Snapping Graph Drawings to the Grid Optimally

Andre Löffler, Thomas C. van Dijk, Alexander Wolff

Chair for Computer Science I:
Algorithms, Complexity and Knowledge-Based Systems
University of Würzburg

19. September 2016

Introduction

Snap-rounding:

Introduction

Snap-rounding:

- Used to overcome precision-related problems of computational geometry.

Introduction

Snap-rounding:

- Used to overcome precision-related problems of computational geometry.
- Conform to list of desired properties:
  - Fixed-precision representation (e.g. integer coordinates)
  - Geometric similarity (no large vertex movements)
  - Topological similarity (equivalence up to the collapsing of features)

Introduction

Snap-rounding:

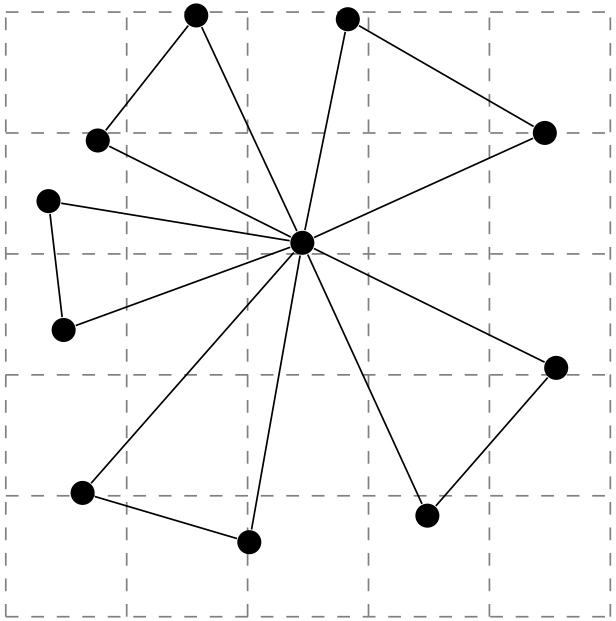
- Used to overcome precision-related problems of computational geometry.
- Conform to list of desired properties:
  - Fixed-precision representation (e.g. integer coordinates)
  - Geometric similarity (no large vertex movements)
  - Topological similarity (equivalence up to the collapsing of features)

Our question:

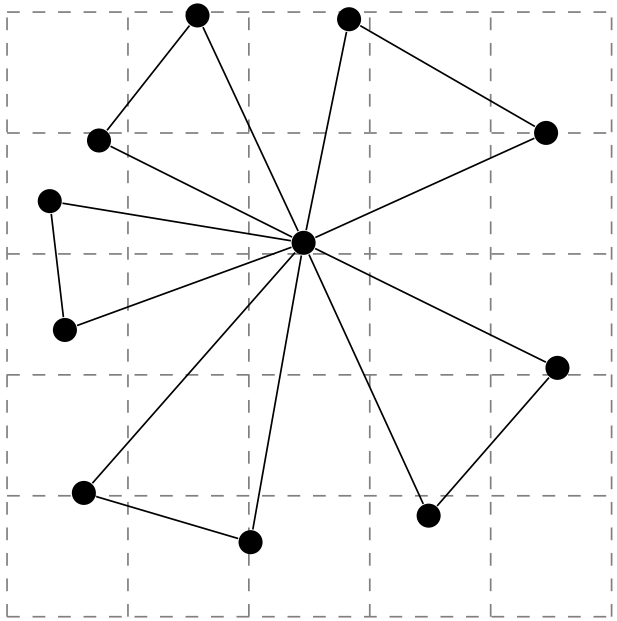
What about topological equivalence?

Intuition

Snap-rounding:

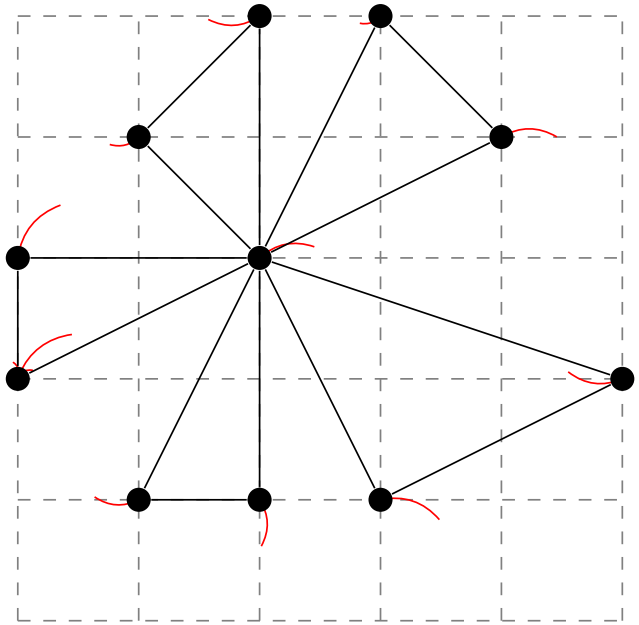


Topologically valid:

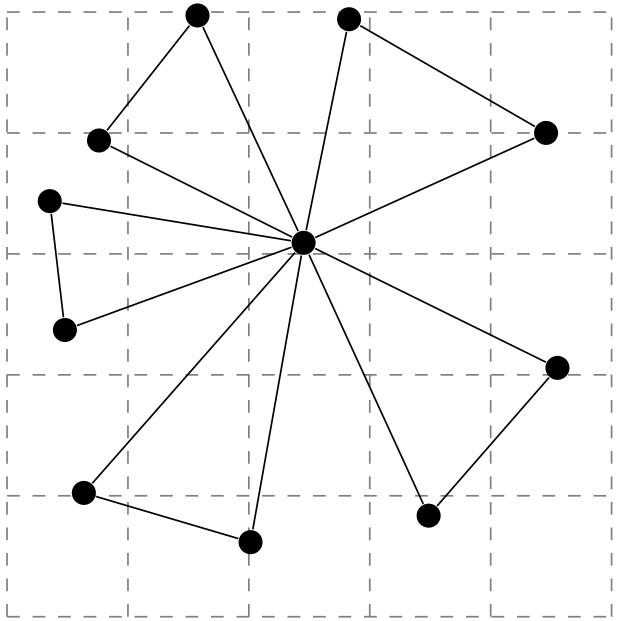


Intuition

Snap-rounding:

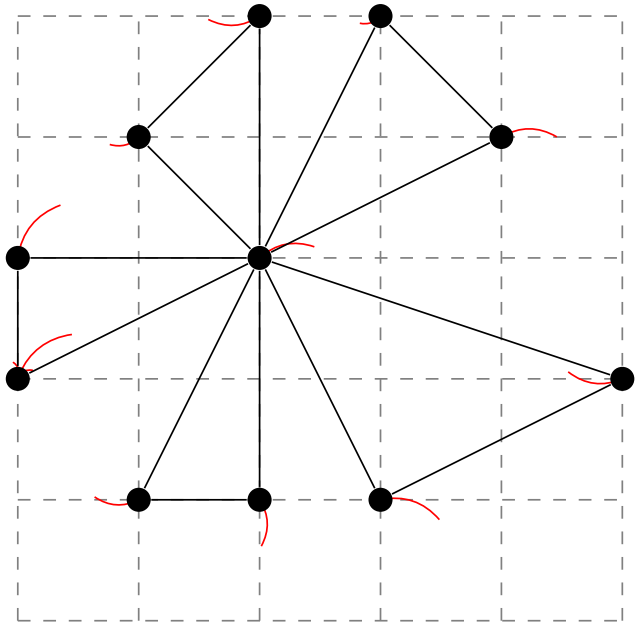


Topologically valid:

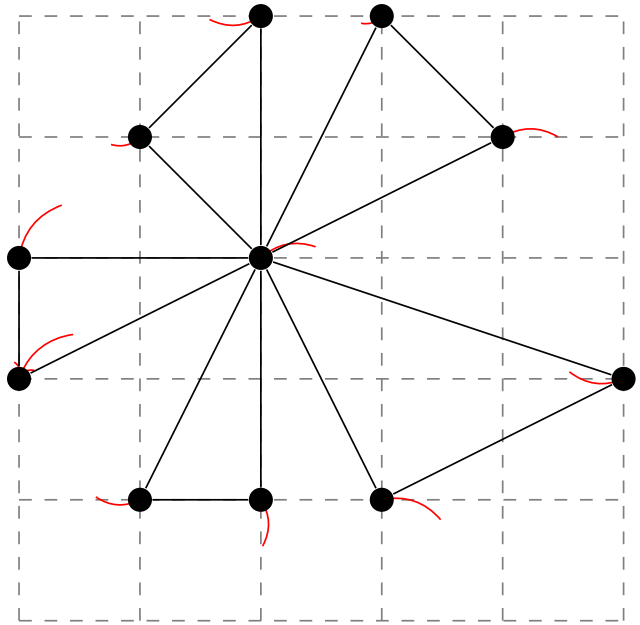


Intuition

Snap-rounding:



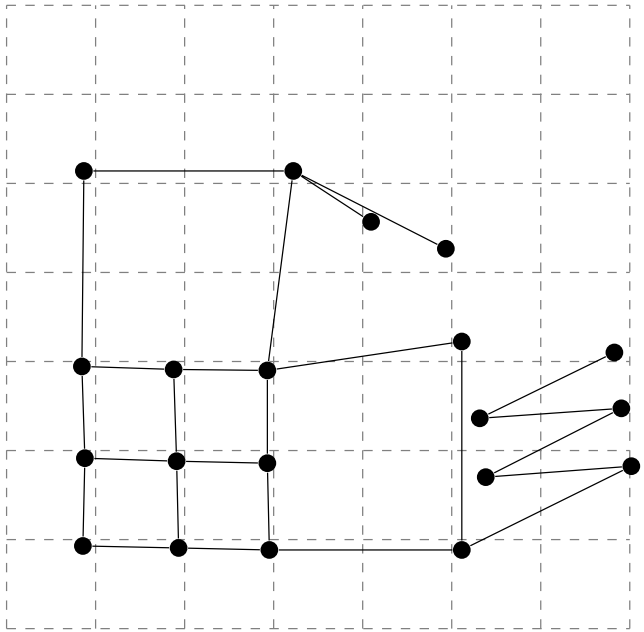
Topologically valid:



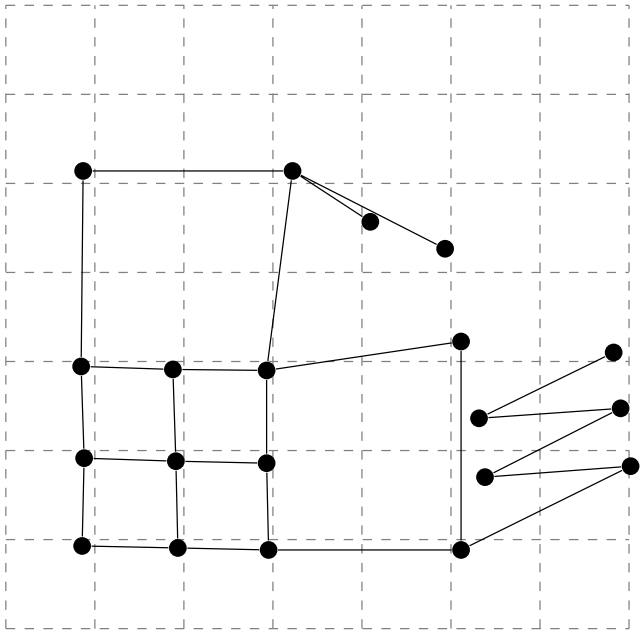
- Snap-rounding already is topologically equivalent.

Intuition

Snap-rounding:

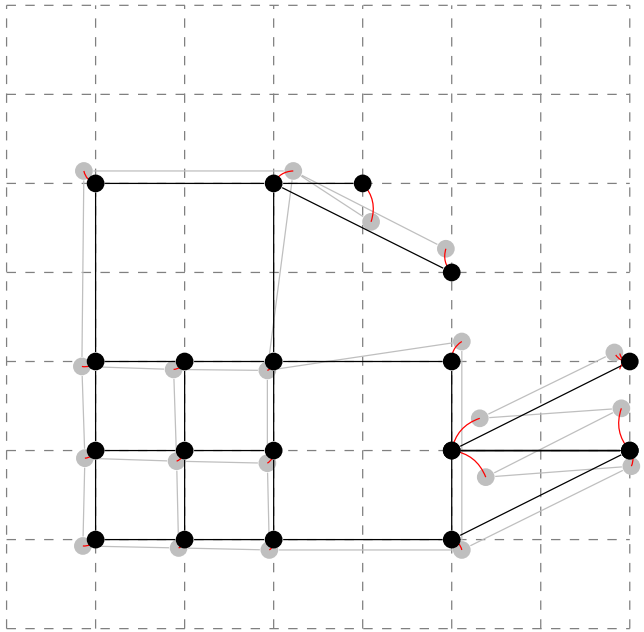


Topologically valid:

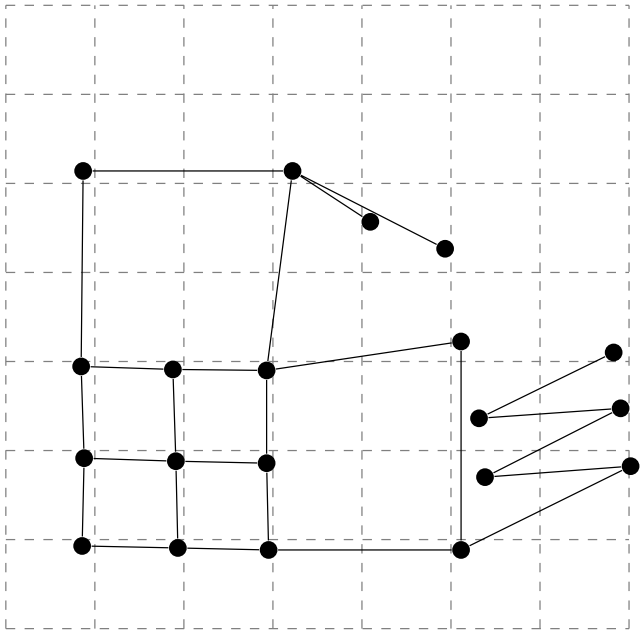


Intuition

Snap-rounding:

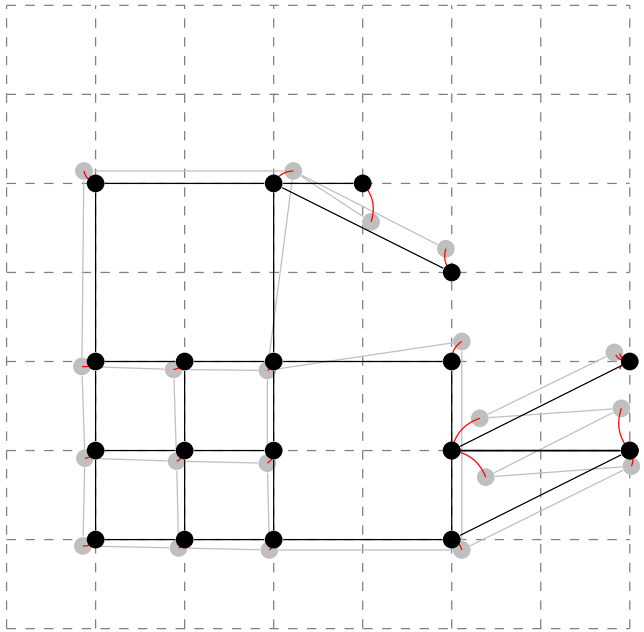


Topologically valid:

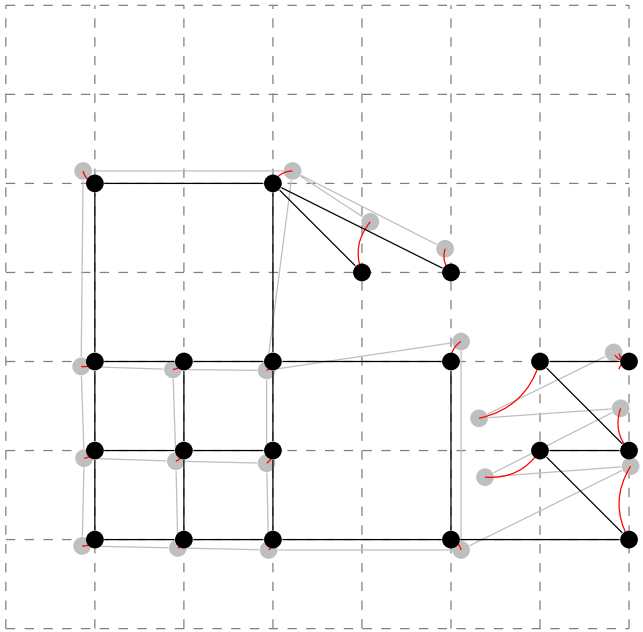


Intuition

Snap-rounding:

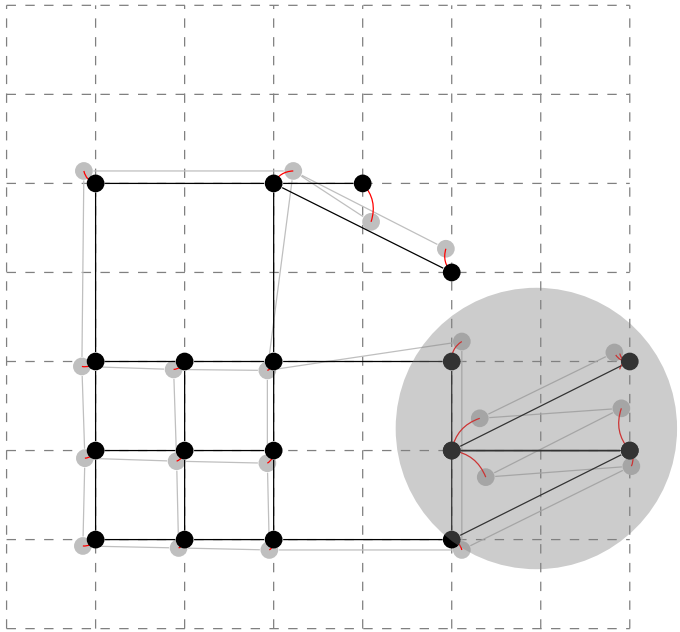


Topologically valid:

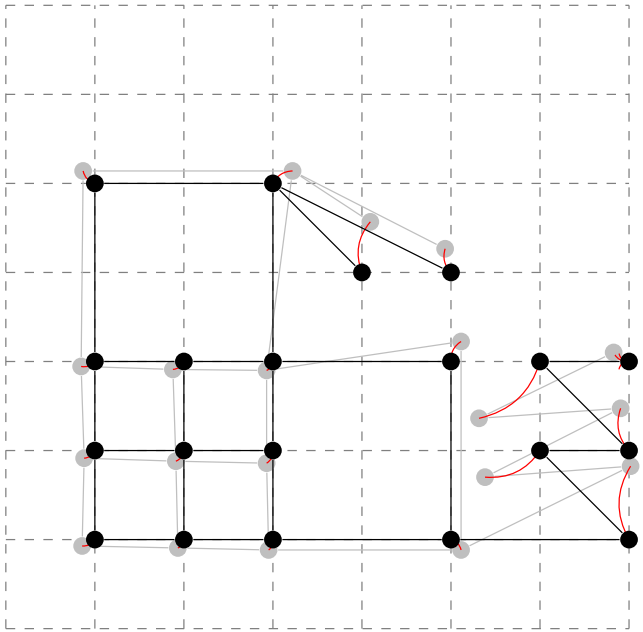


Intuition

Snap-rounding:



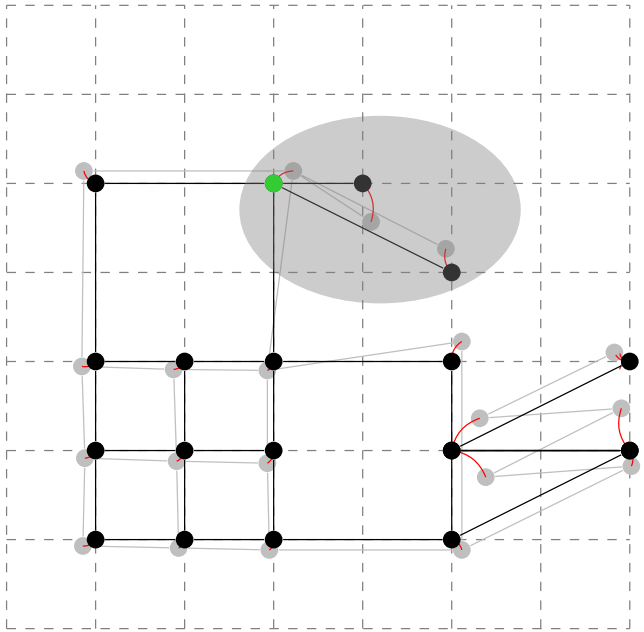
Topologically valid:



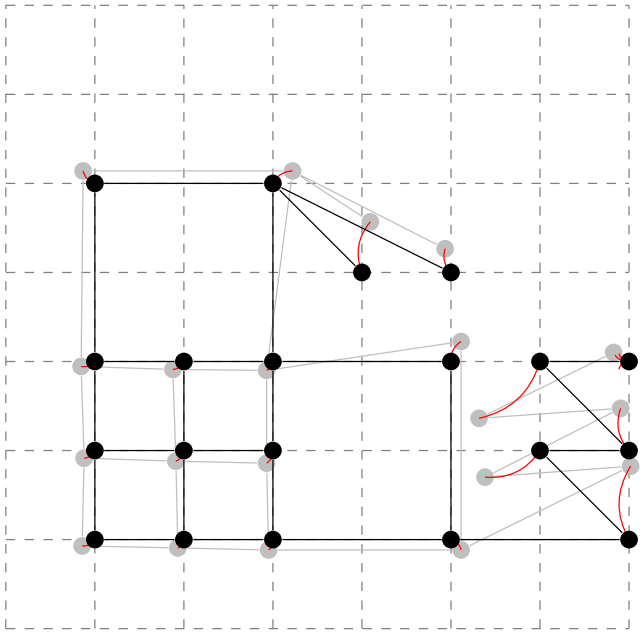
- Snap-rounding alters incidences and forces edges to collapse.

Intuition

Snap-rounding:



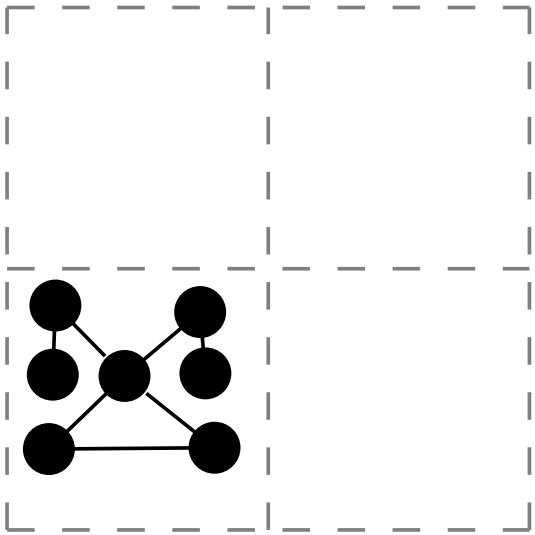
Topologically valid:



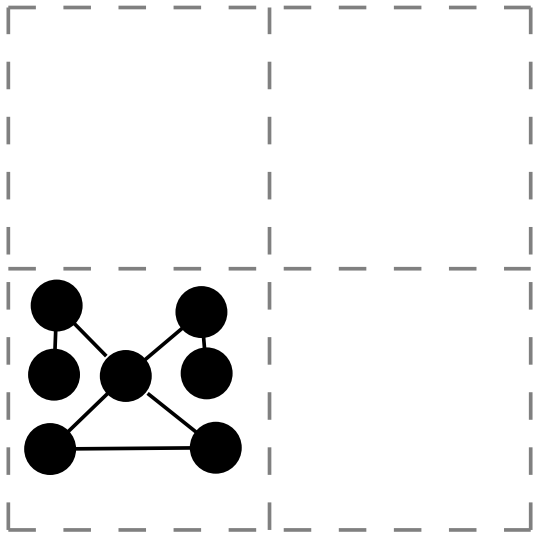
- Snap-rounding alters incidences and forces edges to collapse.
- Rounding to the nearest grid point changes the embedding of the upper-left vertex.

Intuition

Snap-rounding:

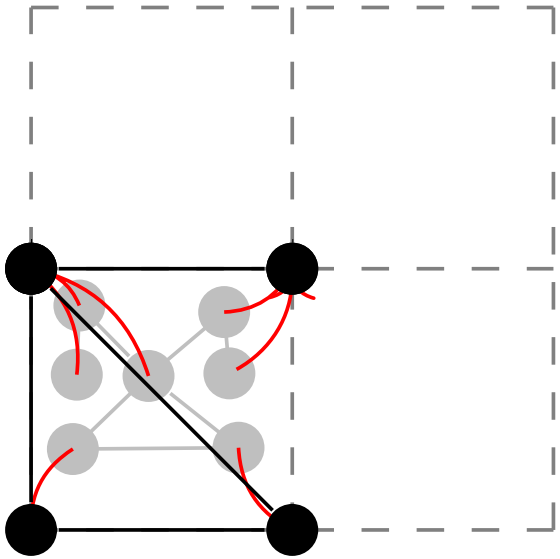


Topologically valid:

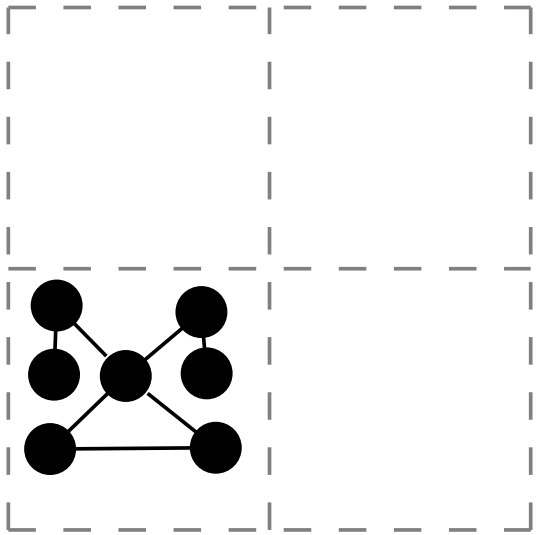


Intuition

Snap-rounding:

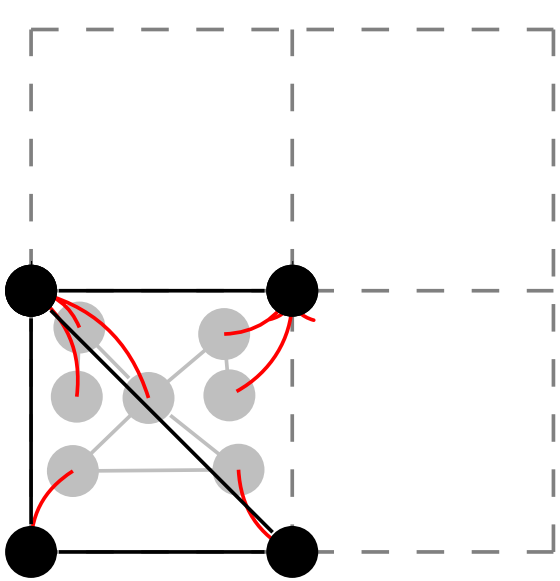


Topologically valid:

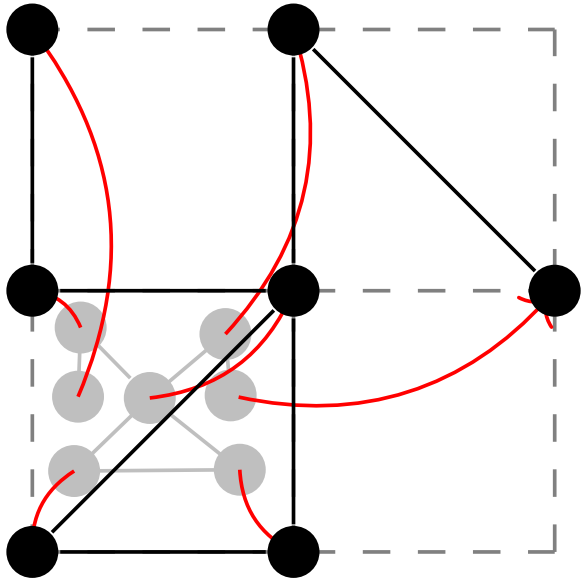


Intuition

Snap-rounding:

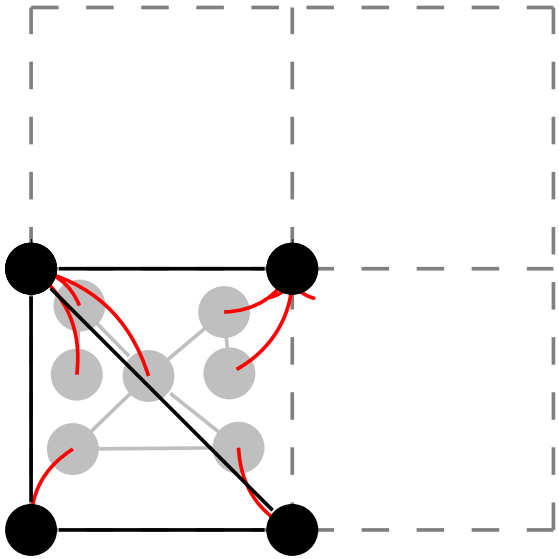


Topologically valid:

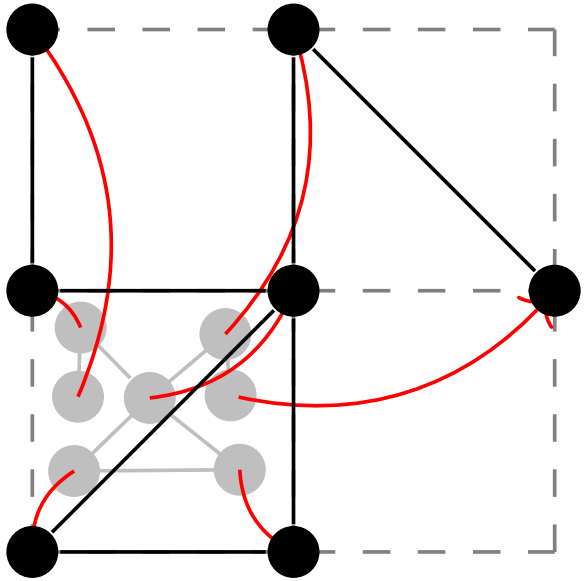


Intuition

Snap-rounding:



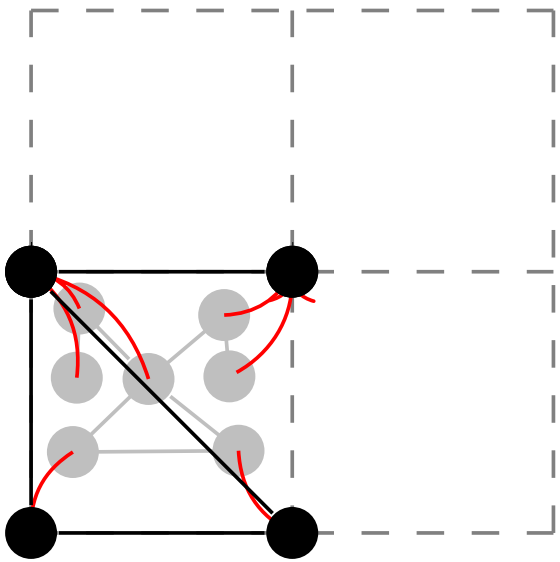
Topologically valid:



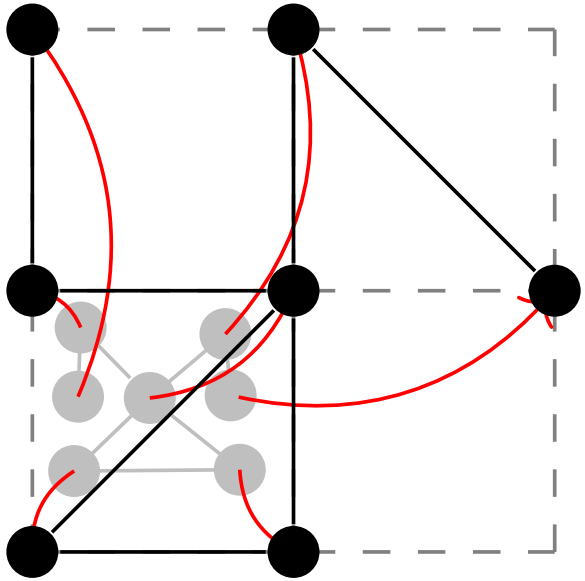
- Snap-rounding heavily modifies this graph.

Intuition

Snap-rounding:



Topologically valid:



- Snap-rounding heavily modifies this graph.
- "Rounding" dense structures with no features collapsing is closely related to creating minimum-area drawings.

Topologically Safe Snapping

- We relax on geometric similarity and allow for larger vertex movements.

Topologically Safe Snapping

- We relax on geometric similarity and allow for larger vertex movements.
- We do not allow for features to collapse.

Topologically Safe Snapping

- We relax on geometric similarity and allow for larger vertex movements.
- We do not allow for features to collapse.

Problem (Topologically Safe Snapping)

Graph $G = (V, E)$ with given embedding, bounding box $B = [0, X_{\max}] \times [0, Y_{\max}]$.

Round G to integer coordinates within B, preserving the given embedding and minimizing total vertex movement.

Topologically Safe Snapping

- We relax on geometric similarity and allow for larger vertex movements.
- We do not allow for features to collapse.

Problem (Topologially Safe Snapping)

Graph $G = (V, E)$ with given embedding, bounding box $B = [0, X_{\max}] \times [0, Y_{\max}]$.

Round G to integer coordinates within B, preserving the given embedding and minimizing total vertex movement.

- Movement is measured in Manhattan-distance.

Our Contribution

- $\mathcal{NP}$-hardness proof for Topologically Safe Snapping.

Our Contribution

- $\mathcal{NP}$-hardness proof for Topologically Safe Snapping.
- Integer Linear Program (ideas only)

Our Contribution

- $\mathcal{NP}$-hardness proof for Topologically Safe Snapping.
- Integer Linear Program (ideas only)
- Experimental Evaluation

The $\mathcal{NP}$-hardness proof

NP-Hardness

- We reduce from Planar Monotone 3SAT.

NP-Hardness

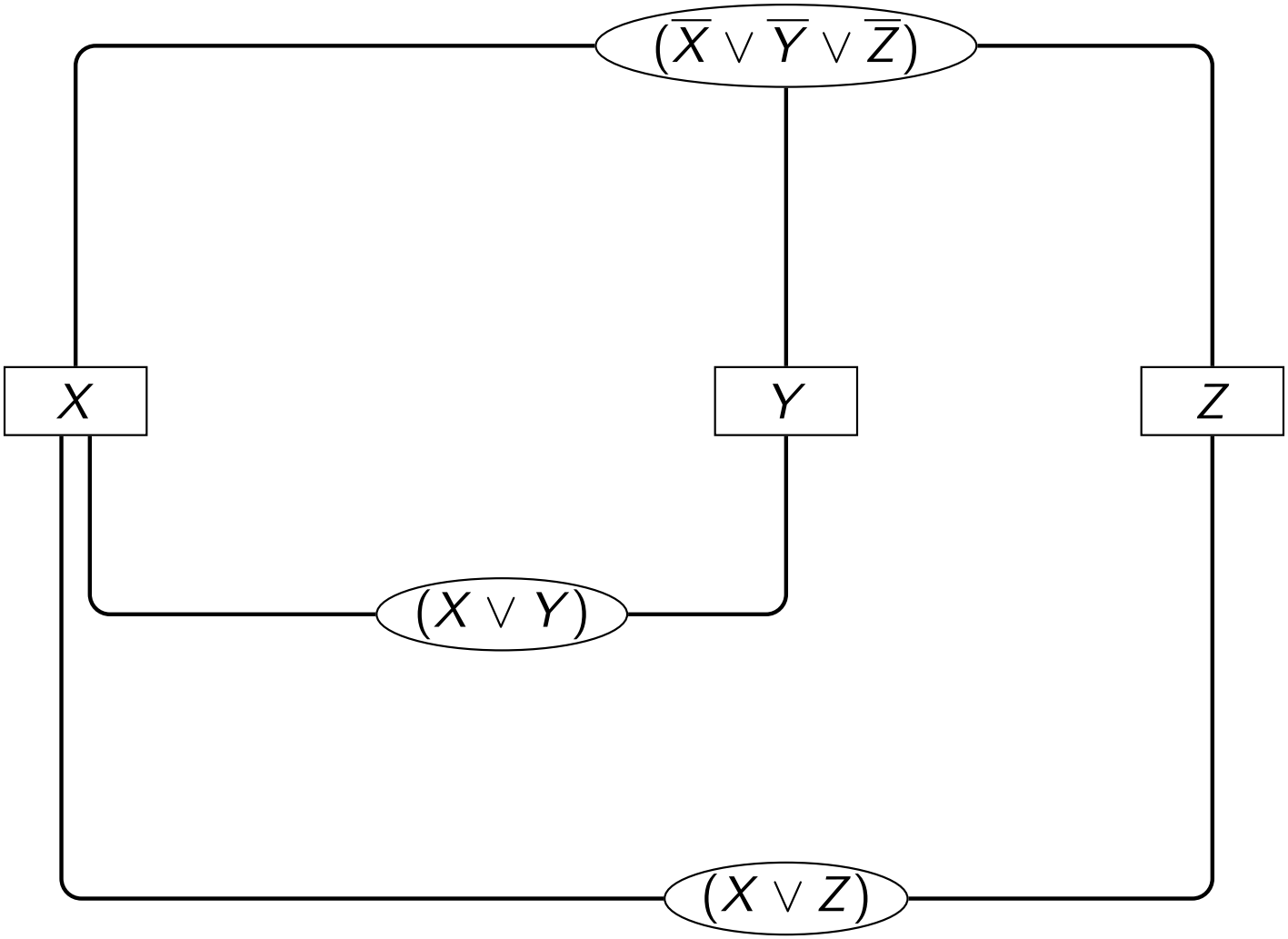
- We reduce from Planar Monotone 3SAT.
- For reduction, consider a decision variant:

Problem (Cost-bound Topologically Safe Snapping)

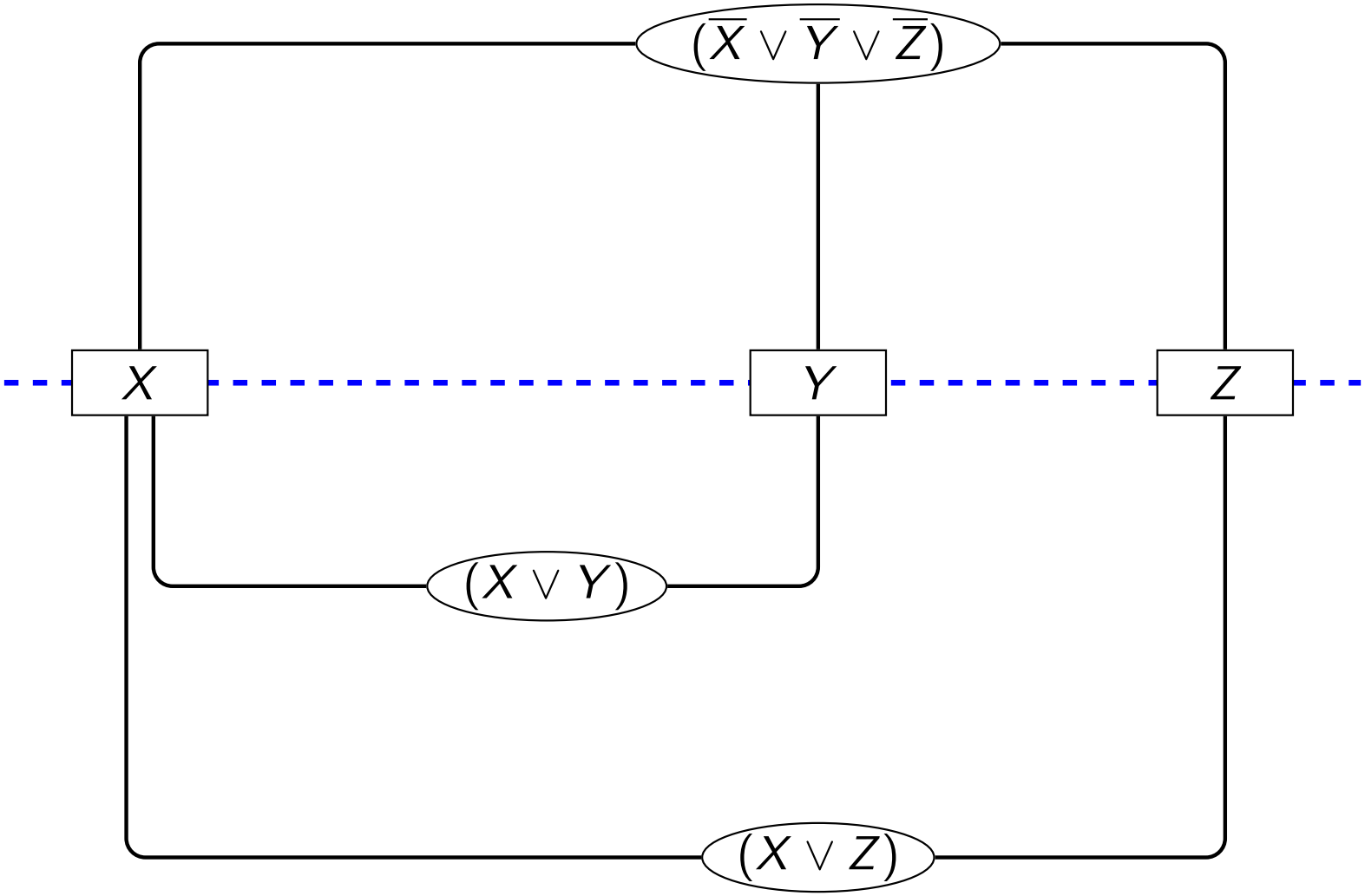
Graph $G = (V, E)$ with given embedding,
bounding box $B = [0, X_{\max}] \times [0, Y_{\max}]$, cost-bound $c_{\min} \in \mathbb{R}^+$.

Can G be rounded to integer coordinates within B, preserving the given embedding with total movement of $c_{\min}$?

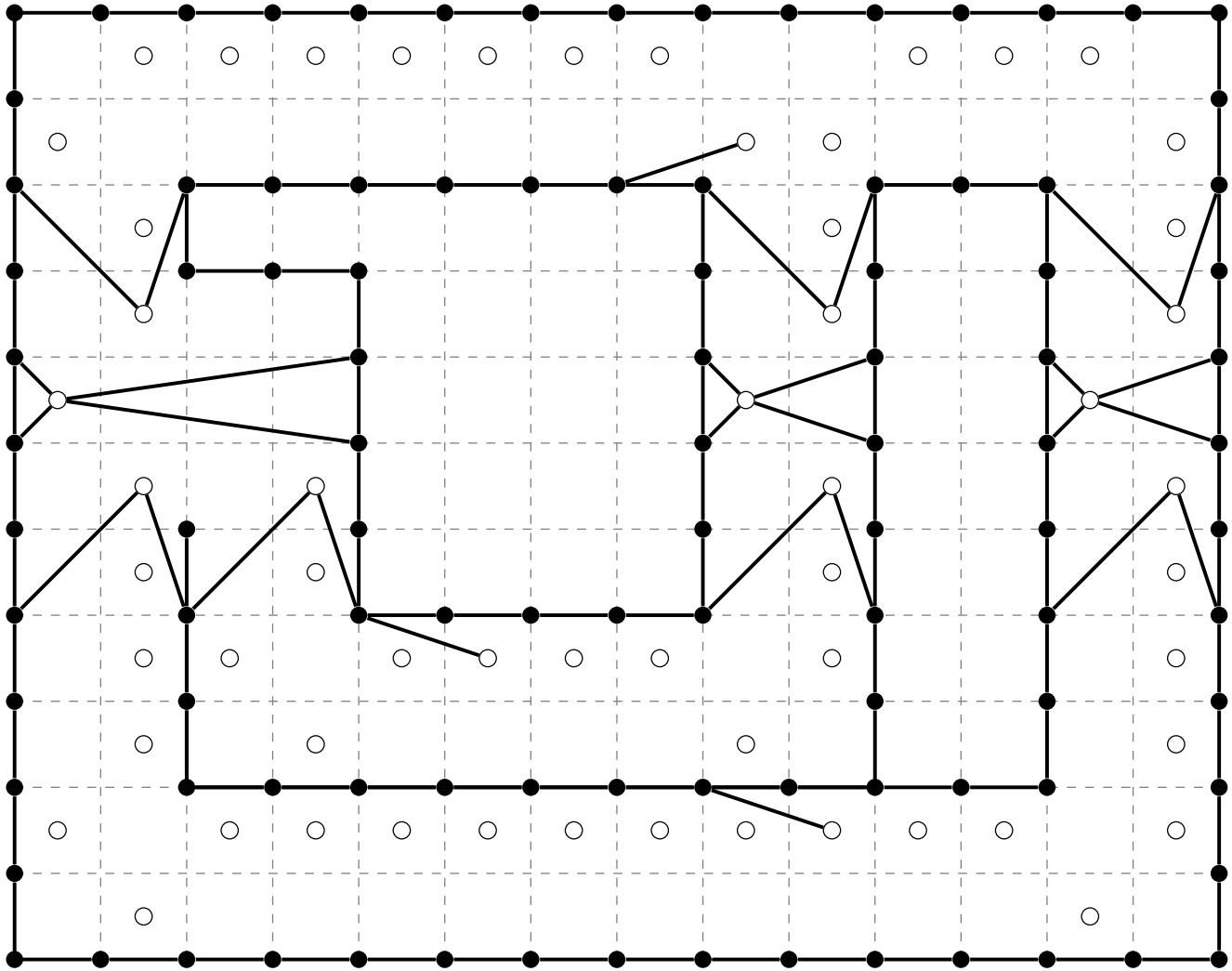
PM3SAT-formula



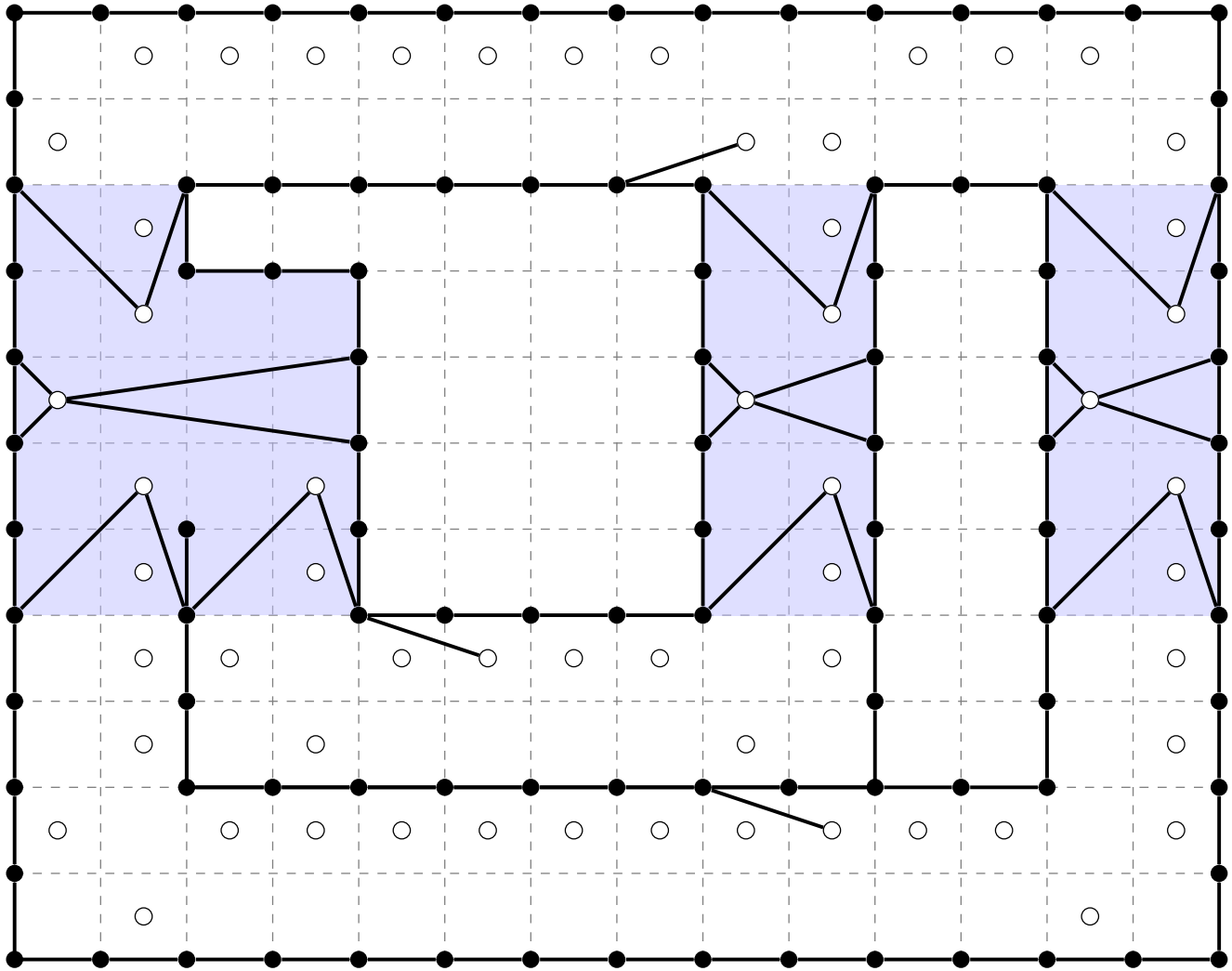
PM3SAT-formula



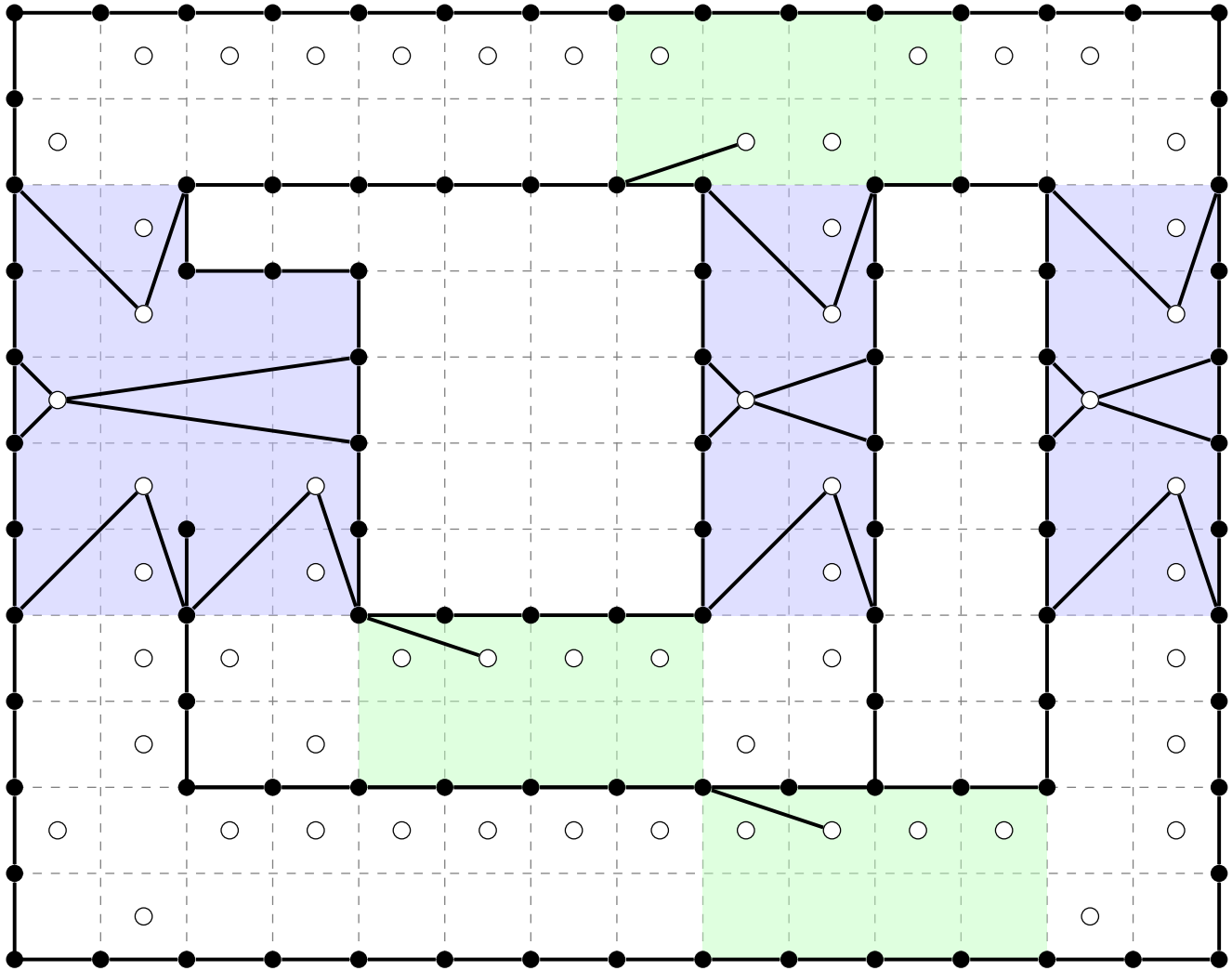
Our Construction



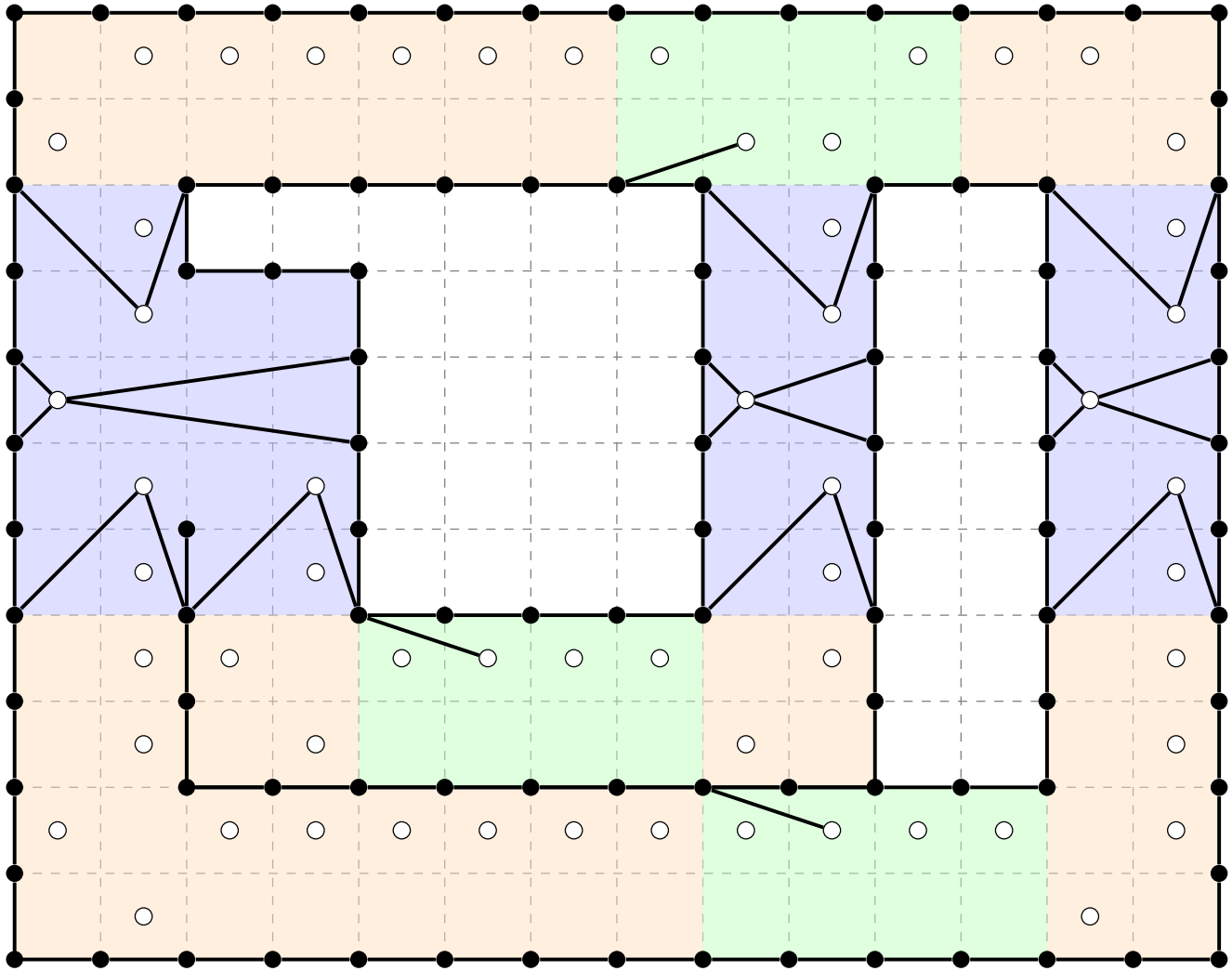
Our Construction



Our Construction



Our Construction



Tunnels & Pushes

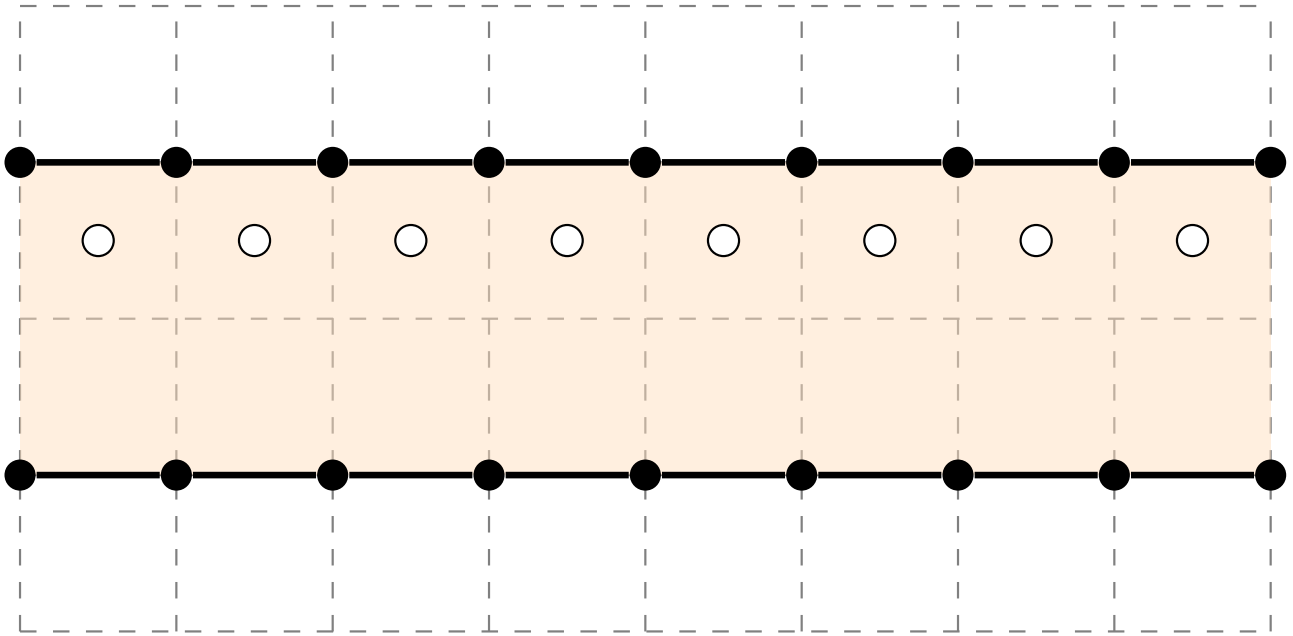
- For a $\mathrm{PM3SAT}$-formula F, our construction resembles its graph.

Tunnels & Pushes

- For a PM3SAT-formula F, our construction resembles its graph.
- White vertices always cost at least 1 to be rounded.
- If F is satisfiable, no black vertex needs to be moved.

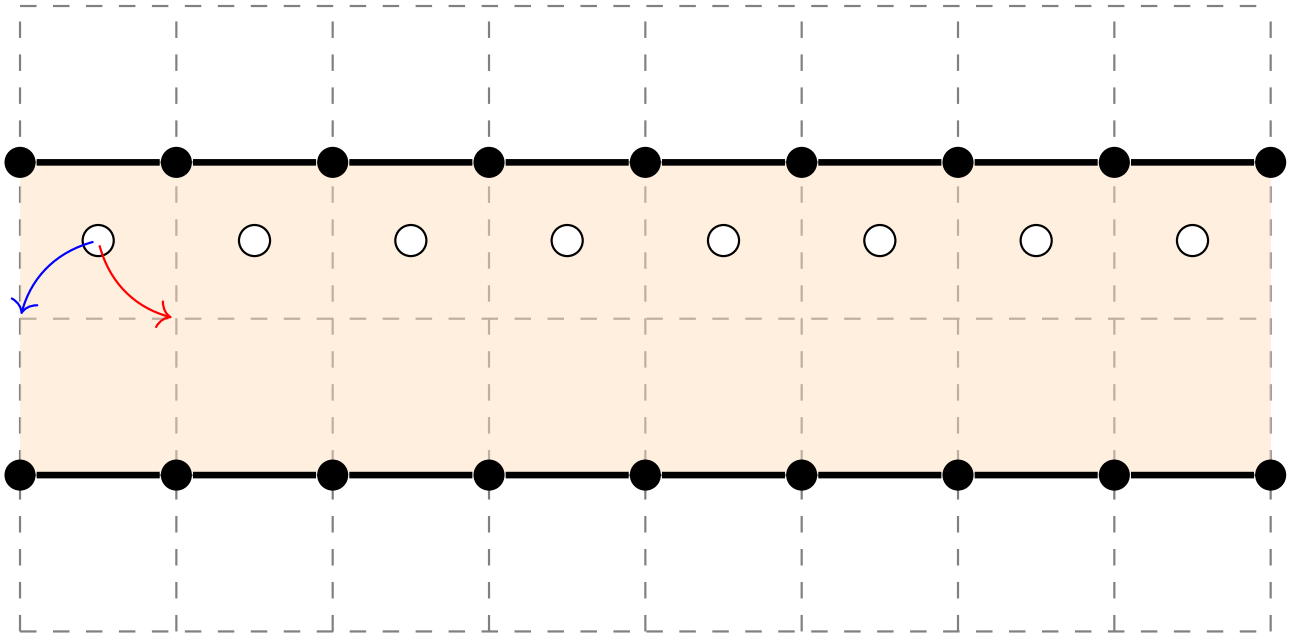
Tunnels & Pushes

- For a PM3SAT-formula F, our construction resembles its graph.
- White vertices always cost at least 1 to be rounded.
- If F is satisfiable, no black vertex needs to be moved.
- Edges form tunnels



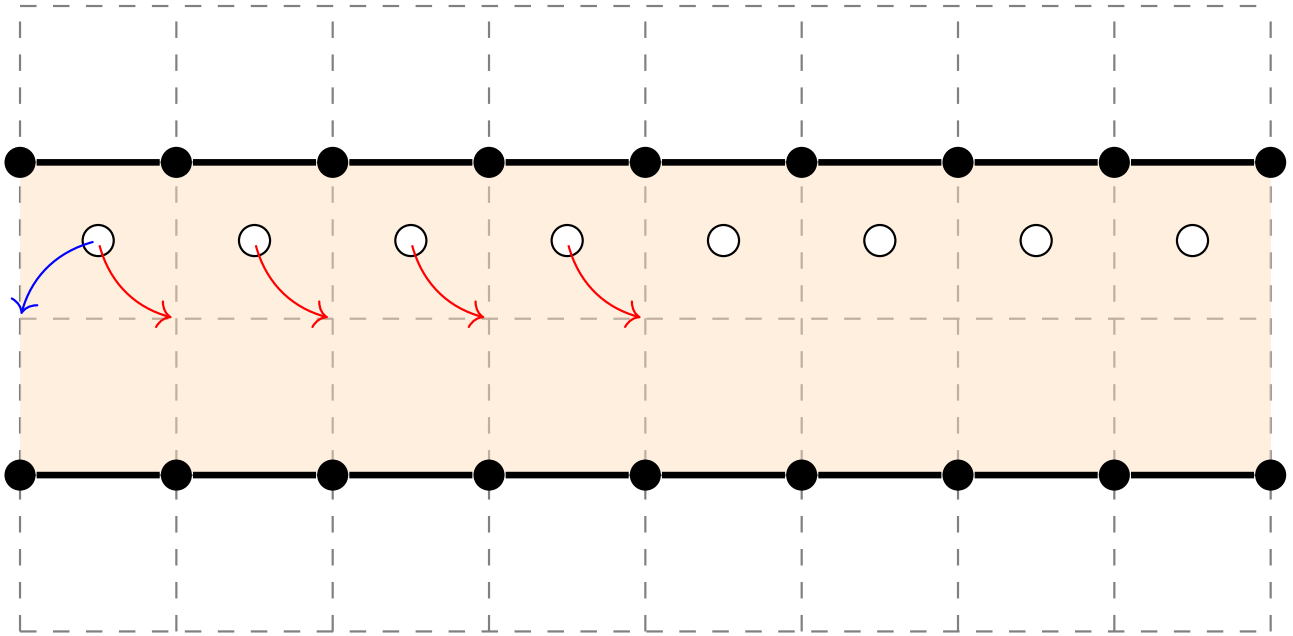
Tunnels & Pushes

- For a PM3SAT-formula F, our construction resembles its graph.
- White vertices always cost at least 1 to be rounded.
- If F is satisfiable, no black vertex needs to be moved.
- Edges form tunnels



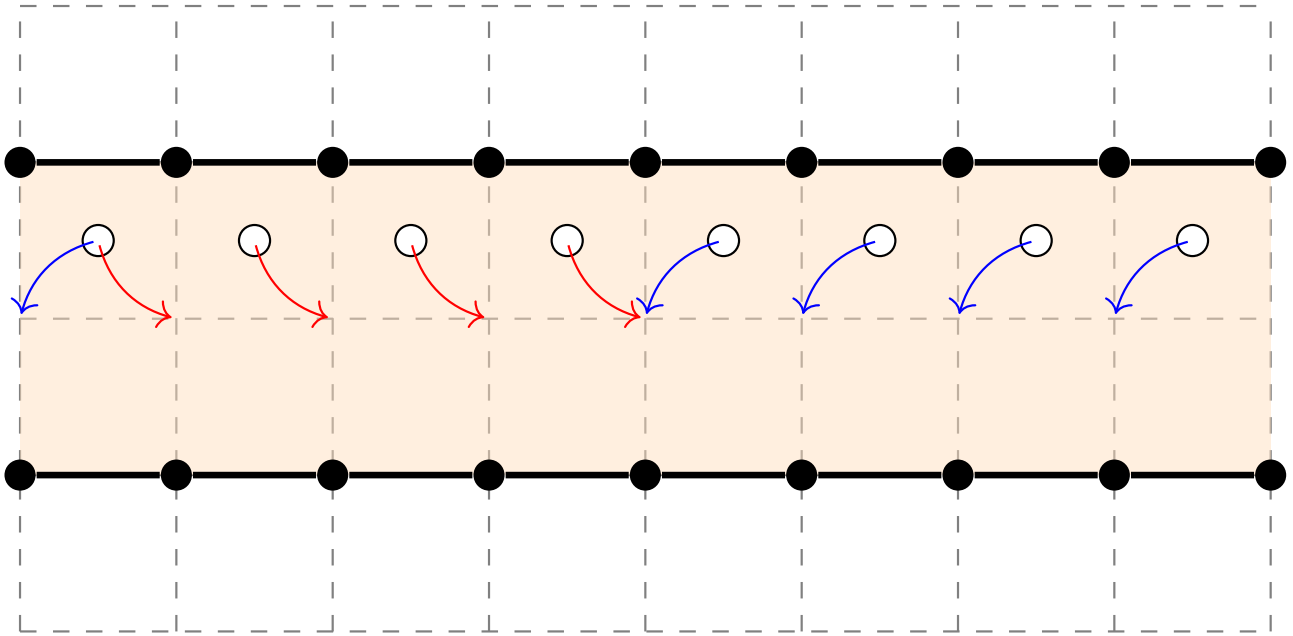
Tunnels & Pushes

- For a PM3SAT-formula F, our construction resembles its graph.
- White vertices always cost at least 1 to be rounded.
- If F is satisfiable, no black vertex needs to be moved.
- Edges form tunnels that transmit pushes.



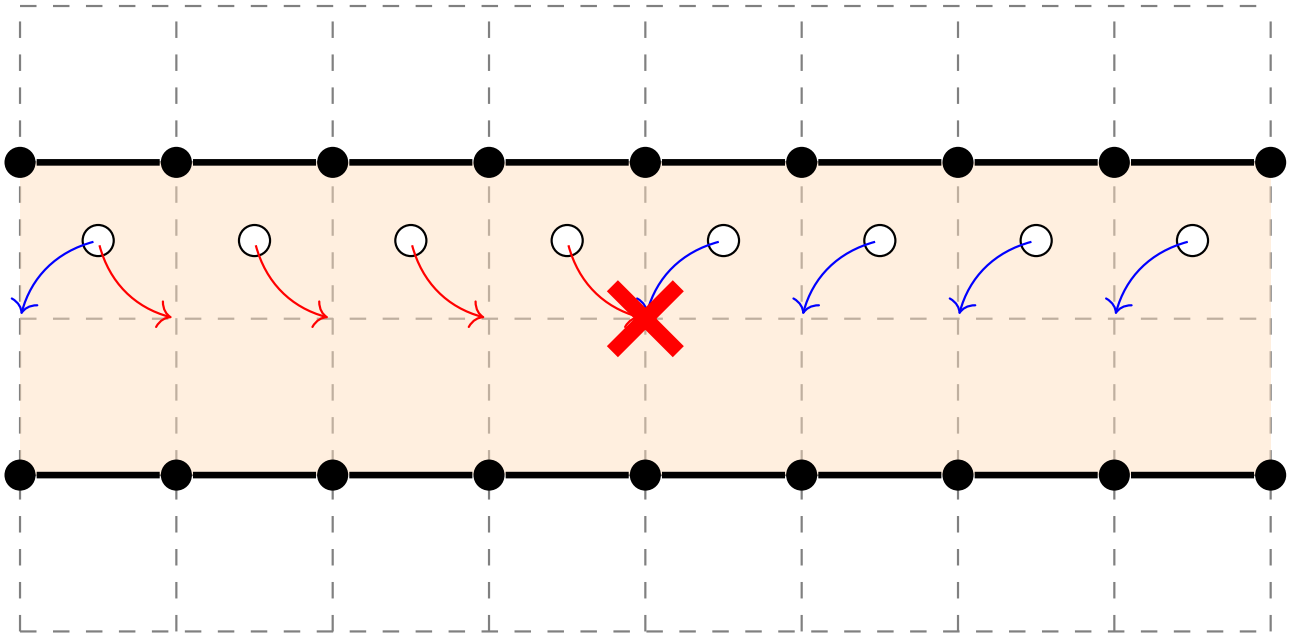
Tunnels & Pushes

- For a PM3SAT-formula F, our construction resembles its graph.
- White vertices always cost at least 1 to be rounded.
- If F is satisfiable, no black vertex needs to be moved.
- Edges form tunnels that transmit pushes.

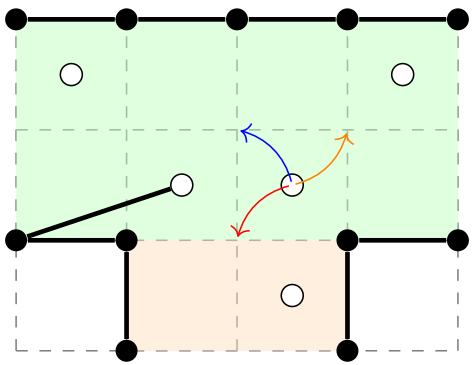


Tunnels & Pushes

- For a PM3SAT-formula F, our construction resembles its graph.
- White vertices always cost at least 1 to be rounded.
- If F is satisfiable, no black vertex needs to be moved.
- Edges form tunnels that transmit pushes.
- Topological safety ensures consistency of transmission.

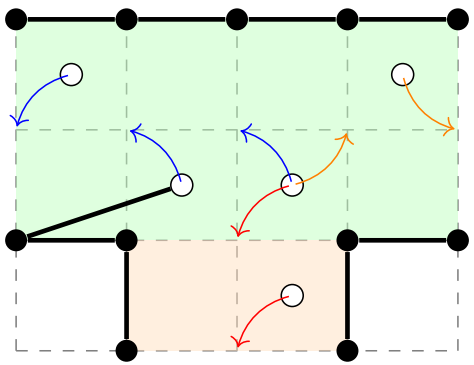


Clauses



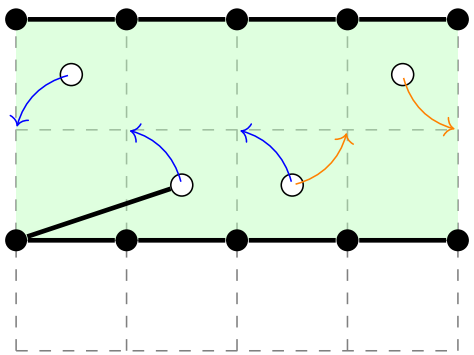
- At the center, there is a decider vertex with (up to) three possibible target grid points.

Clauses



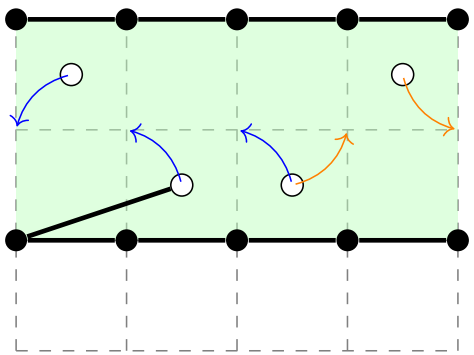
- At the center, there is a decider vertex with (up to) three possibible target grid points.
- Following one arrow, rounding generates pushes.

Clauses



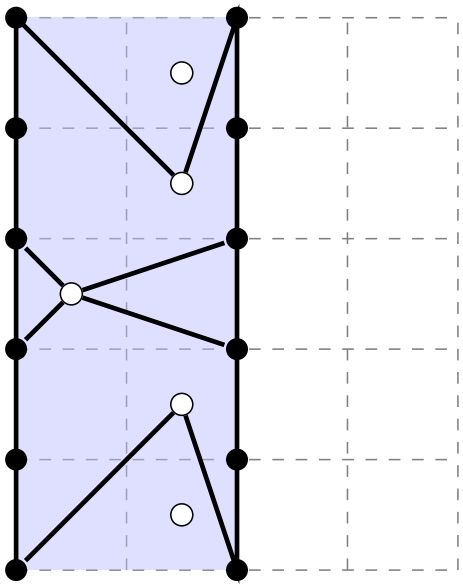
- At the center, there is a decider vertex with (up to) three possibible target grid points.
- Following one arrow, rounding generates pushes.
- Blocking the bottom tunnel gives clause-gadgets for two variables.

Clauses



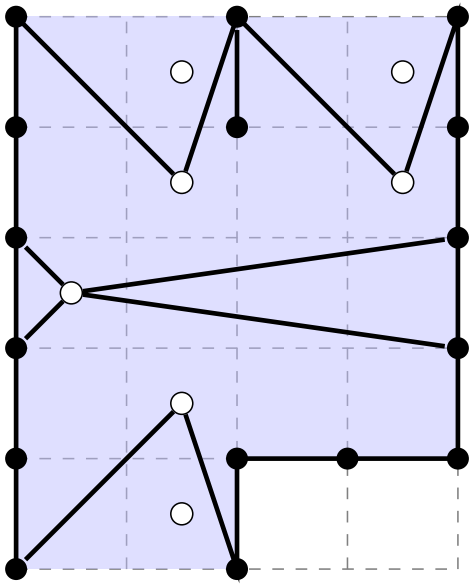
- At the center, there is a decider vertex with (up to) three possibible target grid points.
- Following one arrow, rounding generates pushes.
- Blocking the bottom tunnel gives clause-gadgets for two variables.
- All-unnegated gadgets are constructed mirroring at a horizontal line.

Variables



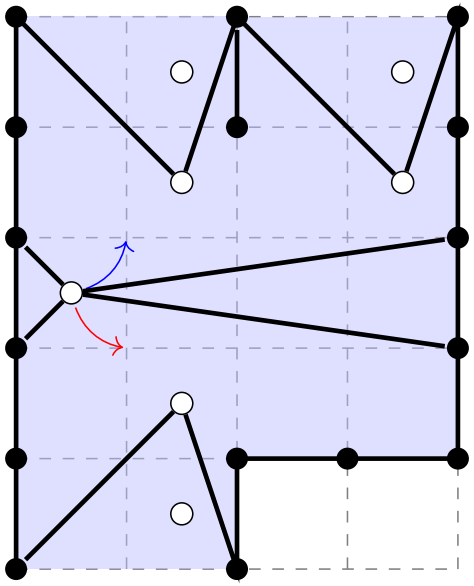
- Has tunnel connections for negated and unnegated occurances.

Variables



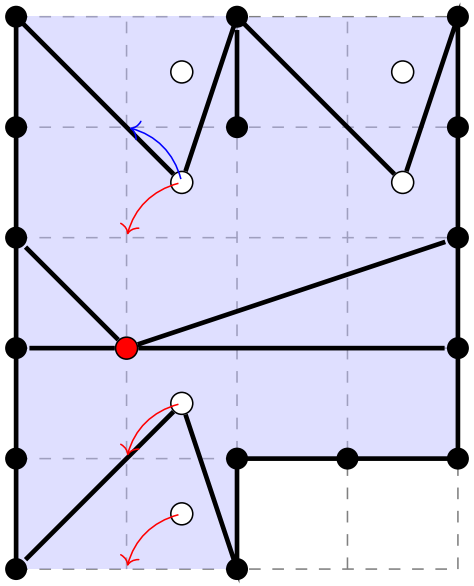
- Has tunnel connections for negated and unnegated occurances.
- Grows horizontally with number of occurances.

Variables



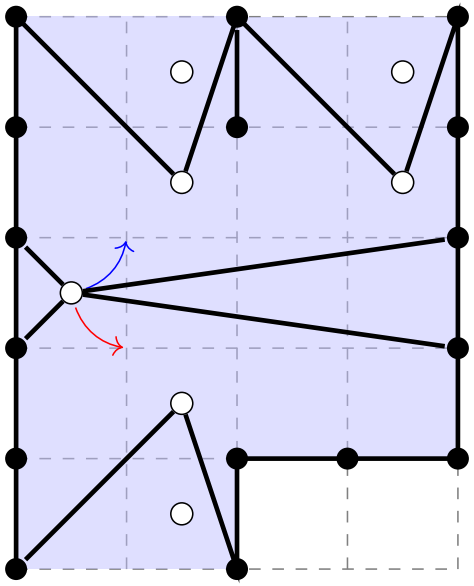
- Has tunnel connections for negated and unnegated occurances.
- Grows horizontally with number of occurances.
- At the left wall, there is an assignment vertex.

Variables



- Has tunnel connections for negated and unnegated occurances.
- Grows horizontally with number of occurances.
- At the left wall, there is an assignment vertex.
- Following one arrow blocks tunnels on this side and creates pushes.

Variables



- Has tunnel connections for negated and unnegated occurances.
- Grows horizontally with number of occurances.
- At the left wall, there is an assignment vertex.
- Following one arrow blocks tunnels on this side and creates pushes.
- Moving the assignment vertex up equals a TRUE-assignment, FALSE otherwise.

Hardness Proof

Theorem

Cost-bound Topologically Safe Snapping *is $\mathcal{NP}$-complete.*

Sketch of proof:

Hardness Proof

Theorem

Cost-bound Topologically Safe Snapping *is $\mathcal{NP}$-complete.*

Sketch of proof:

- Combine gadgets according to formula-graphs structure.

Hardness Proof

Theorem

Cost-bound Topologically Safe Snapping *is $\mathcal{NP}$-complete.*

Sketch of proof:

- Combine gadgets according to formula-graphs structure.
- Cost-bound $c_{\min}$ equals number of white vertices.

Hardness Proof

Theorem

Cost-bound Topologically Safe Snapping *is $\mathcal{NP}$-complete.*

Sketch of proof:

- Combine gadgets according to formula-graphs structure.
- Cost-bound $c_{\min}$ equals number of white vertices.
- If total movement cost equals $c_{\min}$, truth-assignment is obtained from assignment vertices.

Hardness Proof

Theorem

Cost-bound Topologically Safe Snapping *is $\mathcal{NP}$-complete.*

Sketch of proof:

- Combine gadgets according to formula-graphs structure.
- Cost-bound $c_{\min}$ equals number of white vertices.
- If total movement cost equals $c_{\min}$, truth-assignment is obtained from assignment vertices.
- If the formula is unsatisfiable, at least one black vertex has to be moved $\Rightarrow$ $c_{\min}$ is exceeded.

Other results

Corollary

Topologically Safe Snapping *is also $\mathcal{NP}$-hard when using* **Euclidean** *distance.*

Other results

Corollary

Topologically Safe Snapping *is also $\mathcal{NP}$-hard when using* **Euclidean** *distance. In this case it is also $\mathcal{NP}$-hard to minimize the* **maximum** *movement instead of the sum.*

Other results

Corollary

Topologically Safe Snapping *is also $\mathcal{NP}$-hard when using* **Euclidean** *distance. In this case it is also $\mathcal{NP}$-hard to minimize the* **maximum** *movement instead of the sum.*

Corollary

Euclidean Topologically Safe Snapping *with the objective to minimize* **maximum** *movement is $\mathcal{APX}$-hard.*

Integer Linear Program

Overview

Things to handle:

Overview

Things to handle:

- Unique vertex coordinates

Overview

Things to handle:

- Unique vertex coordinates (very simple)

Overview

Things to handle:

- Unique vertex coordinates (very simple)
- Planarity

Overview

Things to handle:

- Unique vertex coordinates (very simple)
- Planarity
- Embeddings

Overview

Things to handle:

- Unique vertex coordinates (very simple)
- Planarity
- Embeddings

Basics:

Overview

Things to handle:

- Unique vertex coordinates (very simple)
- Planarity
- Embeddings

Basics:

- x_v, y_v are output coordinates.

Overview

Things to handle:

- Unique vertex coordinates (very simple)
- Planarity
- Embeddings

Basics:

- x_v, y_v are output coordinates.
- Objective function:

$$\textsc{Minimize} \sum_{v \in V} (|x_v - X_v| + |y_v - Y_v|)$$

Overview

Things to handle:

- Unique vertex coordinates (very simple)
- Planarity
- Embeddings

Basics:

- x_v, y_v are output coordinates.
- Objective function:

$$\textsc{Minimize} \sum_{v \in V} (|x_v - X_v| + |y_v - Y_v|)$$

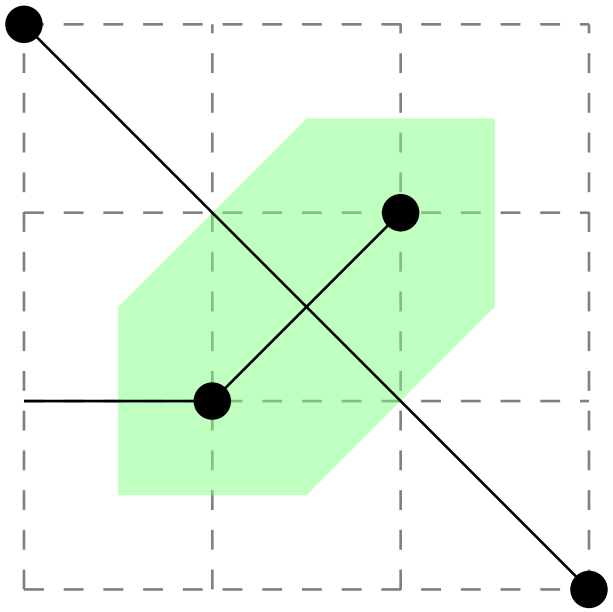
- Constraint: distinct vertex coordinates.

Planarity

- Similar to Metro-Map Drawing by Nöllenburg & Wolff. [GD '05]

Planarity

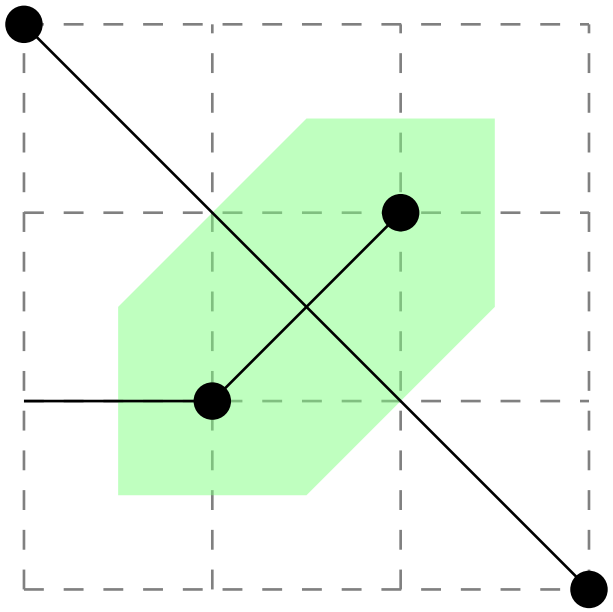
- Similar to Metro-Map Drawing by Nöllenburg & Wolff. [GD '05]
- **Idea:** every edge has some D_{min}-neighborhood that only incident edges are allowed to intersect.



Octilinear, $D_{\text{min}} = 0.5$

Planarity

- Similar to Metro-Map Drawing by Nöllenburg & Wolff. [GD '05]
- **Idea:** every edge has some D_{min}-neighborhood that only incident edges are allowed to intersect.

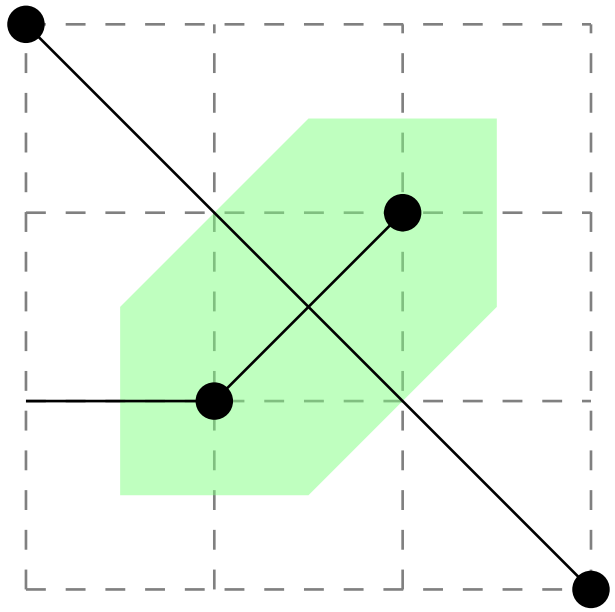


Octilinear, $D_{\text{min}} = 0.5$

- We consider any possible direction (not only octilinear ones).

Planarity

- Similar to Metro-Map Drawing by Nöllenburg & Wolff. [GD '05]
- **Idea:** every edge has some D_{min}-neighborhood that only incident edges are allowed to intersect.



Octilinear, $D_{\text{min}} = 0.5$

- We consider any possible direction (not only octilinear ones).
- According to bounding box size:

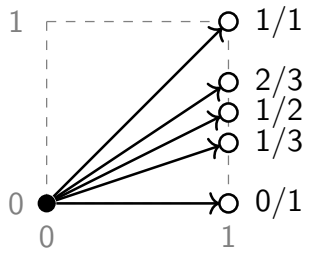
$$D_{\text{min}} = \frac{1}{\max\{X_{\text{max}}, Y_{\text{max}}\} + 1}$$

Directions

- Generated using the Farey sequence:

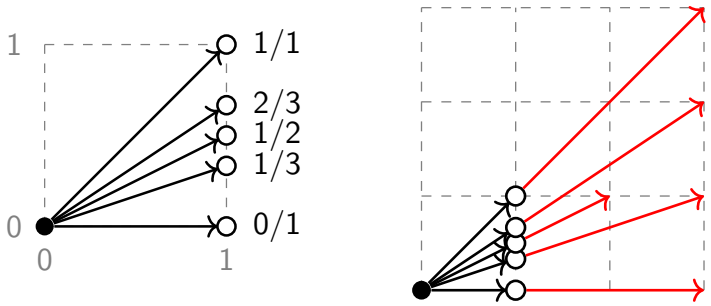
Directions

- Generated using the Farey sequence:



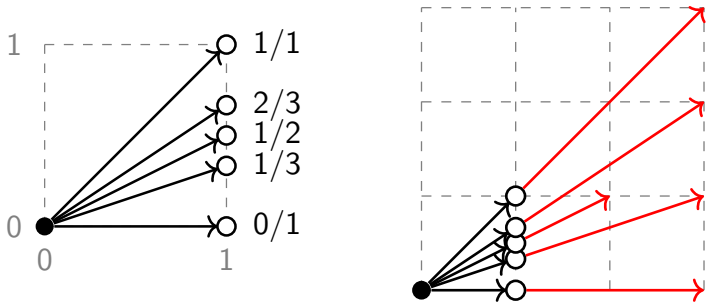
Directions

- Generated using the Farey sequence:



Directions

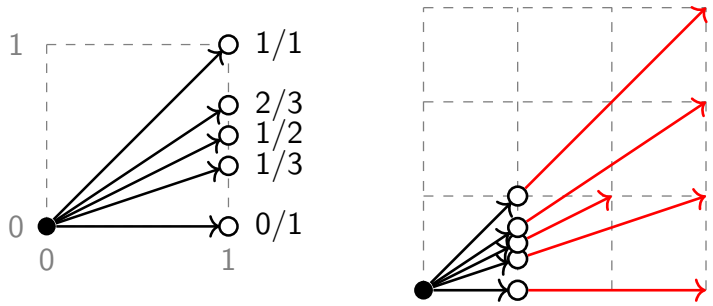
- Generated using the Farey sequence:



- Inside $[-k, k] \times [-k, k]$ area, there are $\Theta(k^2)$ directions to consider.

Directions

- Generated using the Farey sequence:



- Inside $[-k, k] \times [-k, k]$ area, there are $\Theta(k^2)$ directions to consider.
- Consider them to be ordered counter-clockwise.

Embeddings

- Circular order of neighbors arround any vertex must not change.

Embeddings

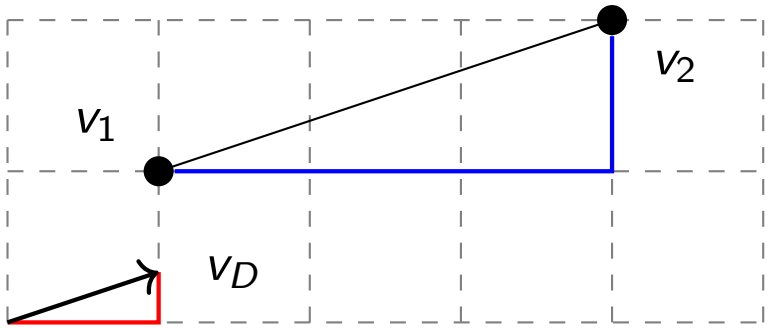
- Circular order of neighbors arround any vertex must not change.
- **Idea:** for every vertex-neighbor pair, detect direction of that edge.

Embeddings

- Circular order of neighbors arround any vertex must not change.
- **Idea:** for every vertex-neighbor pair, detect direction of that edge.
- Compare direction slopes to edge slope.

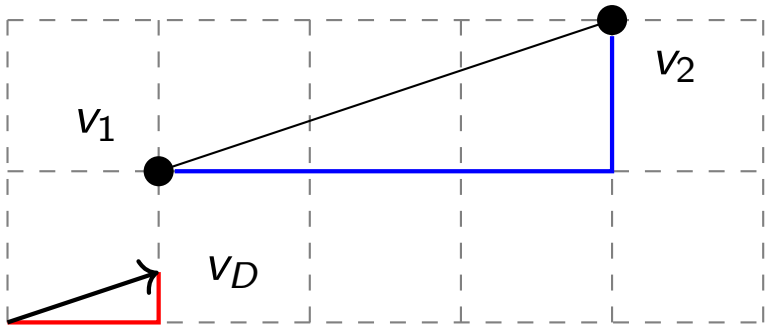
Embeddings

- Circular order of neighbors arround any vertex must not change.
- **Idea:** for every vertex-neighbor pair, detect direction of that edge.
- Compare direction slopes to edge slope.



Embeddings

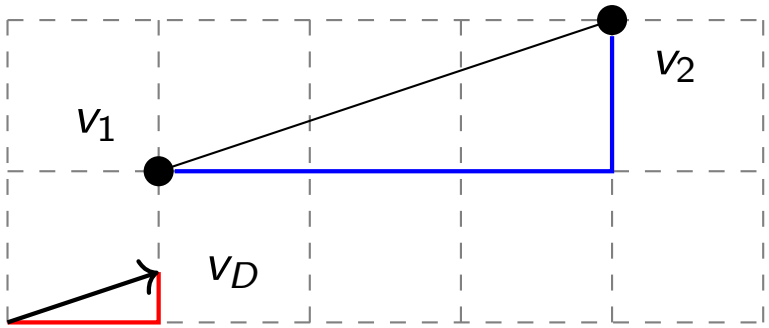
- Circular order of neighbors arround any vertex must not change.
- **Idea:** for every vertex-neighbor pair, detect direction of that edge.
- Compare direction slopes to edge slope.



- Map edges to directions

Embeddings

- Circular order of neighbors arround any vertex must not change.
- **Idea:** for every vertex-neighbor pair, detect direction of that edge.
- Compare direction slopes to edge slope.



- Map edges to directions and compare the ordering of those directions to the given embedding.

Integer Linear Program

Theorem

This ILP solves Topologically Safe Snapping.

Integer Linear Program

Theorem

This ILP solves Topologically Safe Snapping.

- In practice, our model easily becomes too large to solve (in reasonable time).

Integer Linear Program

Theorem

This ILP solves Topologically Safe Snapping.

- In practice, our model easily becomes too large to solve (in reasonable time).
- We use delayed constraint generation to iteratively improve our model.

Integer Linear Program

Theorem

This ILP solves Topologically Safe Snapping.

- In practice, our model easily becomes too large to solve (in reasonable time).
- We use delayed constraint generation to iteratively improve our model.
- We generate most constraints on demand:

Integer Linear Program

Theorem

This ILP solves Topologically Safe Snapping.

- In practice, our model easily becomes too large to solve (in reasonable time).
- We use delayed constraint generation to iteratively improve our model.
- We generate most constraints on demand: first iteration is simple rounding (with unique coordinates).

Experimental Evaluation

The Setup

- Using the JAVA bindings for IBM CPLEX.

The Setup

- Using the JAVA bindings for IBM CPLEX.
- Test system: Linux server with 16 cores (2666 MHz, 4 MB cache), 16 GB main memory.

The Setup

- Using the JAVA bindings for IBM CPLEX.
- Test system: Linux server with 16 cores (2666 MHz, 4 MB cache), 16 GB main memory.
- Numbers of rows & columns before CPLEX presolving.

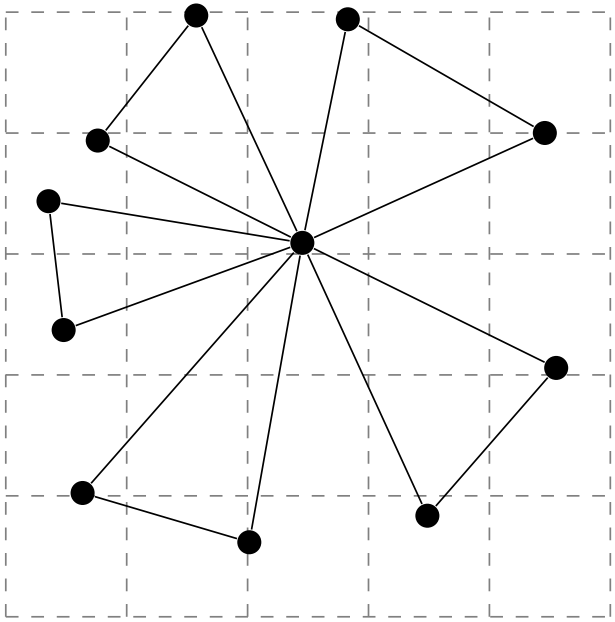
The Setup

- Using the JAVA bindings for IBM CPLEX.
- Test system: Linux server with 16 cores (2666 MHz, 4 MB cache), 16 GB main memory.
- Numbers of rows & columns before CPLEX presolving.
- Runtime in wall-clock time.

The Setup

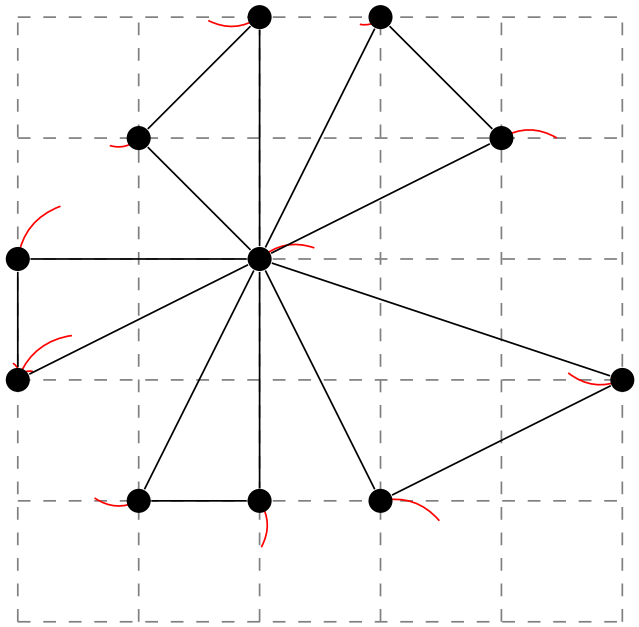
- Using the JAVA bindings for IBM CPLEX.
- Test system: Linux server with 16 cores (2666 MHz, 4 MB cache), 16 GB main memory.
- Numbers of rows & columns before CPLEX presolving.
- Runtime in wall-clock time.
- For delayed constraint generation, time is accumulated total.

The Good



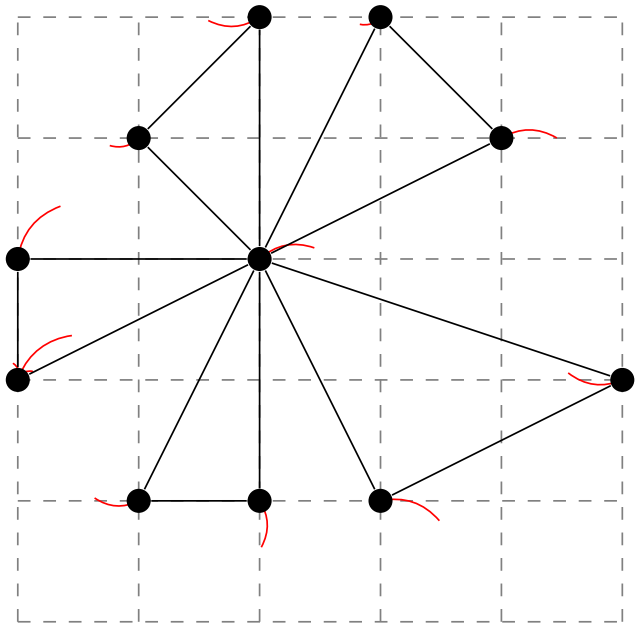
	Full	Delayed
rows		
cols		
time		

The Good



	Full	Delayed
rows		
cols		
time		

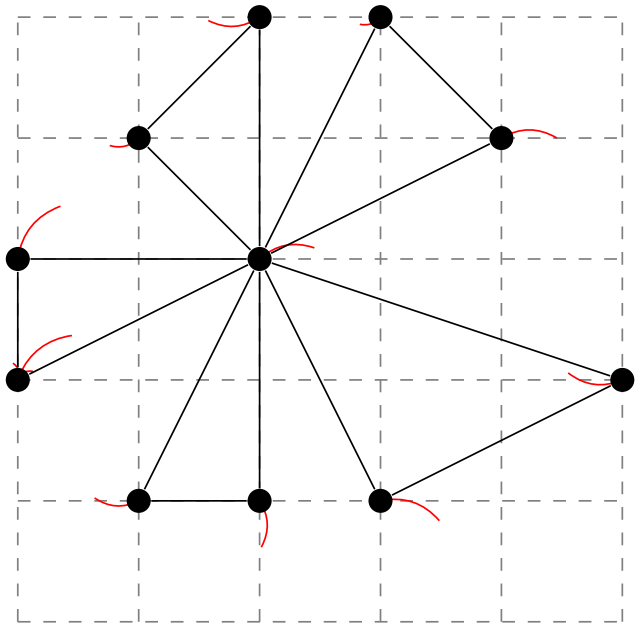
The Good



- Even small examples take several seconds to solve.

	Full	Delayed
rows	42 699	
cols	11 300	
time	10.6 s	

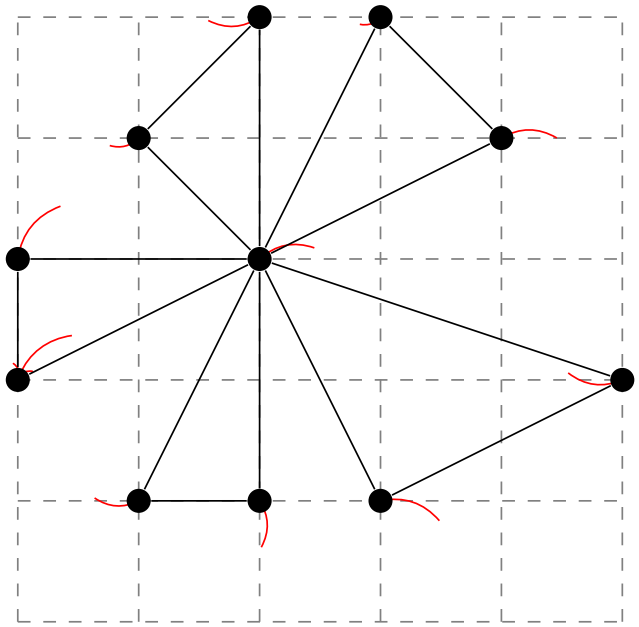
The Good



- Even small examples take several seconds to solve.
- This is a very simple example!

	Full	Delayed
rows	42 699	
cols	11 300	
time	10.6 s	

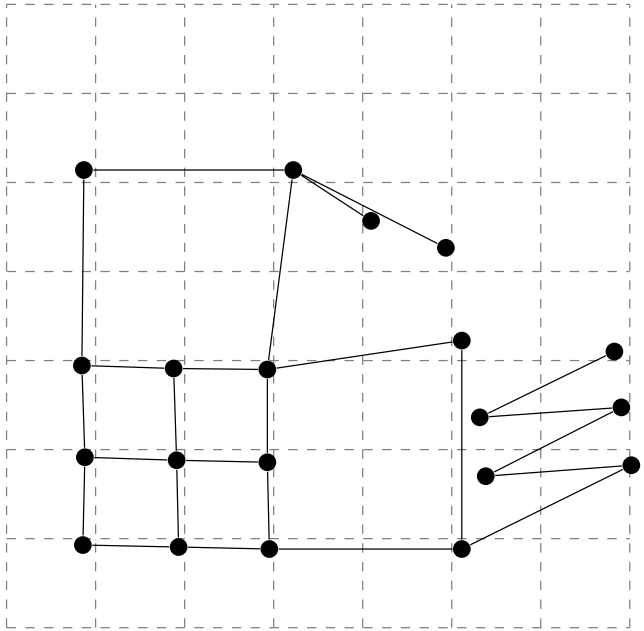
The Good



- Even small examples take several seconds to solve.
- This is a very simple example!
- Delayed constraint generation gives speed-up.

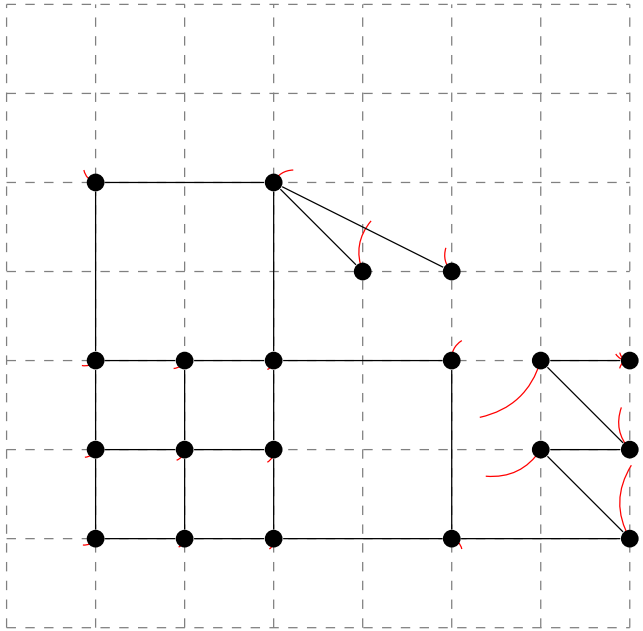
	Full	Delayed
rows	42 699	88
cols	11 300	110
time	10.6 s	0.5 s

The Bad



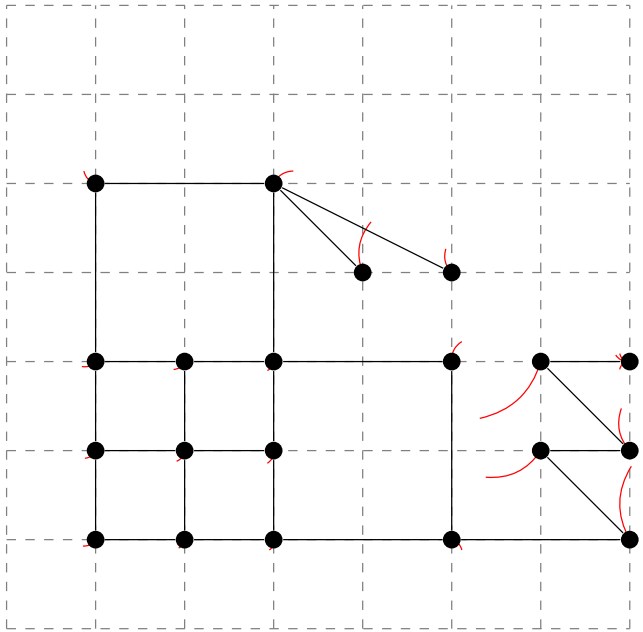
	Full	Delayed
rows		
cols		
time		

The Bad



	Full	Delayed
rows		
cols		
time		

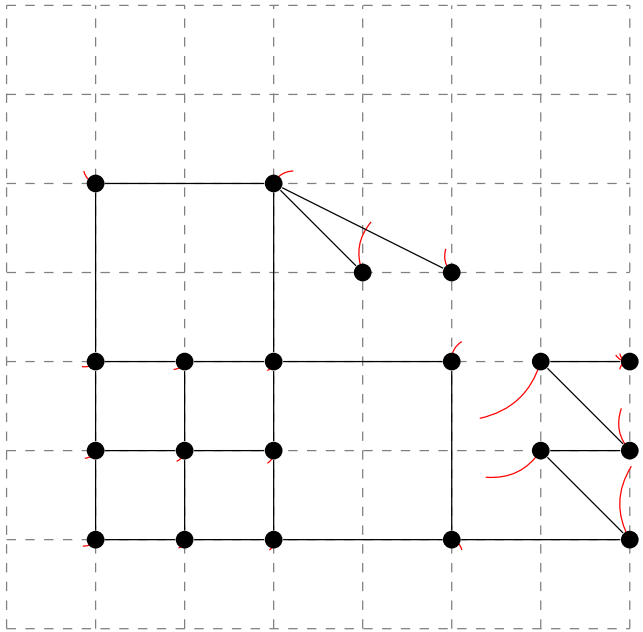
The Bad



- We have canceled this computation after 10 minutes using the full model.

	Full	Delayed
rows	323 441	
cols	82 816	
time	†	

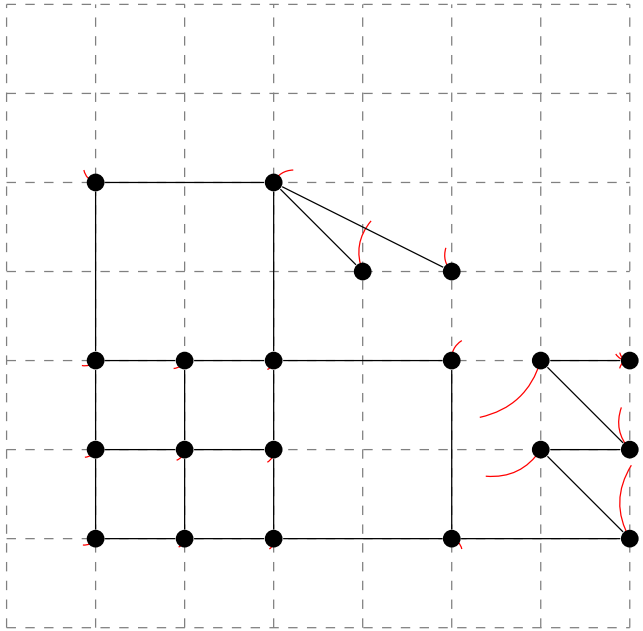
The Bad



- We have canceled this computation after 10 minutes using the full model.
- Delayed constraint generation did cut a lot of "trivial" constraints, but...

	Full	Delayed
rows	323 441	15 161
cols	82 816	4 044
time	†	

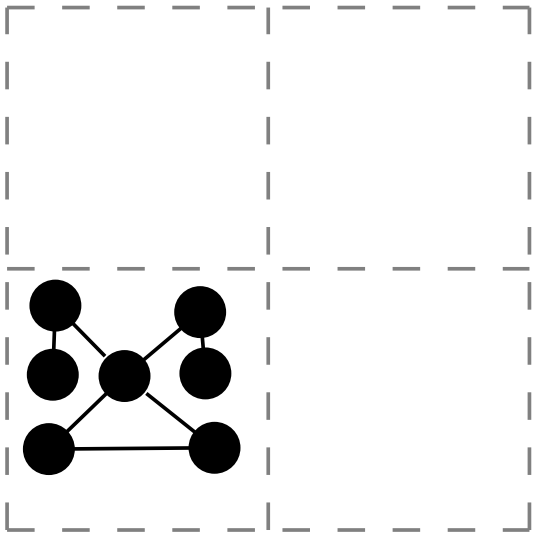
The Bad



- We have canceled this computation after 10 minutes using the full model.
- Delayed constraint generation did cut a lot of "trivial" constraints, but...
- ...waiting more than 3 minutes is too long for a graph on 20 vertices!

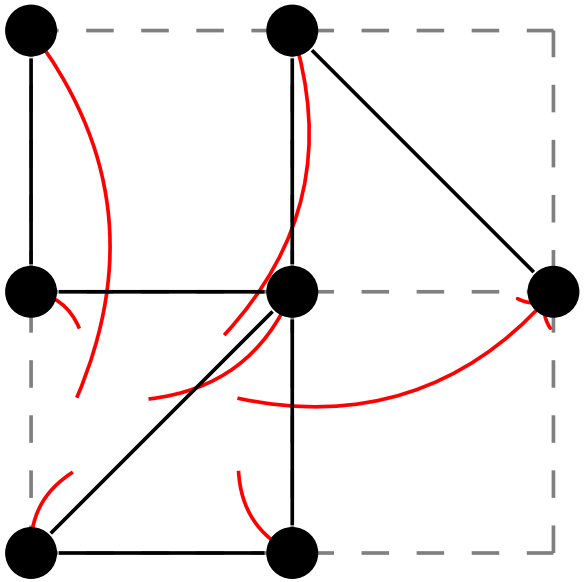
	Full	Delayed
rows	323 441	15 161
cols	82 816	4 044
time	†	211.6 s

The Ugly



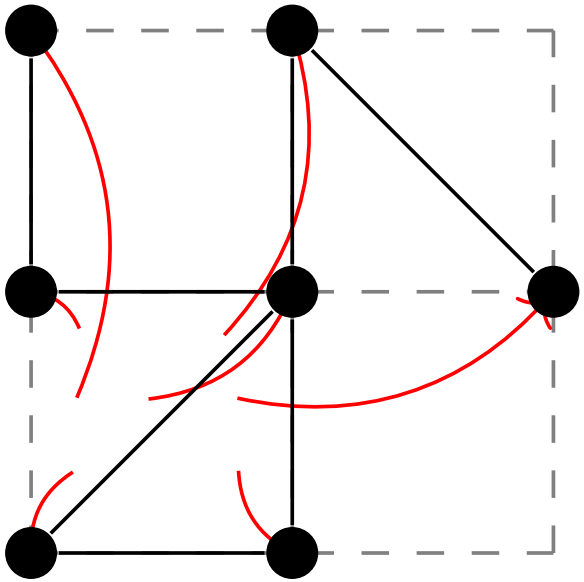
	Full	Delayed
rows		
cols		
time		

The Ugly



	Full	Delayed
rows		
cols		
time		

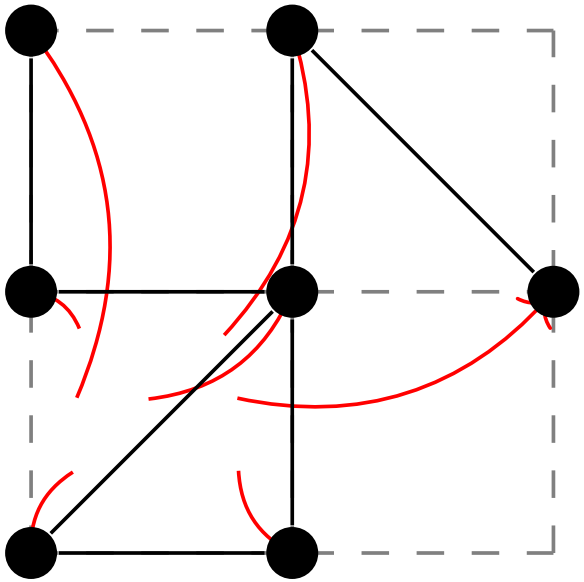
The Ugly



- Graph and bounding box are small, thus the model is small.

	Full	Delayed
rows	2 603	
cols	916	
time	4.8 s	

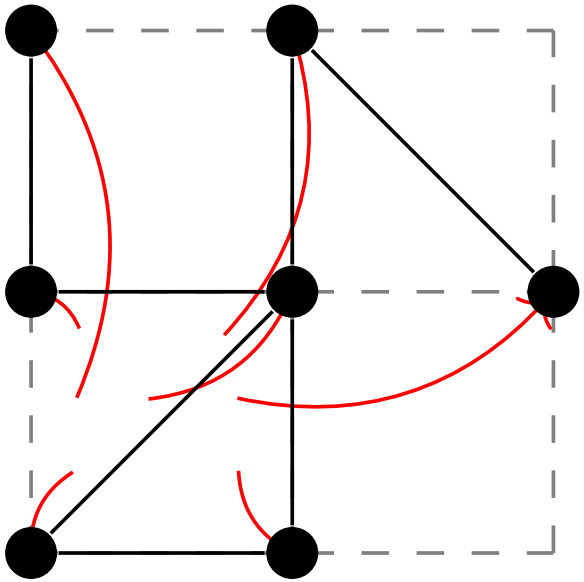
The Ugly



- Graph and bounding box are small, thus the model is small.
- Using delayed constraint generation did worsen runtime.

	Full	Delayed
rows	2 603	2 271
cols	916	816
time	4.8 s	20.2 s

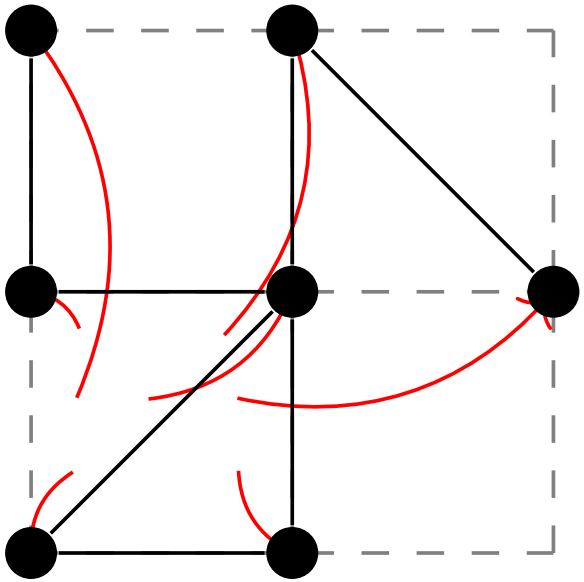
The Ugly



- Graph and bounding box are small, thus the model is small.
- Using delayed constraint generation did worsen runtime.
- Rounding this graph is very similar to finding a minimum-area drawing

	Full	Delayed
rows	2 603	2 271
cols	916	816
time	4.8 s	20.2 s

The Ugly



- Graph and bounding box are small, thus the model is small.
- Using delayed constraint generation did worsen runtime.
- Rounding this graph is very similar to finding a minimum-area drawing, which is also $\mathcal{NP}$-hard.

	Full	Delayed
rows	2 603	2 271
cols	916	816
time	4.8 s	20.2 s

Conclusion

What we did:

Conclusion

What we did:

- We introduce the problem Topologically Safe Snapping

Conclusion

What we did:

- We introduce the problem Topologically Safe Snapping
- and provide a proof that it is $\mathcal{NP}$-hard.

Conclusion

What we did:

- We introduce the problem Topologically Safe Snapping
- and provide a proof that it is $\mathcal{NP}$-hard.
- We give an integer linear program to solve it,

Conclusion

What we did:

- We introduce the problem Topologically Safe Snapping
- and provide a proof that it is $\mathcal{NP}$-hard.
- We give an integer linear program to solve it,
- that can be modified to find minimum-area drawings of graphs as well.

Conclusion

What we did:

- We introduce the problem Topologically Safe Snapping
- and provide a proof that it is $\mathcal{NP}$-hard.
- We give an integer linear program to solve it,
- that can be modified to find minimum-area drawings of graphs as well.

Open problems:

Conclusion

What we did:

- We introduce the problem Topologically Safe Snapping
- and provide a proof that it is $\mathcal{NP}$-hard.
- We give an integer linear program to solve it,
- that can be modified to find minimum-area drawings of graphs as well.

Open problems:

- Find better formulations for the constraints $\Rightarrow$ speed-up ILP.

Conclusion

What we did:

- We introduce the problem Topologically Safe Snapping
- and provide a proof that it is $\mathcal{NP}$-hard.
- We give an integer linear program to solve it,
- that can be modified to find minimum-area drawings of graphs as well.

Open problems:

- Find better formulations for the constraints $\Rightarrow$ speed-up ILP.
- Find some heuristic algorithm.

Conclusion

What we did:

- We introduce the problem Topologically Safe Snapping
- and provide a proof that it is $\mathcal{NP}$-hard.
- We give an integer linear program to solve it,
- that can be modified to find minimum-area drawings of graphs as well.

Open problems:

- Find better formulations for the constraints $\Rightarrow$ speed-up ILP.
- Find some heuristic algorithm.
- Questions about approximability remain open.