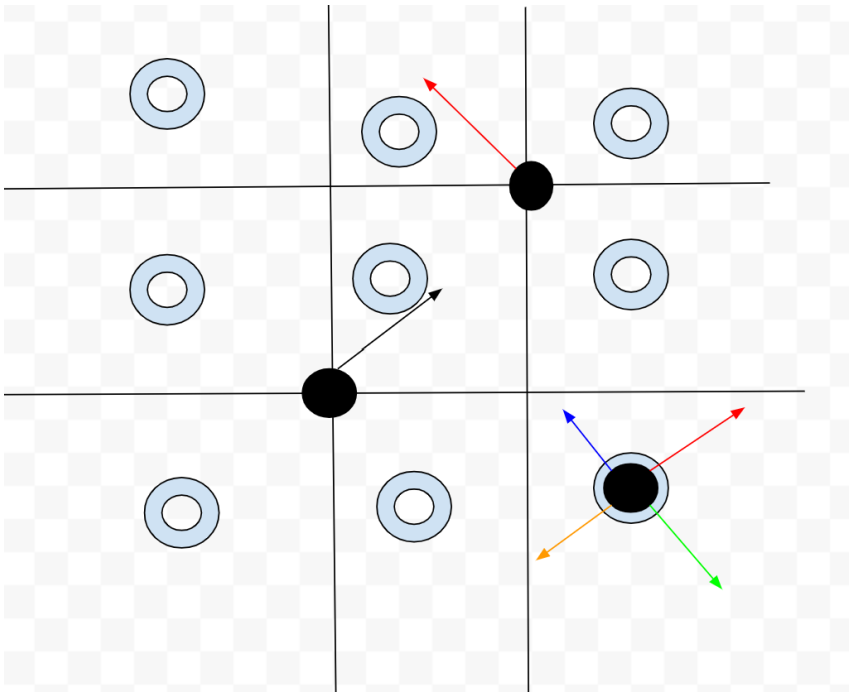


Figure: Caption

One last time, for each of the survivors, place a square in each possible region, and in each possible direction

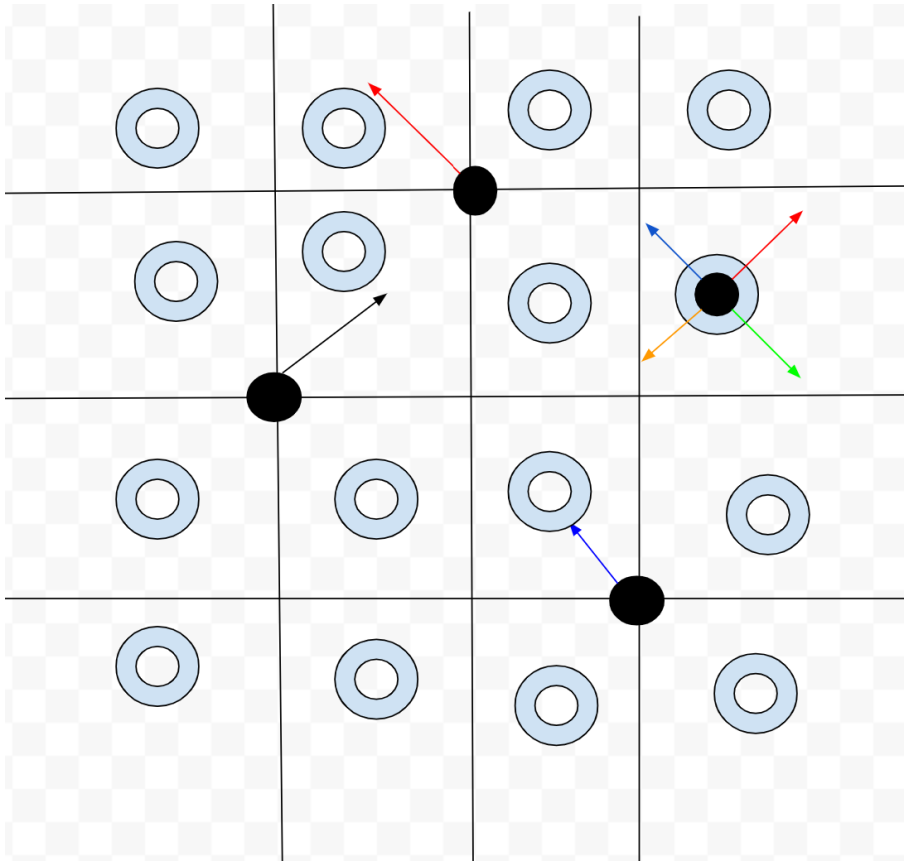


Figure: Even larger grid!

Apply the filter one last time

Apply the filter once more!

Figure: A filter

Voila: The Results!!

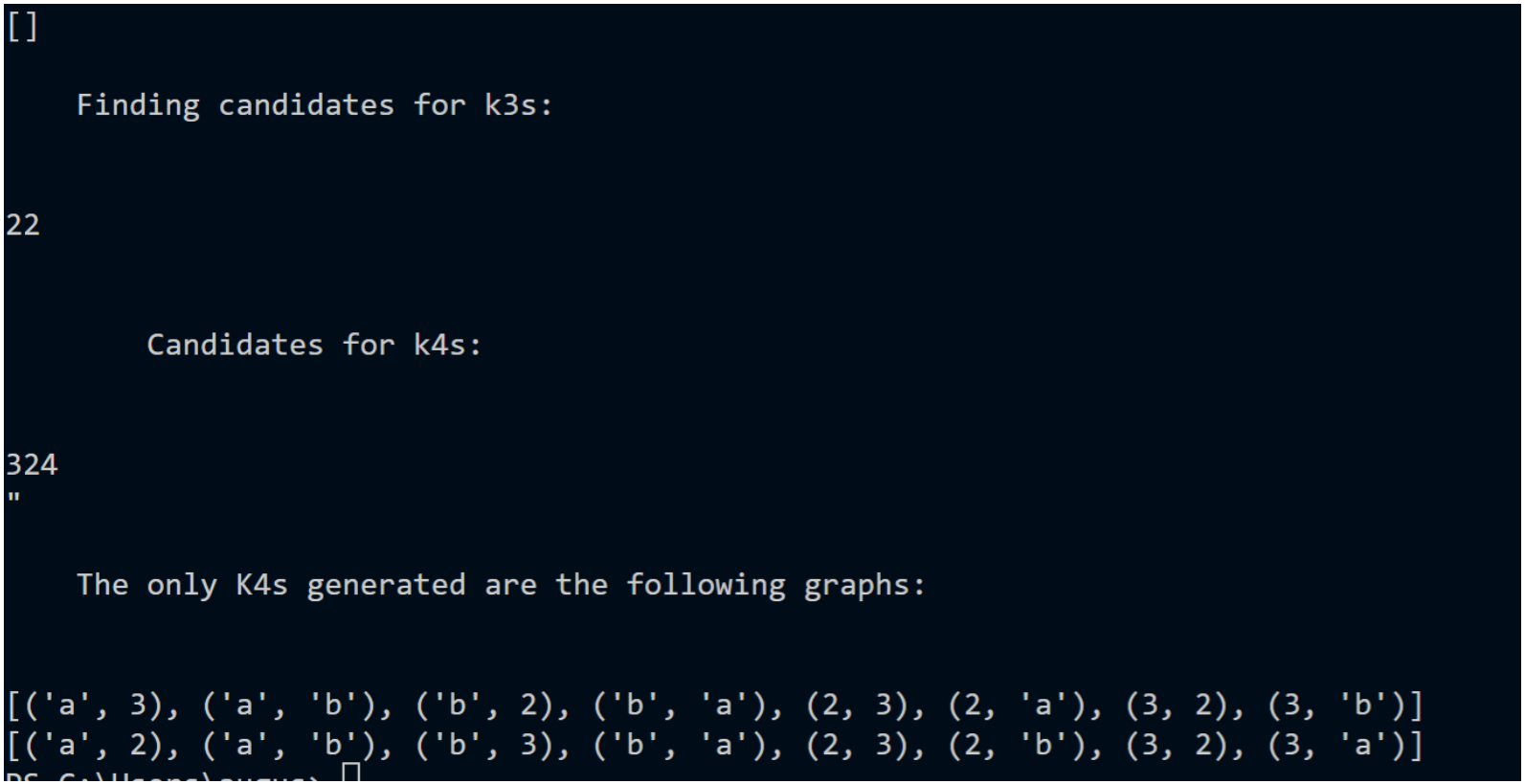


Figure: output

We're almost there