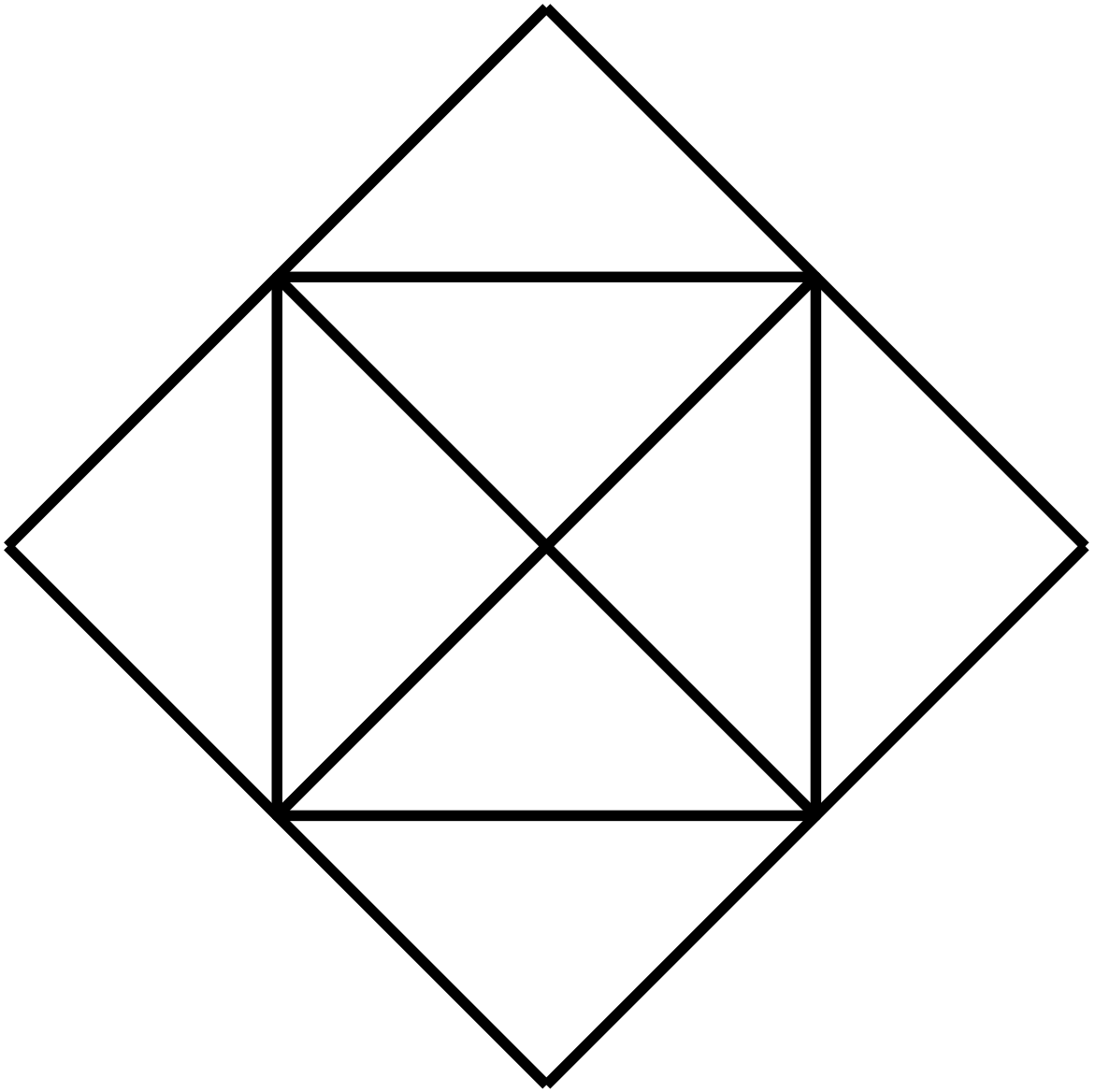
Fun with Discrete Math:
Eulerian and Hamiltonian Graphs

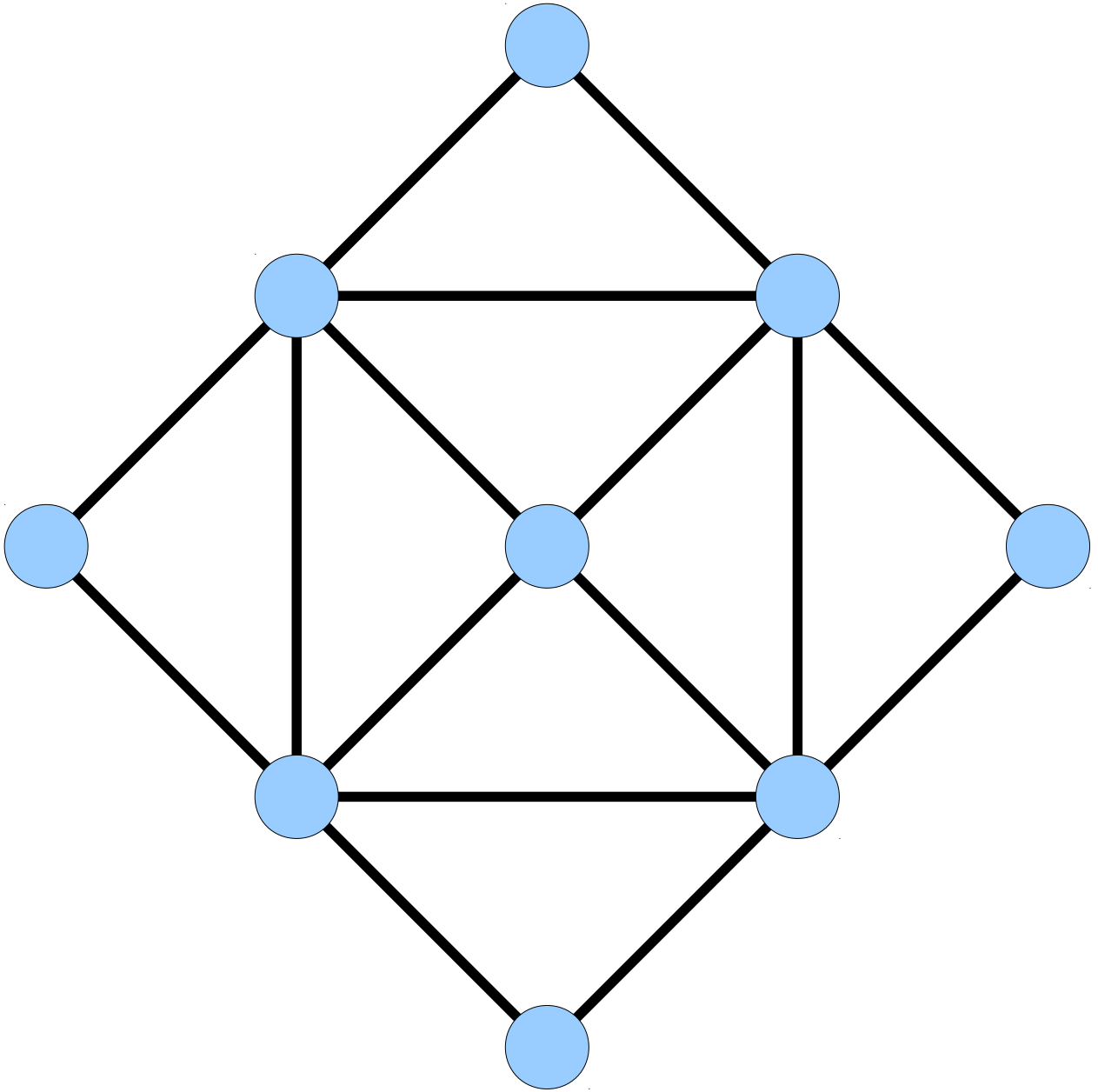
Today's lecture is all about having fun with the topics we've seen so far.

There's a question on PS5 about this material, but in a nifty applied context.

This is our last lecture on discrete structures. Starting Monday, we'll focus on computability theory.

Some Puzzles

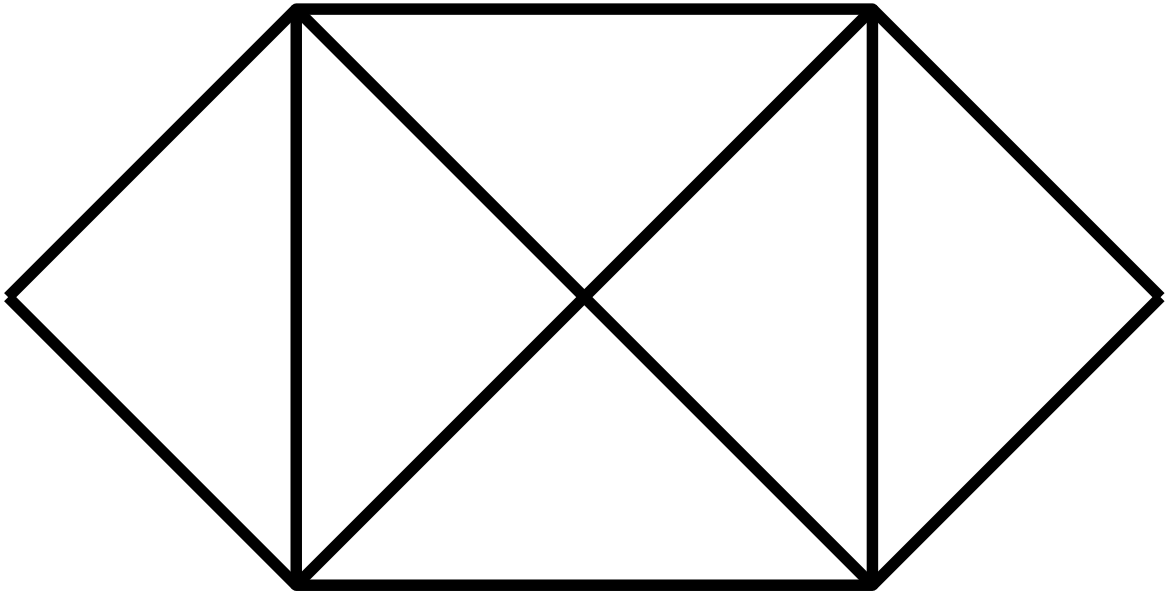


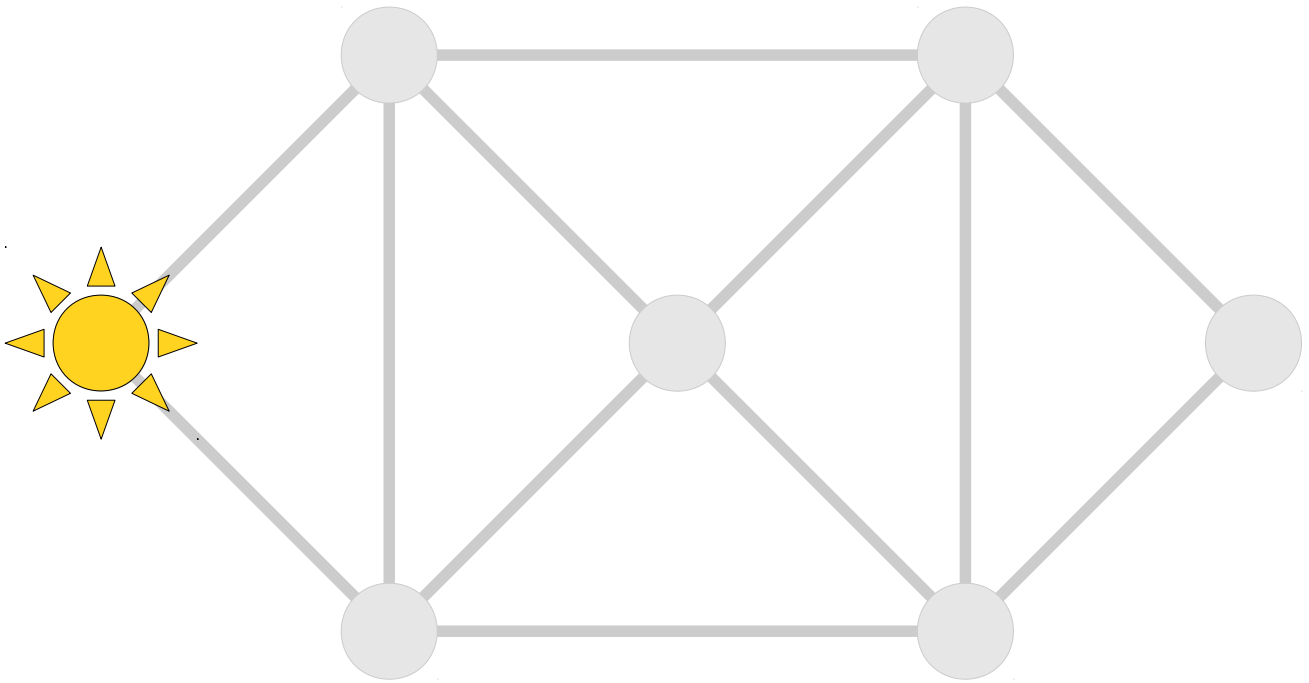


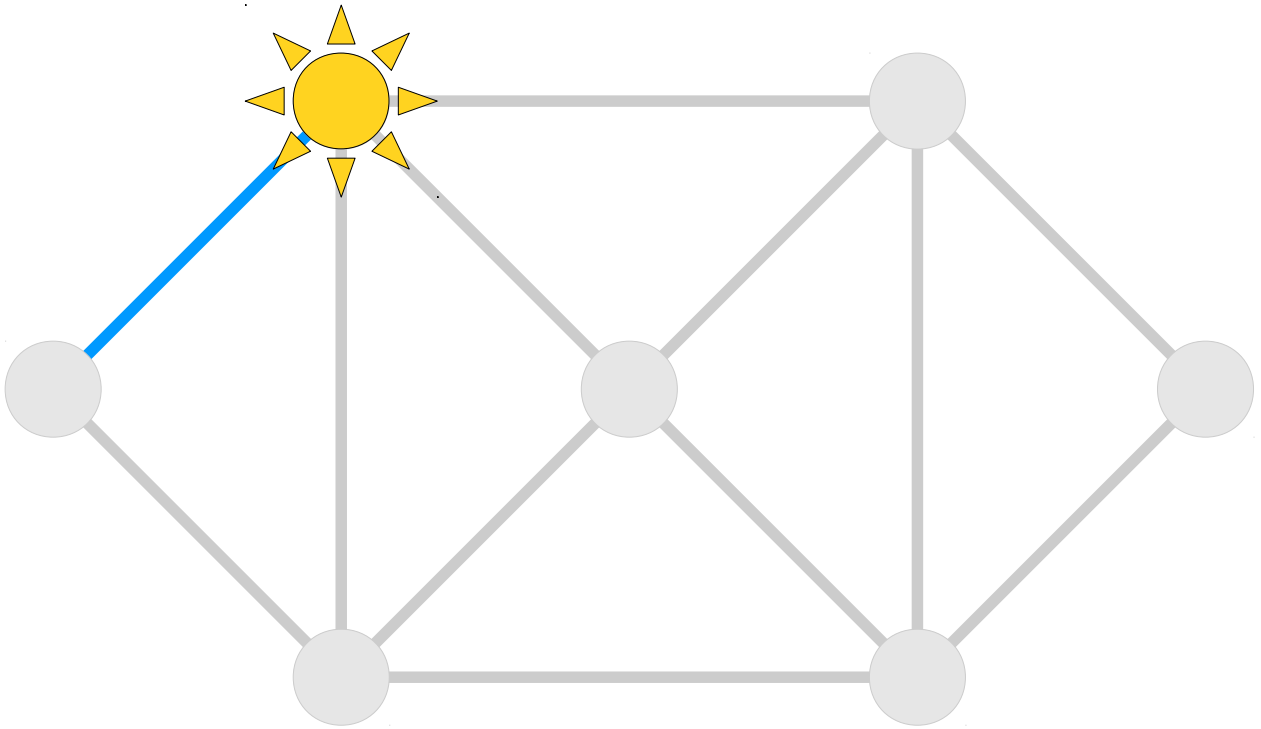
Eulerian Graphs

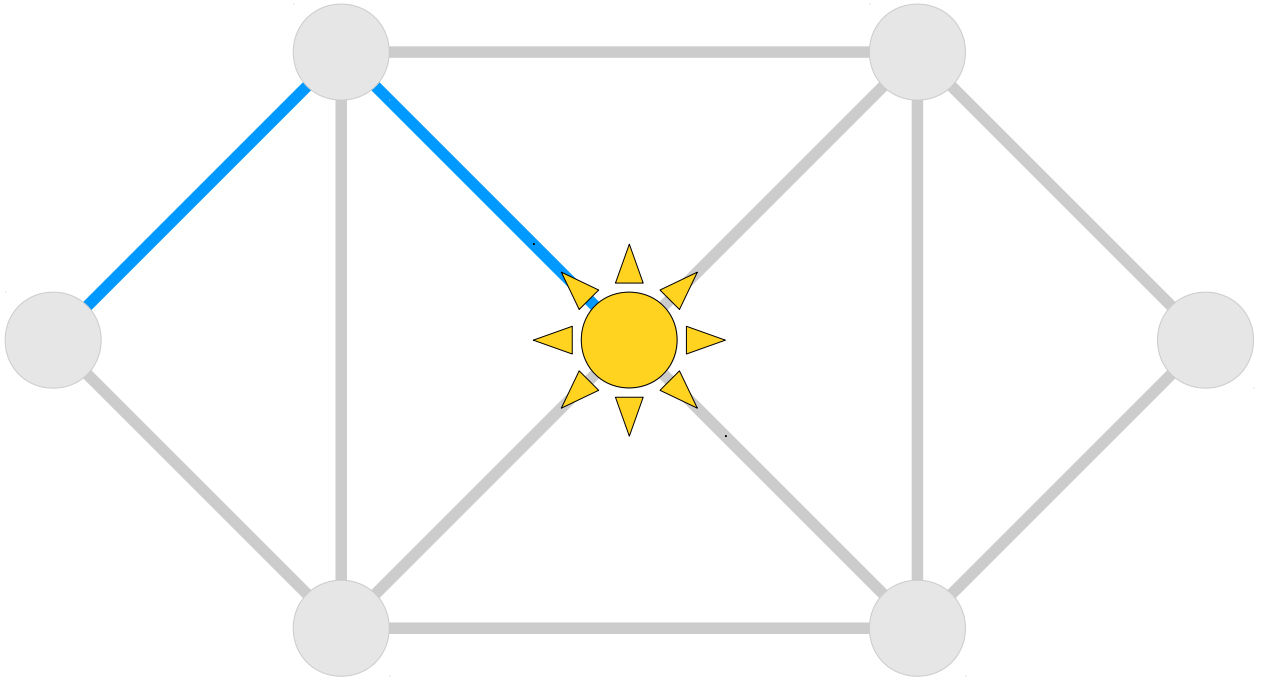
- An ***Eulerian circuit*** is a cycle in a connected graph G that passes through every edge in G exactly once.
- Some graphs have Eulerian circuits; others do not.
- An ***Eulerian graph*** is a connected graph that has an Eulerian circuit.

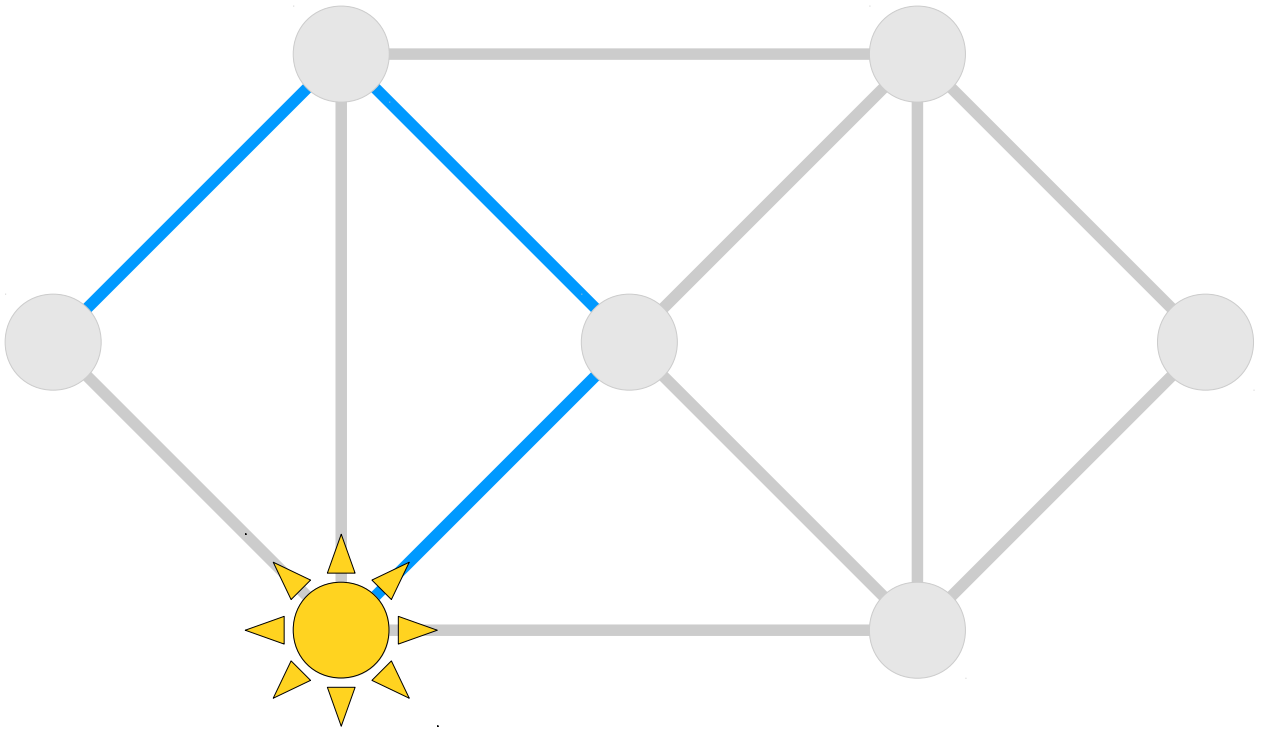
Question: Which graphs are Eulerian?

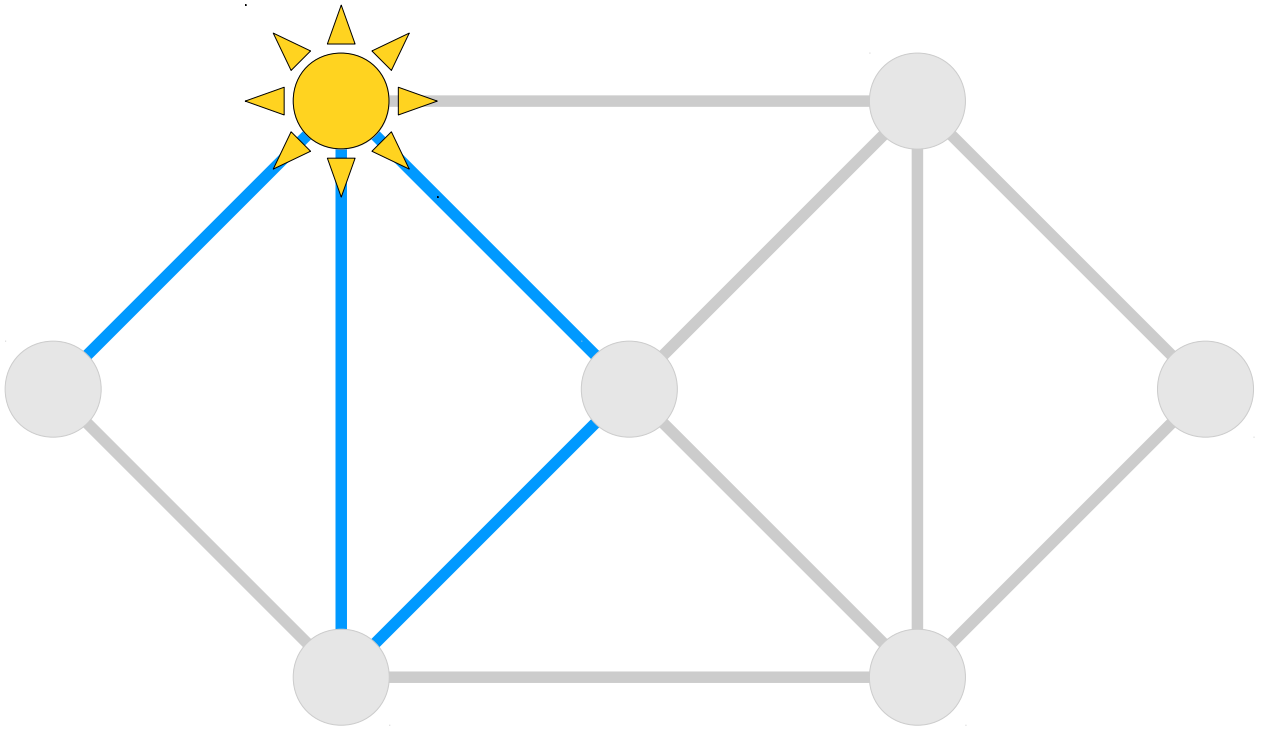


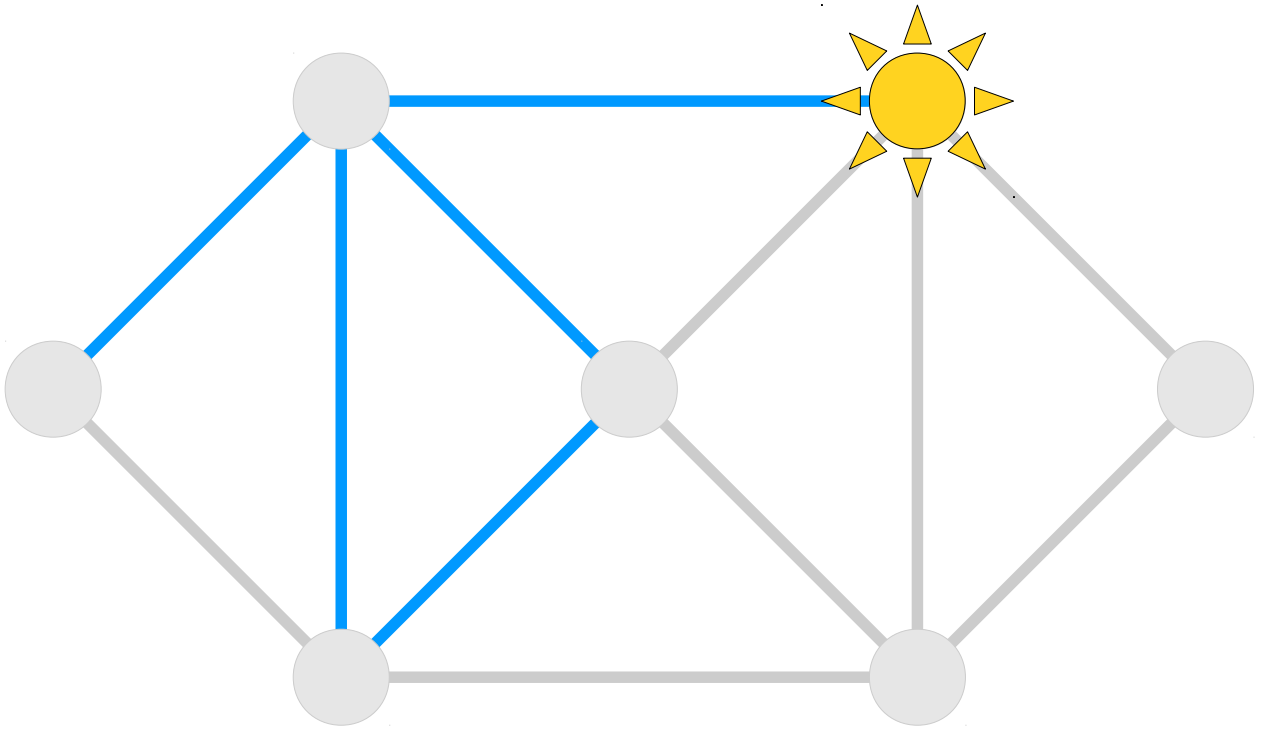


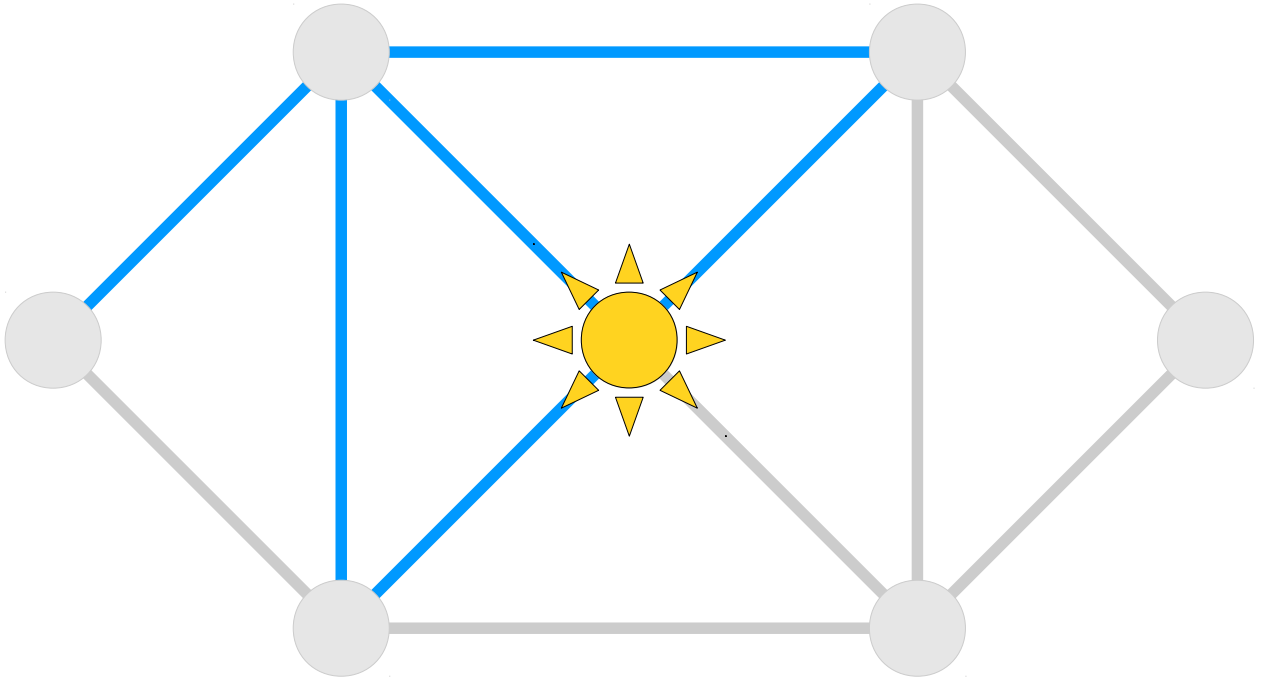


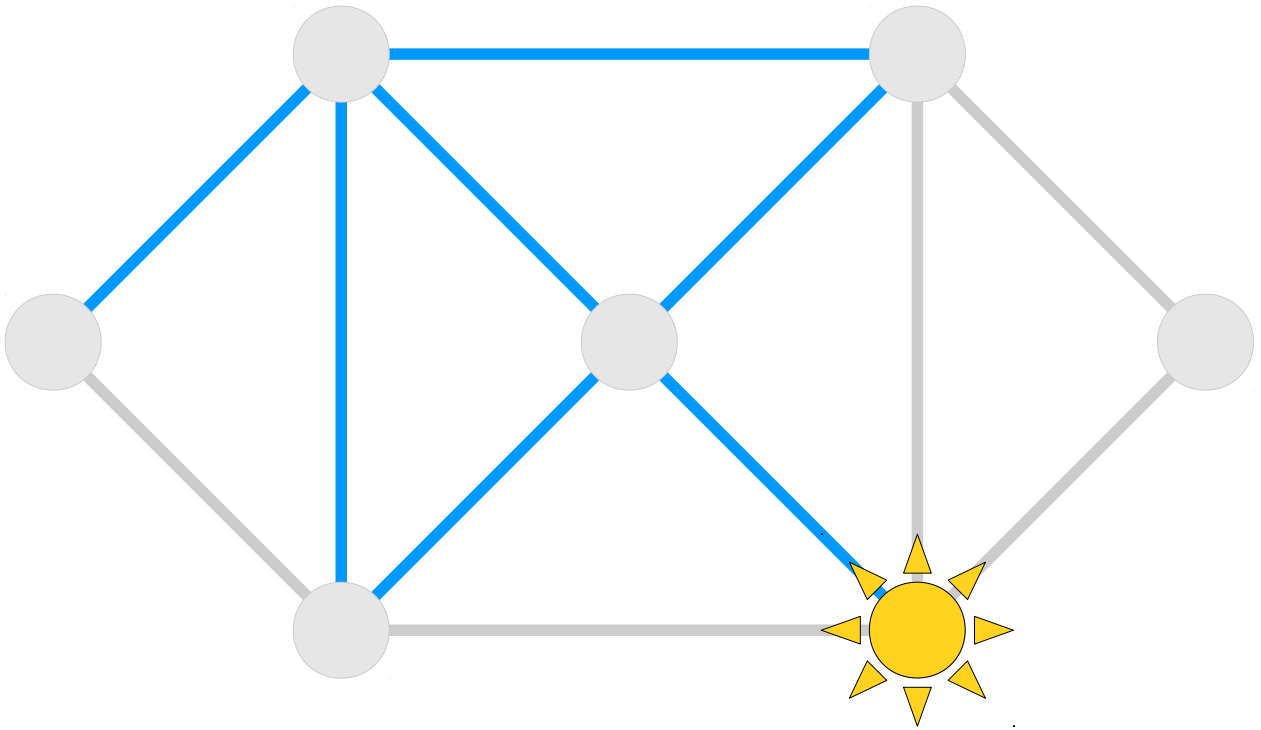


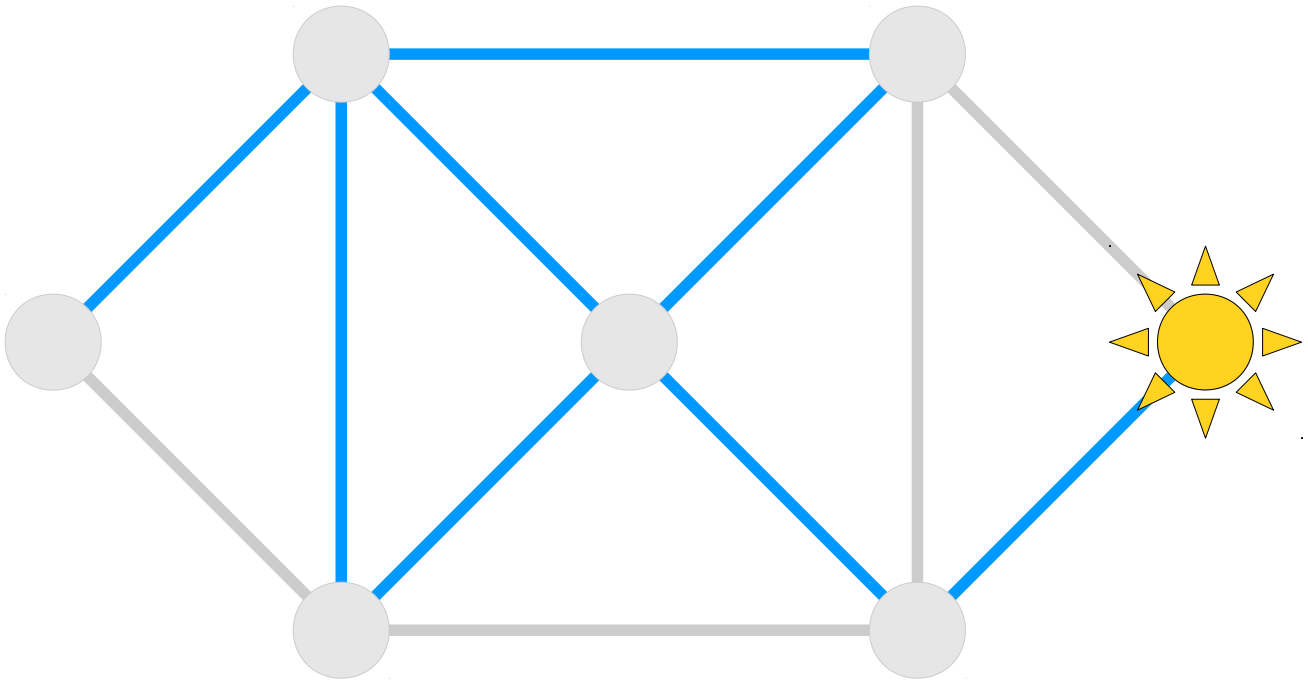


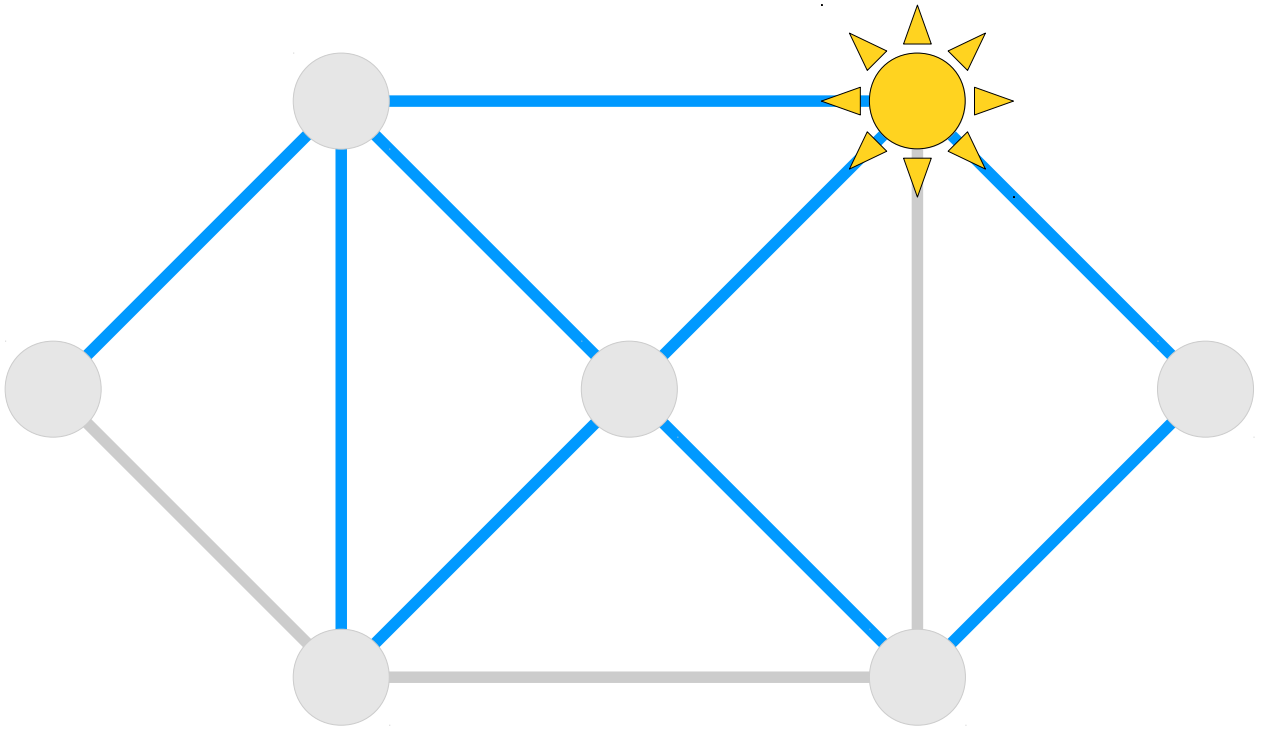


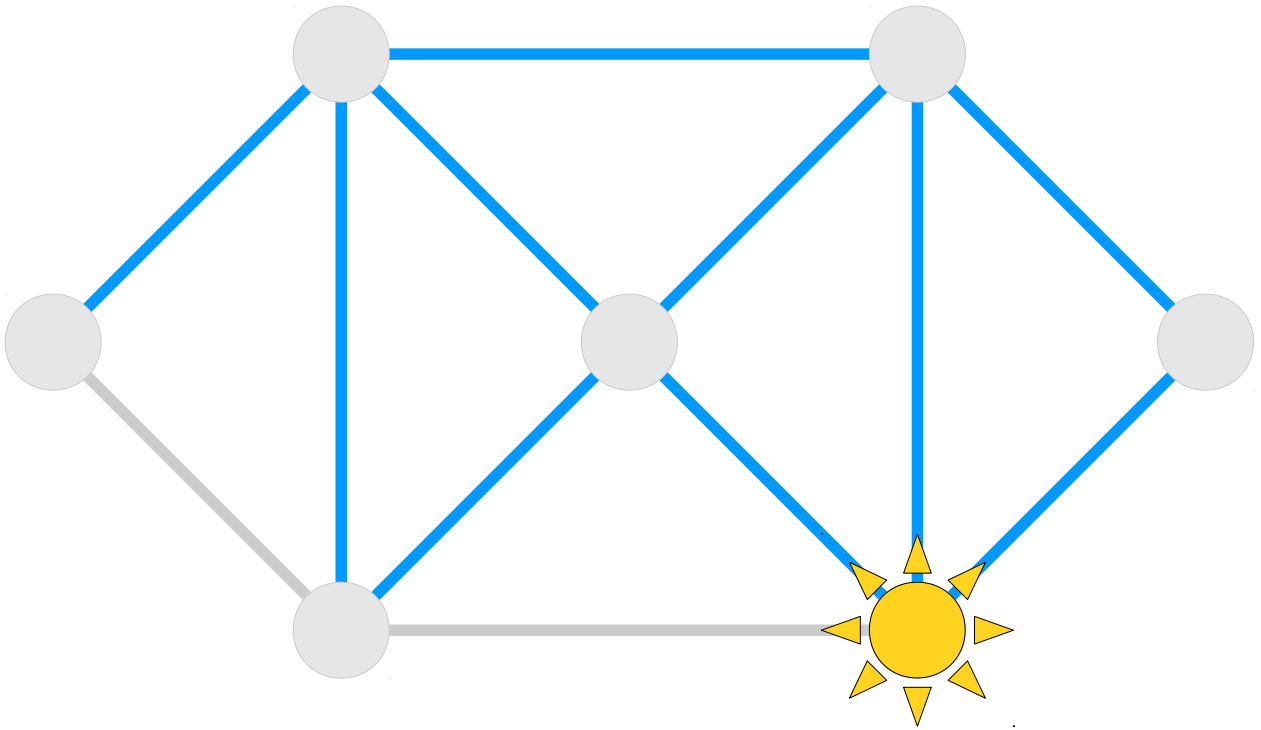


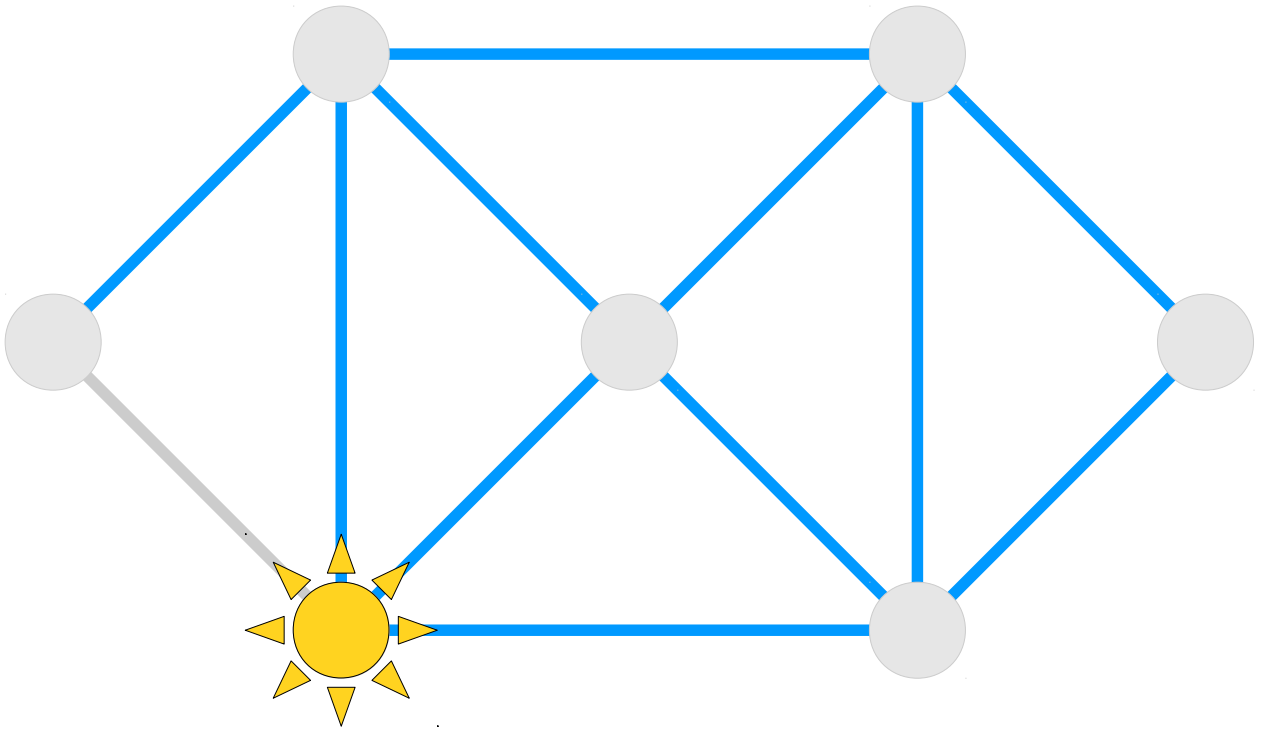


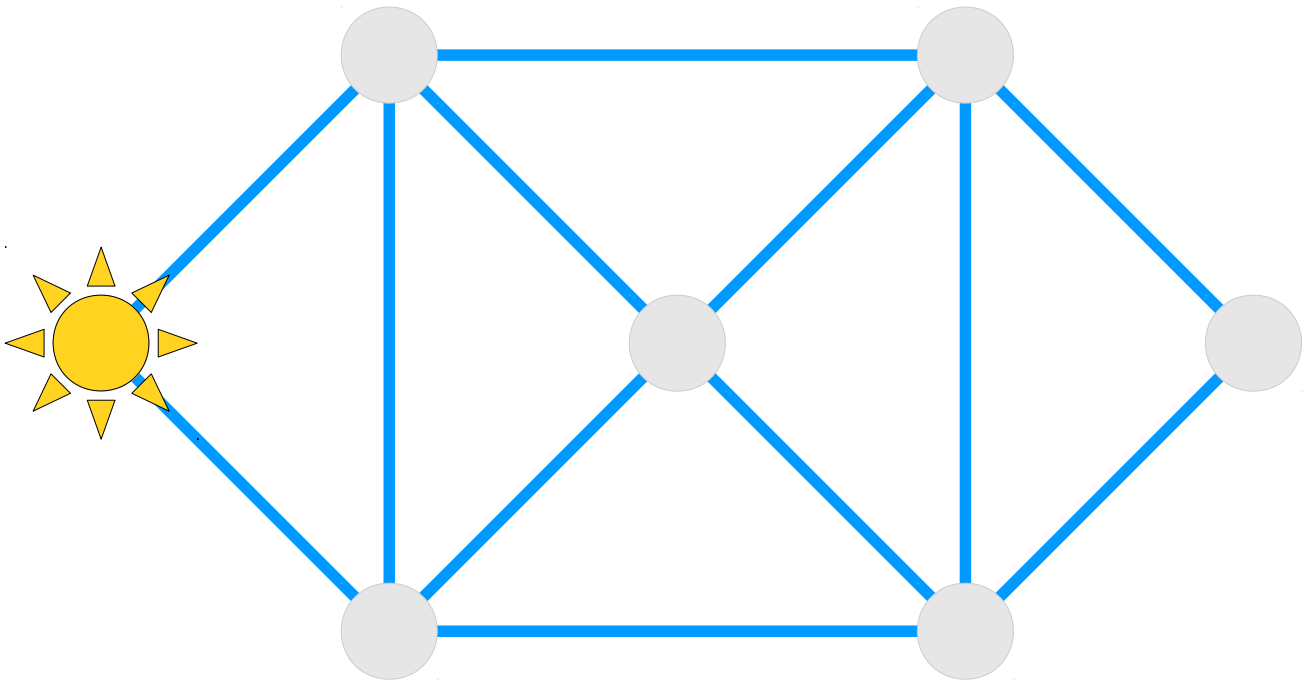


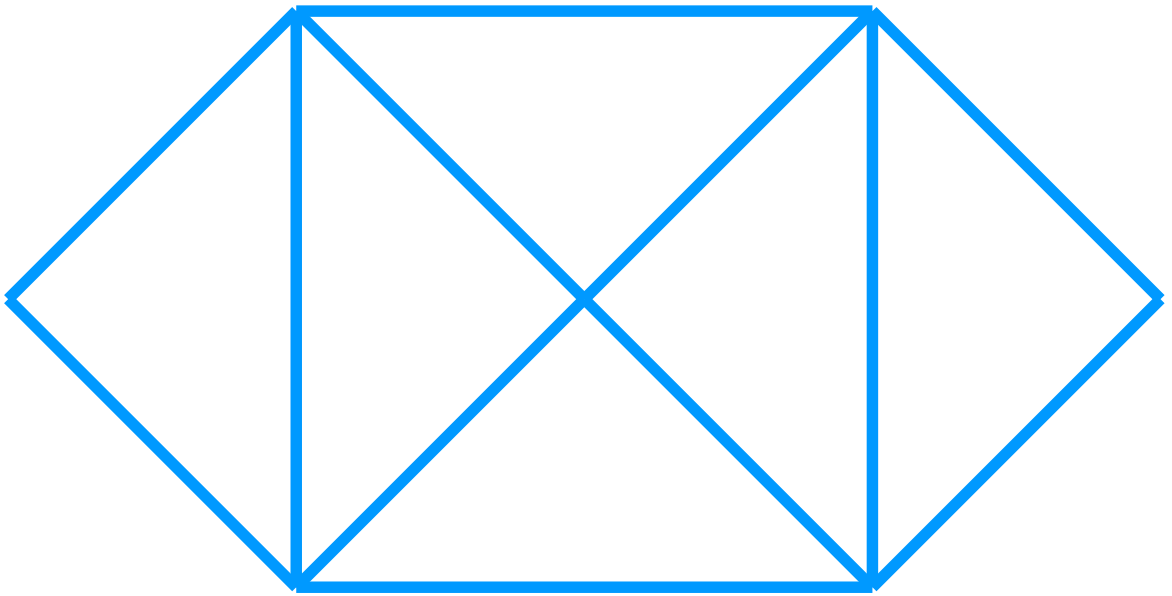


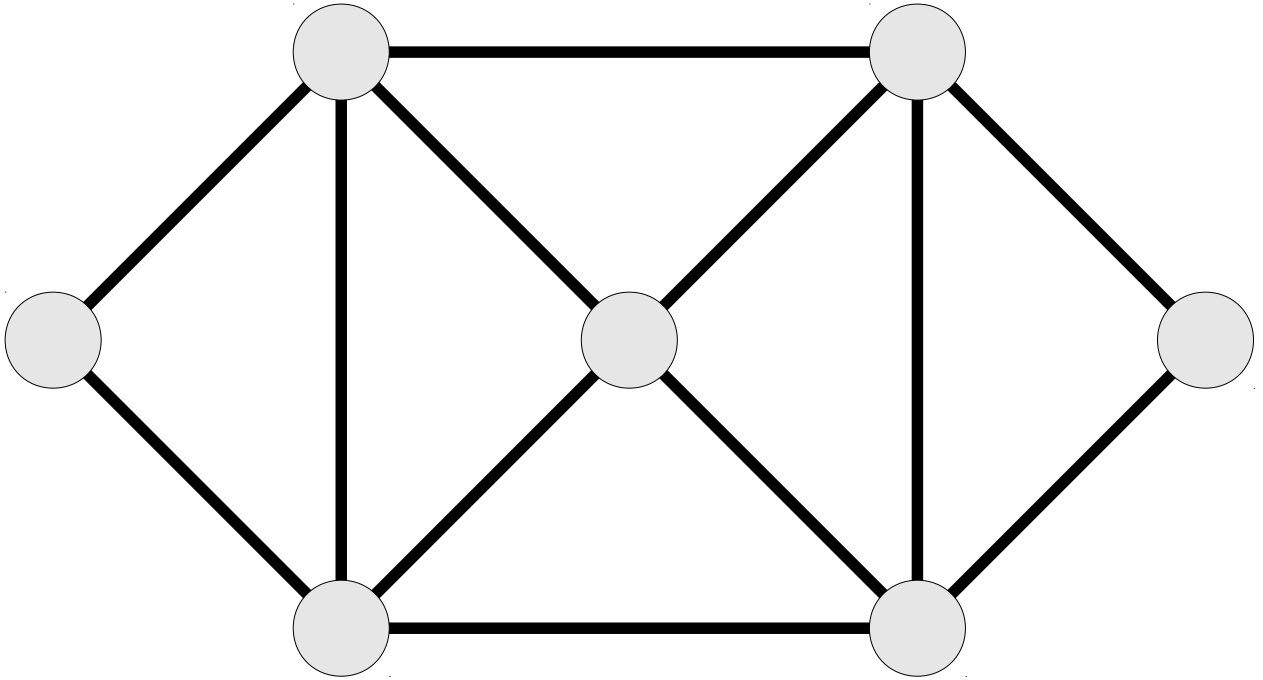


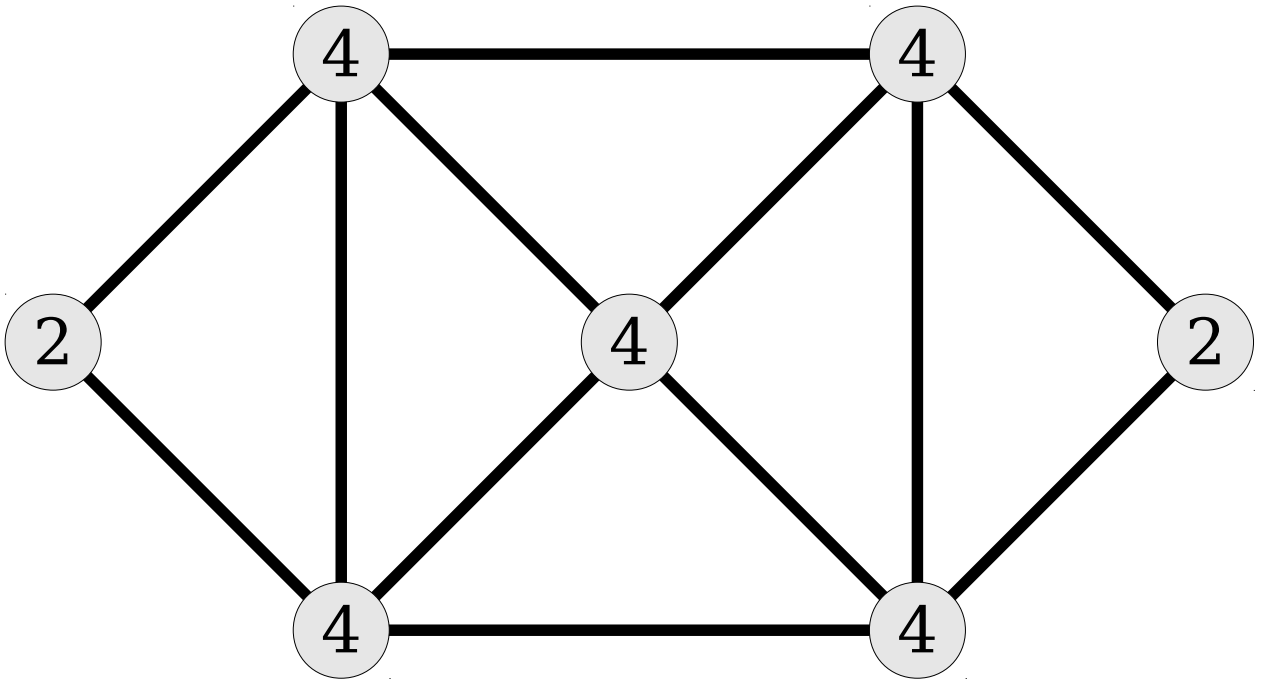


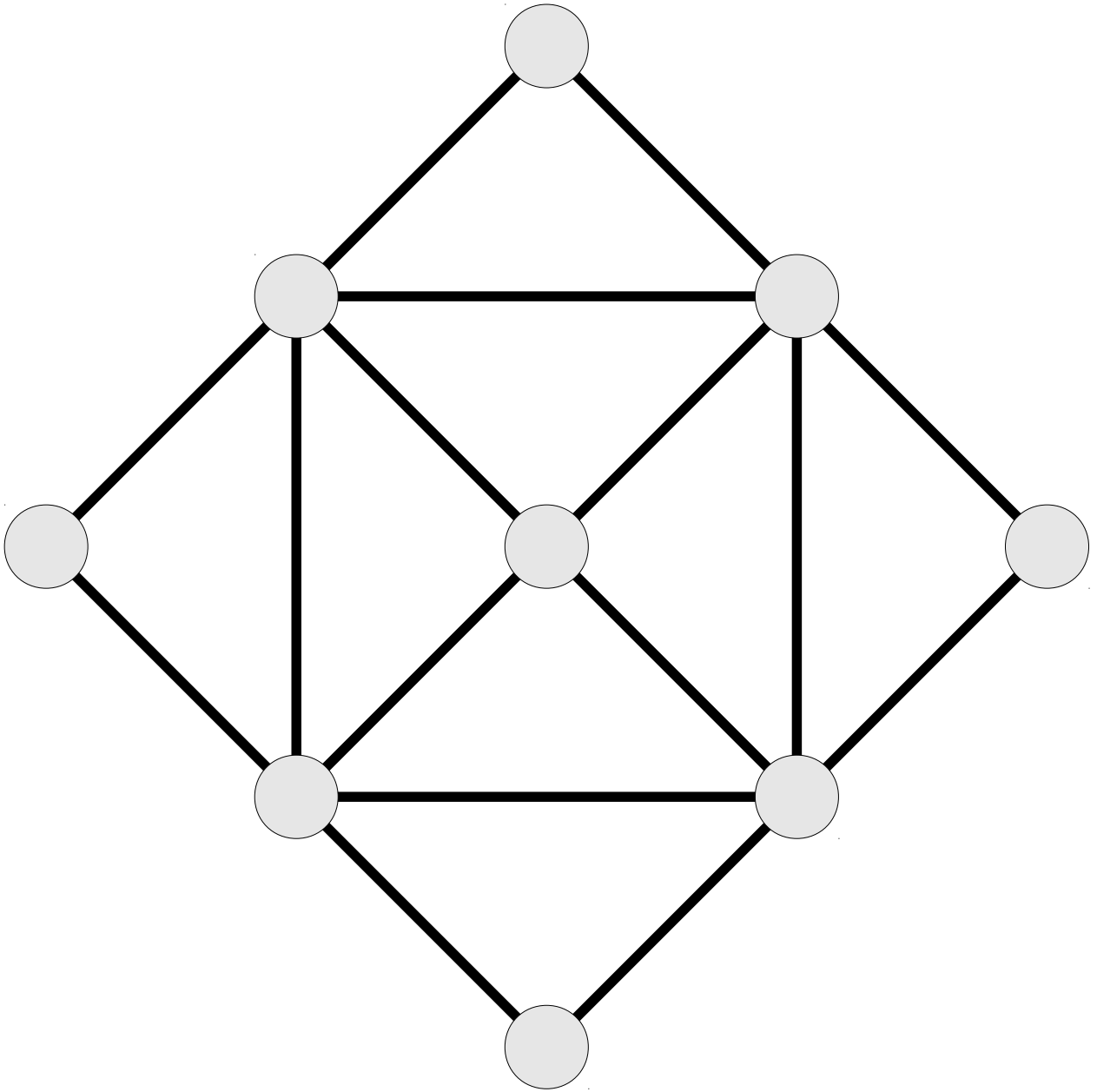


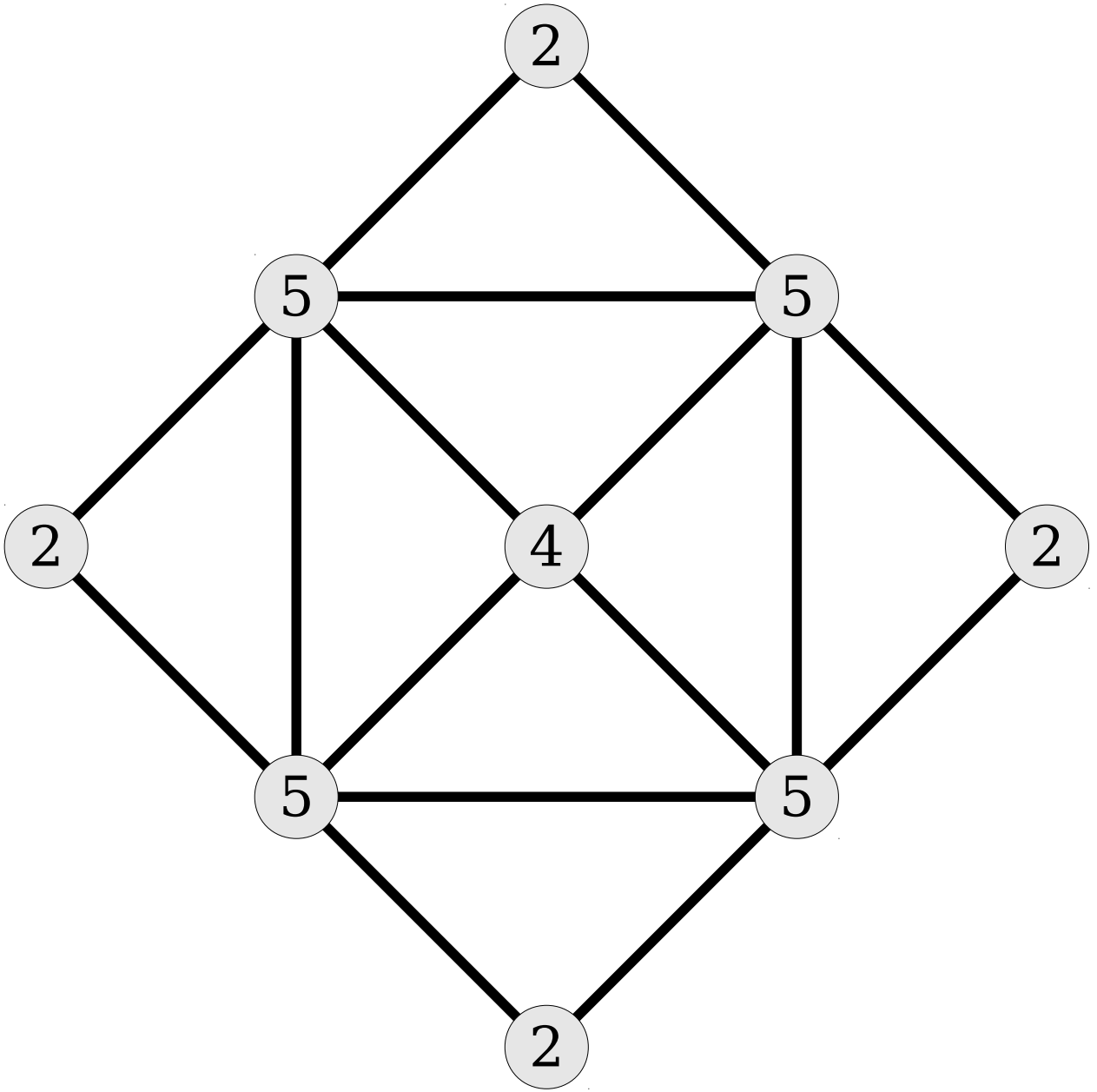


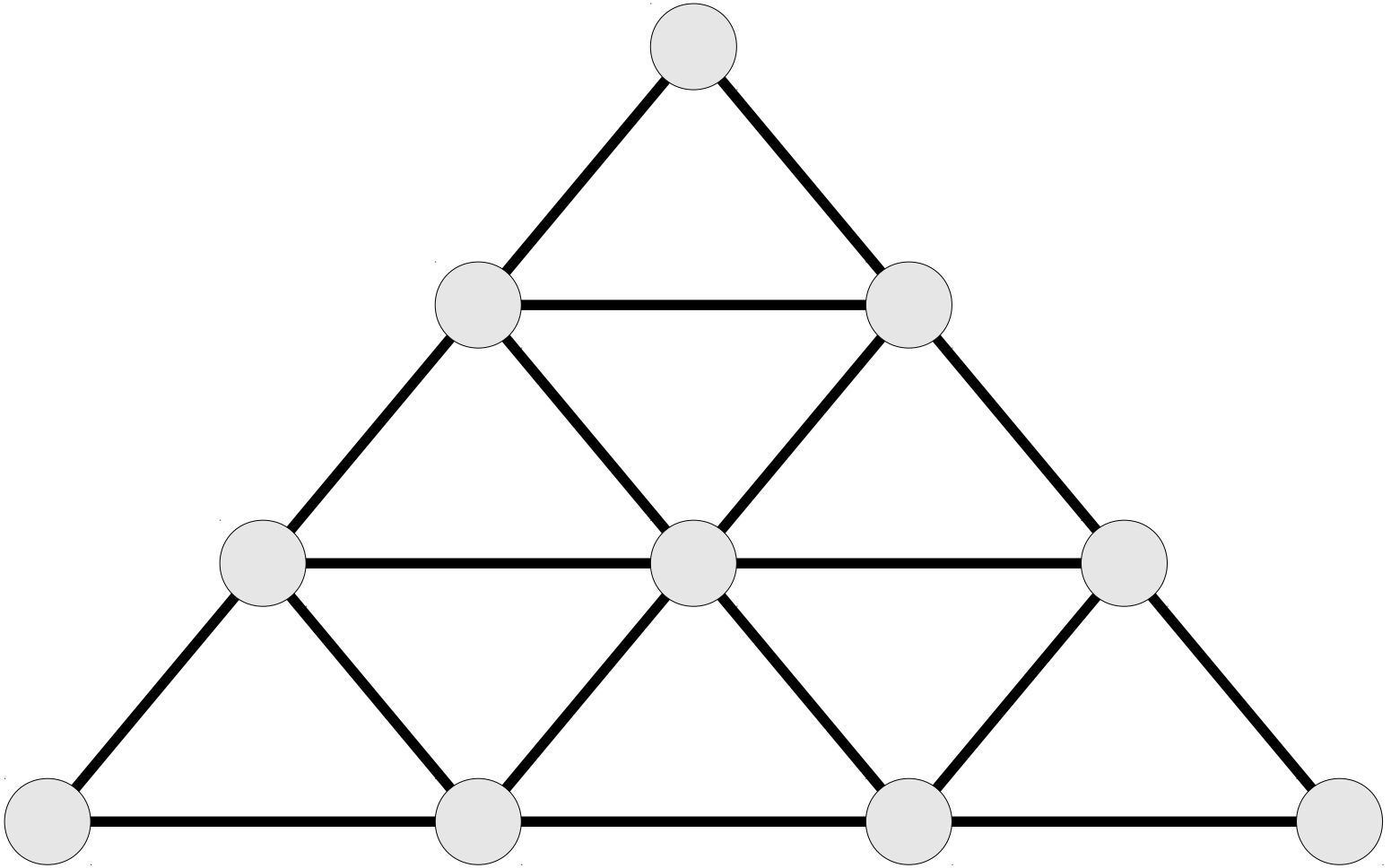


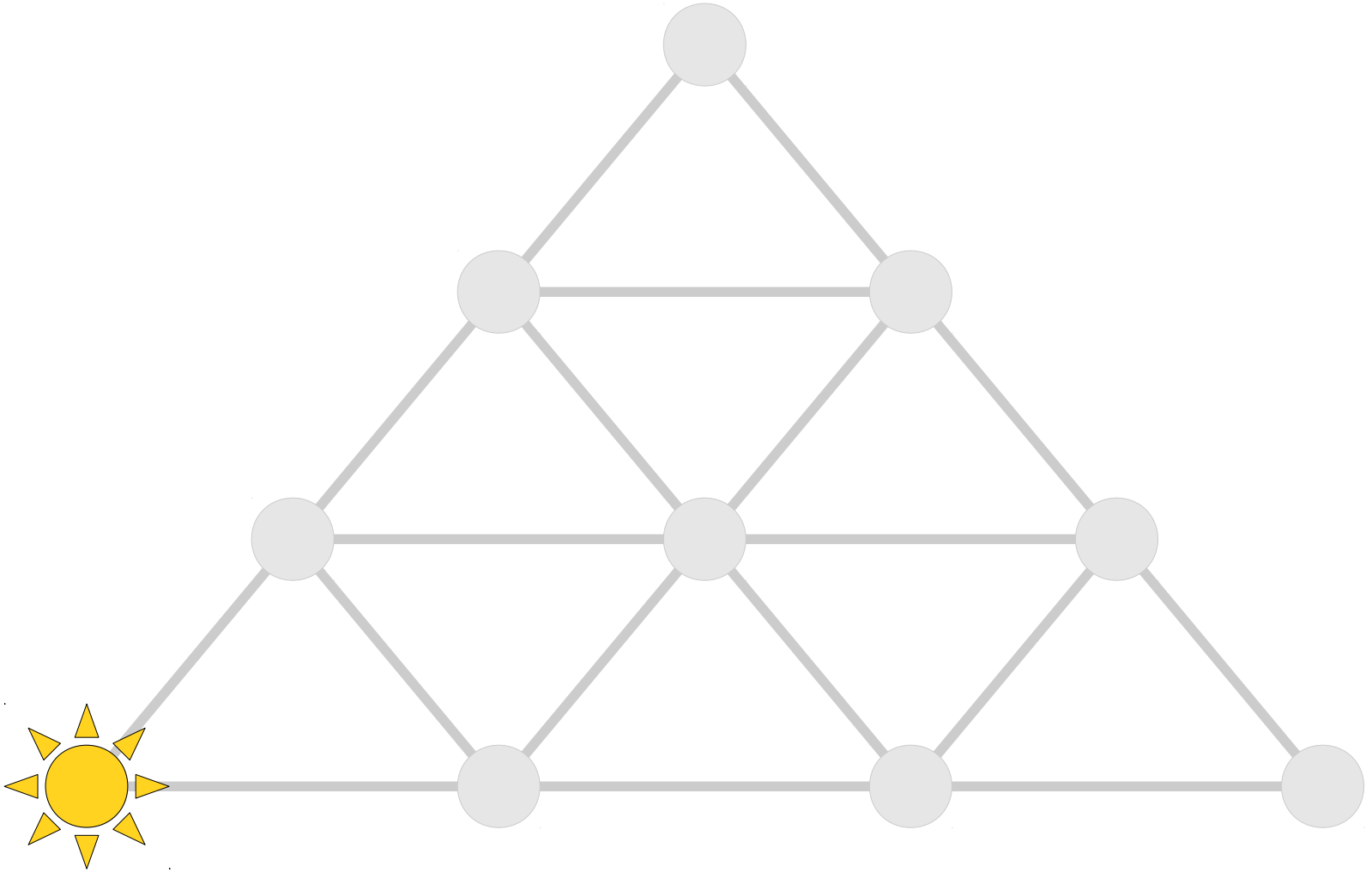


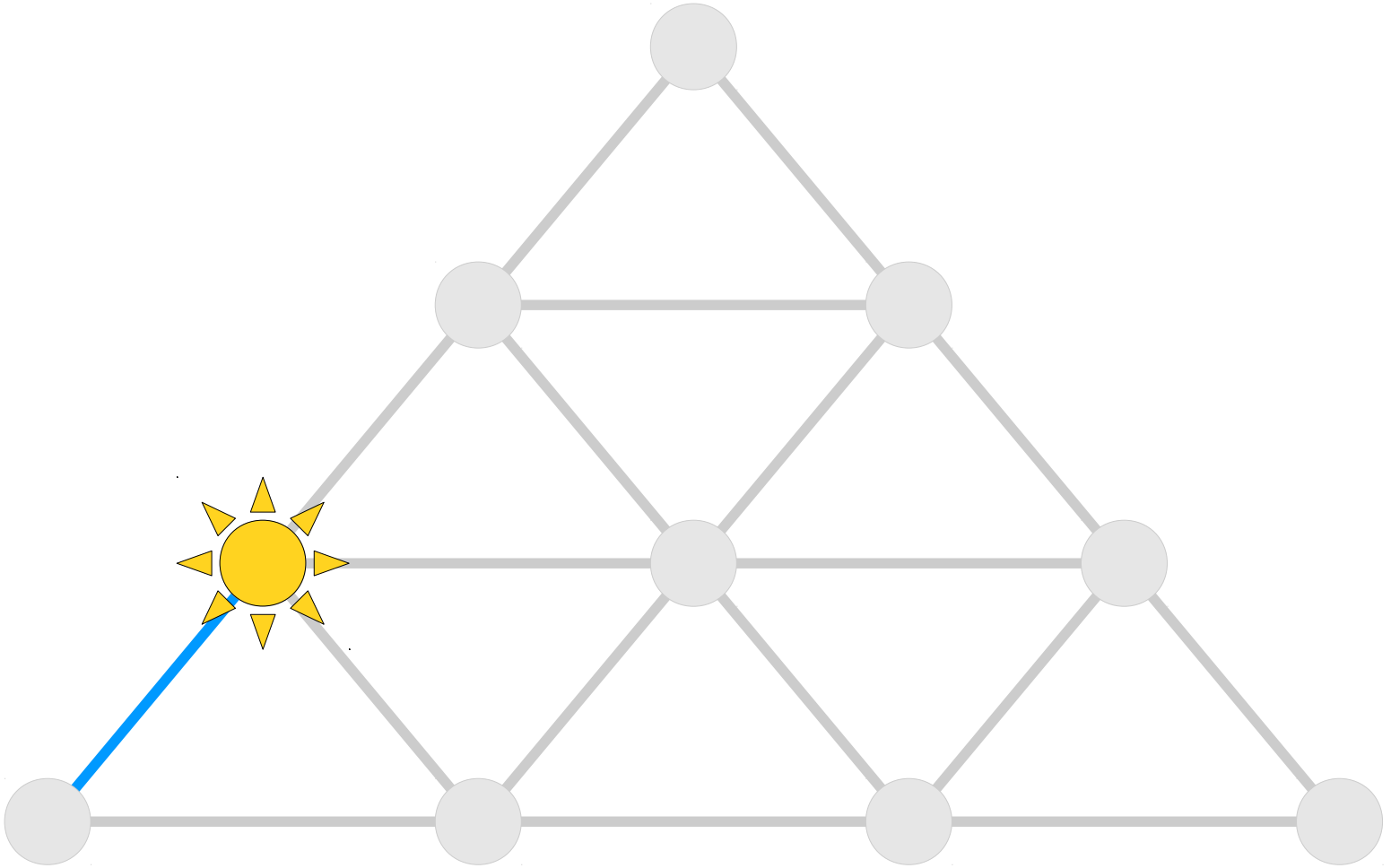

4
4
2
4
2
4
4

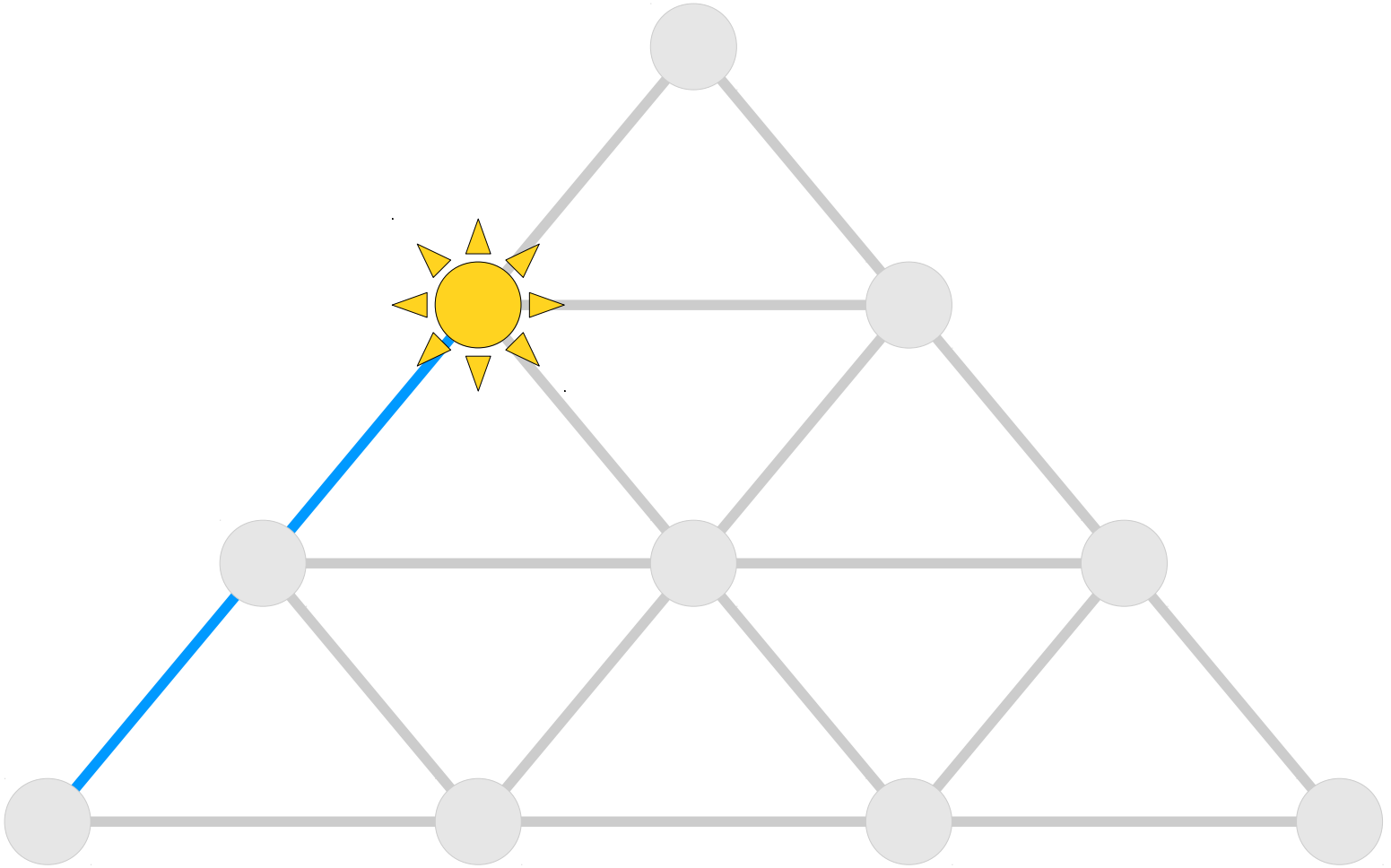


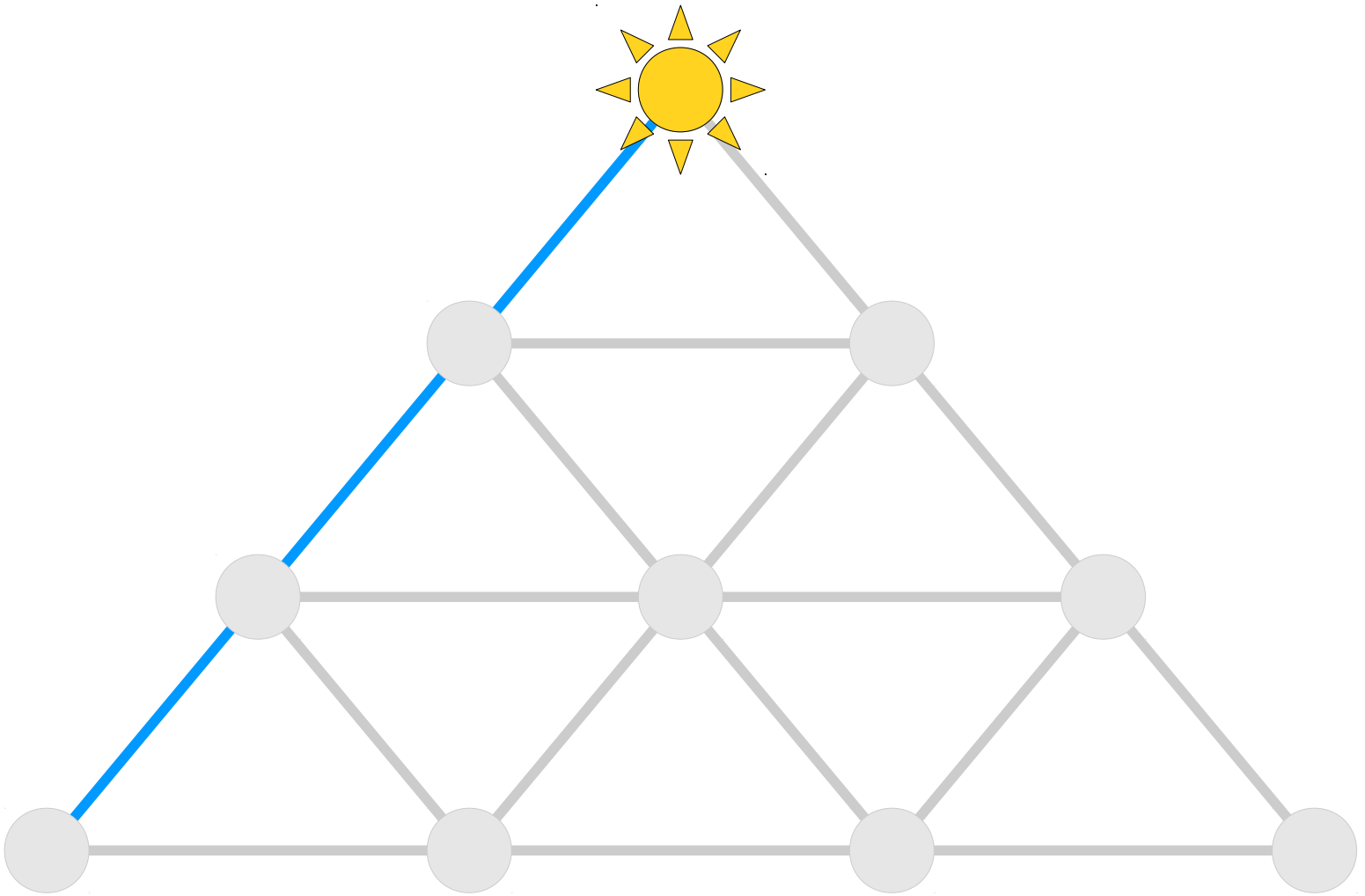

2
5
5
2
4
2
5
5
2

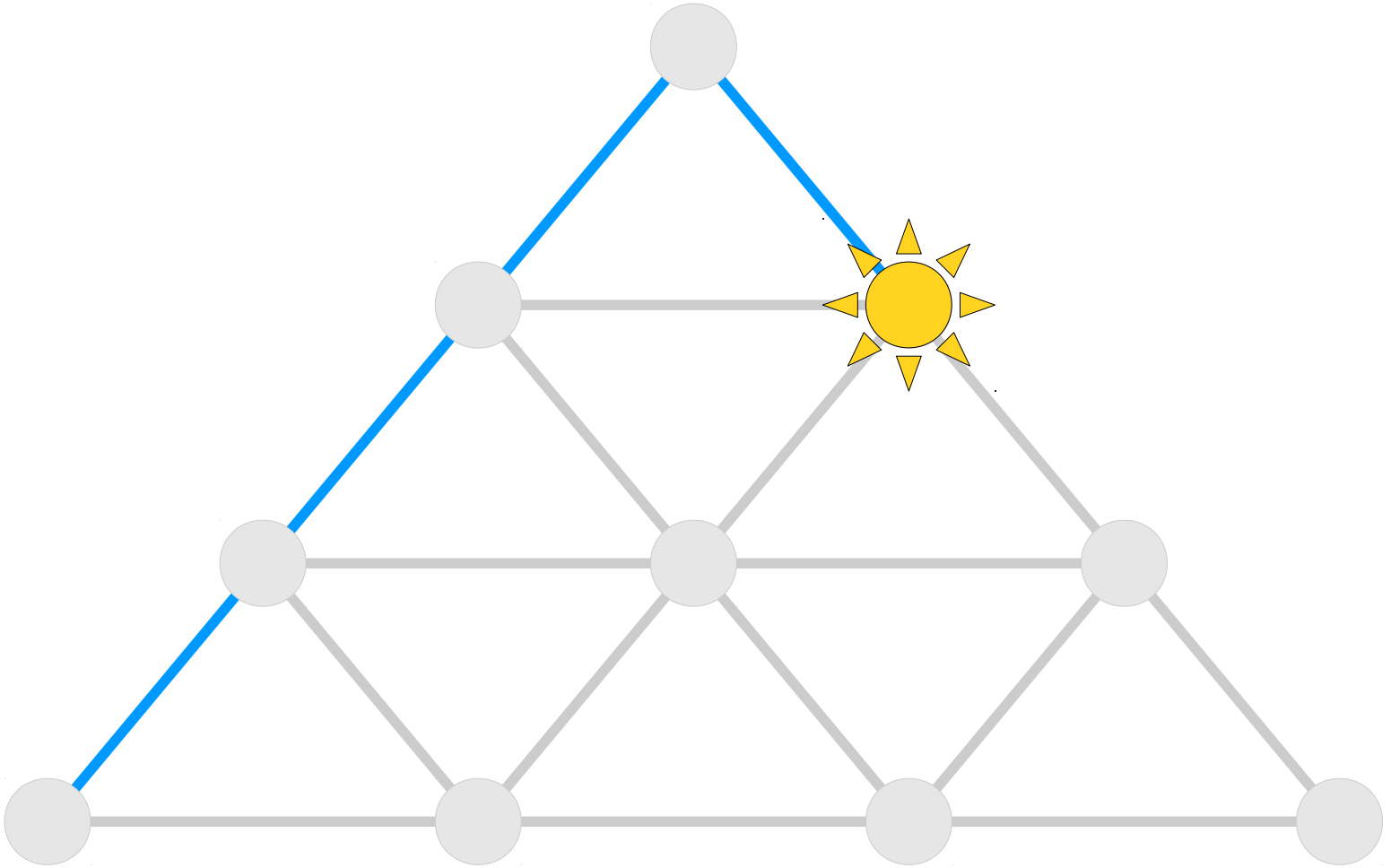


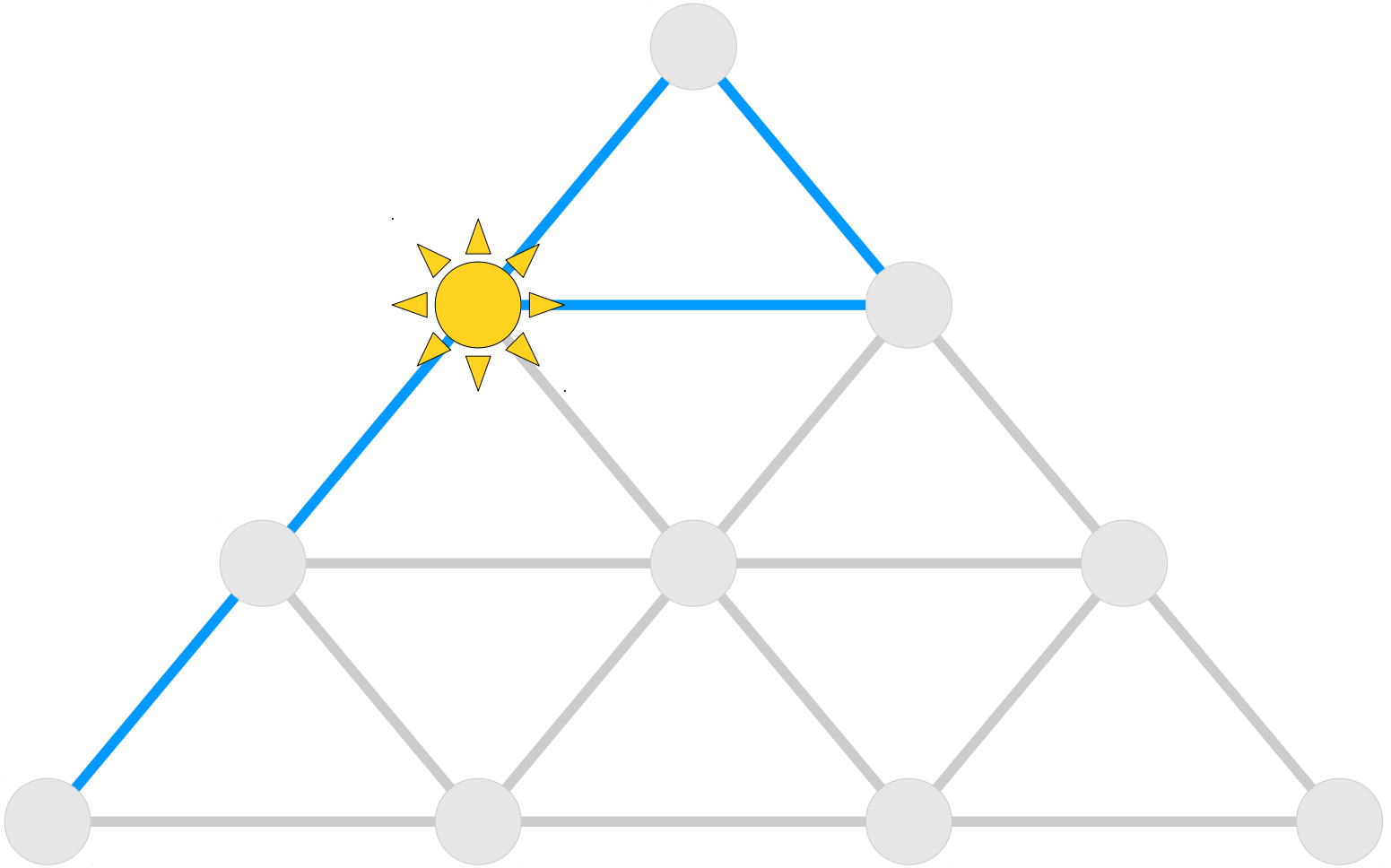


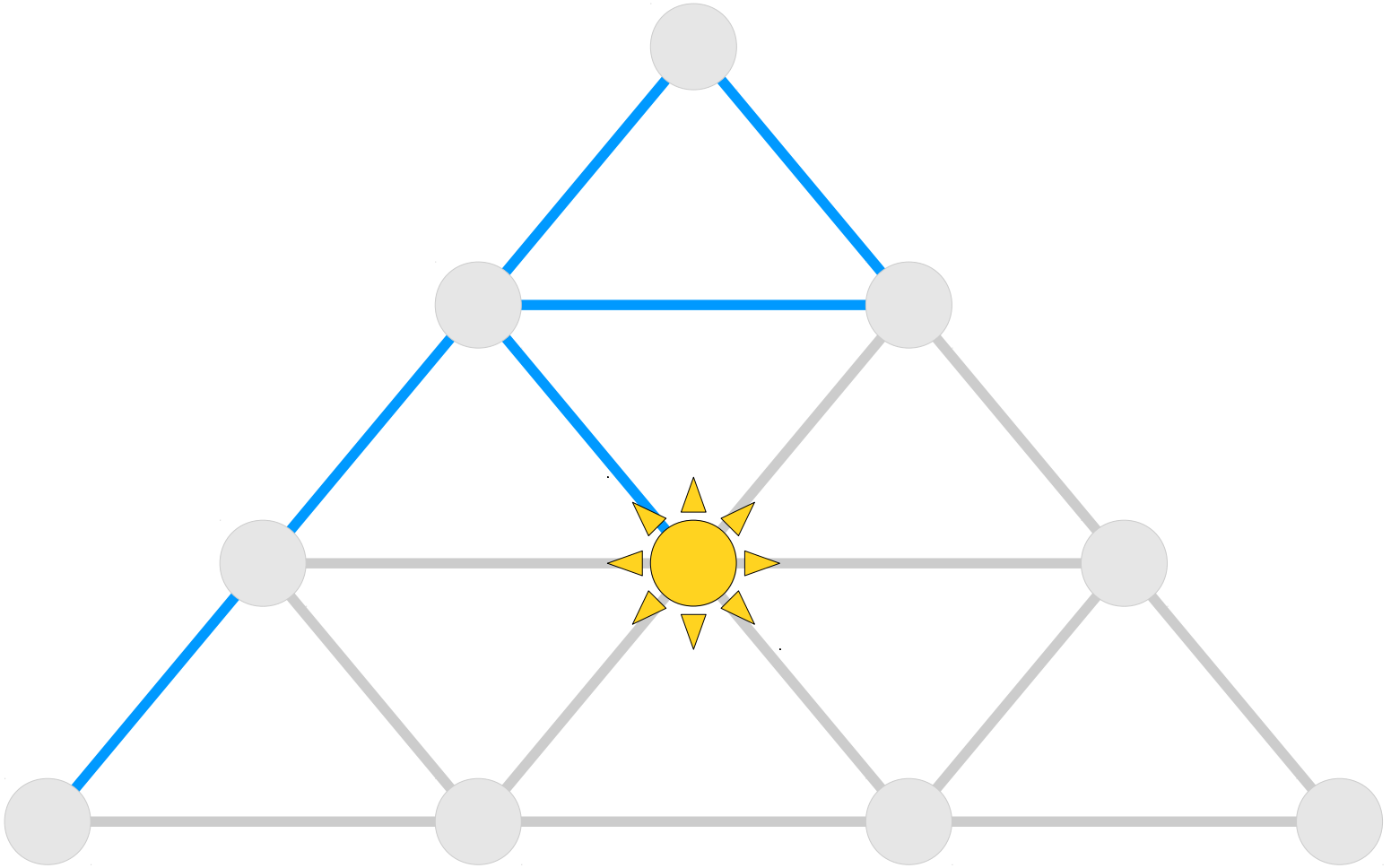


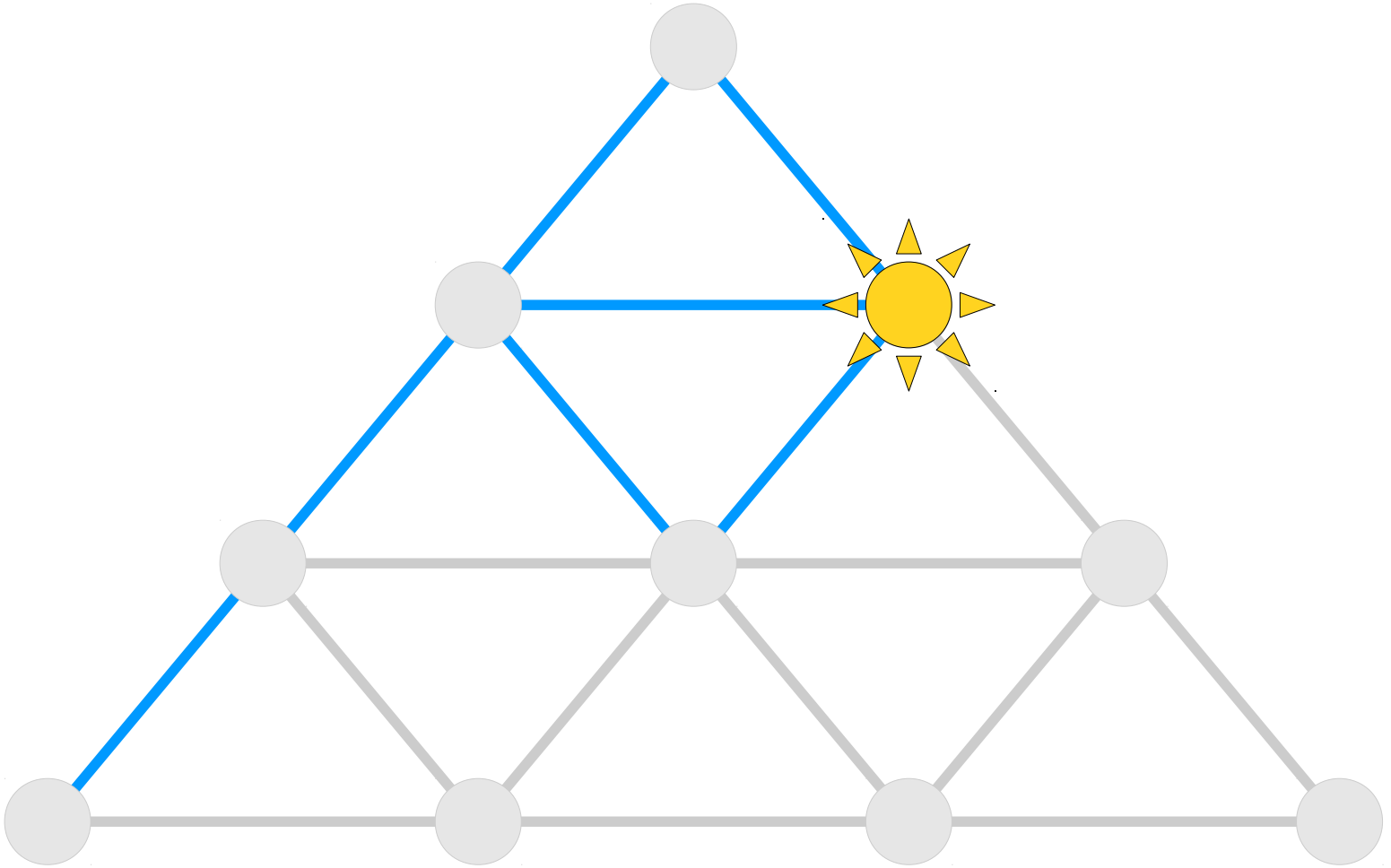


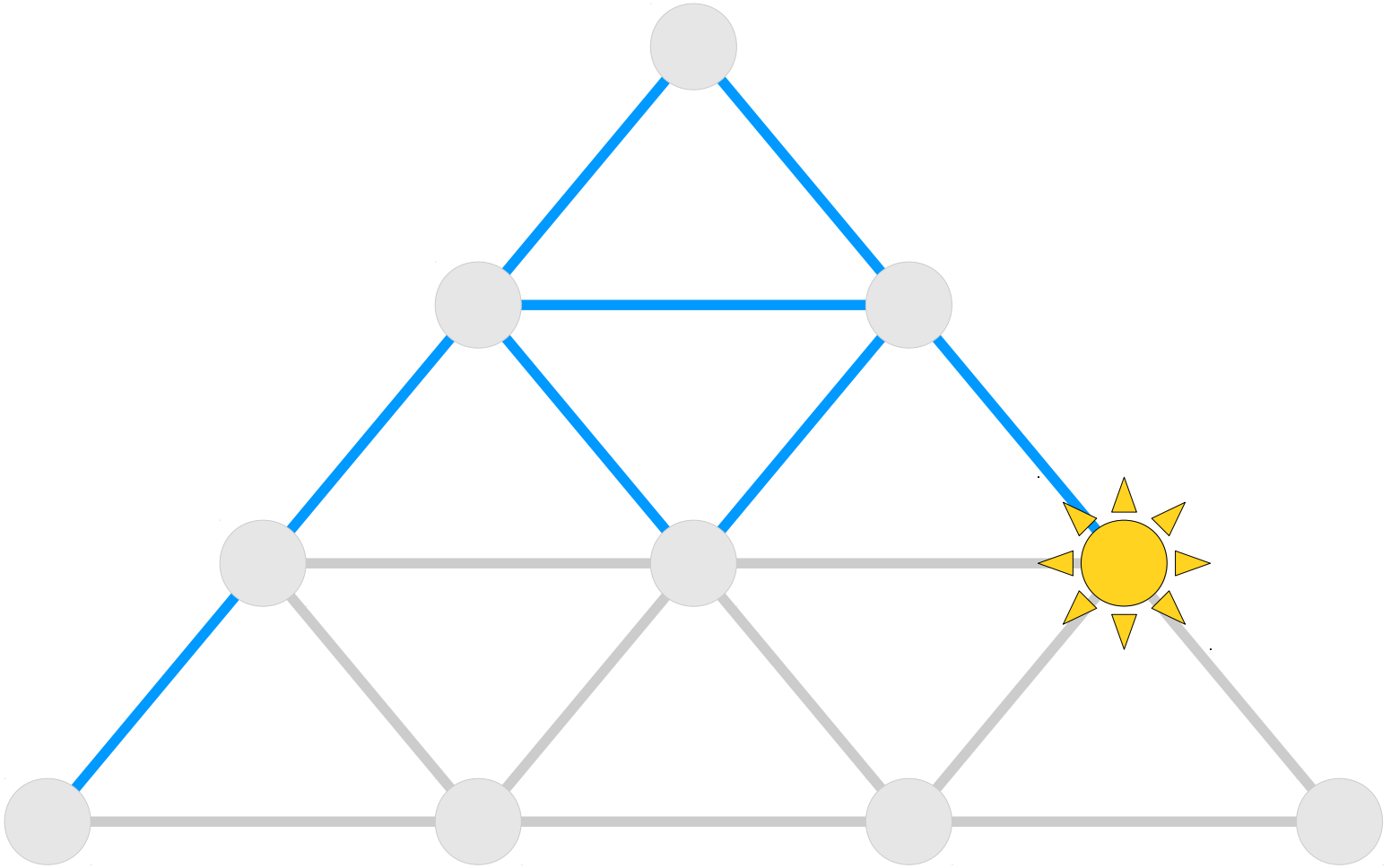


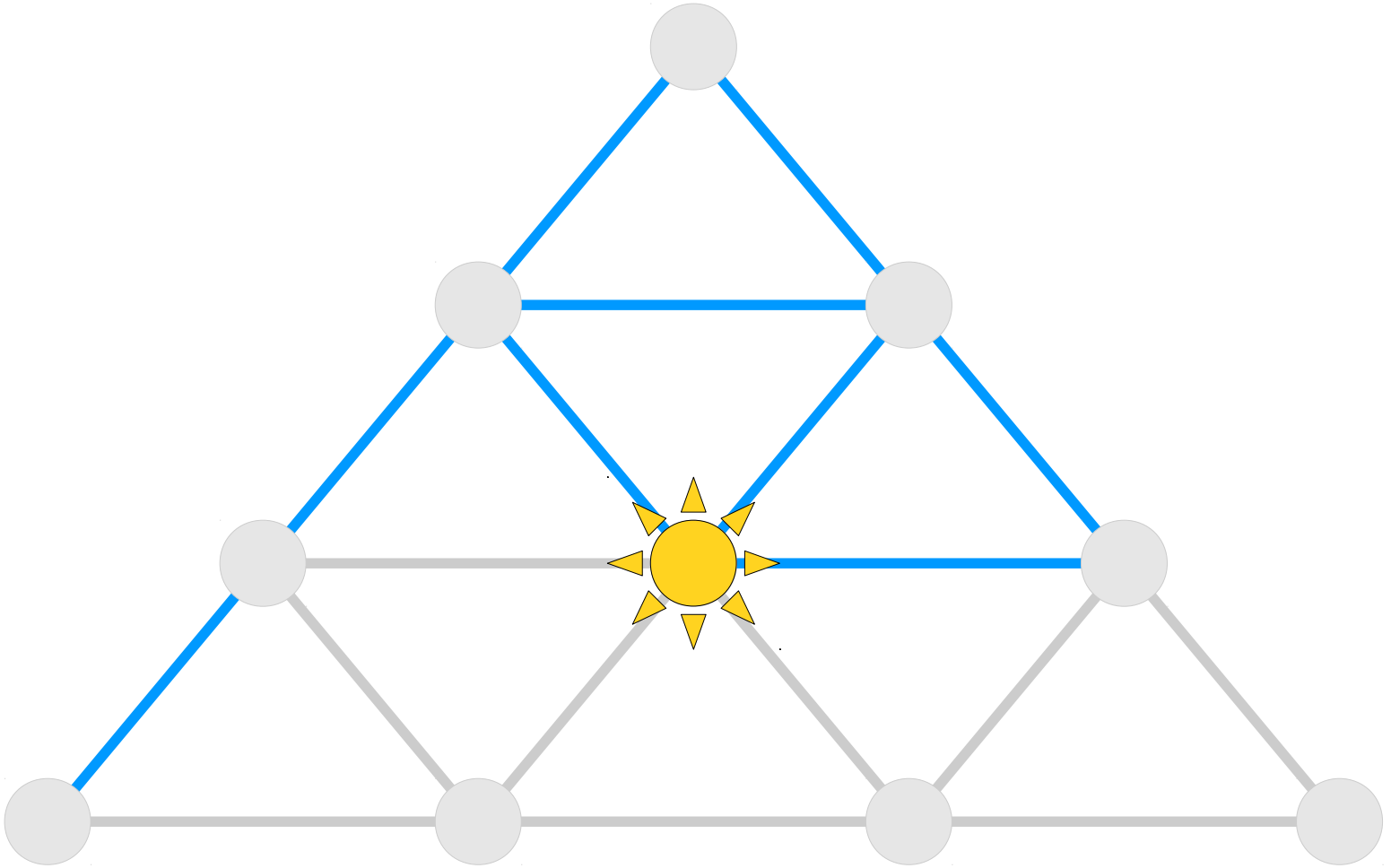


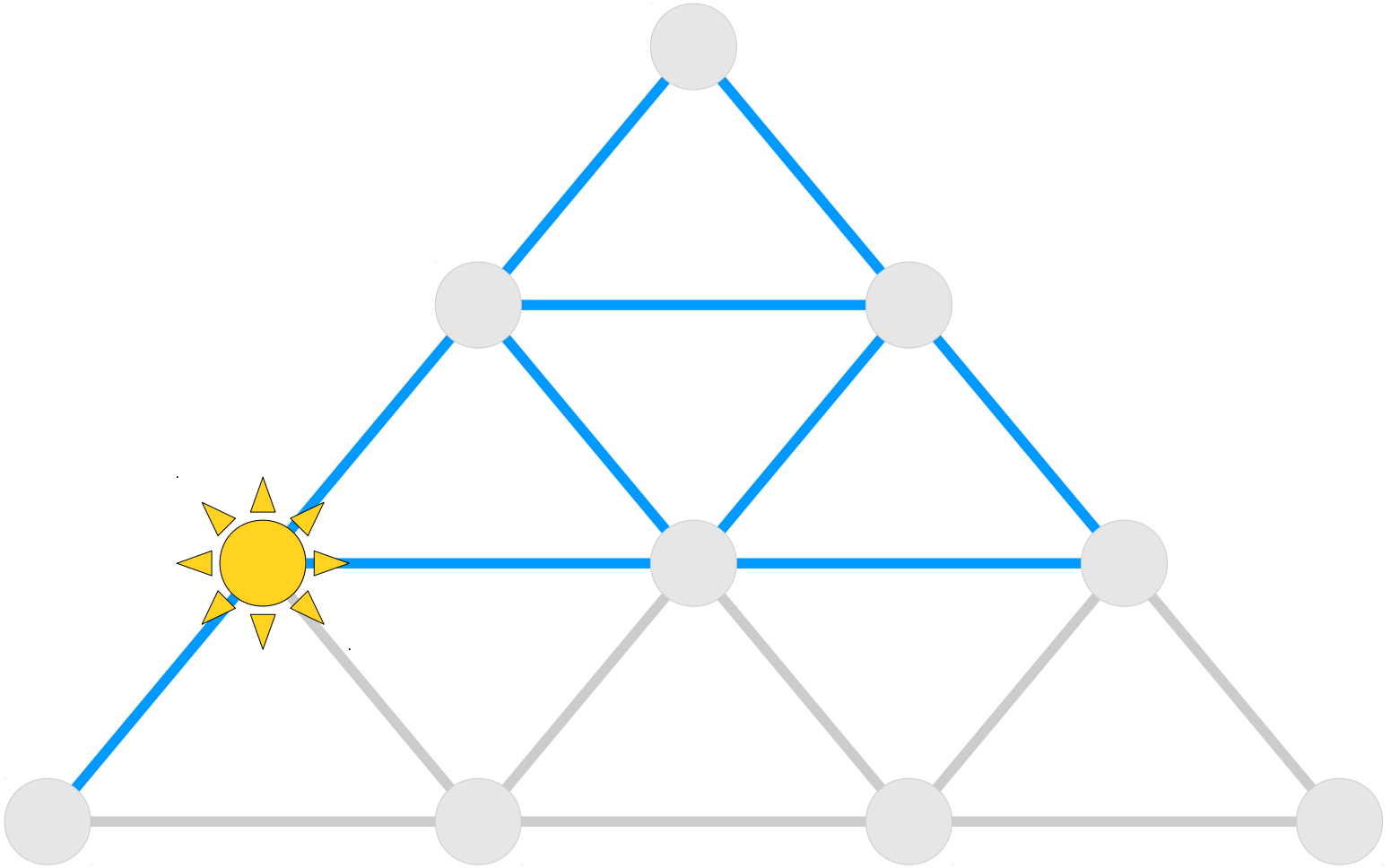


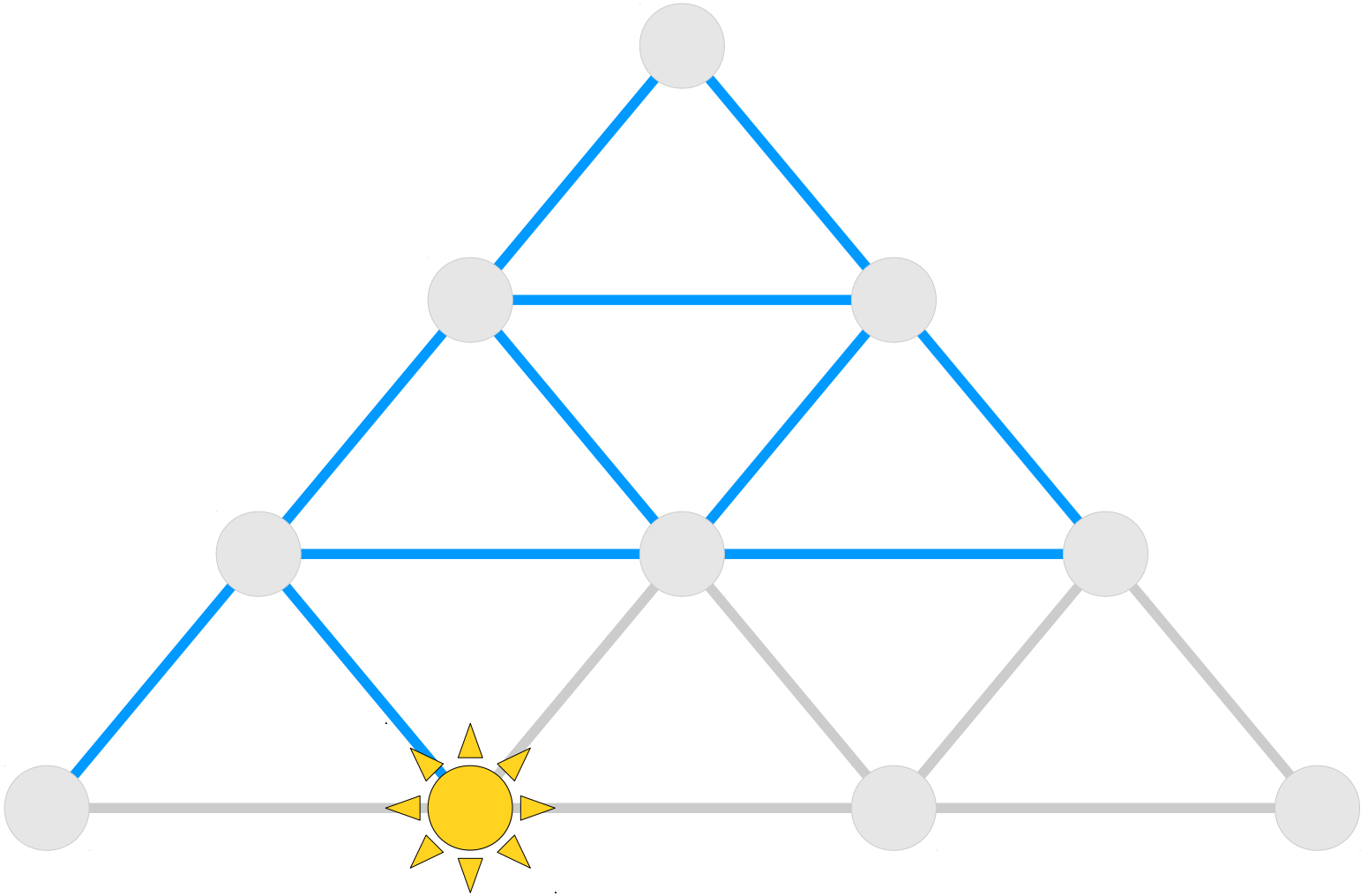


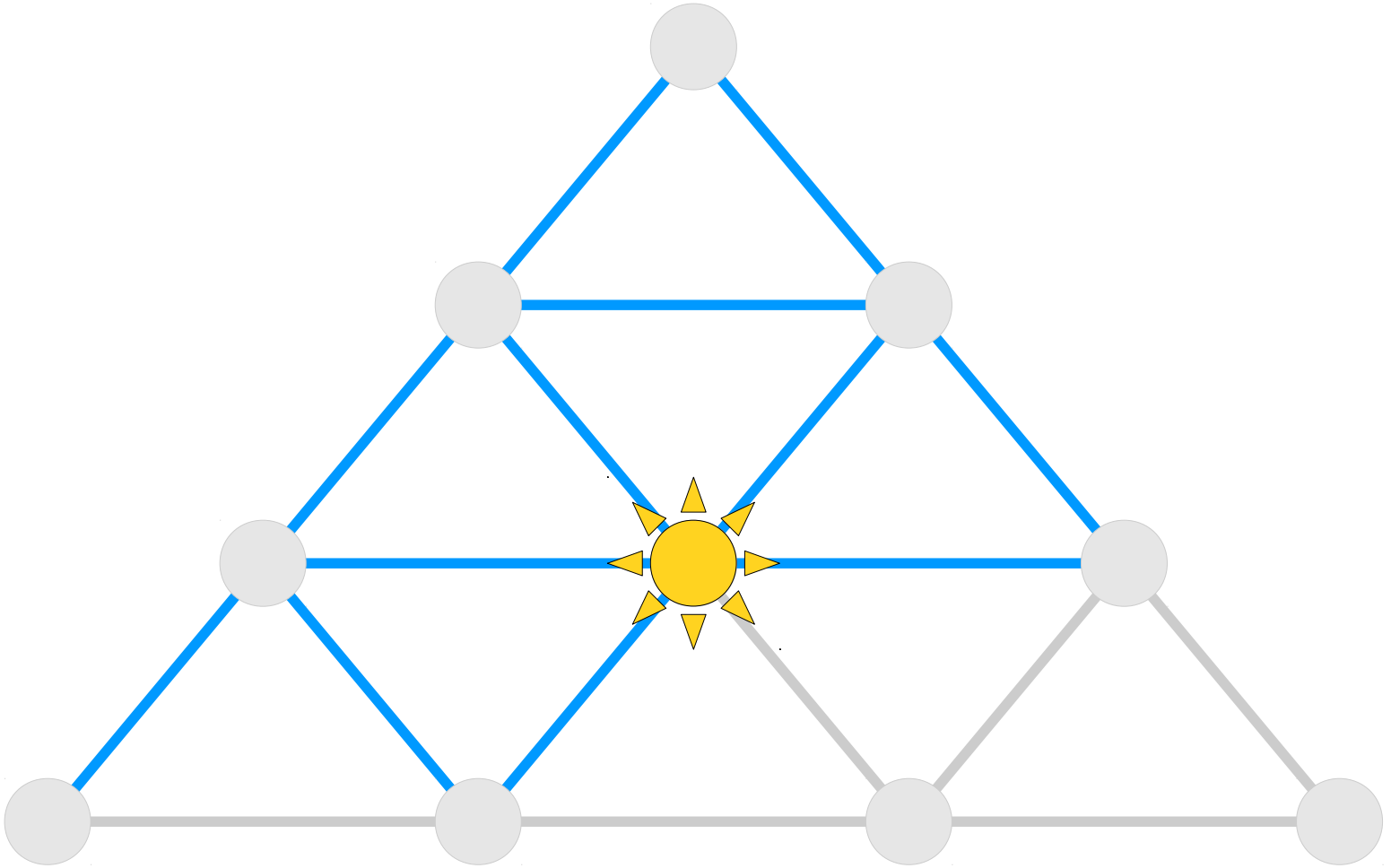


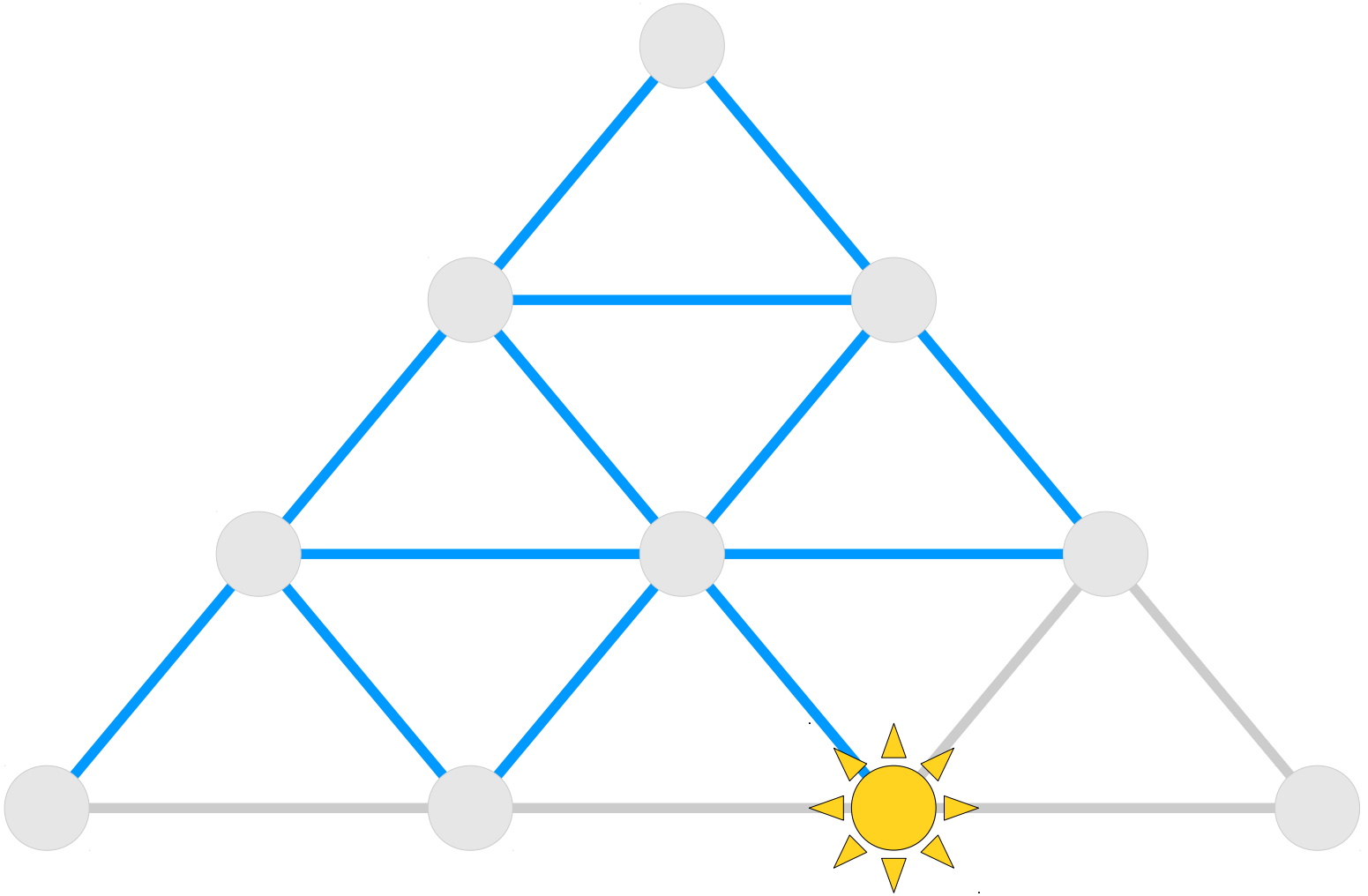


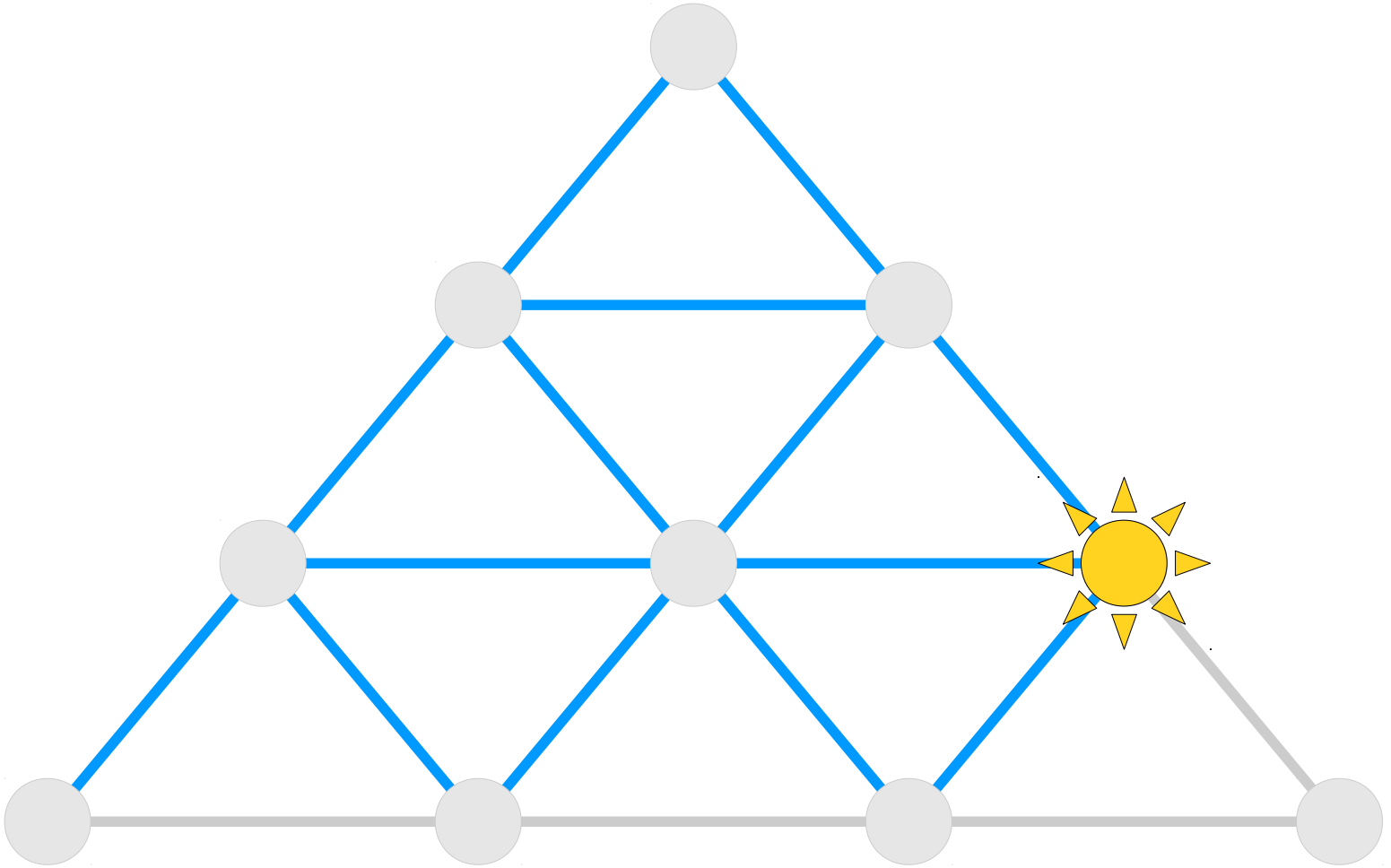


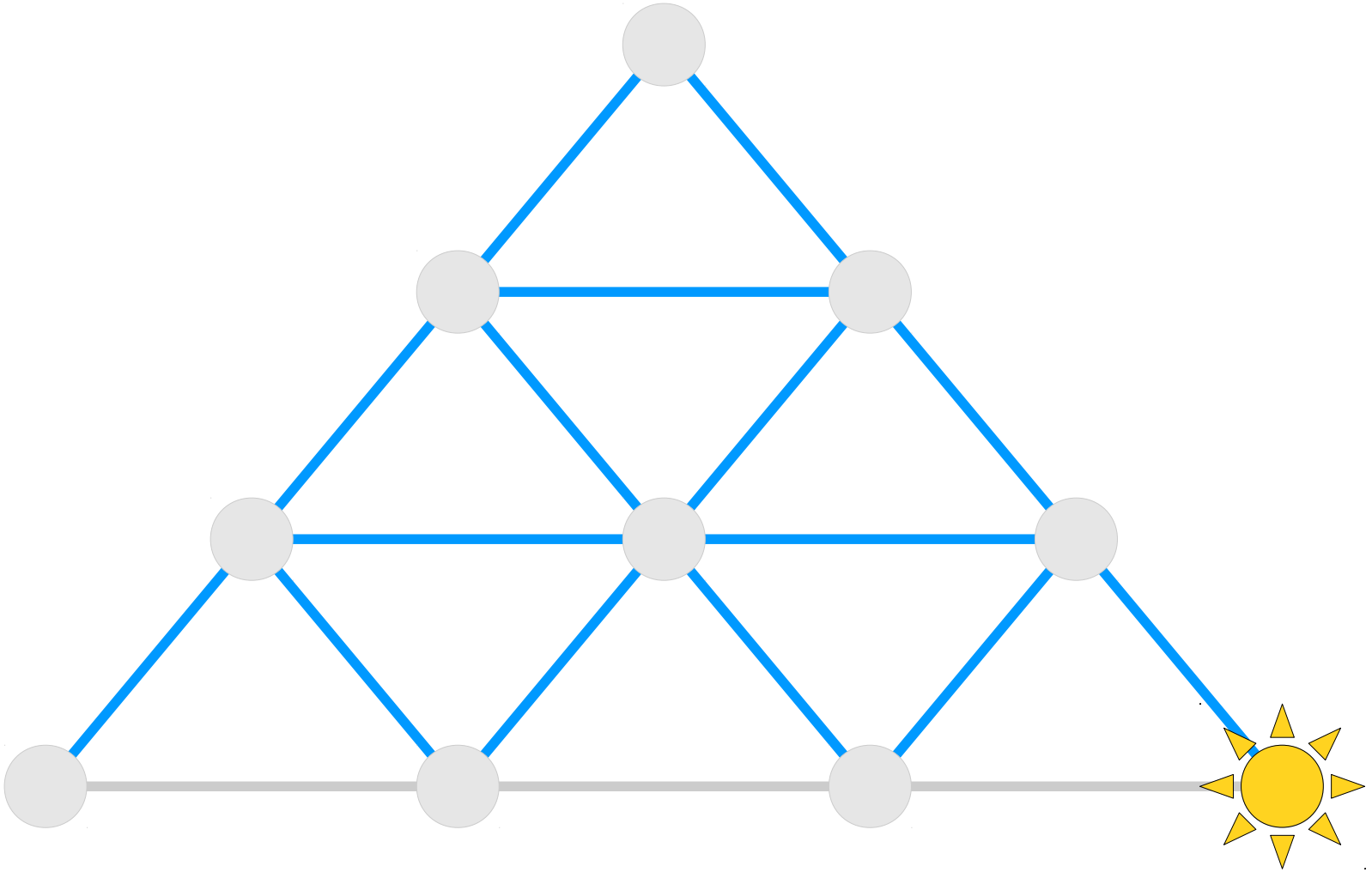


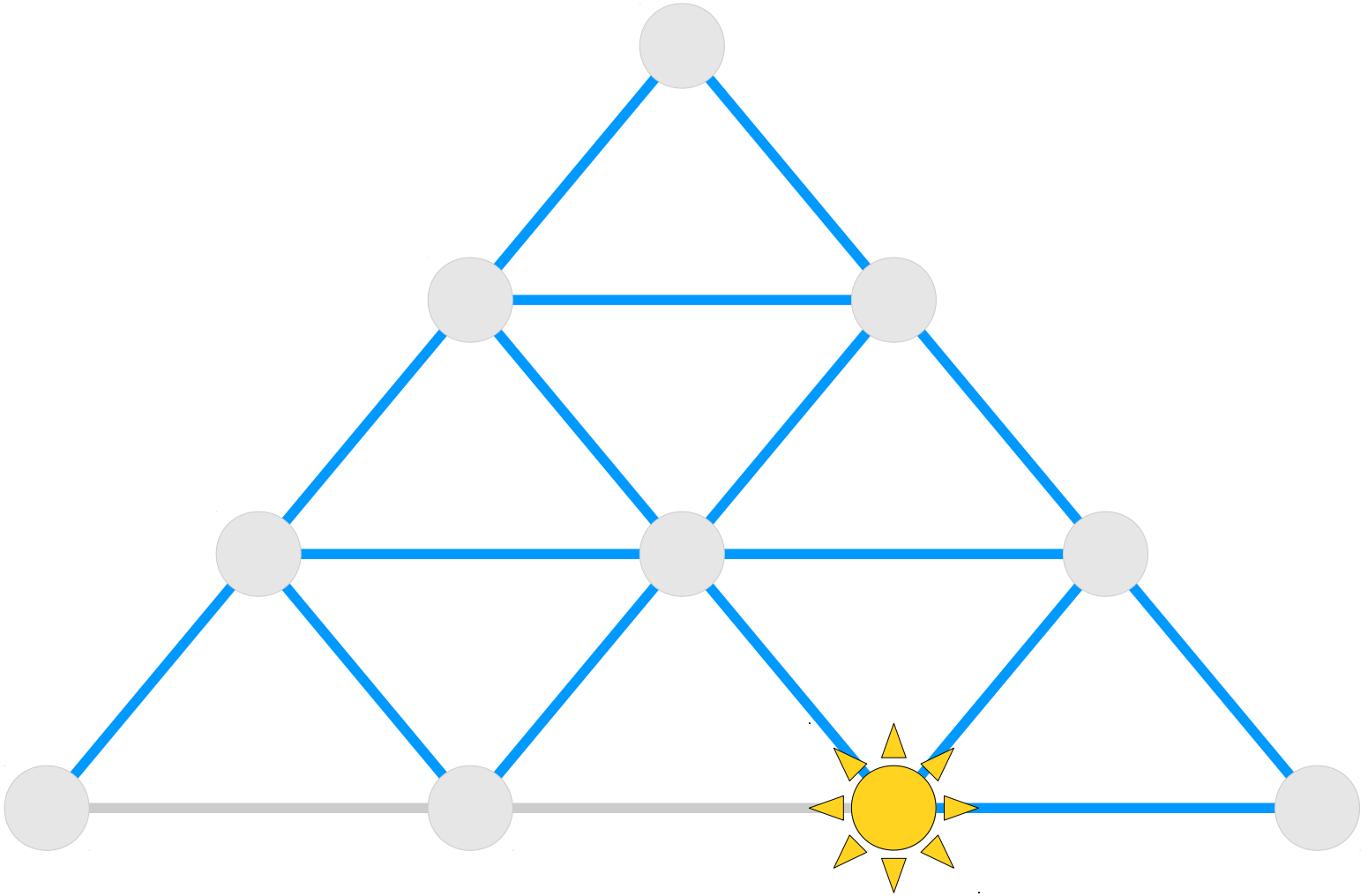


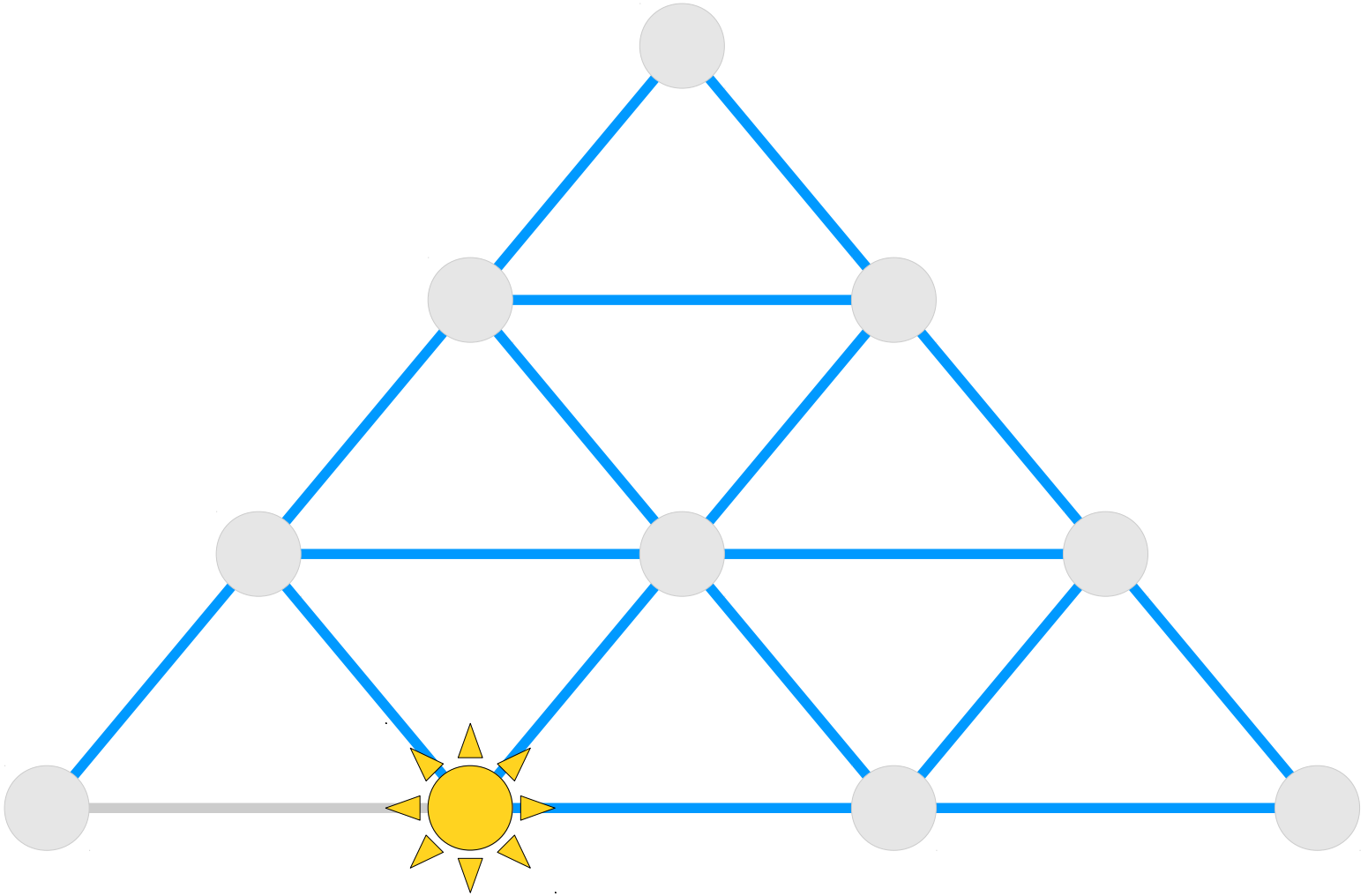


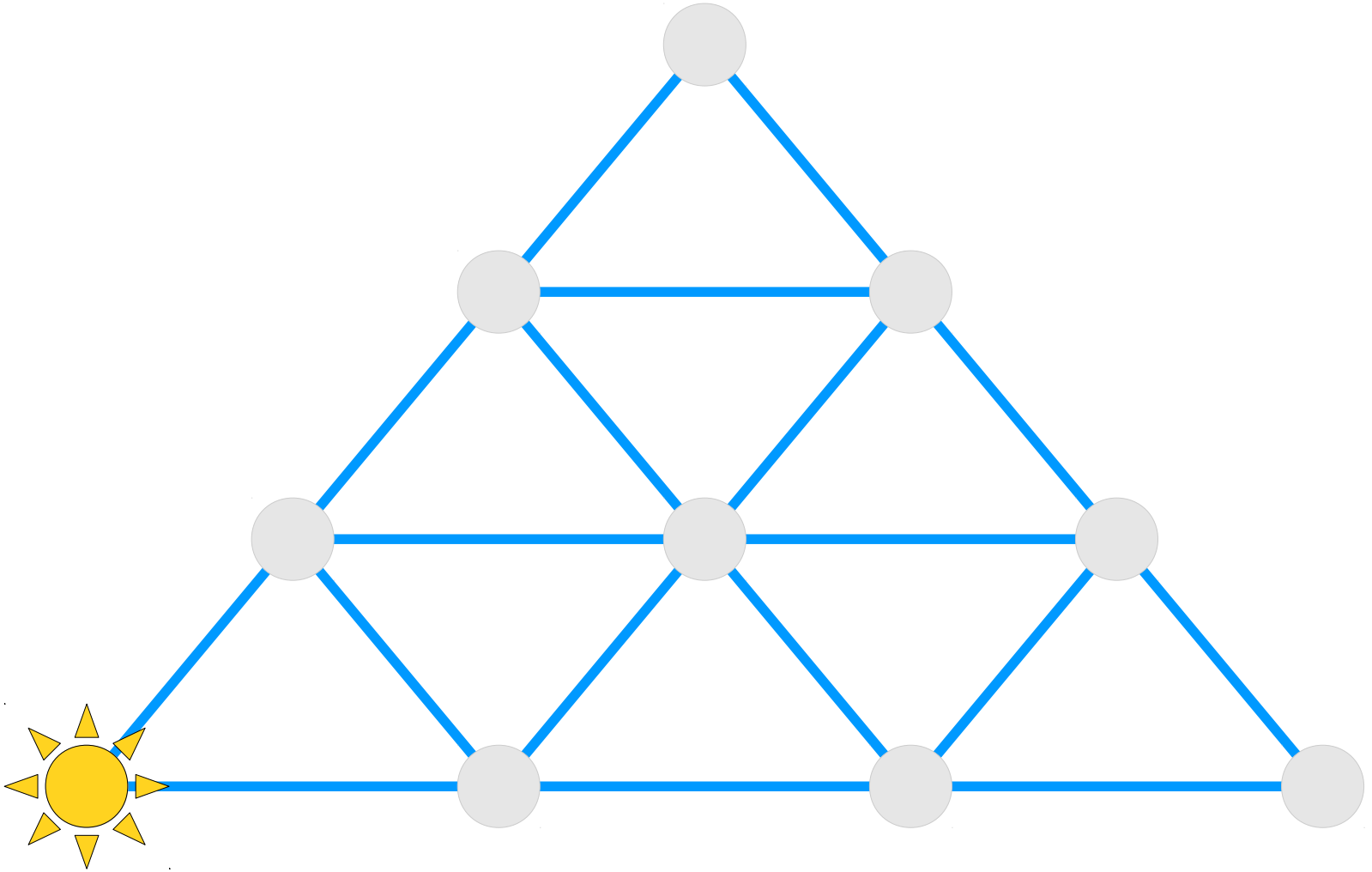


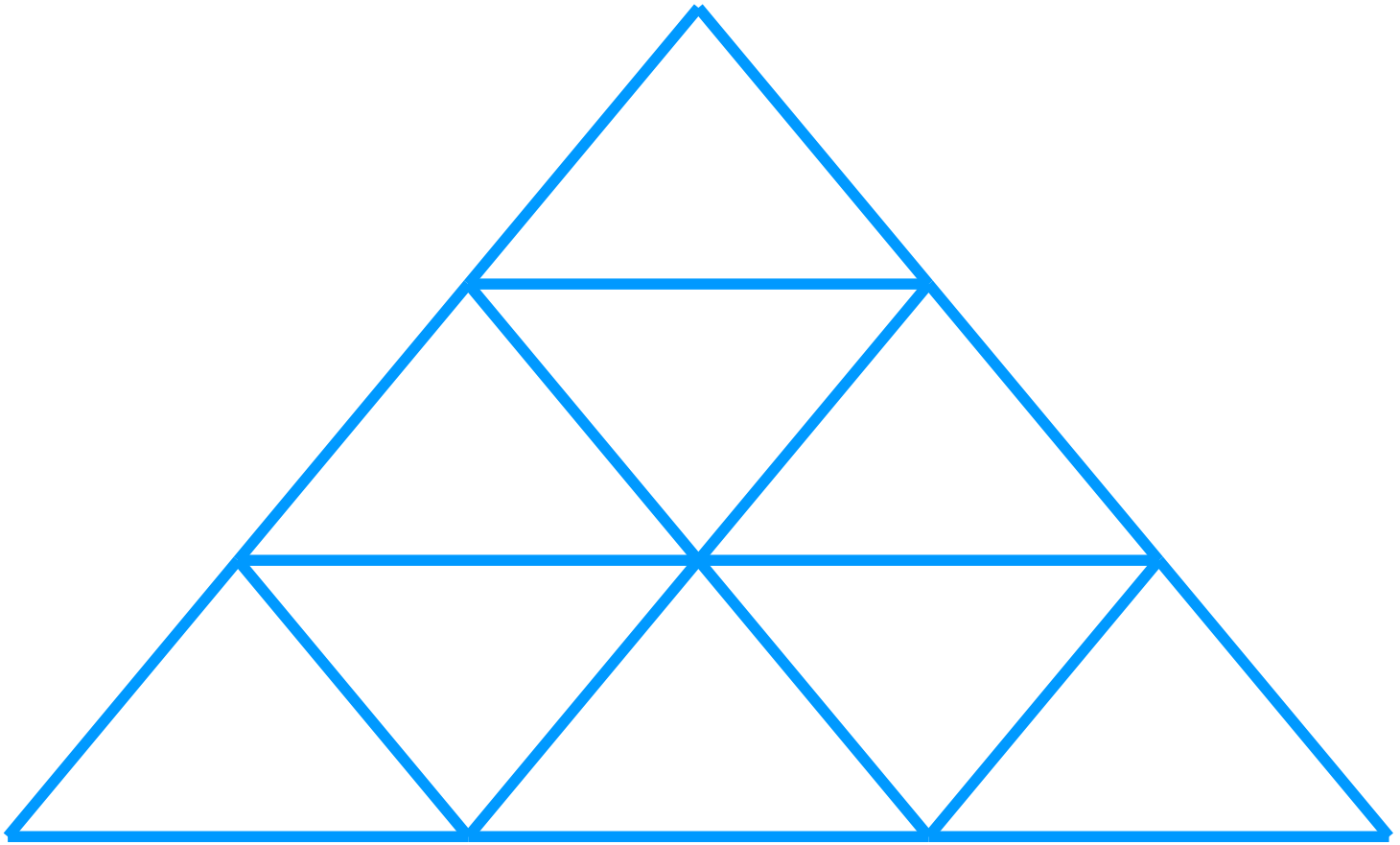


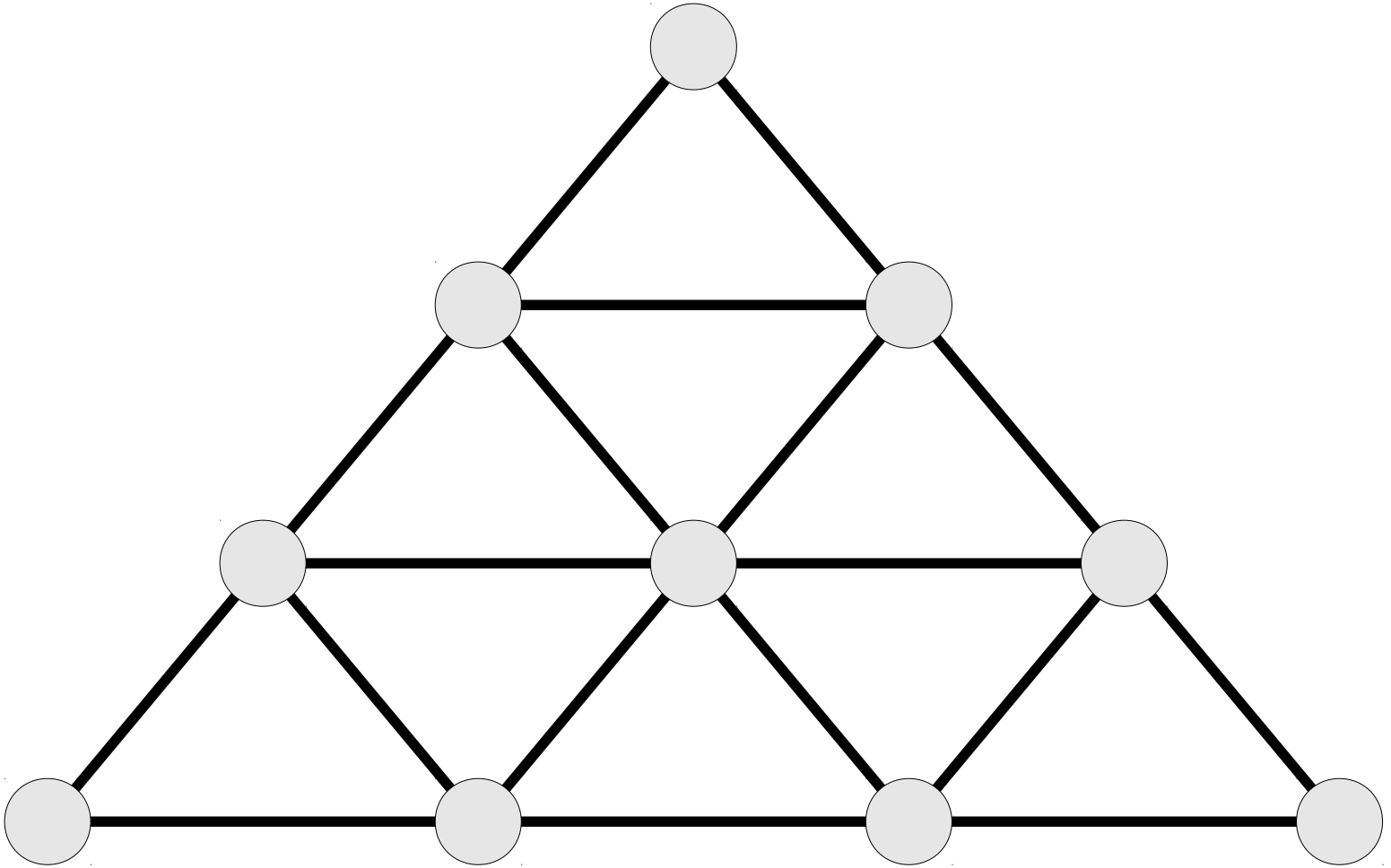


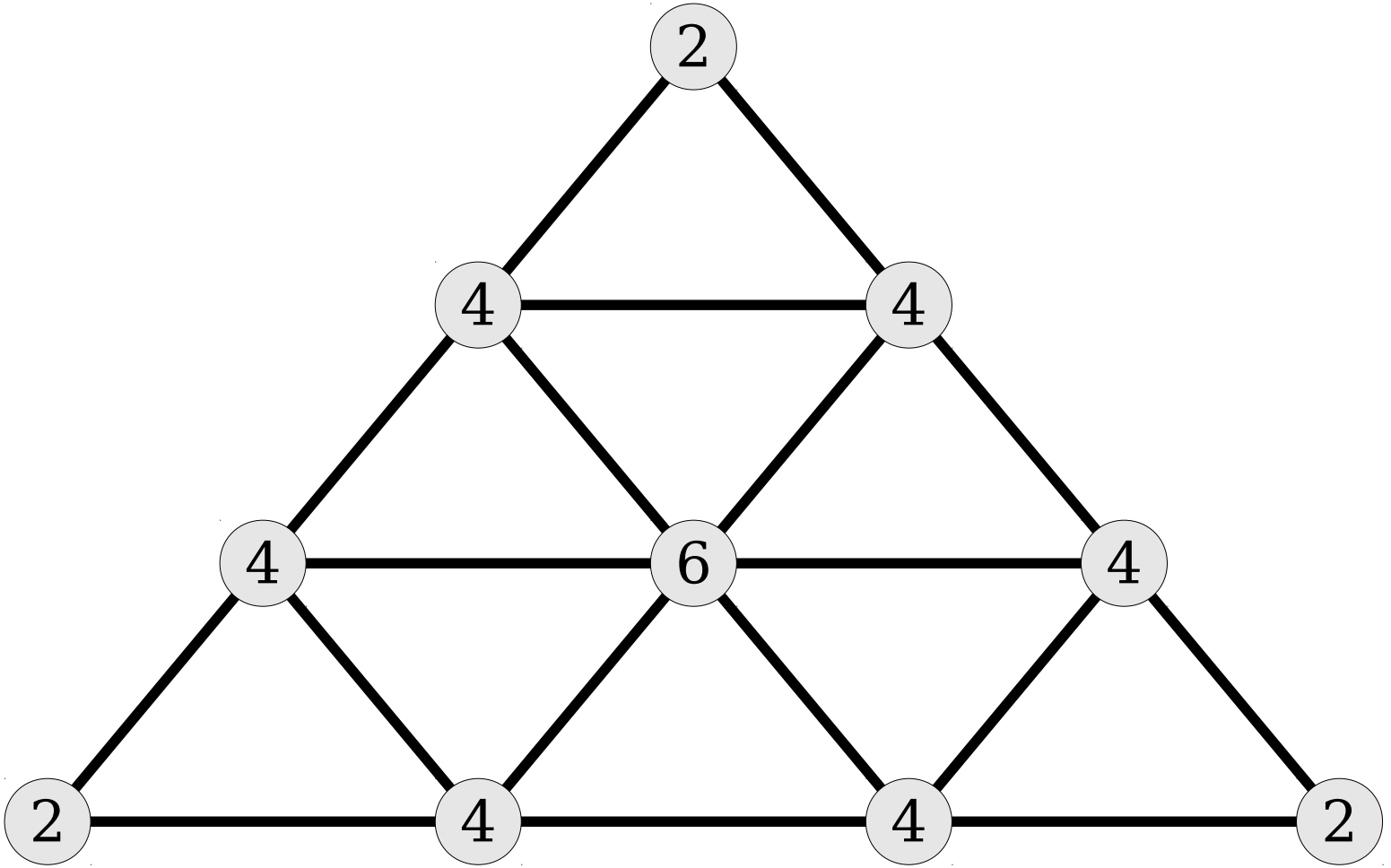










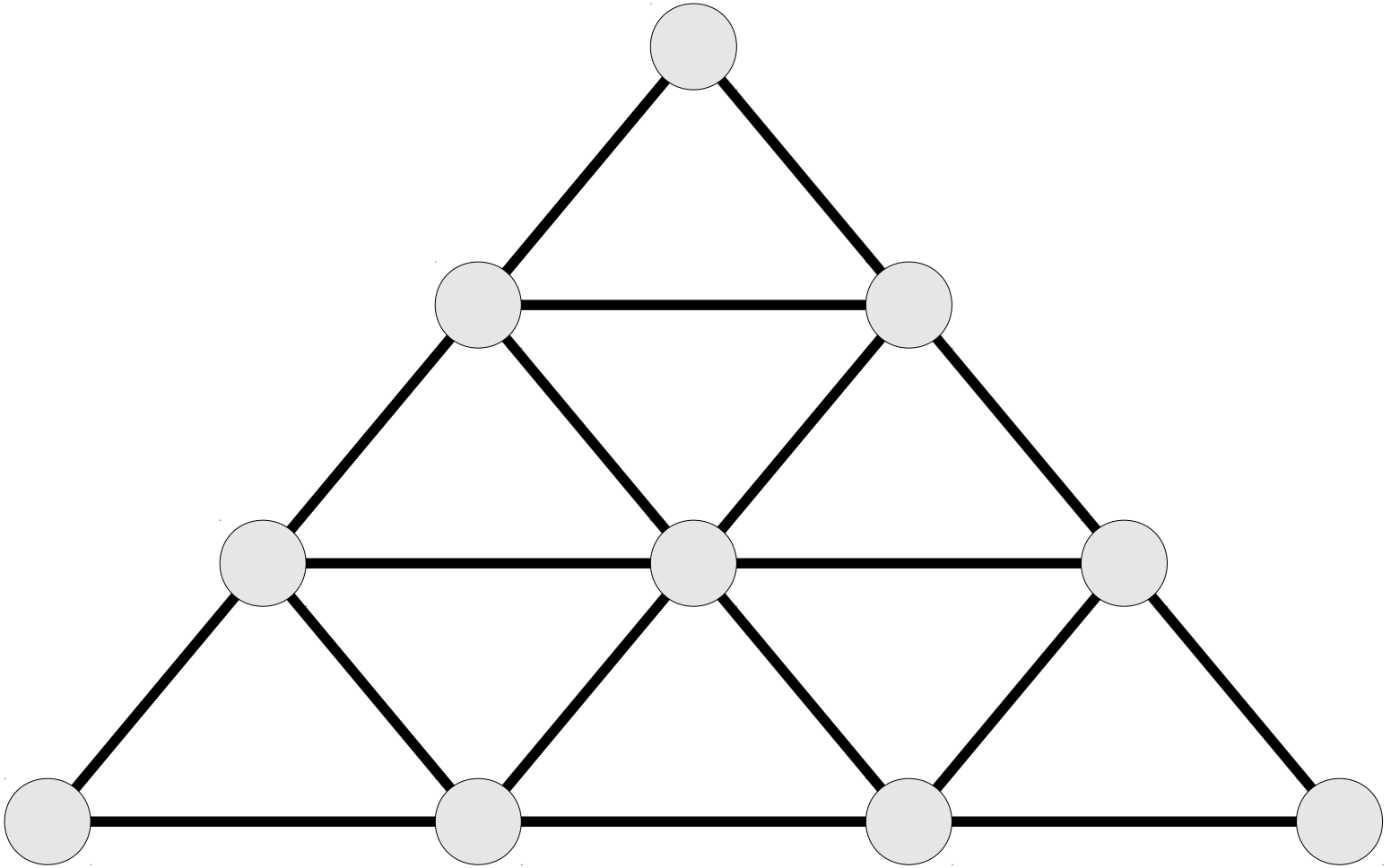

2
4
4
4
6
4
2
4
4
2

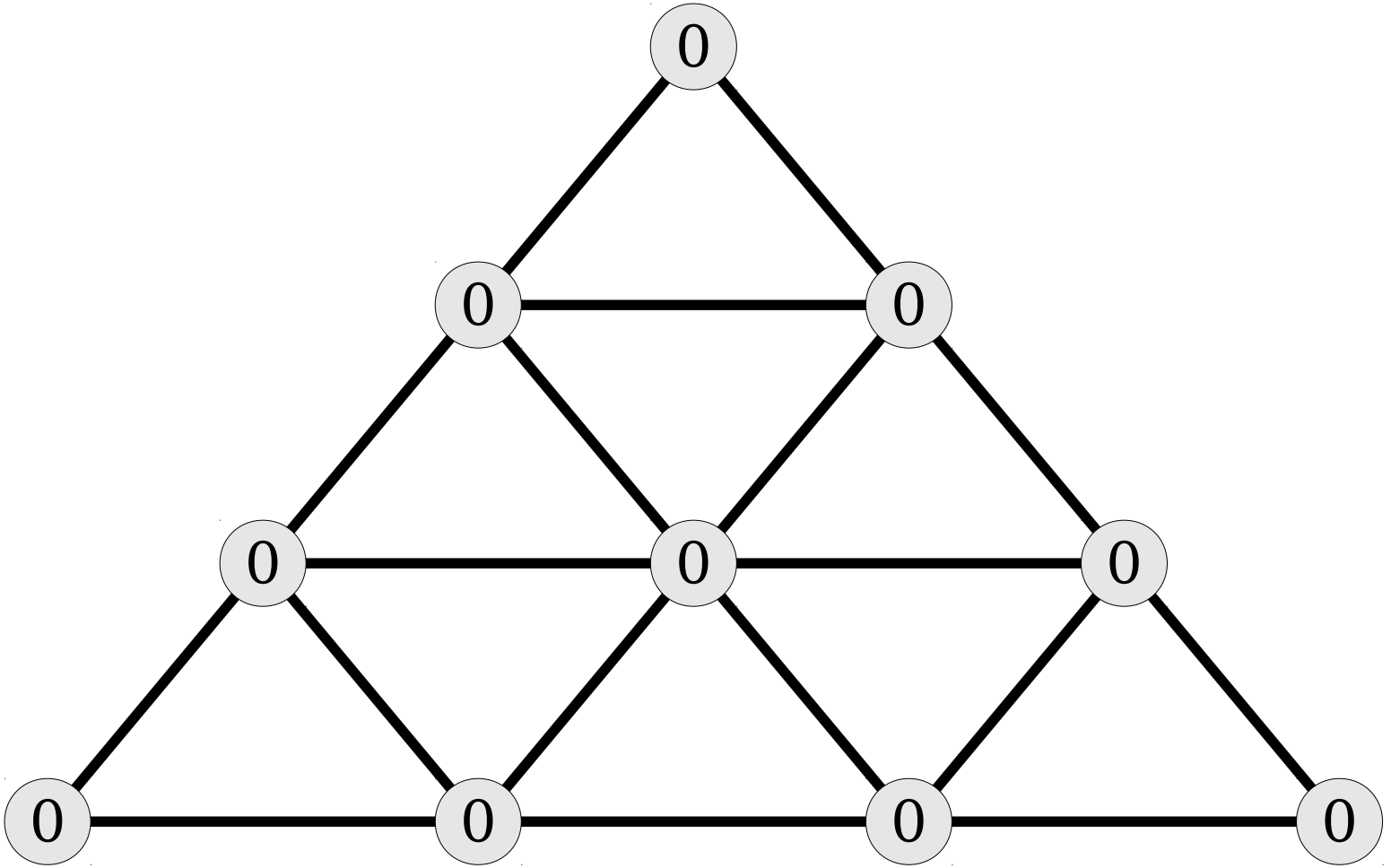
The ***degree*** of a node in a graph is the number of edges touching it (equivalently, the number of nodes it's adjacent to).

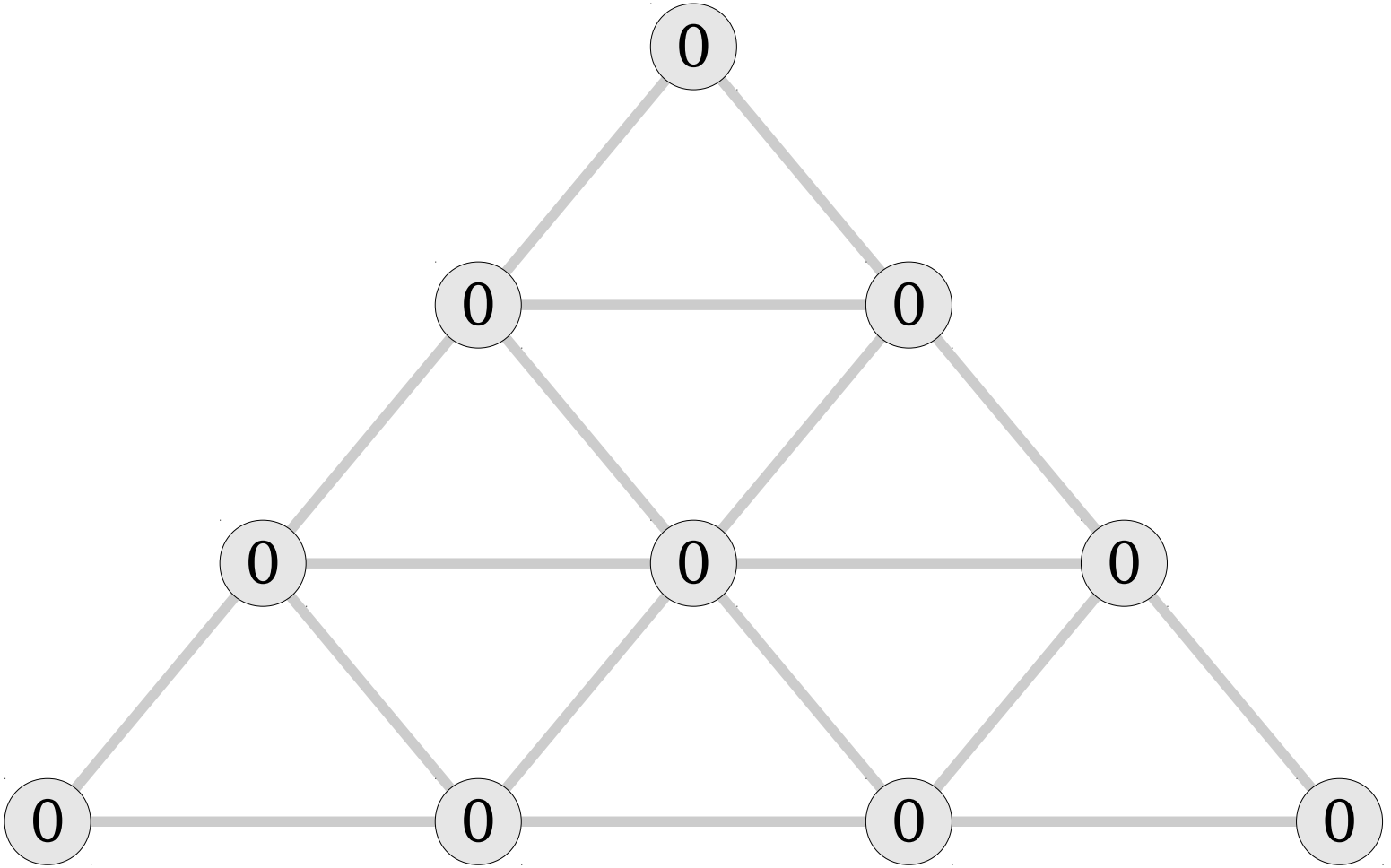
Theorem: An (undirected) graph G is Eulerian if and only if it is connected and every node has even degree.

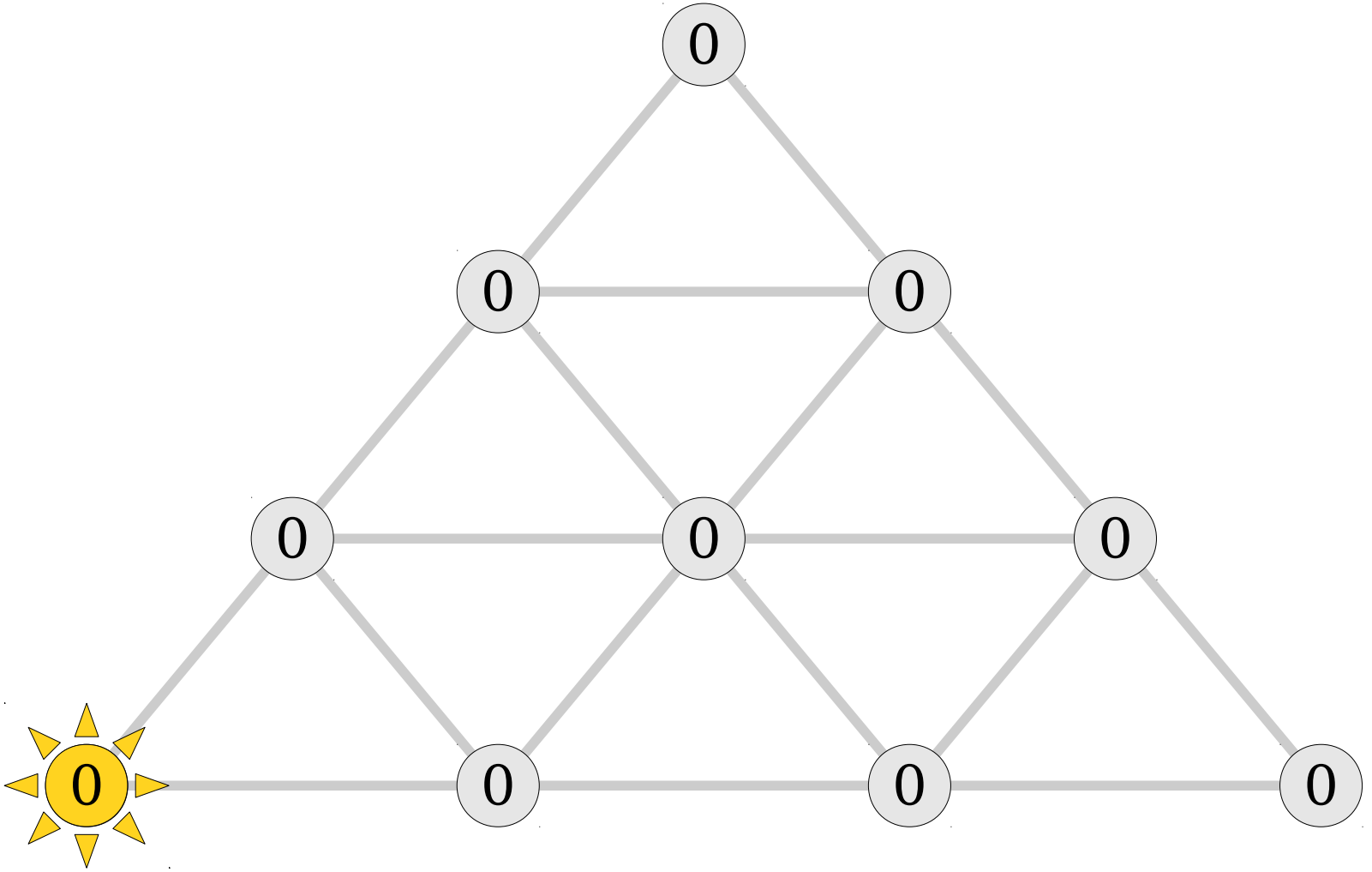
Proving the Theorem

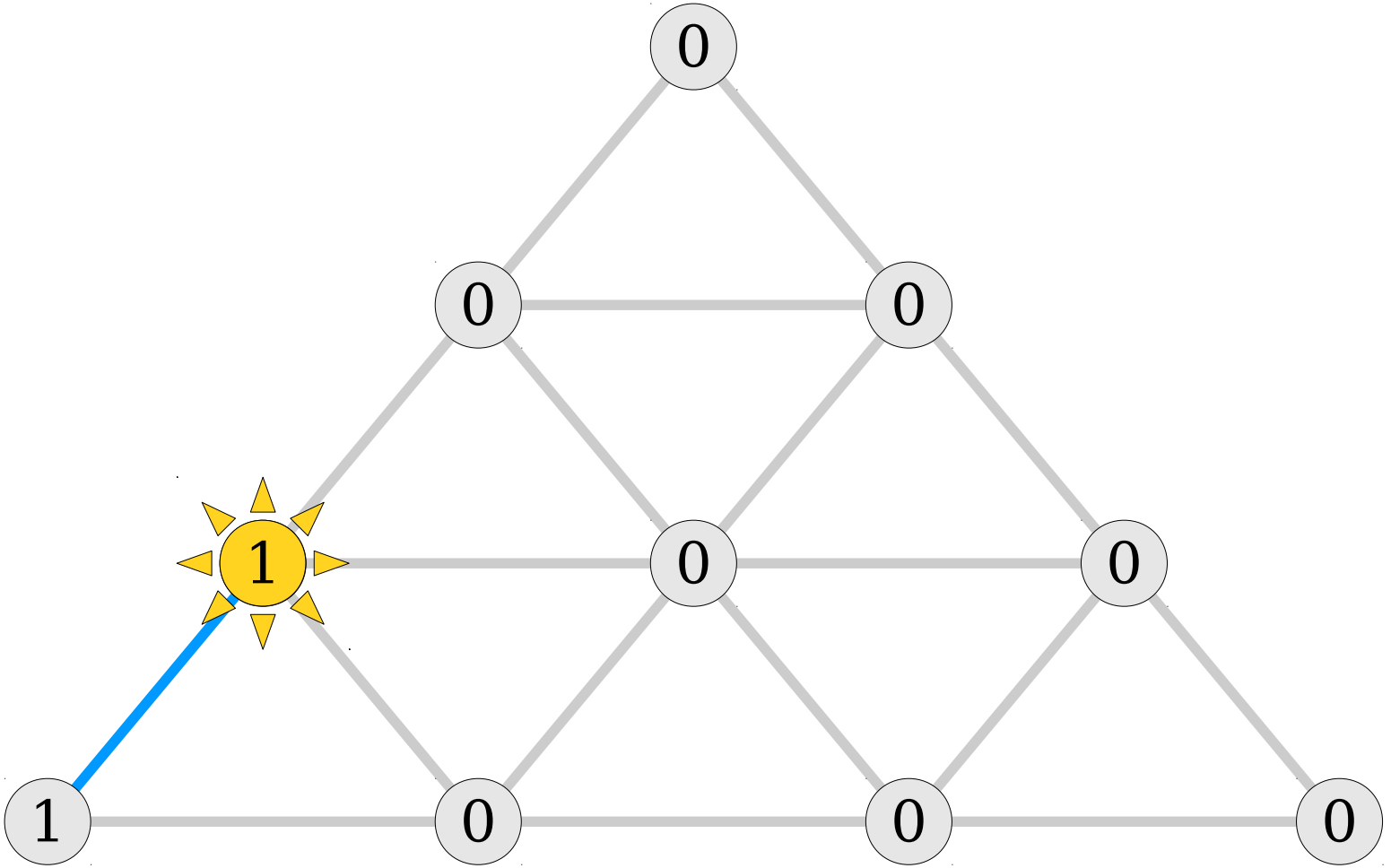
- This theorem is a biconditional, so we need to prove two different directions of implication.
- First, we'll prove that if a graph is Eulerian, then it is connected and every node has even degree. *(Easier)*
- Next, we'll prove that if a graph is connected and every node has even degree, then it is Eulerian. *(Harder)*

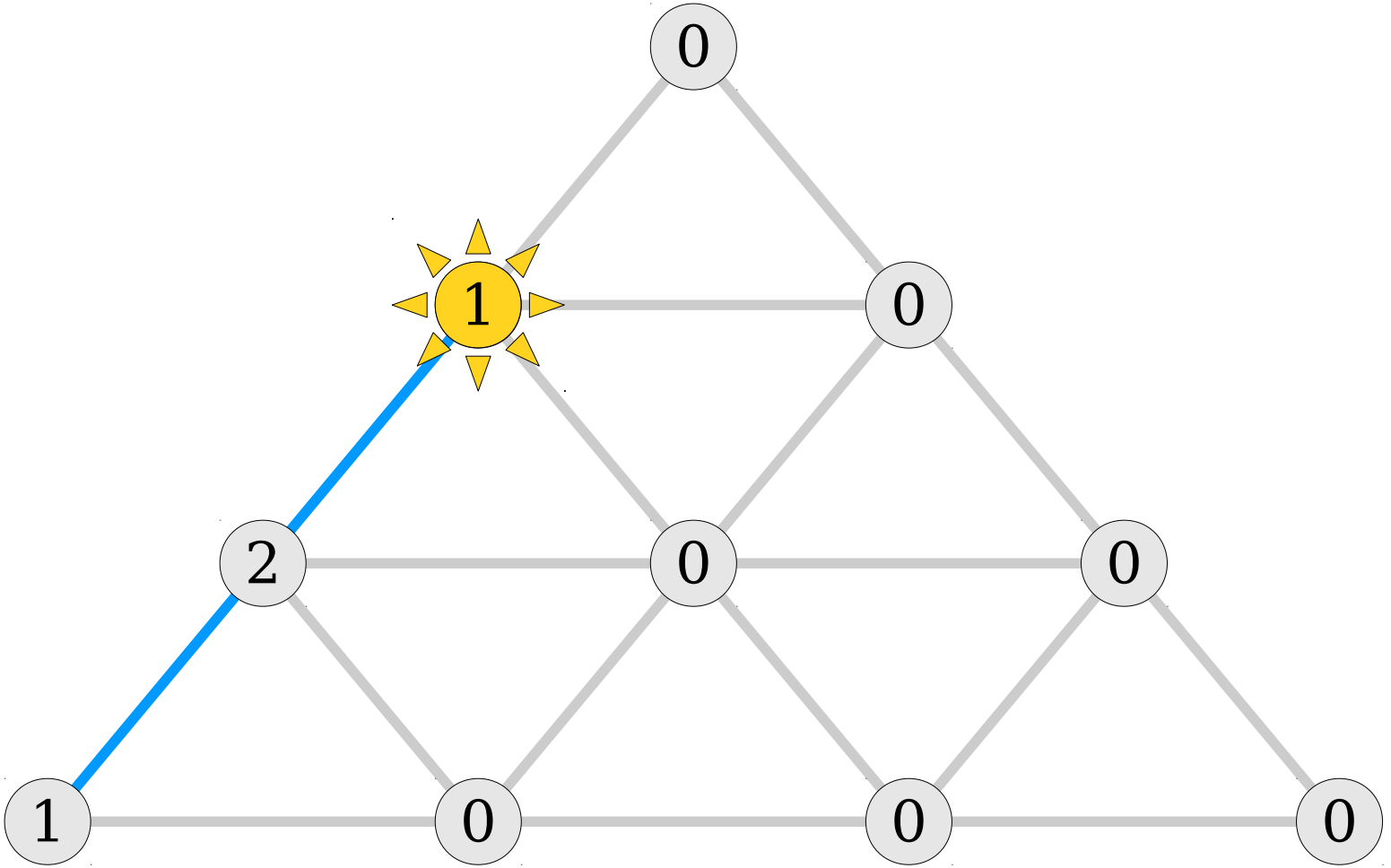


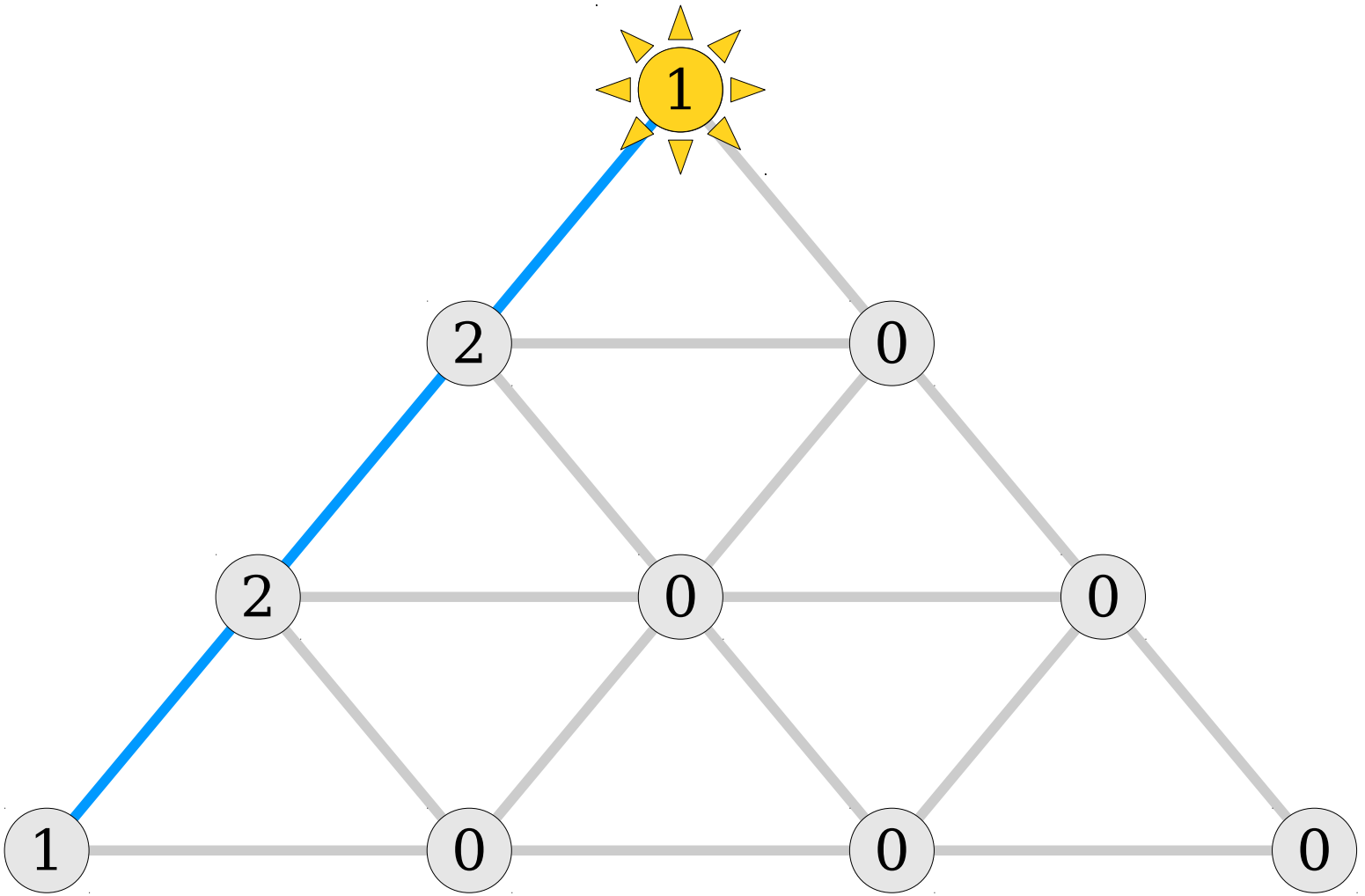

0
0
0
0
0
0
0
0
0
0

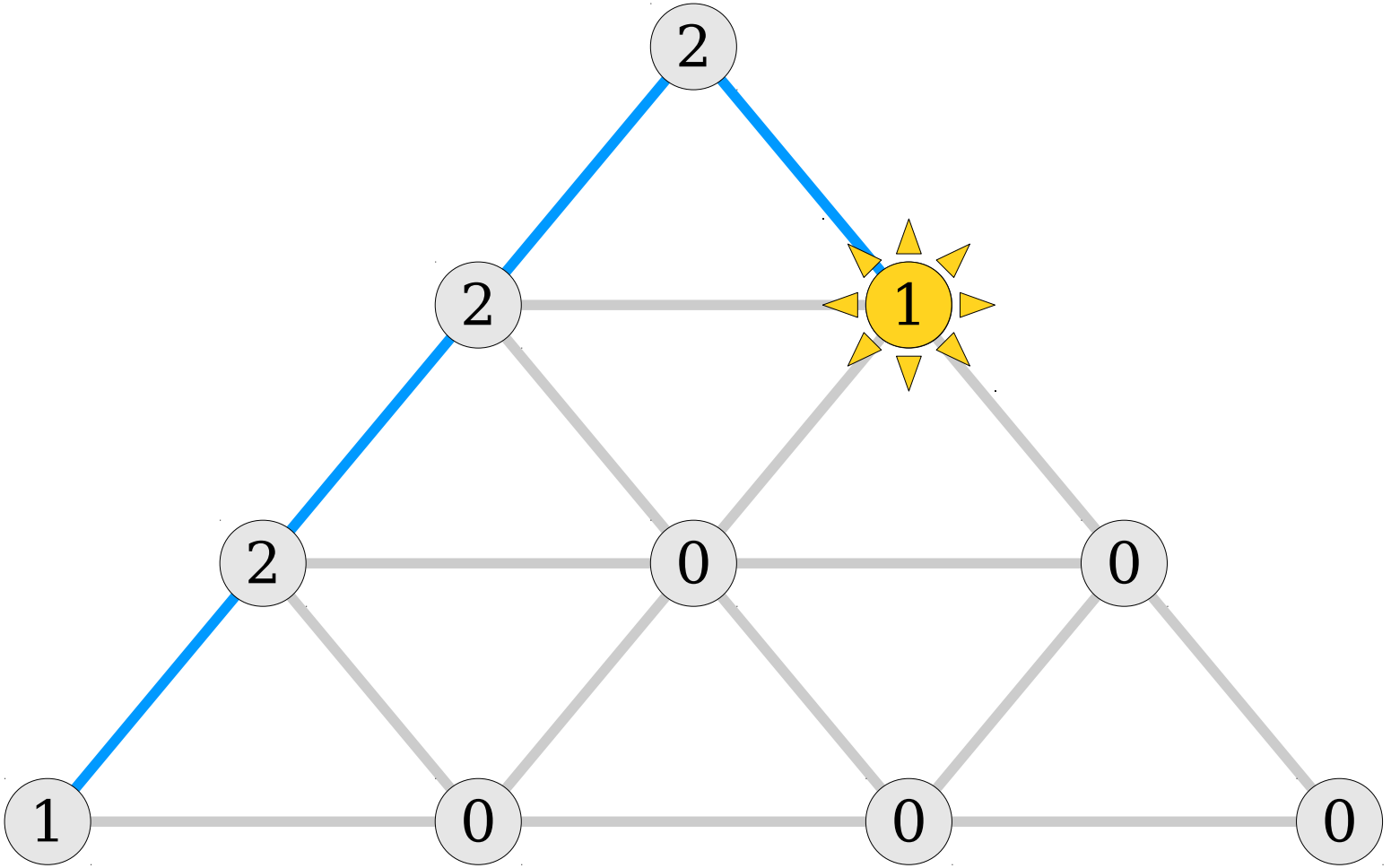

0
0
0
0
0
0
0
0
0
0

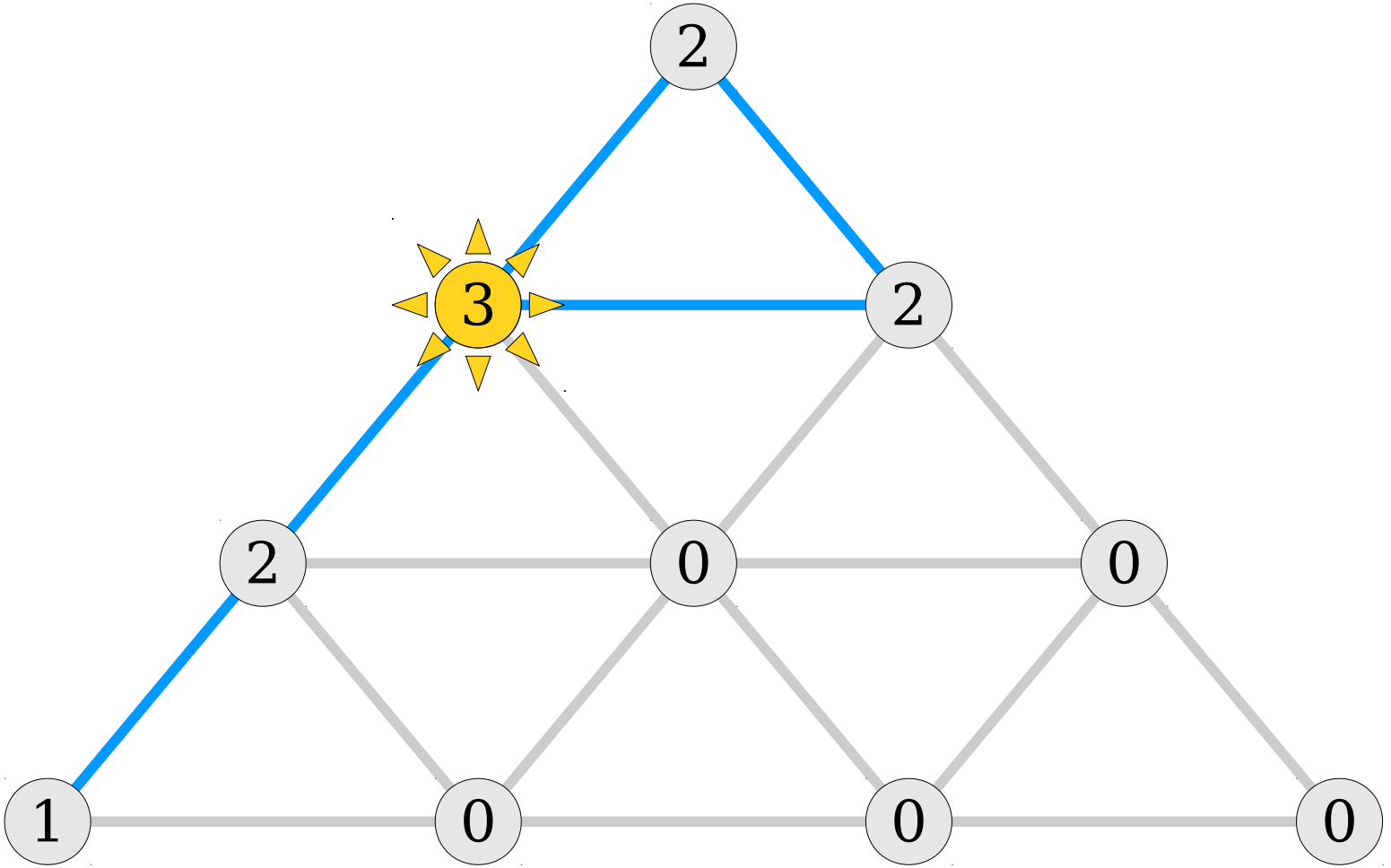

0
0
0
0
0
0
0
0
0
0

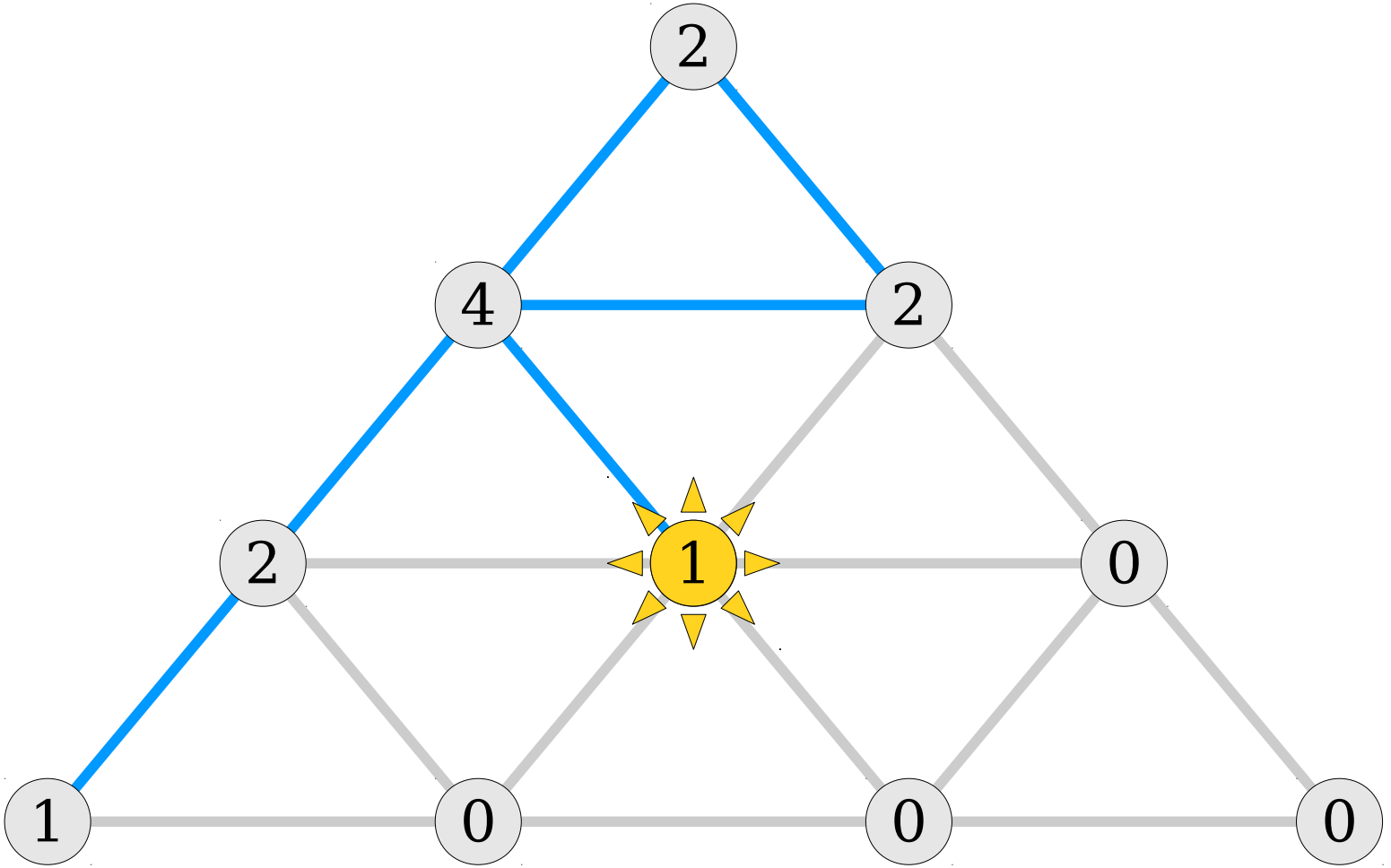

0
0
0
1
0
0
1
0
0
0

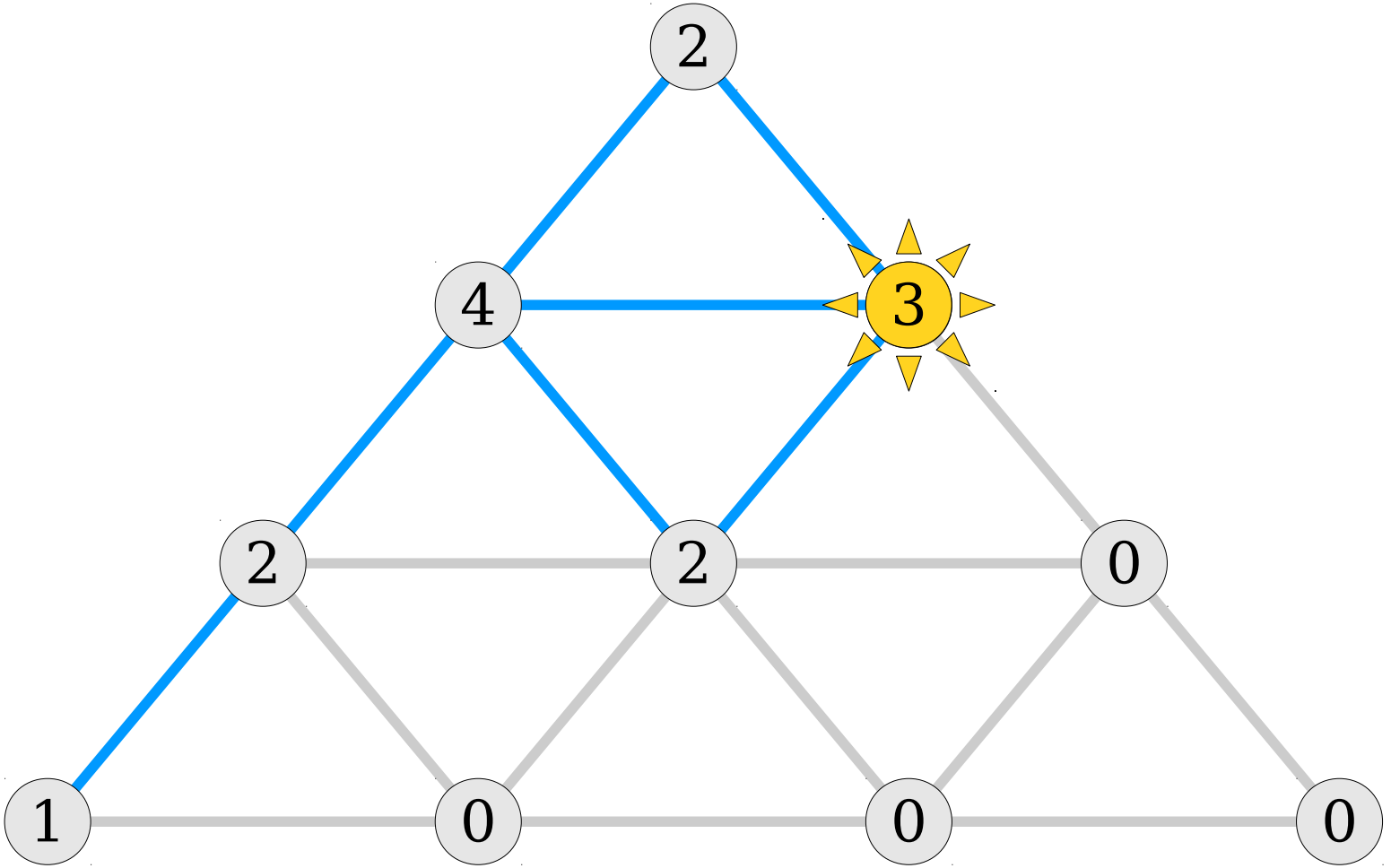

0
1
0
2
0
0
1
0
0
0

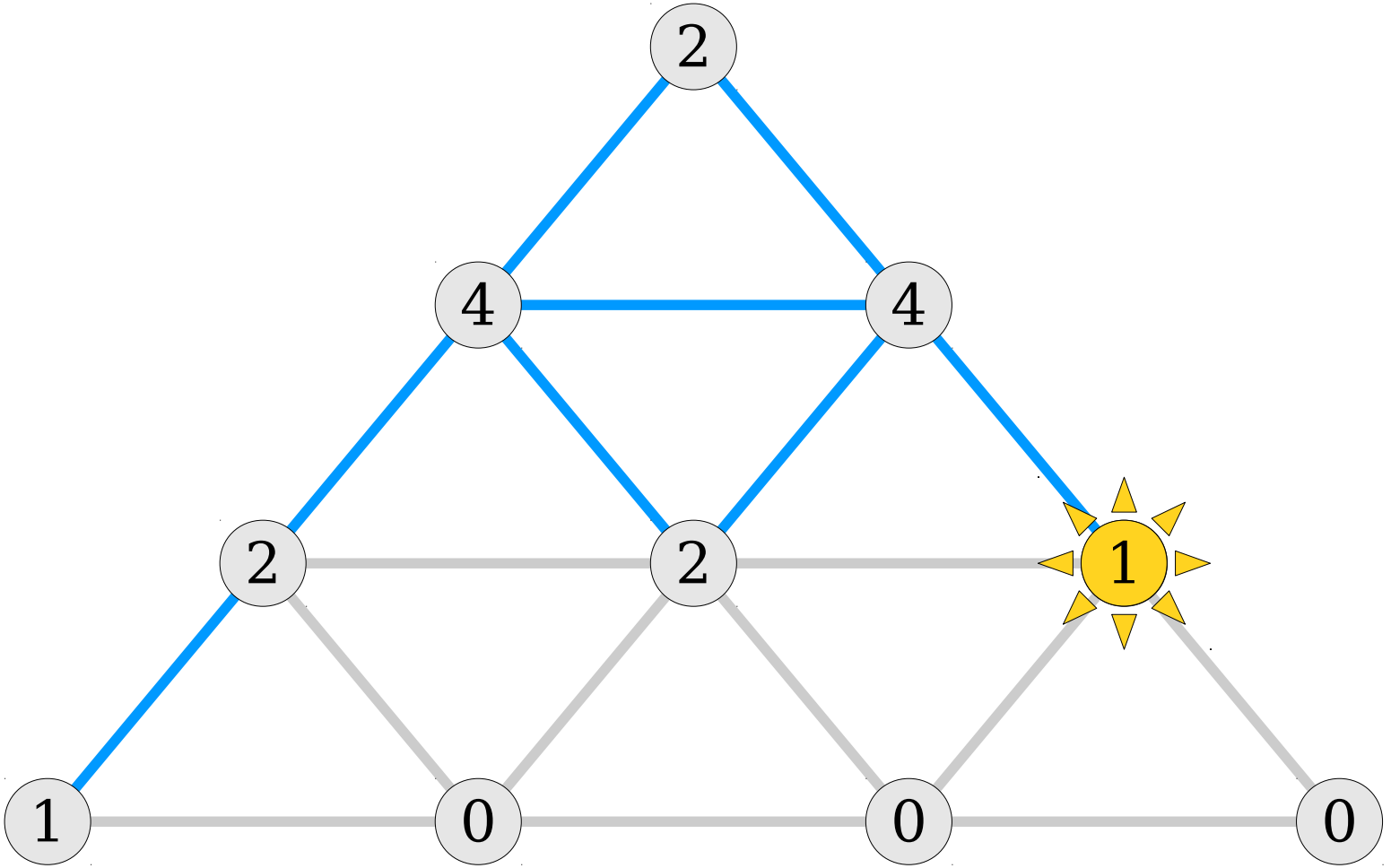

1
2
0
2
0
0
1
0
0
0

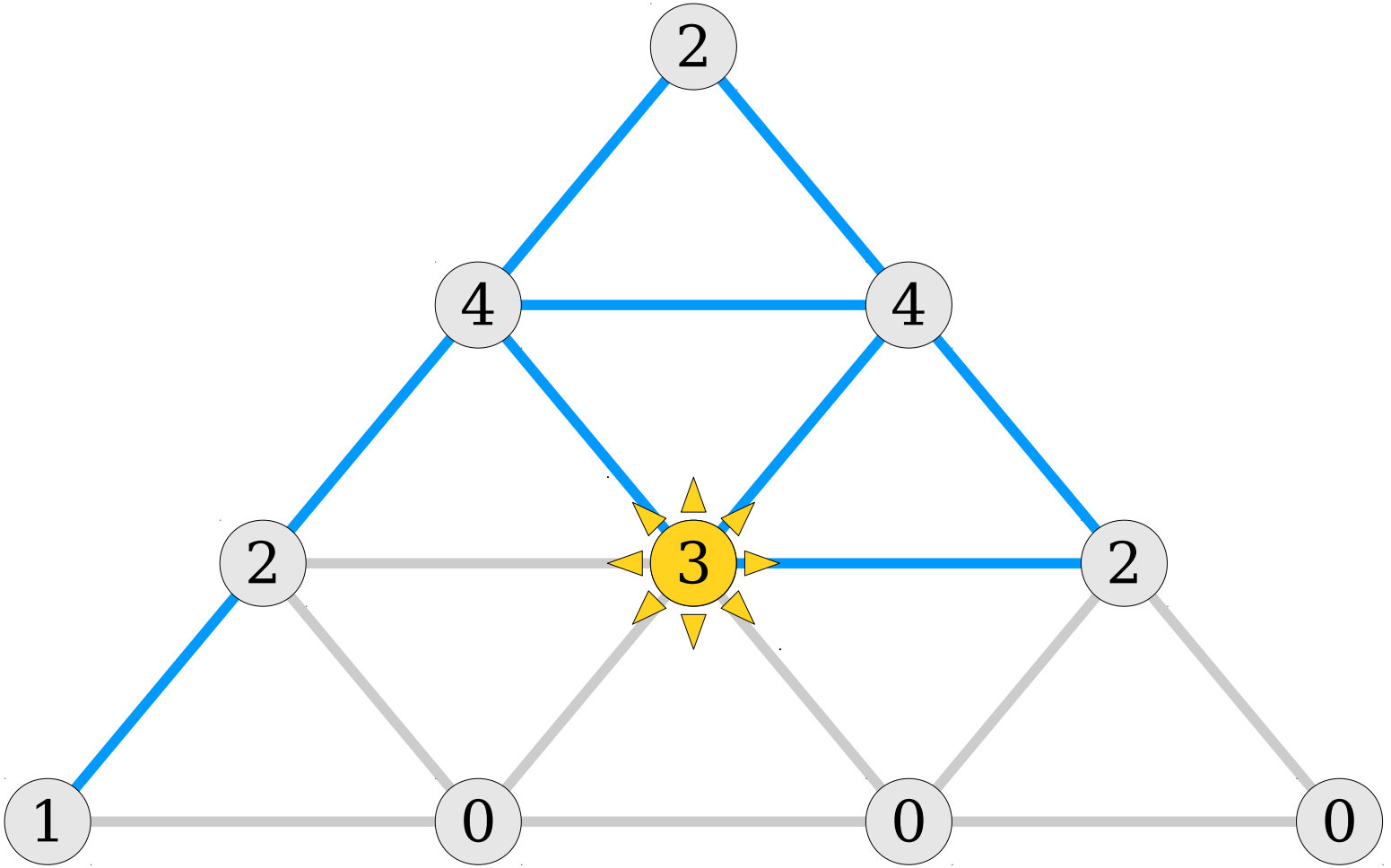

2
2
1
2
0
0
1
0
0
0

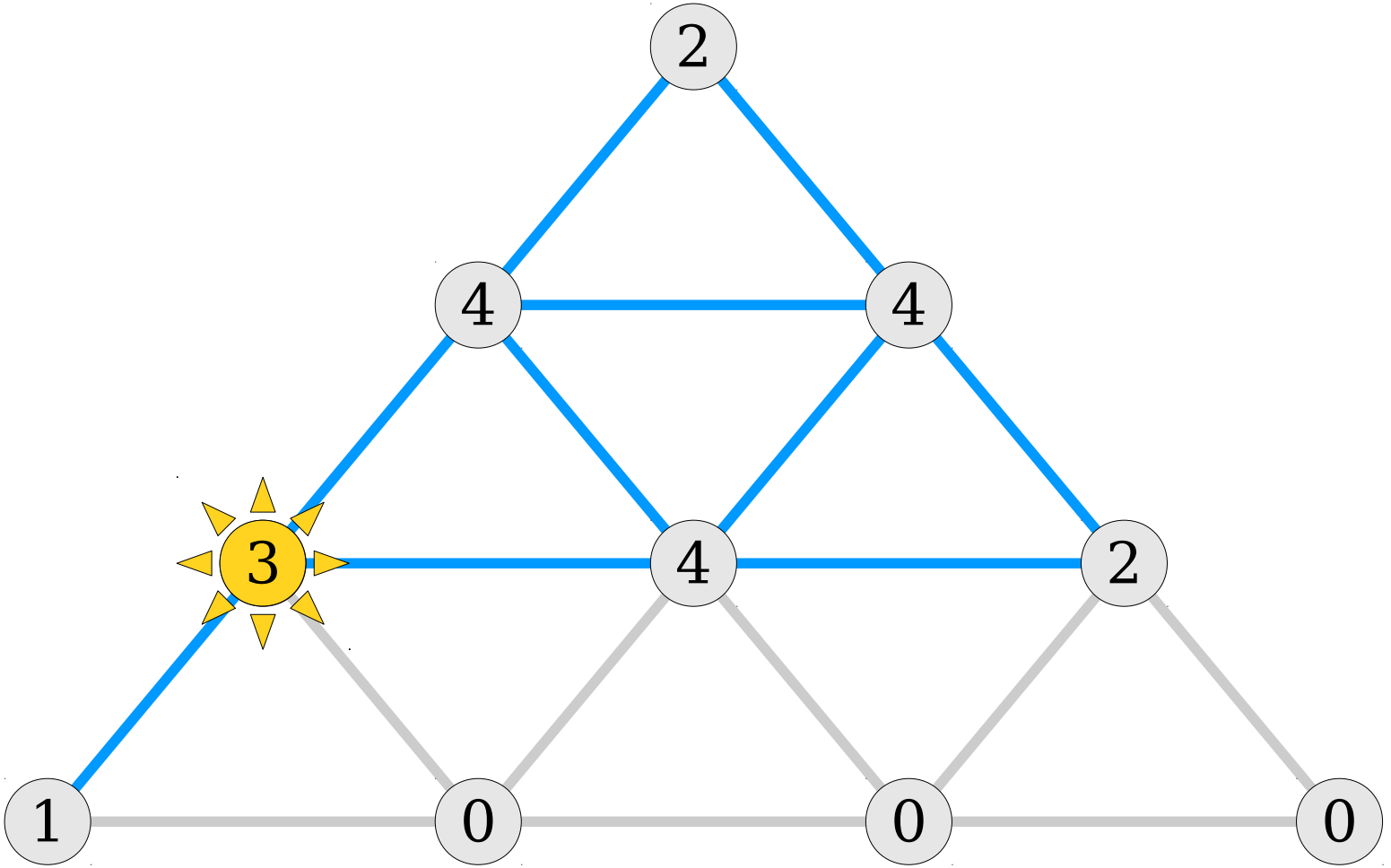

2
3
2
2
0
0
1
0
0
0

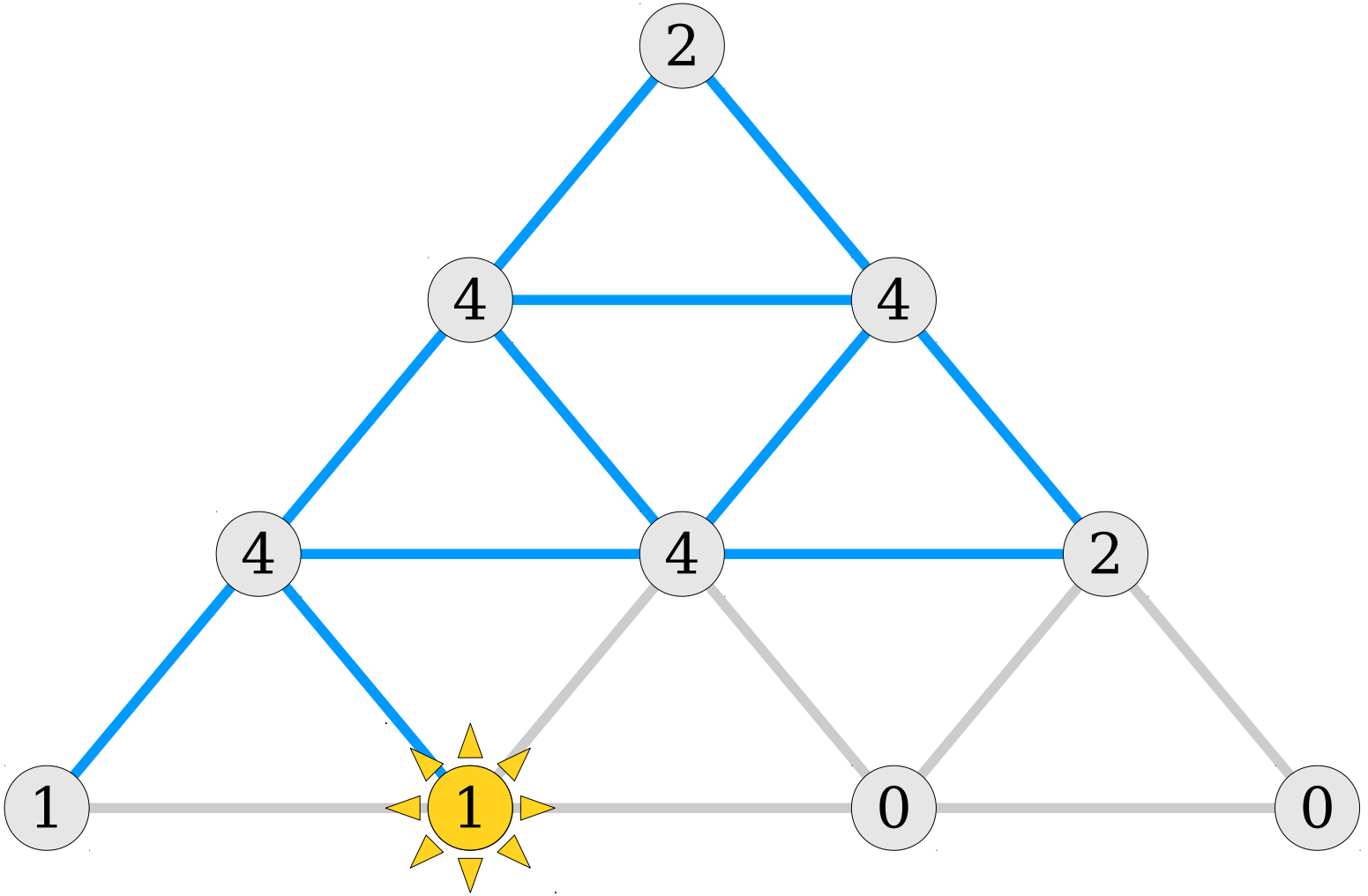

2
4
2
2
1
0
1
0
0
0

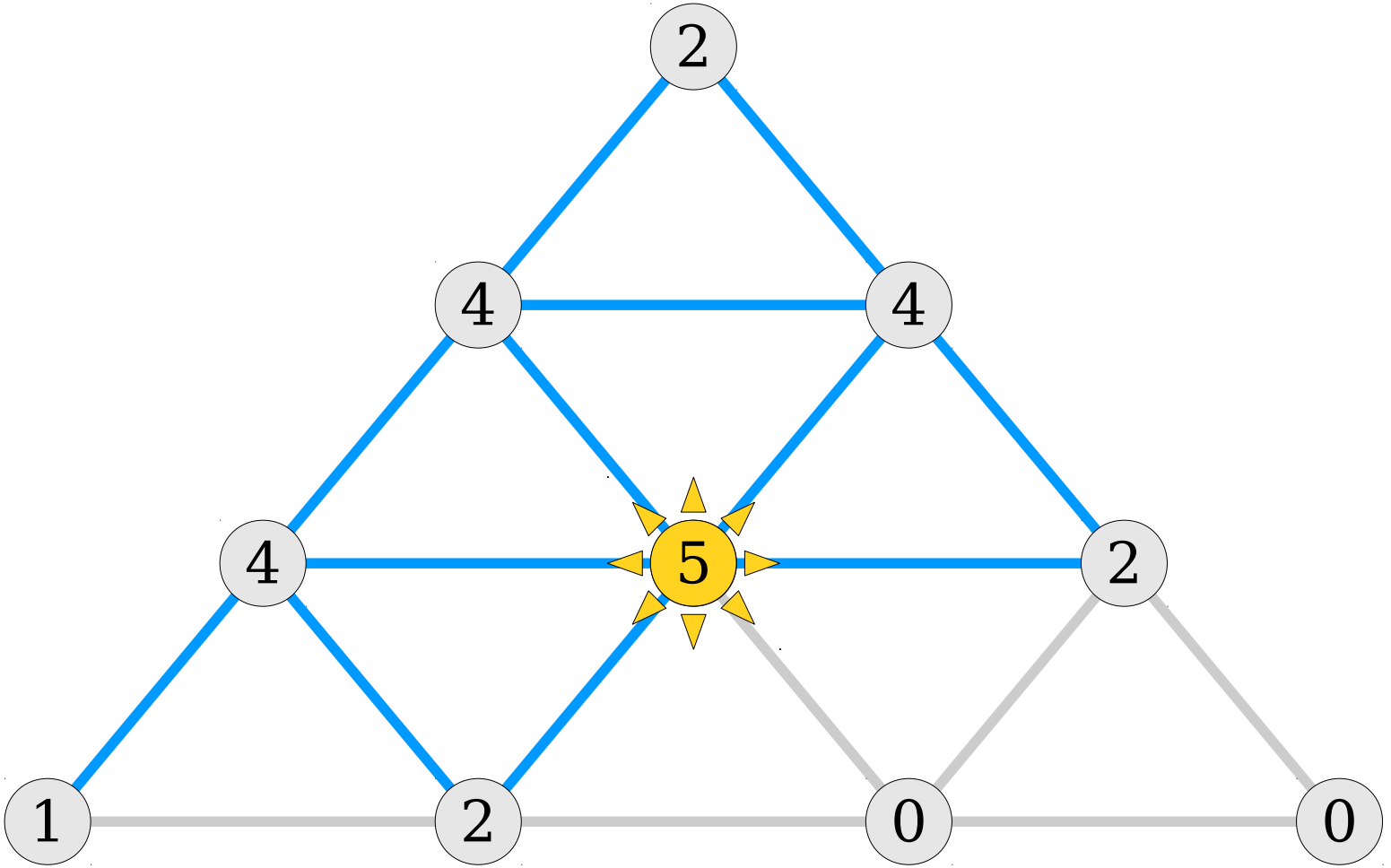

2
4
3
2
2
0
1
0
0
0

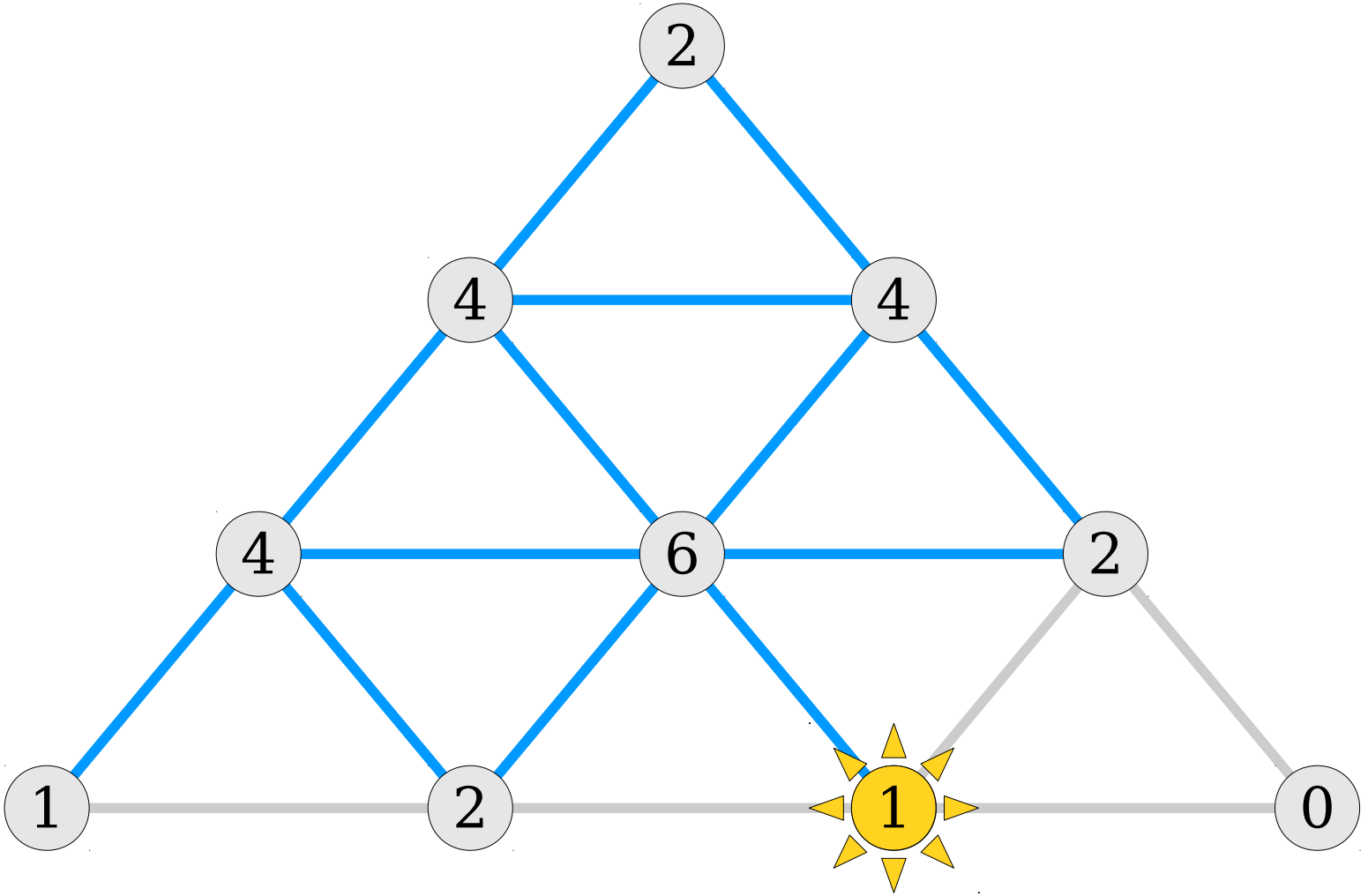

2
4
4
2
2
1
1
0
0
0

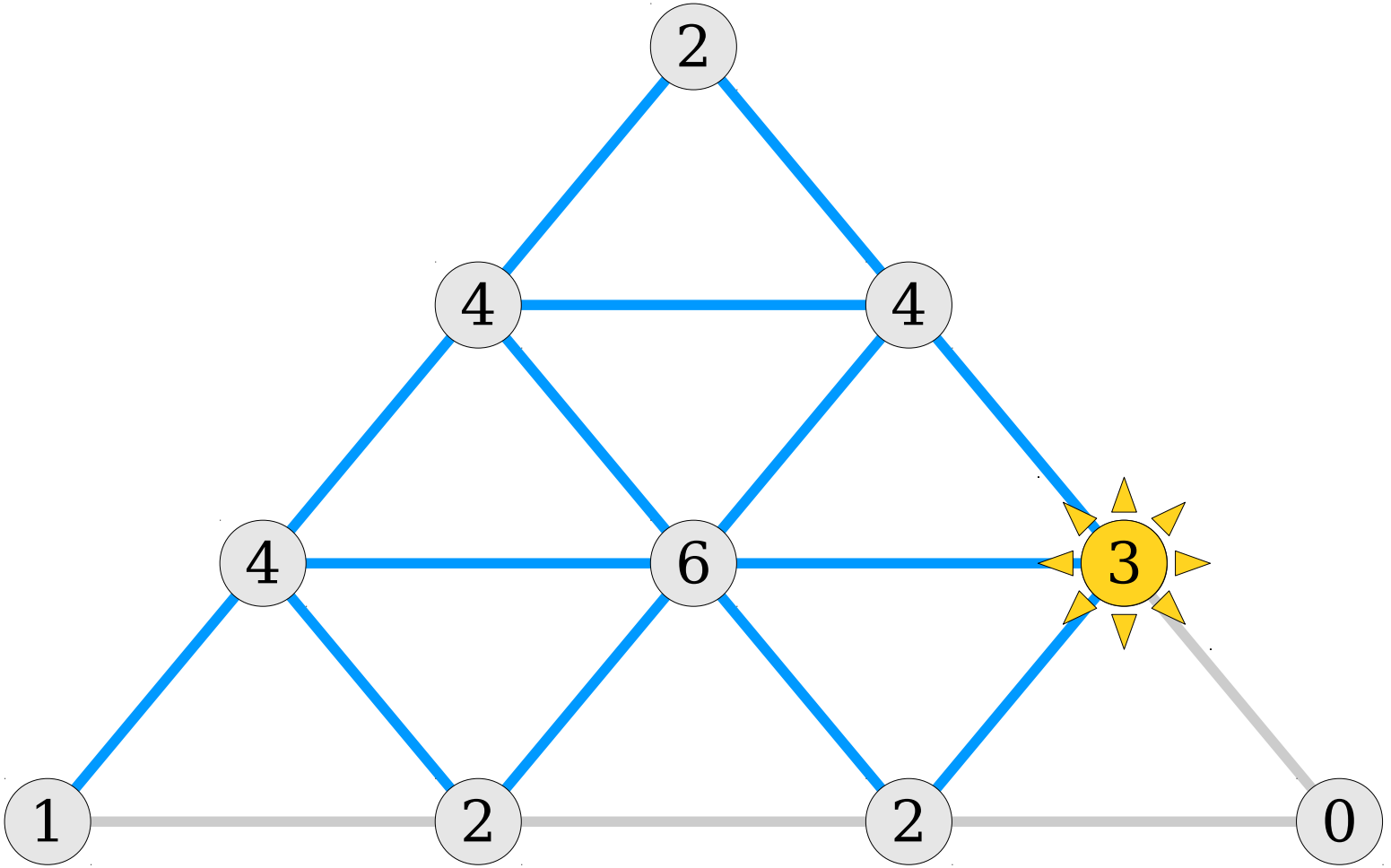

2
4
4
2
3
2
1
0
0
0

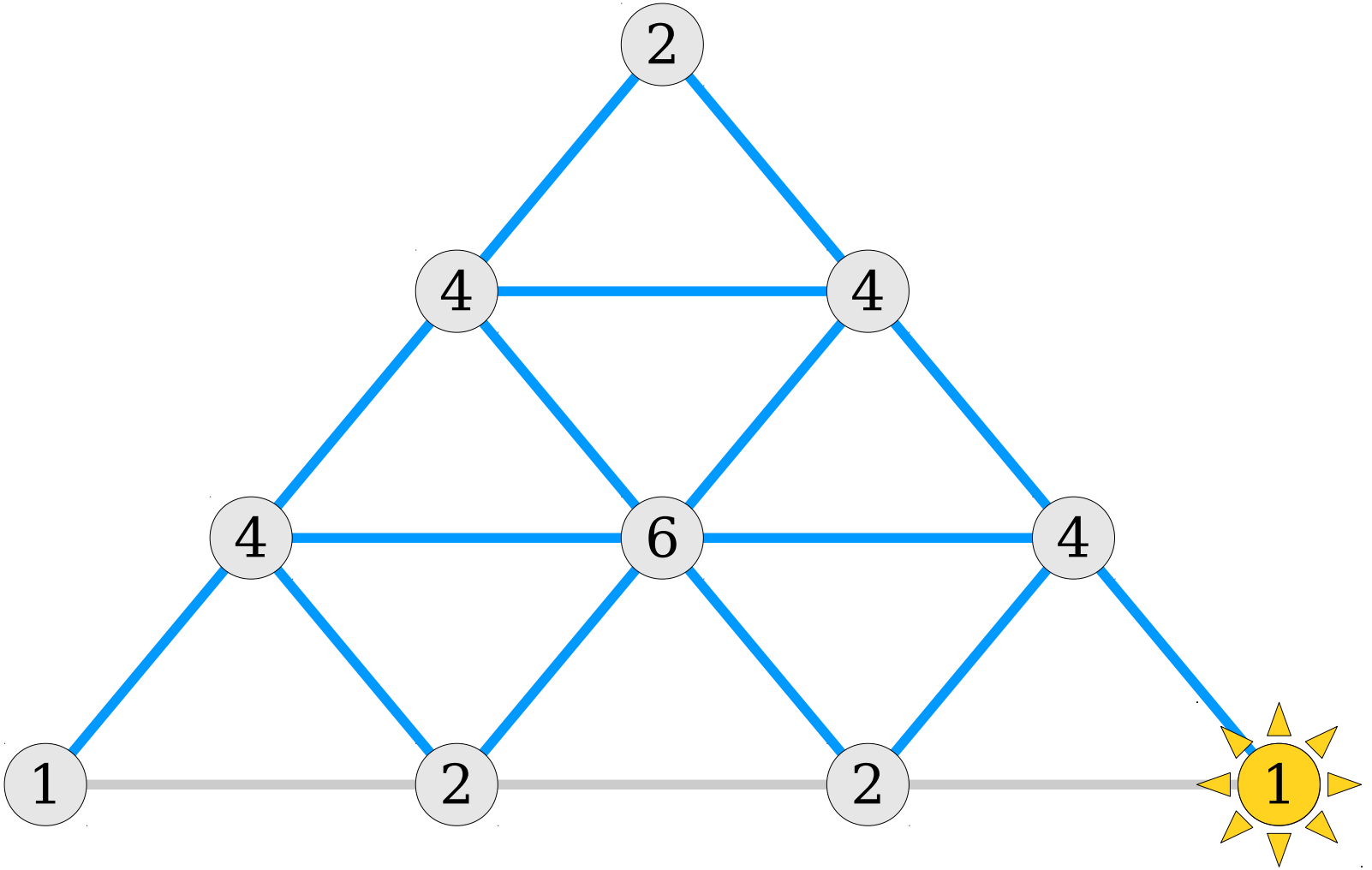

2
4
4
3
4
2
1
0
0
0

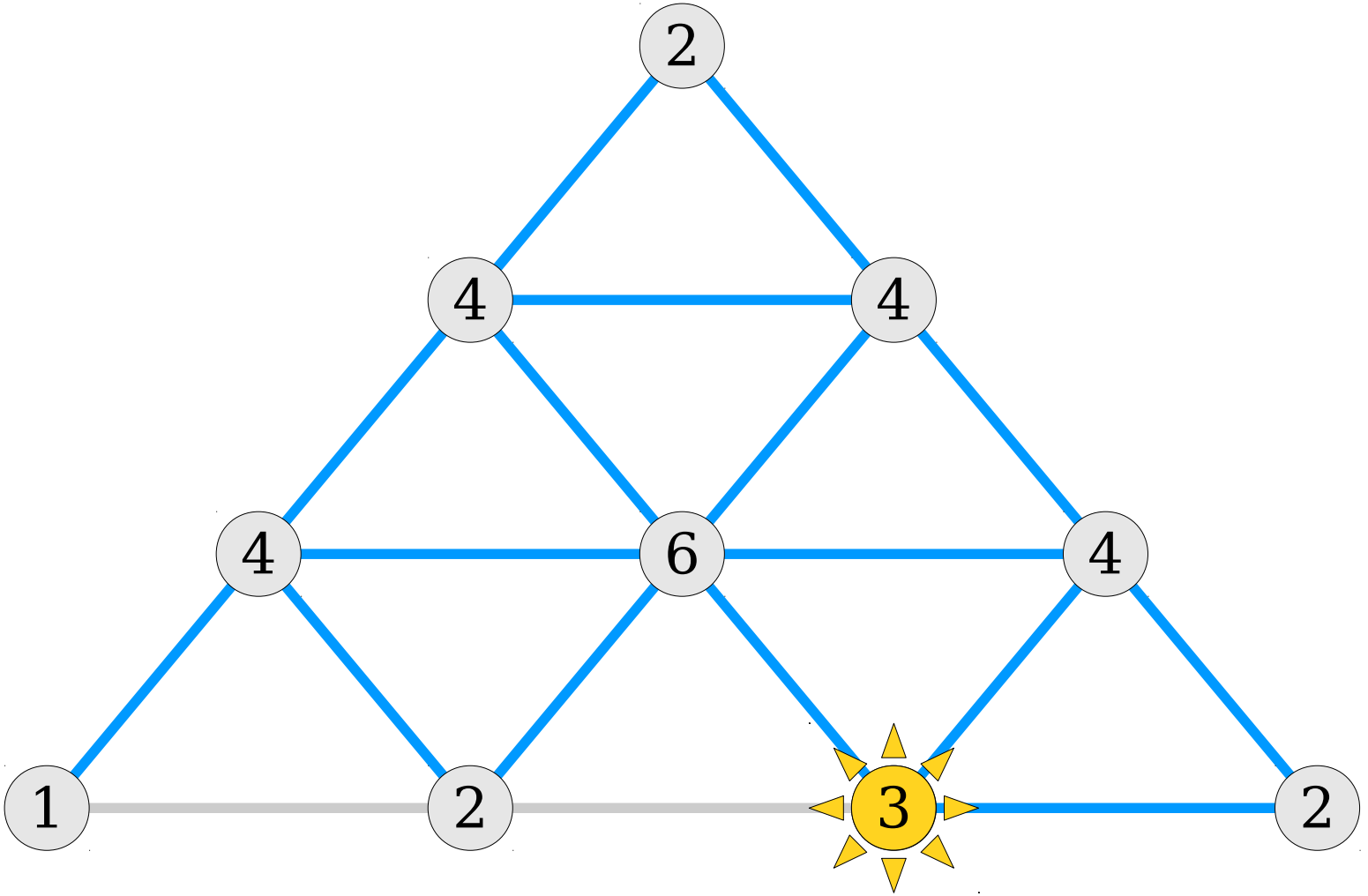

2
4
4
4
4
2
1
1
0
0

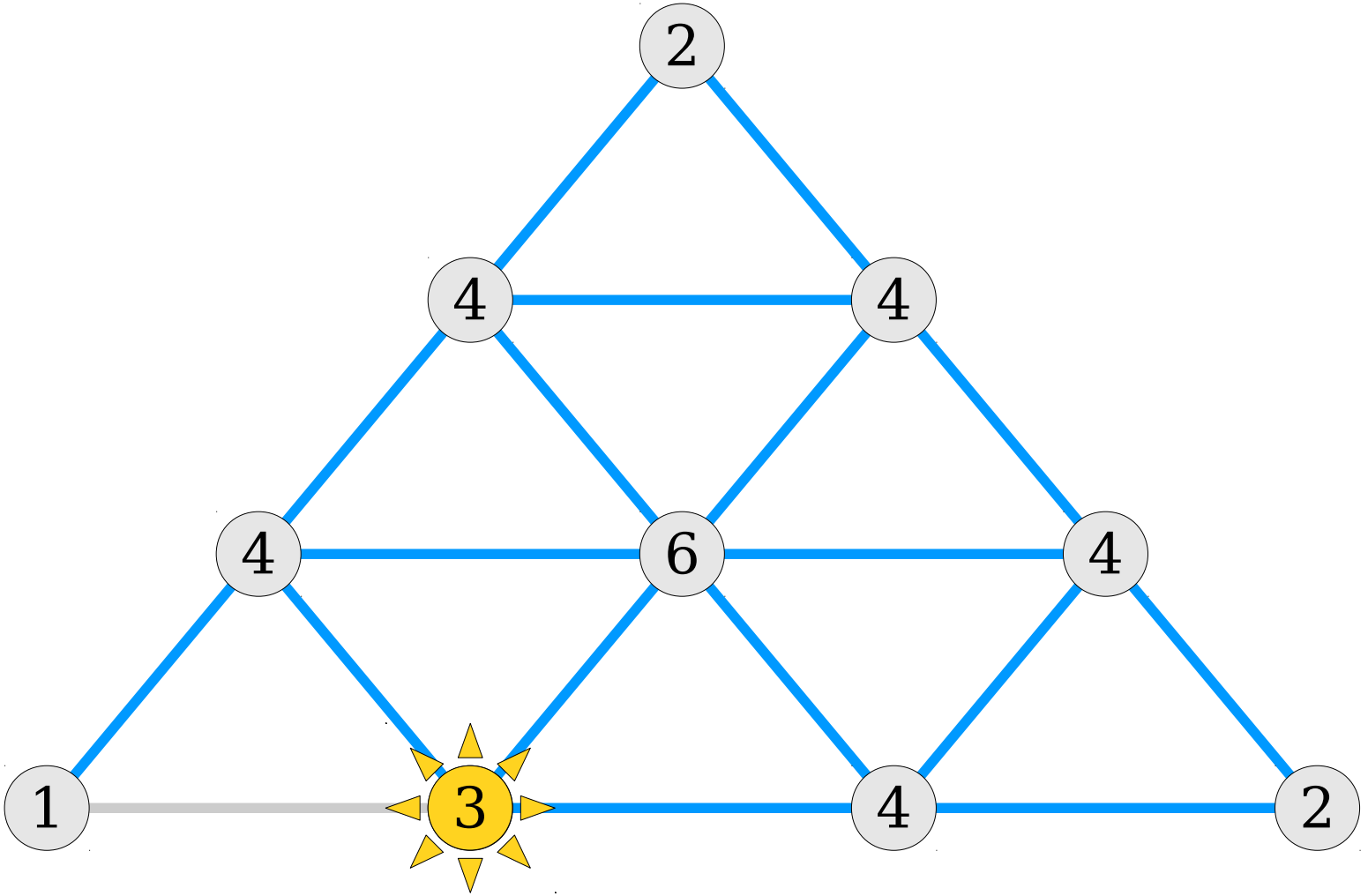

2
4
4
4
5
2
1
2
0
0

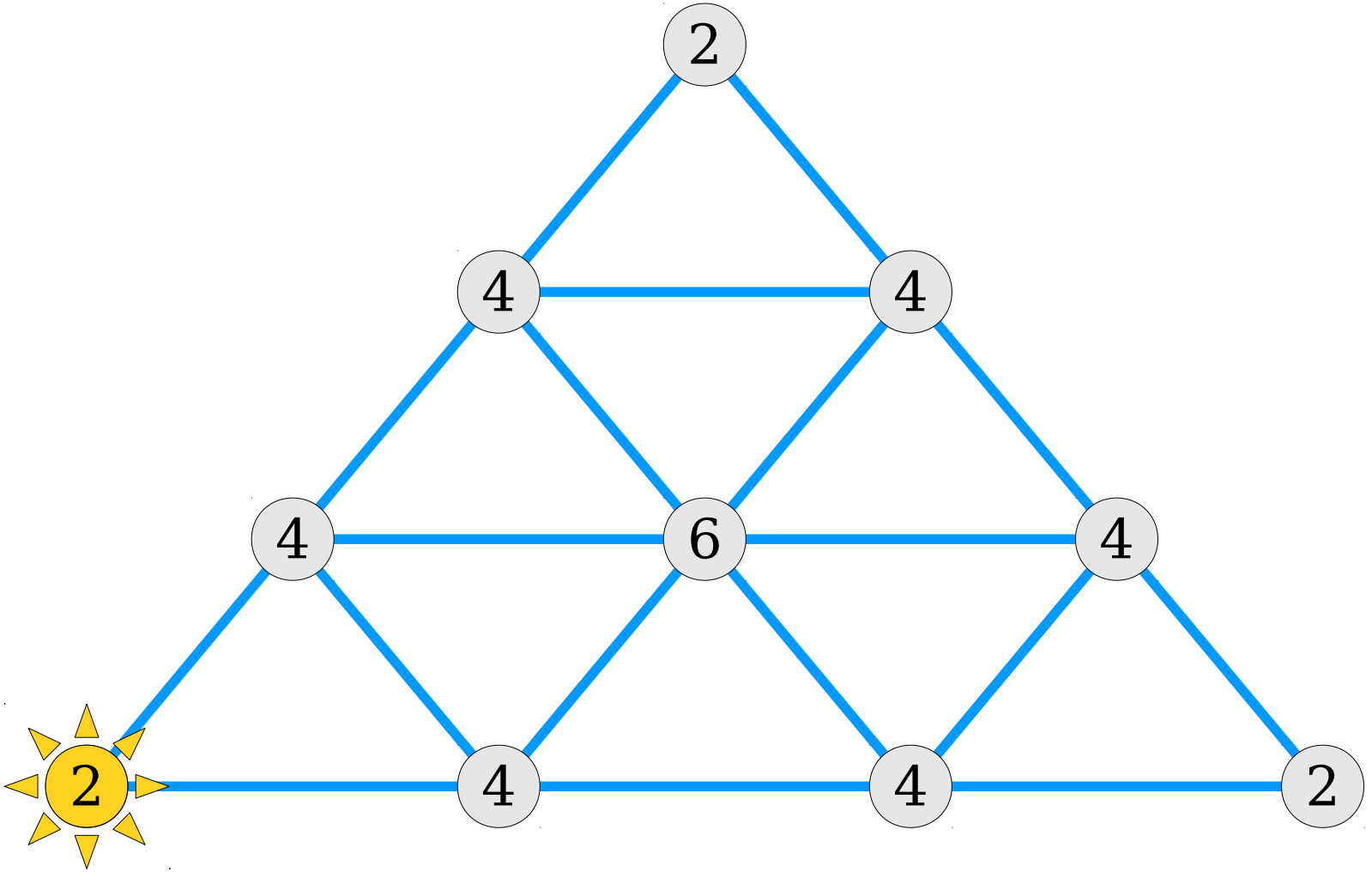

2
4
4
4
6
2
1
2
1
0

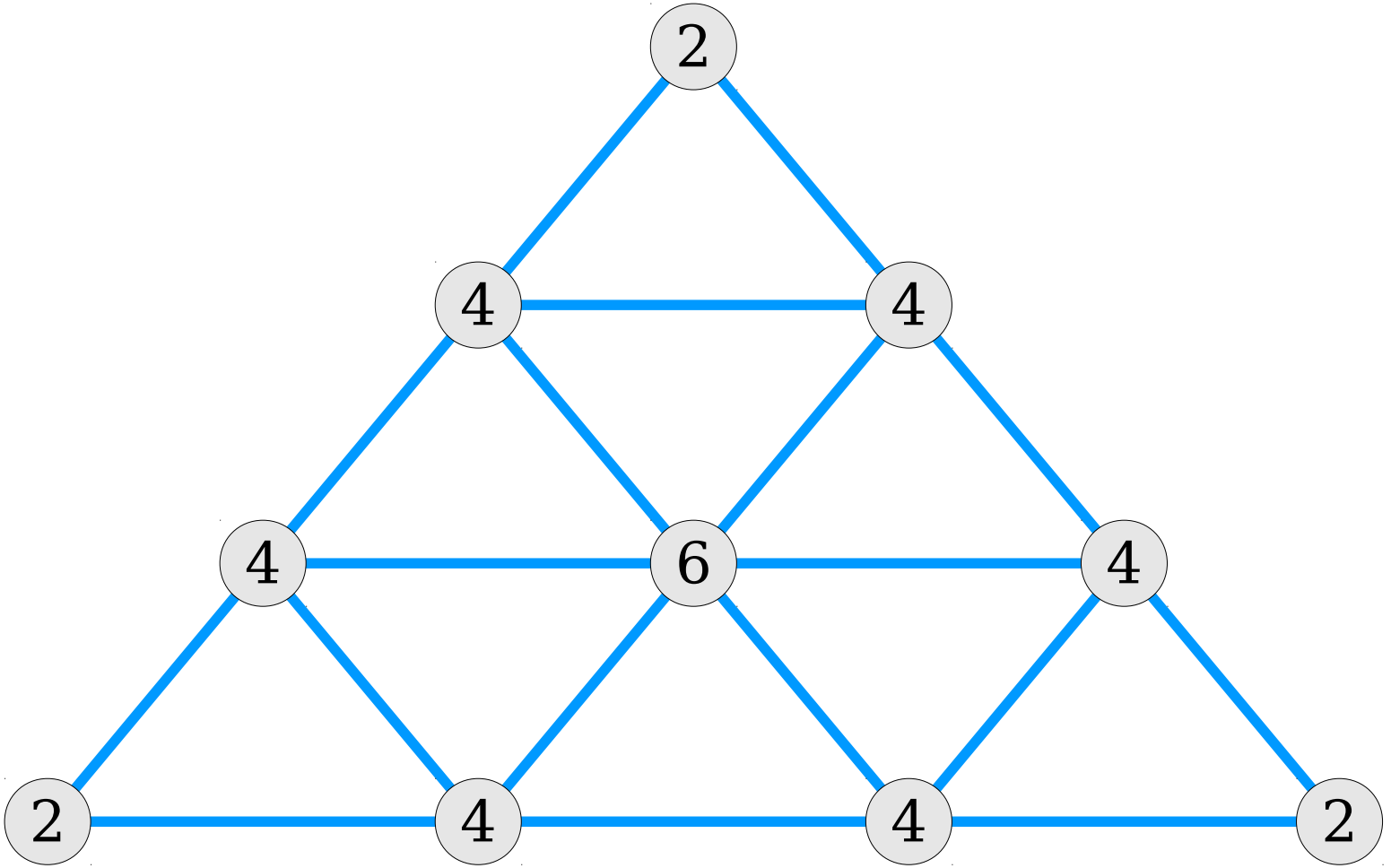

2
4
4
4
6
3
1
2
2
0


2
4
4
4
6
4
1
2
2
1


2
4
4
4
6
4
1
2
3
2


2
4
4
4
6
4
1
3
4
2


2
4
4
4
6
4
2
4
4
2


2
4
4
4
6
4
2
4
4
2

Lemma 1: If G is Eulerian, then every node in G has even degree.

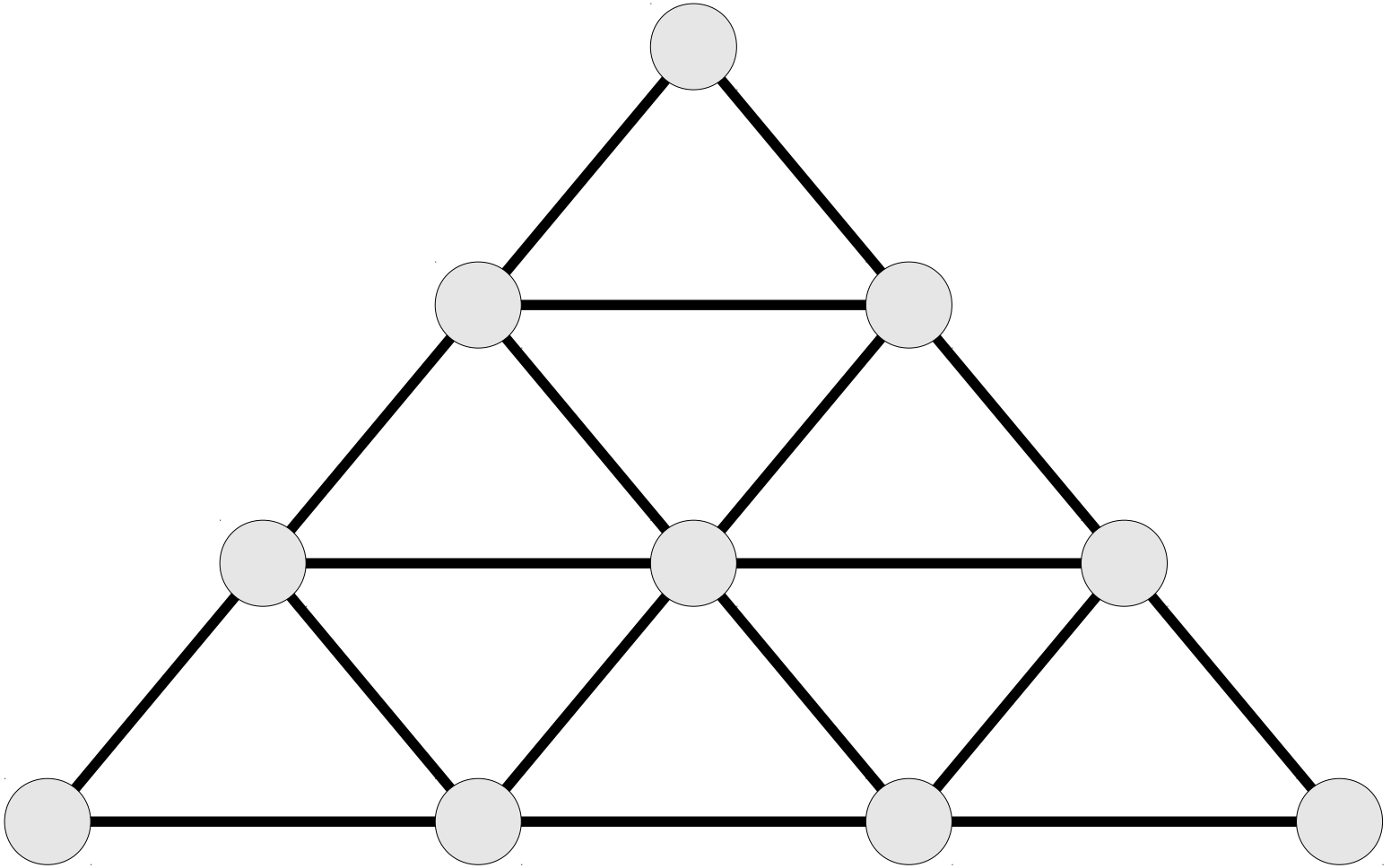
Proof: Let $G = (V, E)$ be an Eulerian graph and let C be an Eulerian circuit in G. Fix any node v. If we trace through circuit C, we will enter v the same number of times that we leave it. This means that the number of edges incident to v that are a part of C is even. Since C contains every edge in the graph exactly once, this means that v is incident to an even number of edges. Therefore, v has even degree, as required. ■

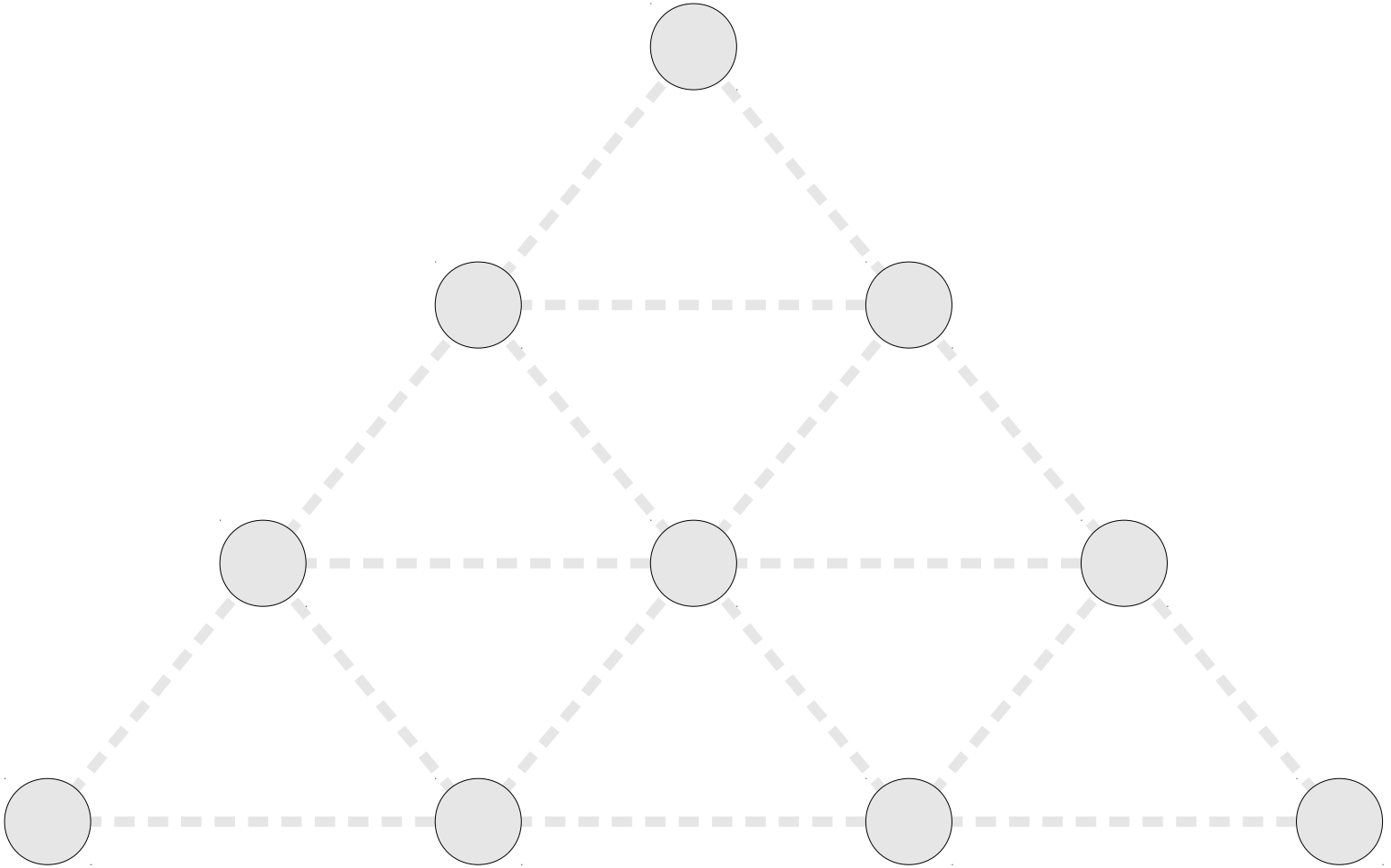
The Other Direction

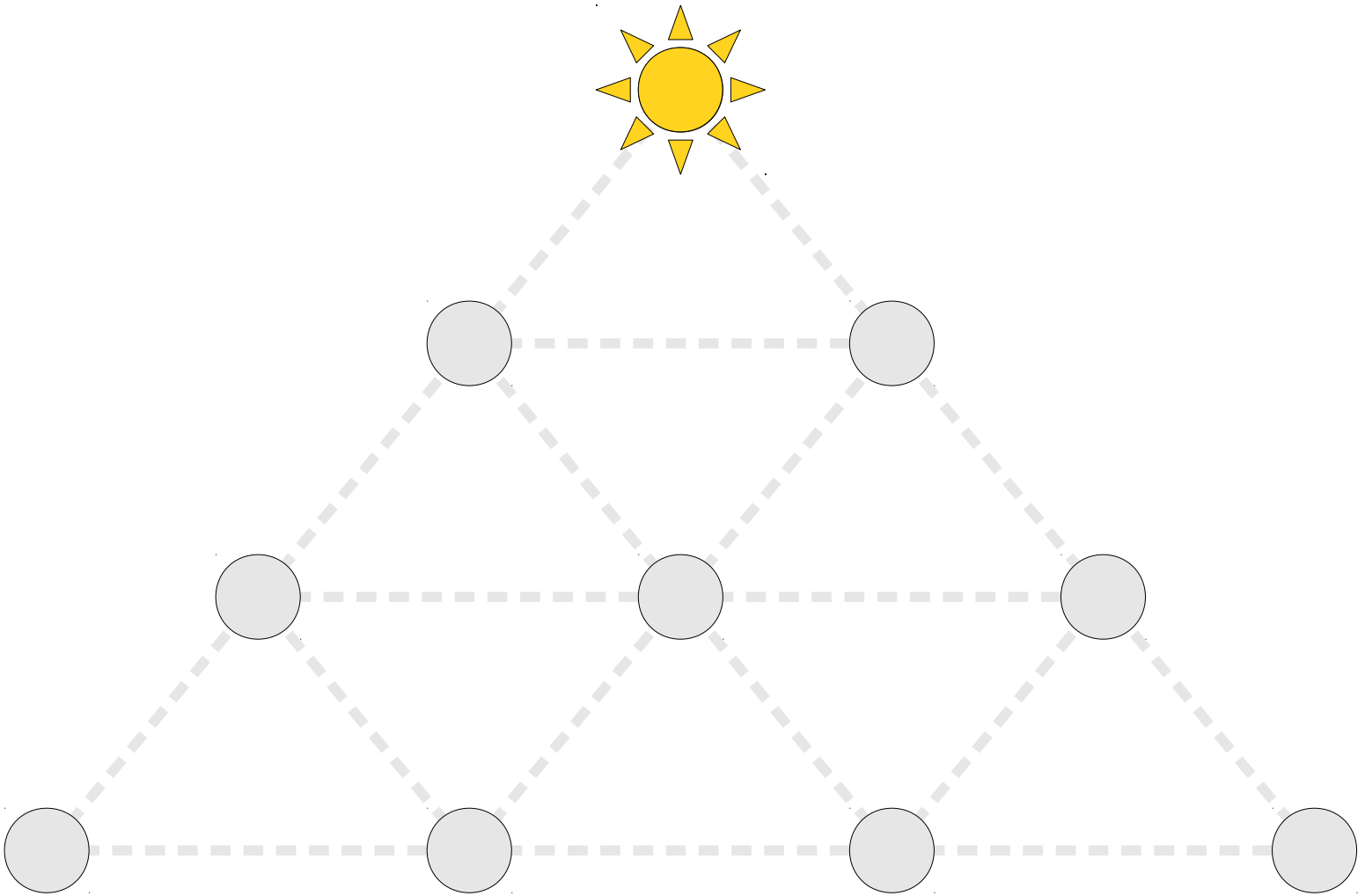
- For the other direction of the proof, we need to show the following:

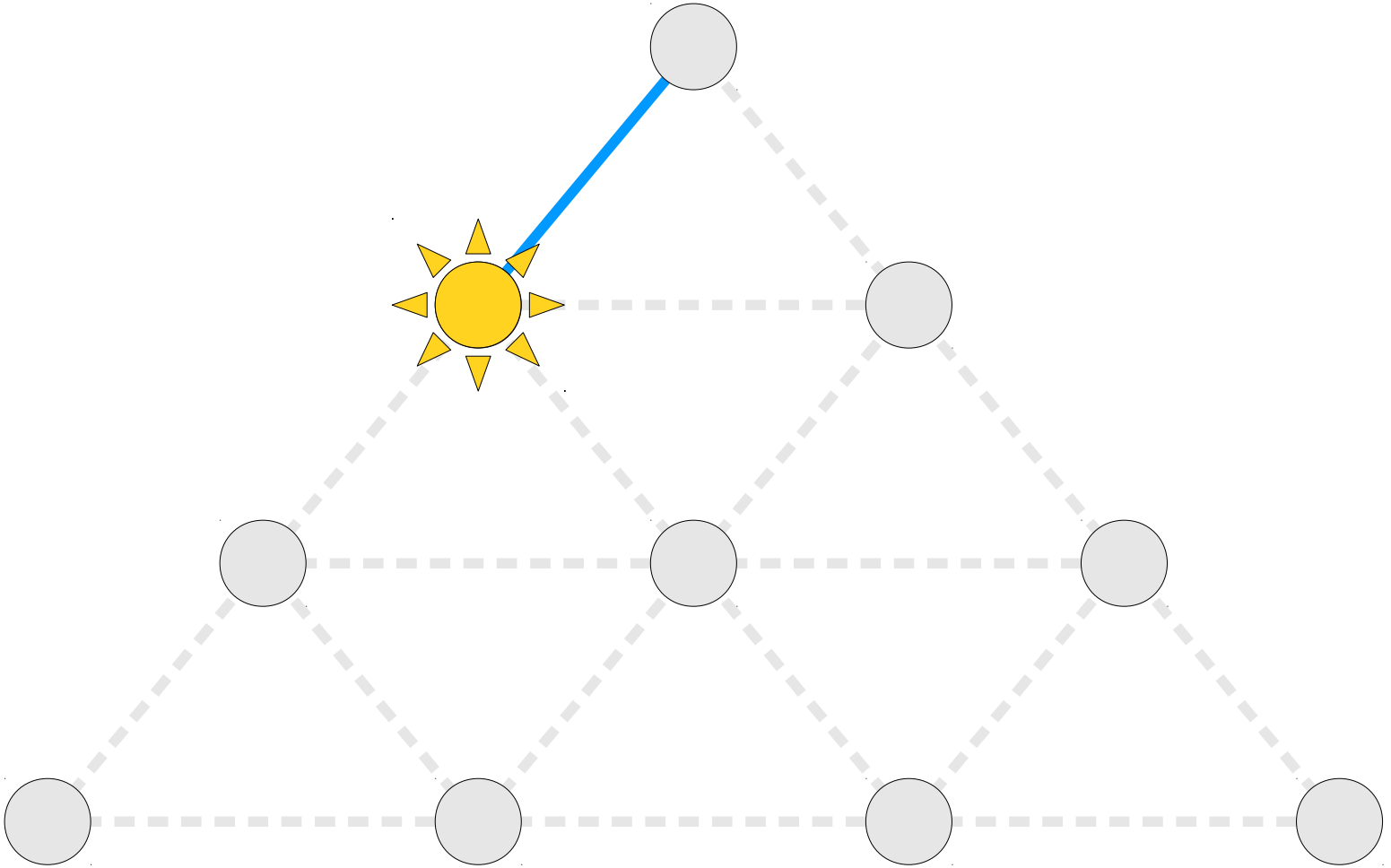
  If *G* is connected and every node has even degree, then *G* has an Eulerian circuit.

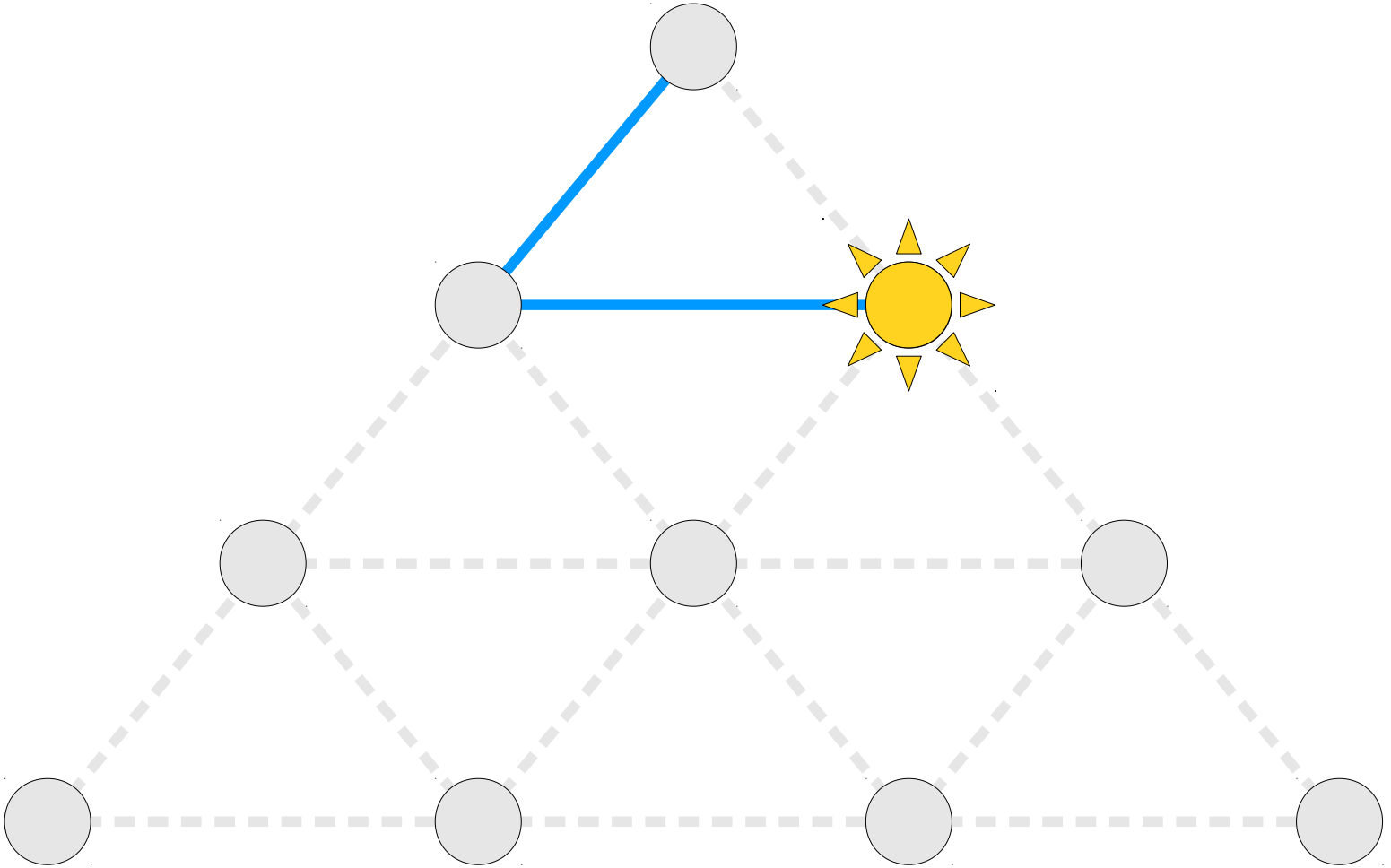
- To do so, we're going to see a technique that lets us start with an empty path and continuously increase its size until it becomes an Eulerian circuit.

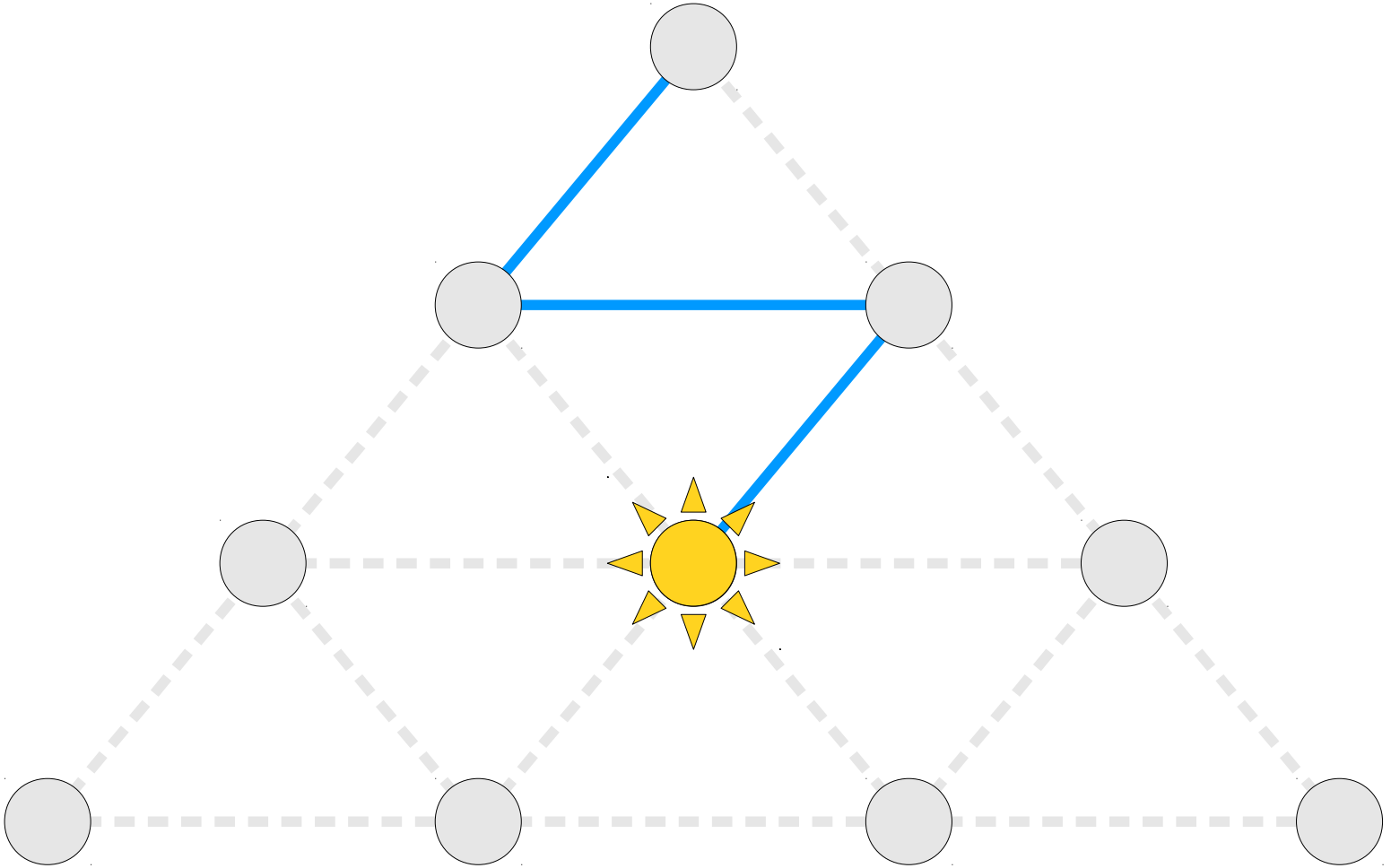


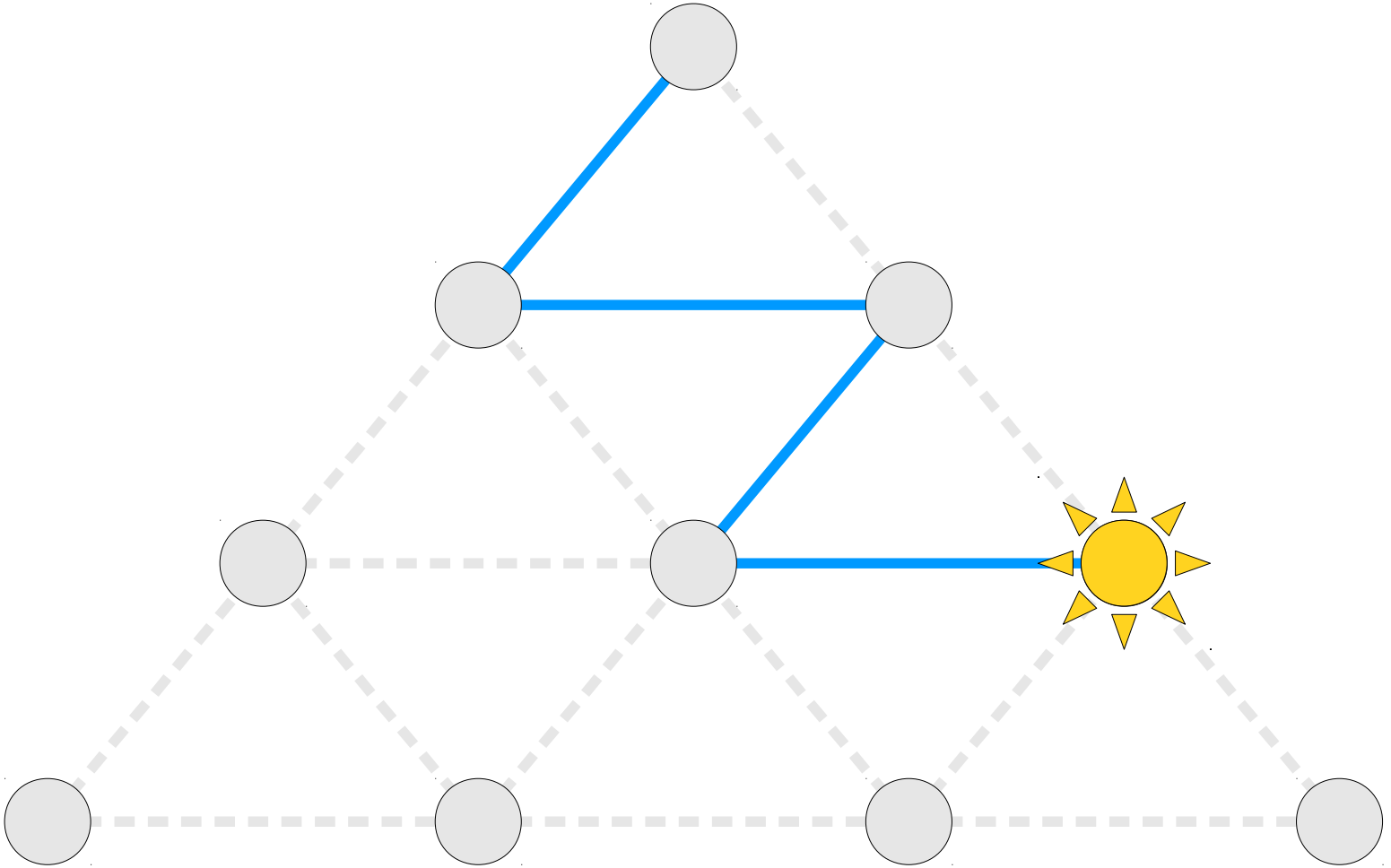


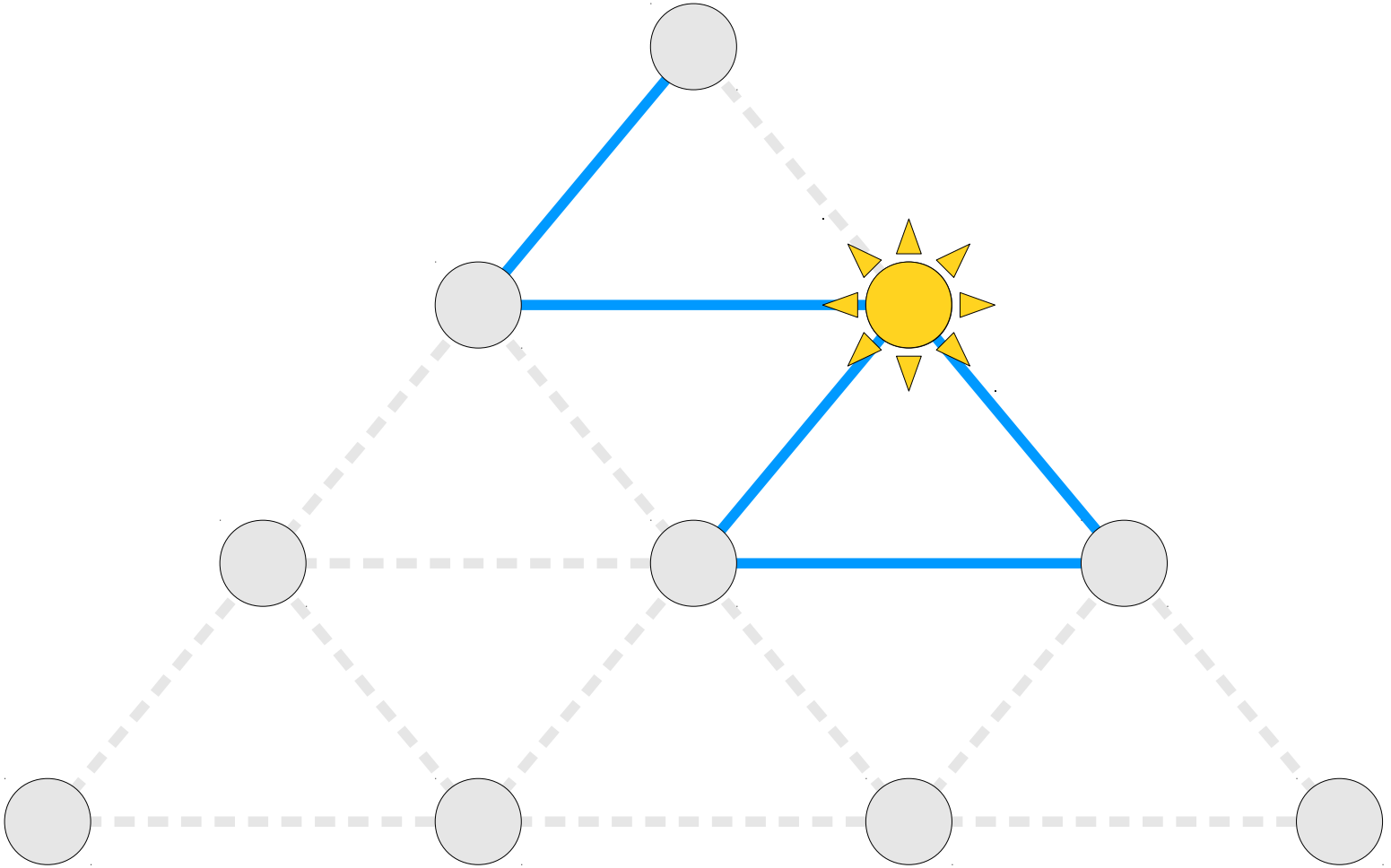


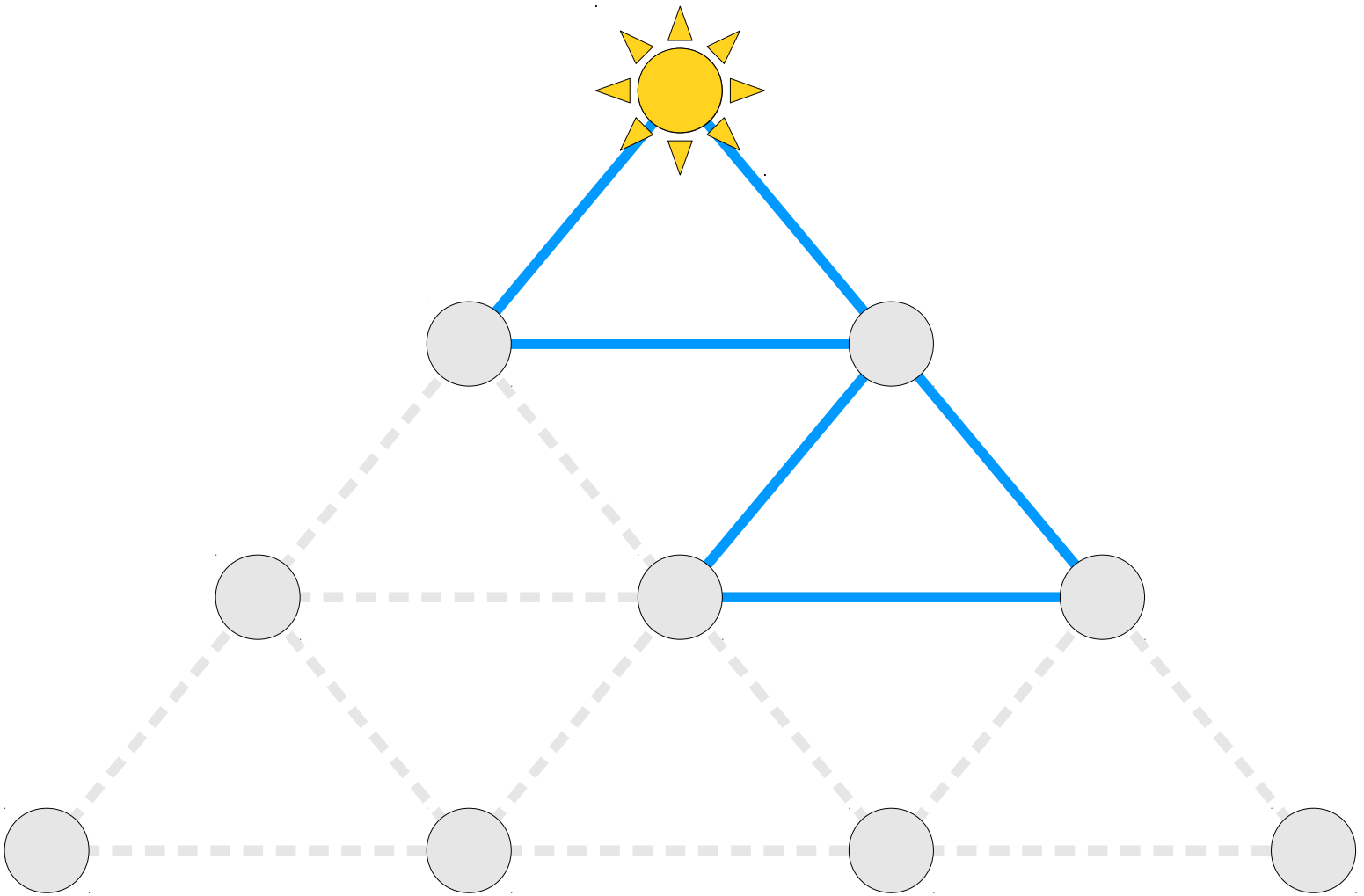


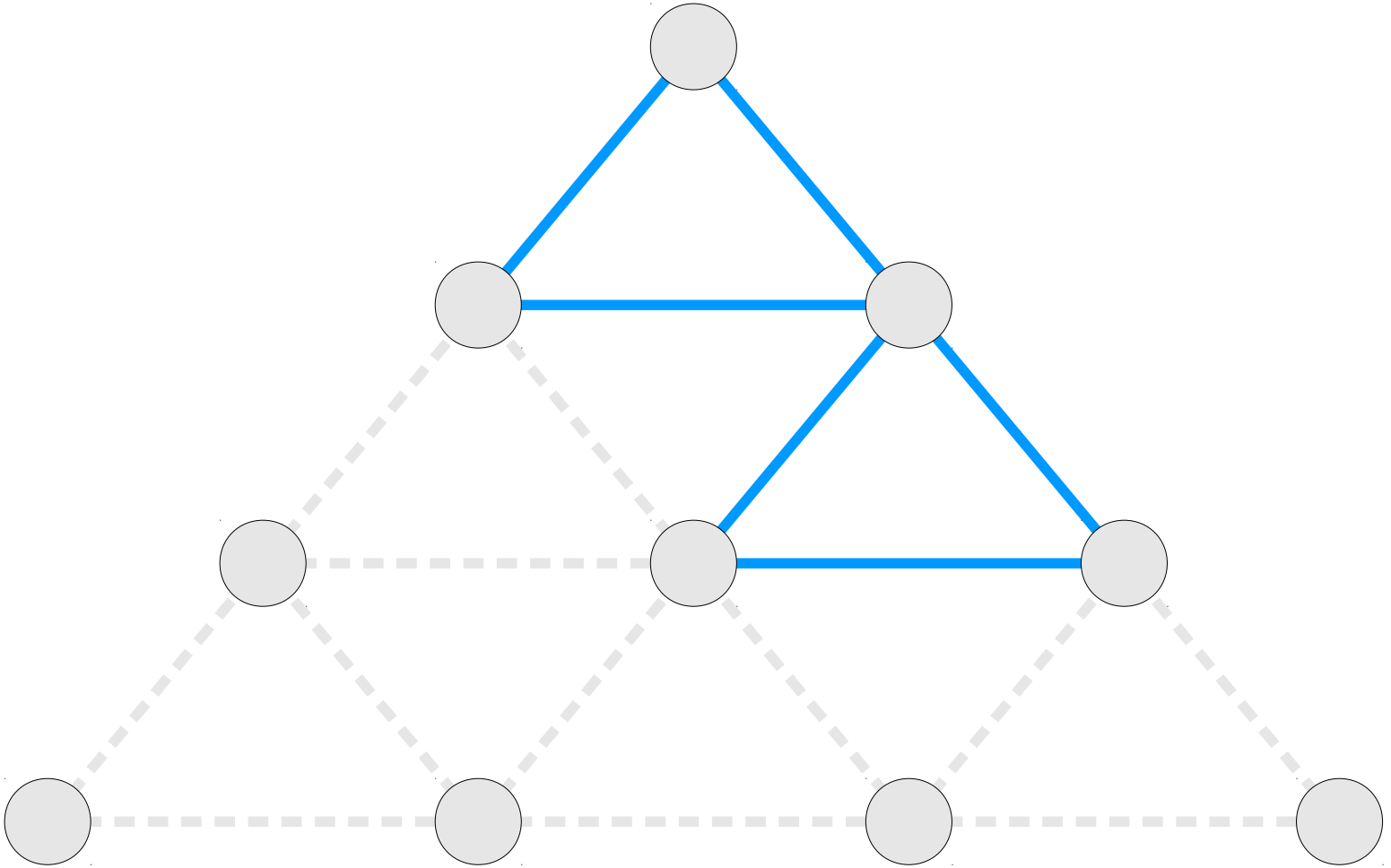


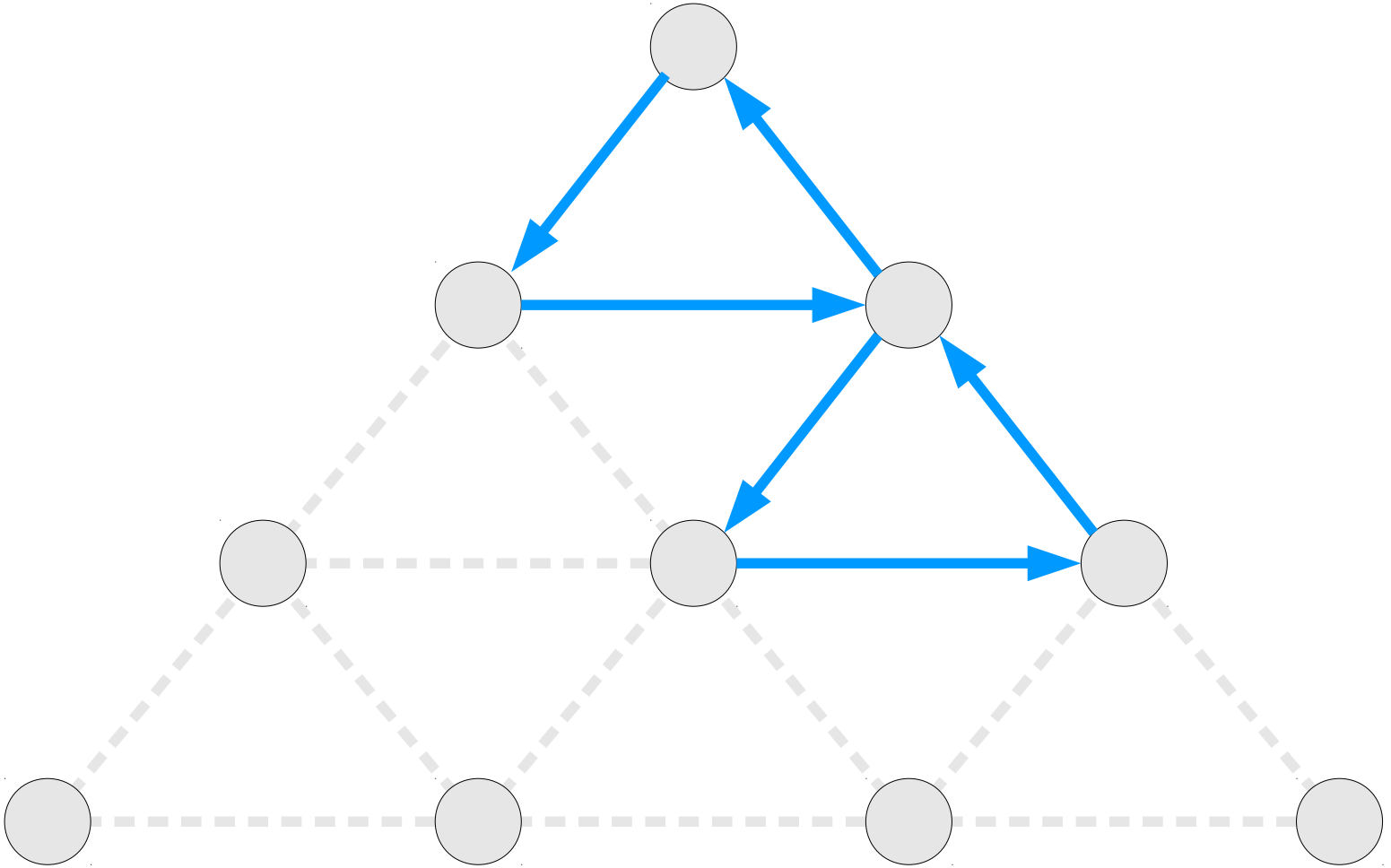


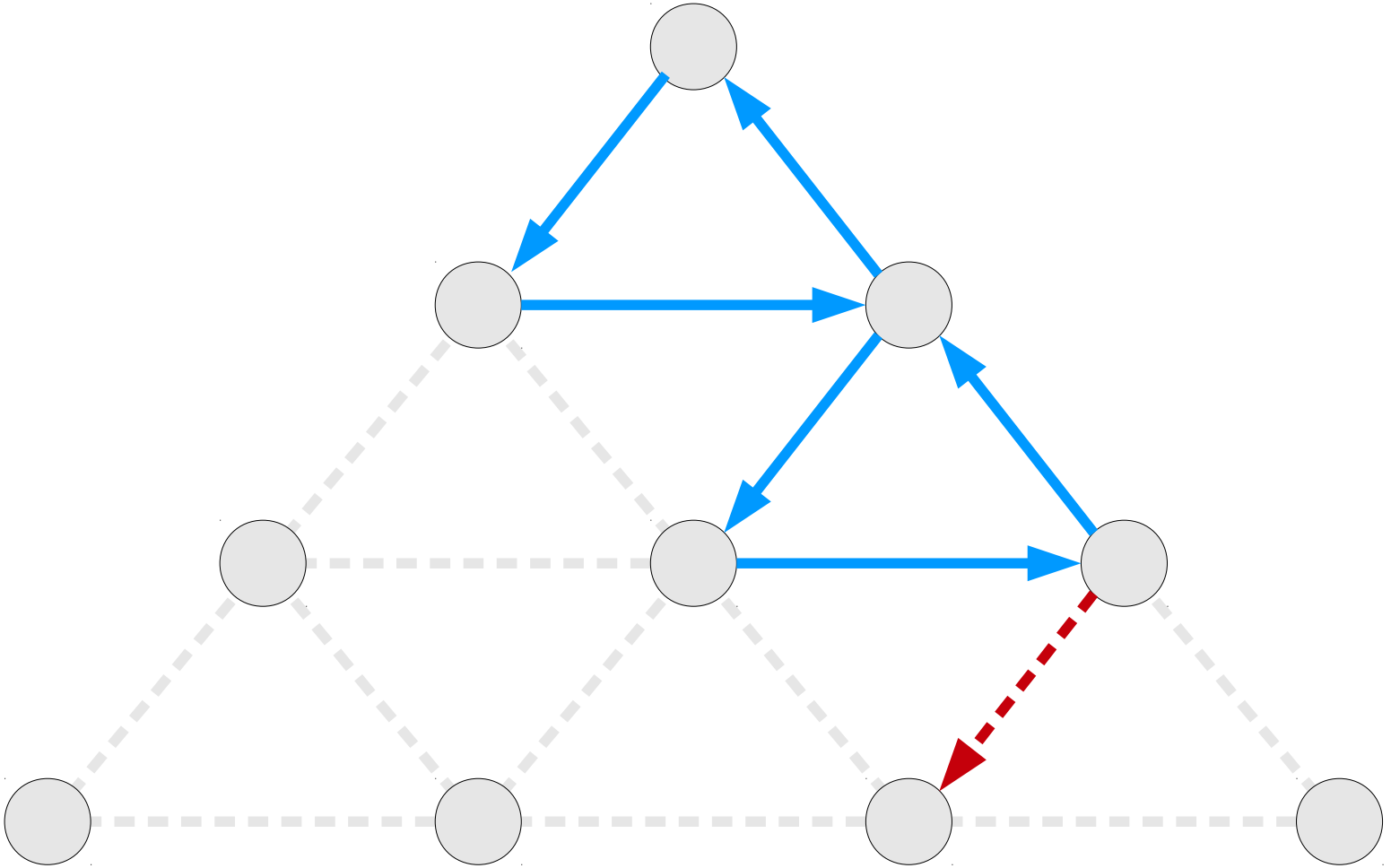


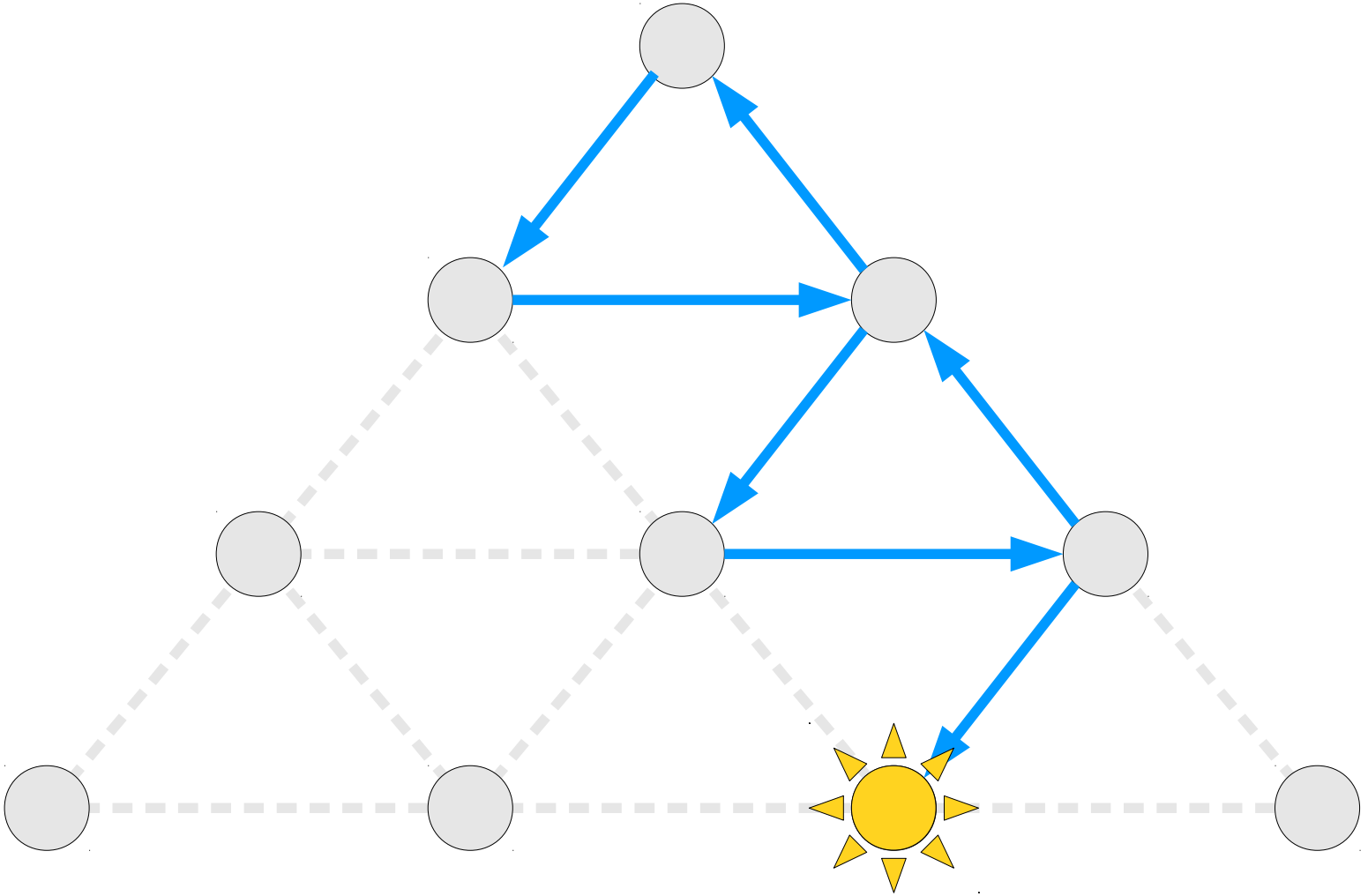


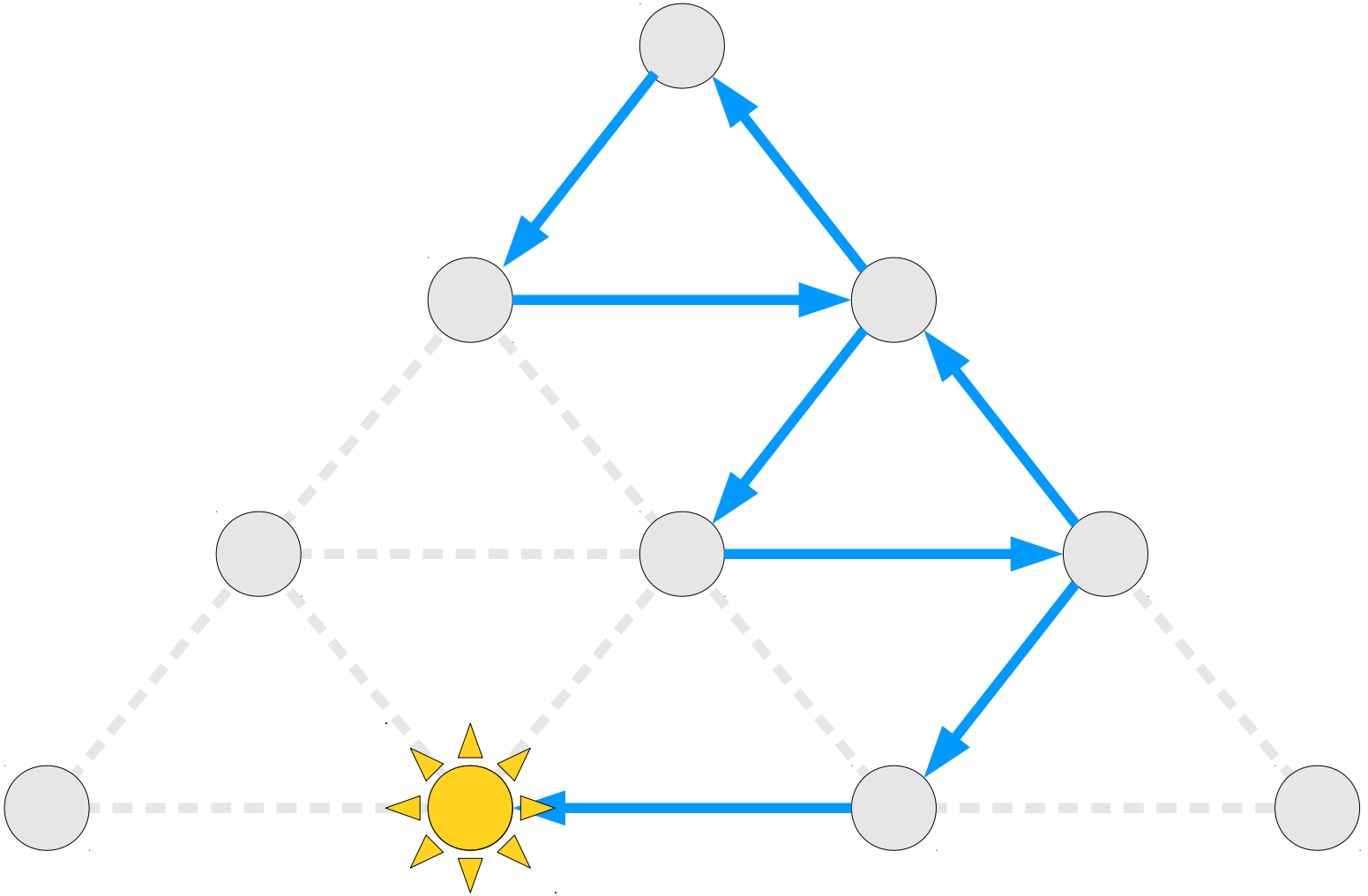


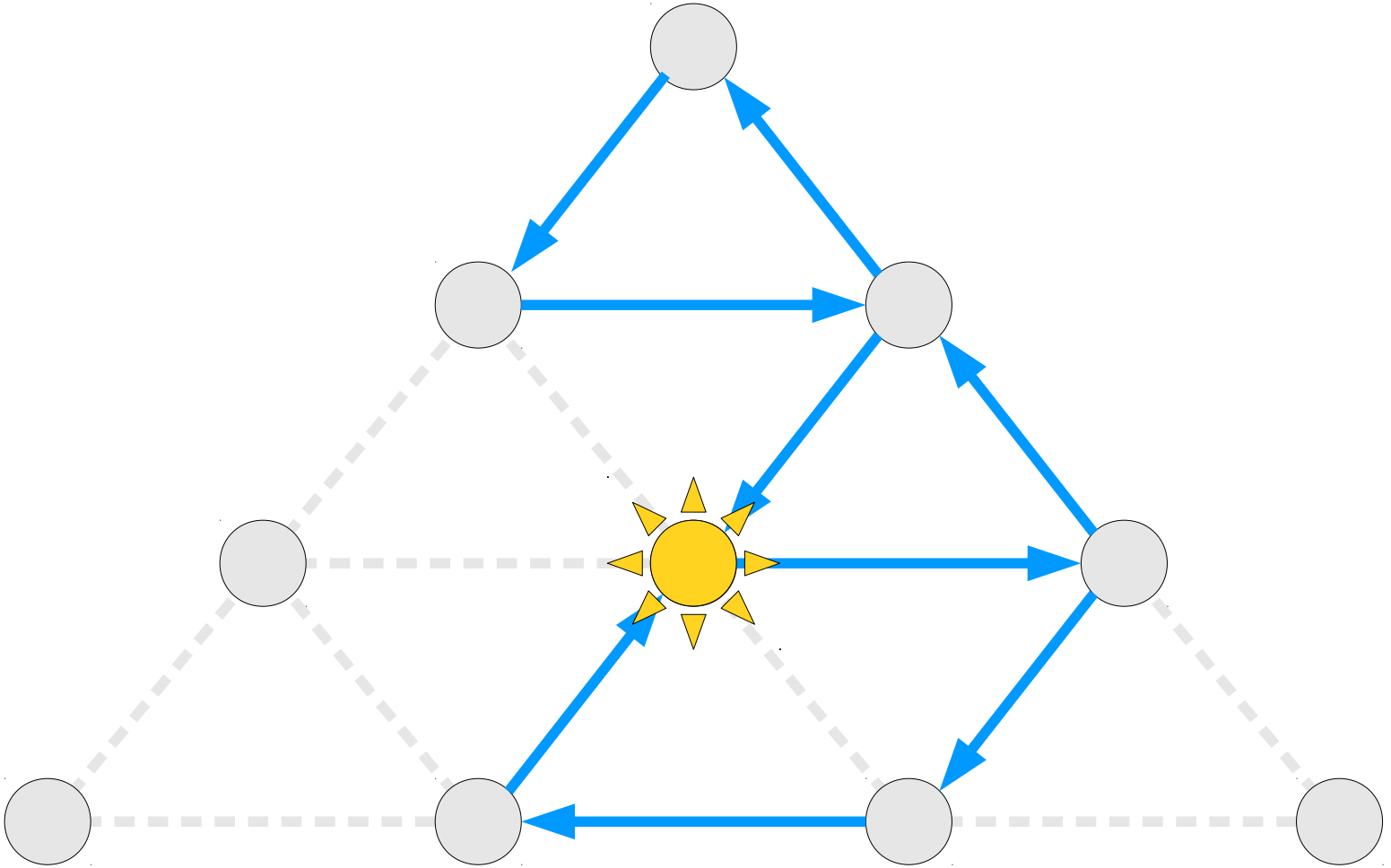


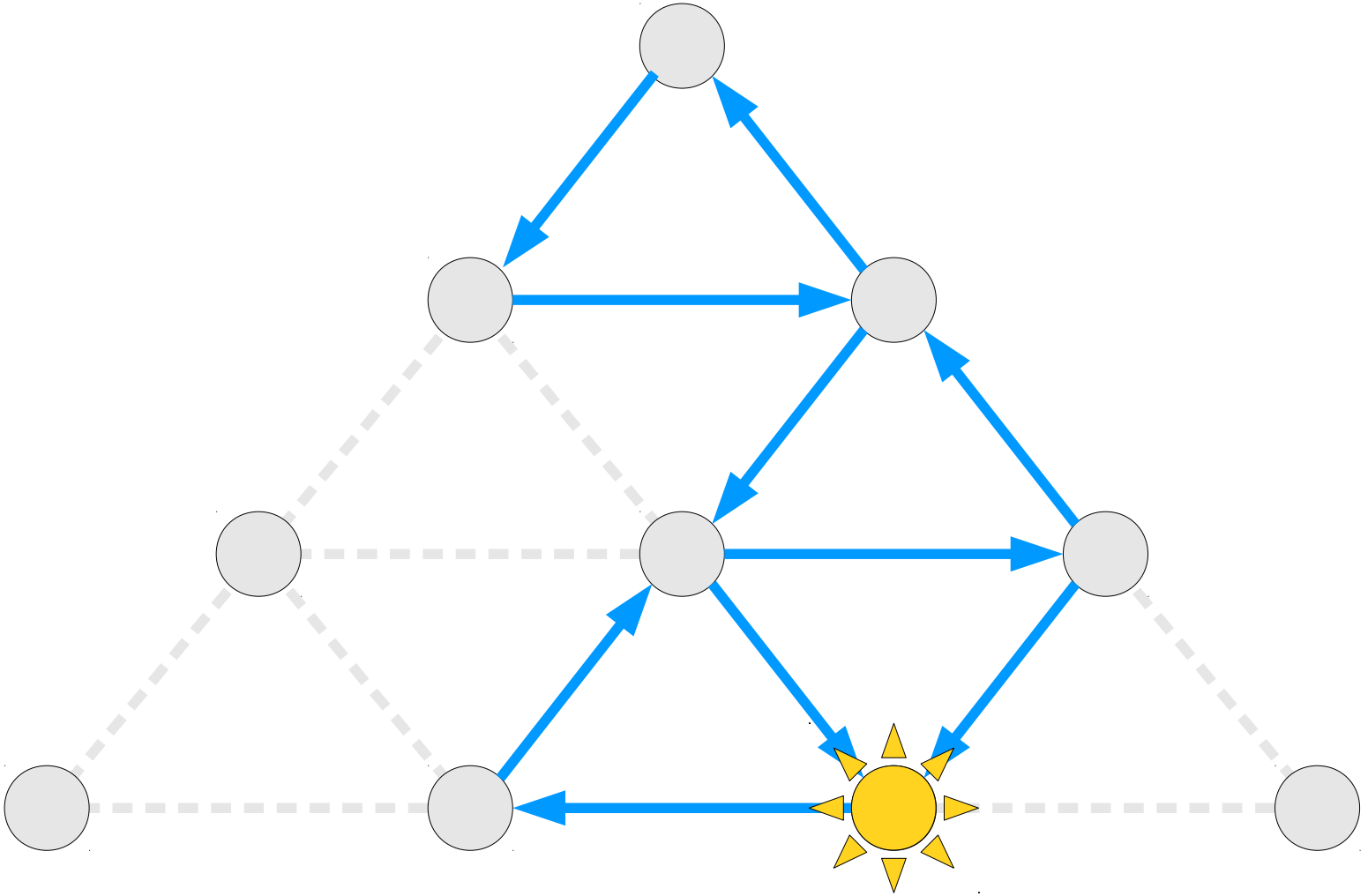


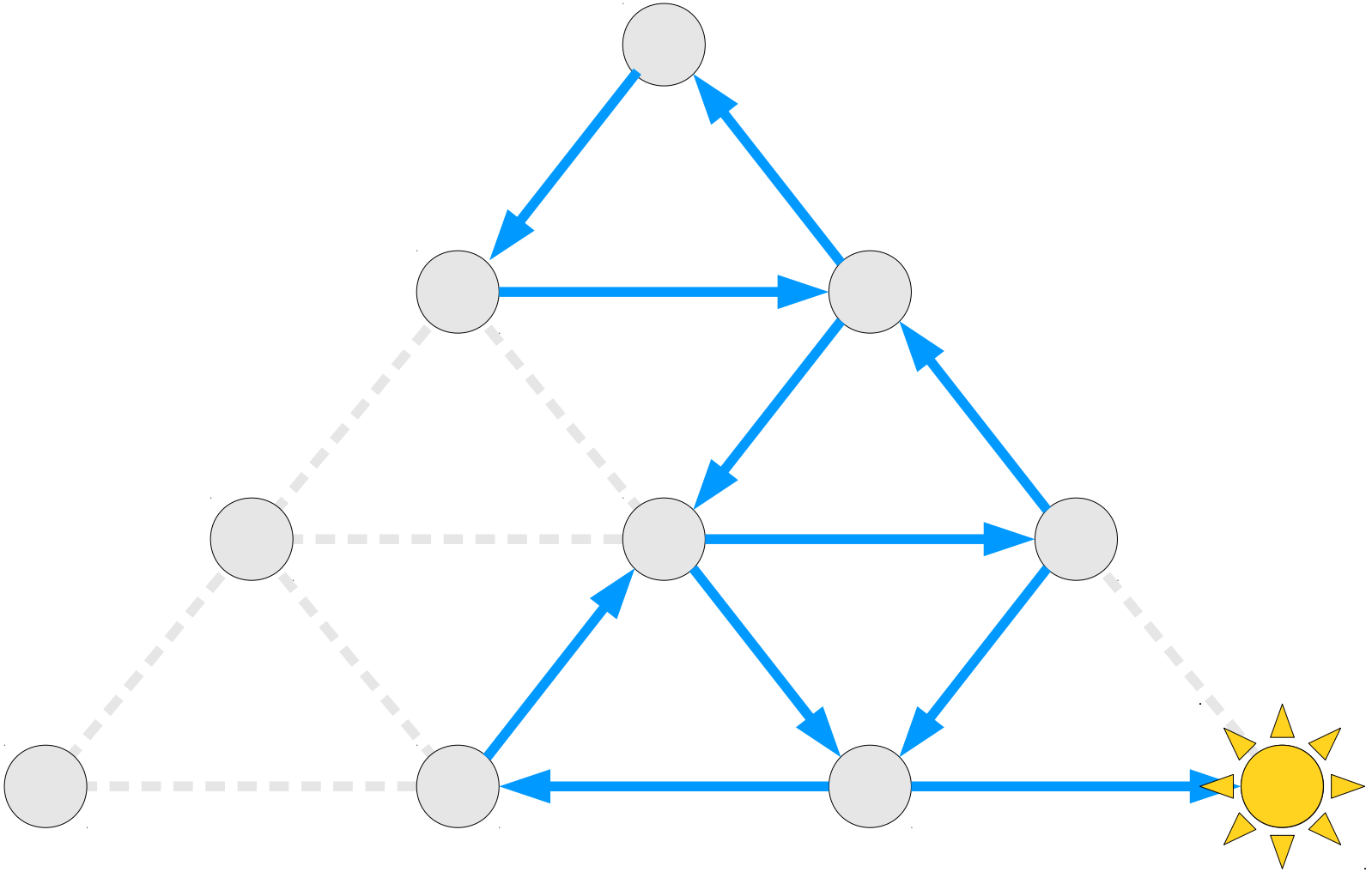


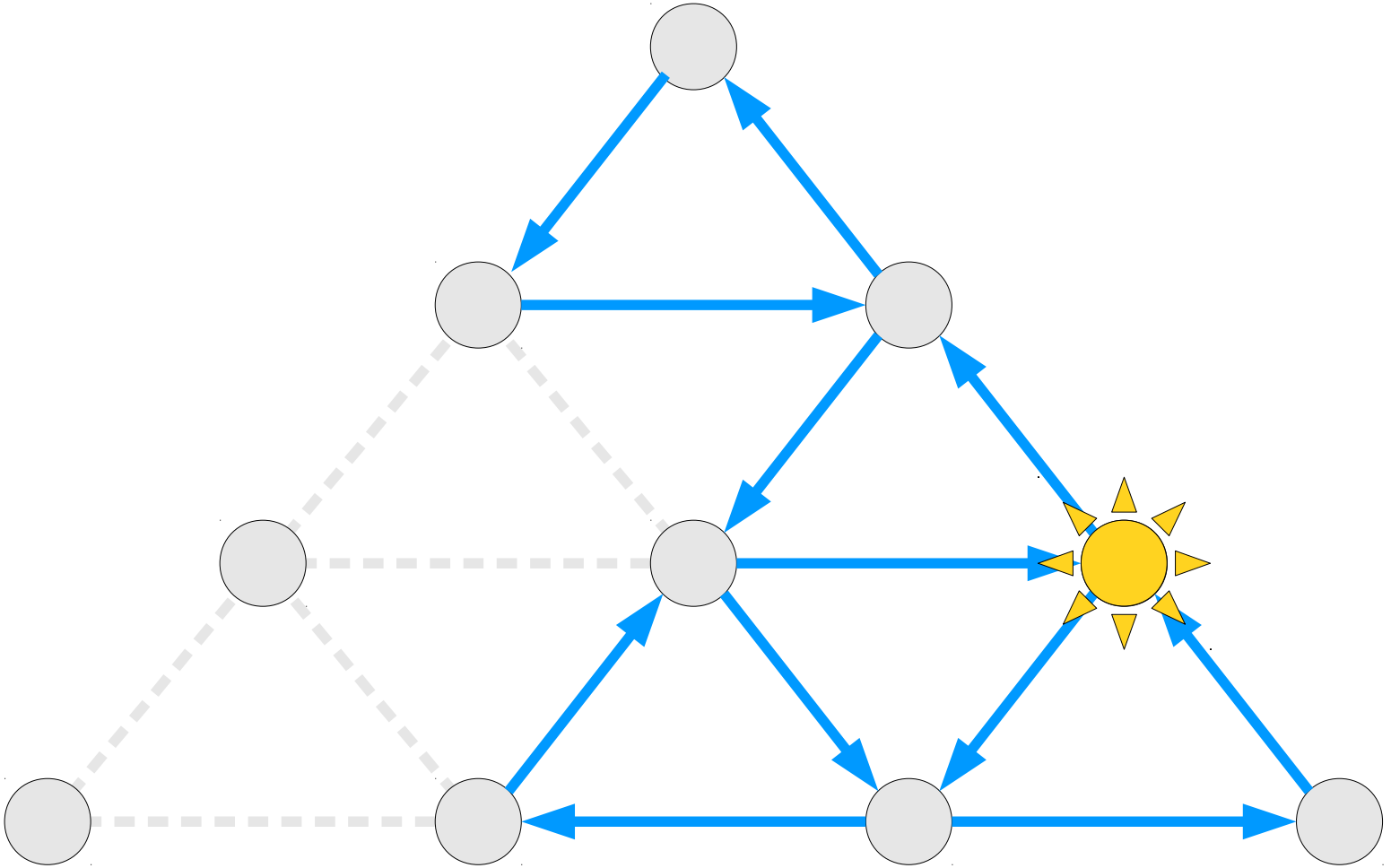


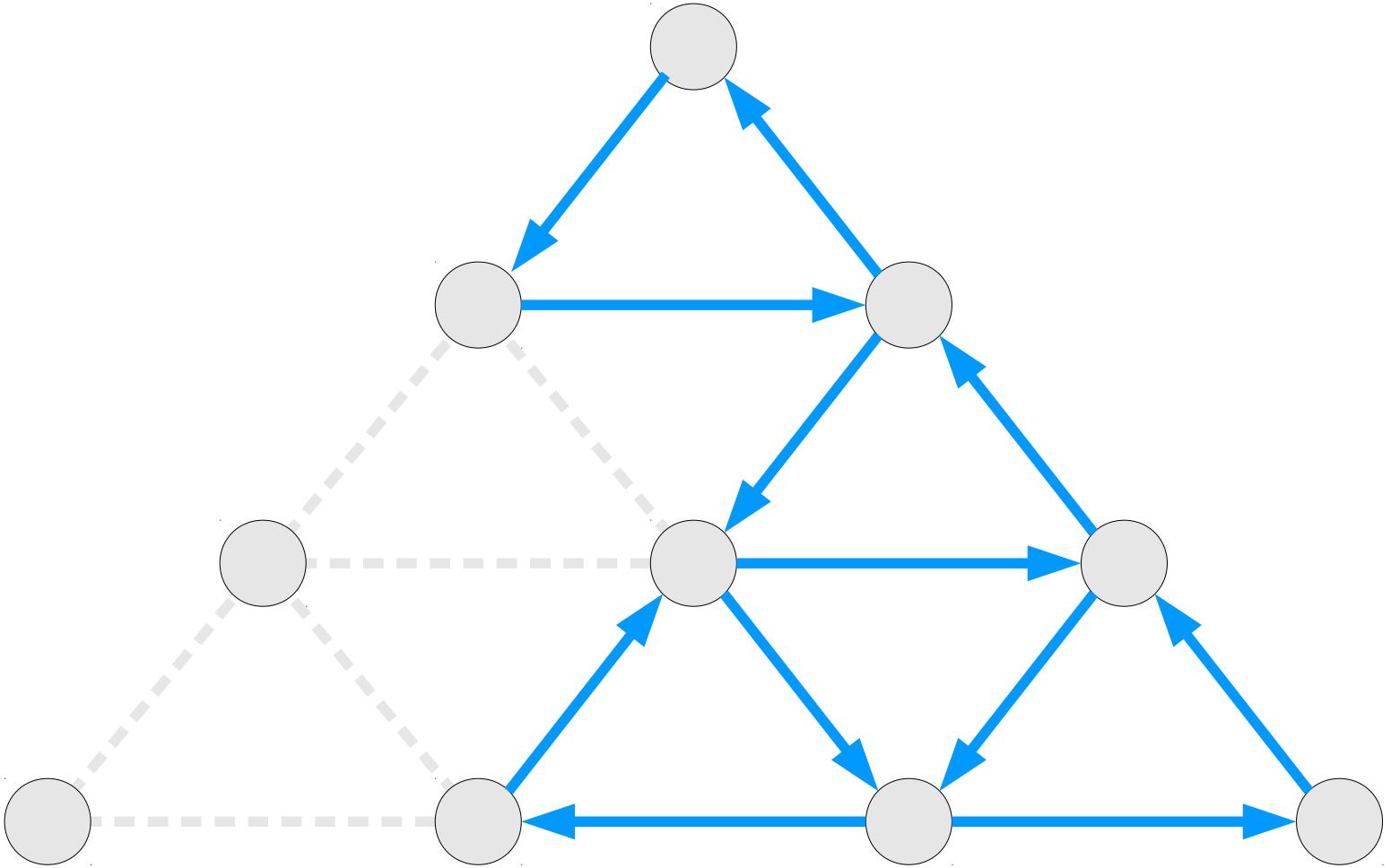


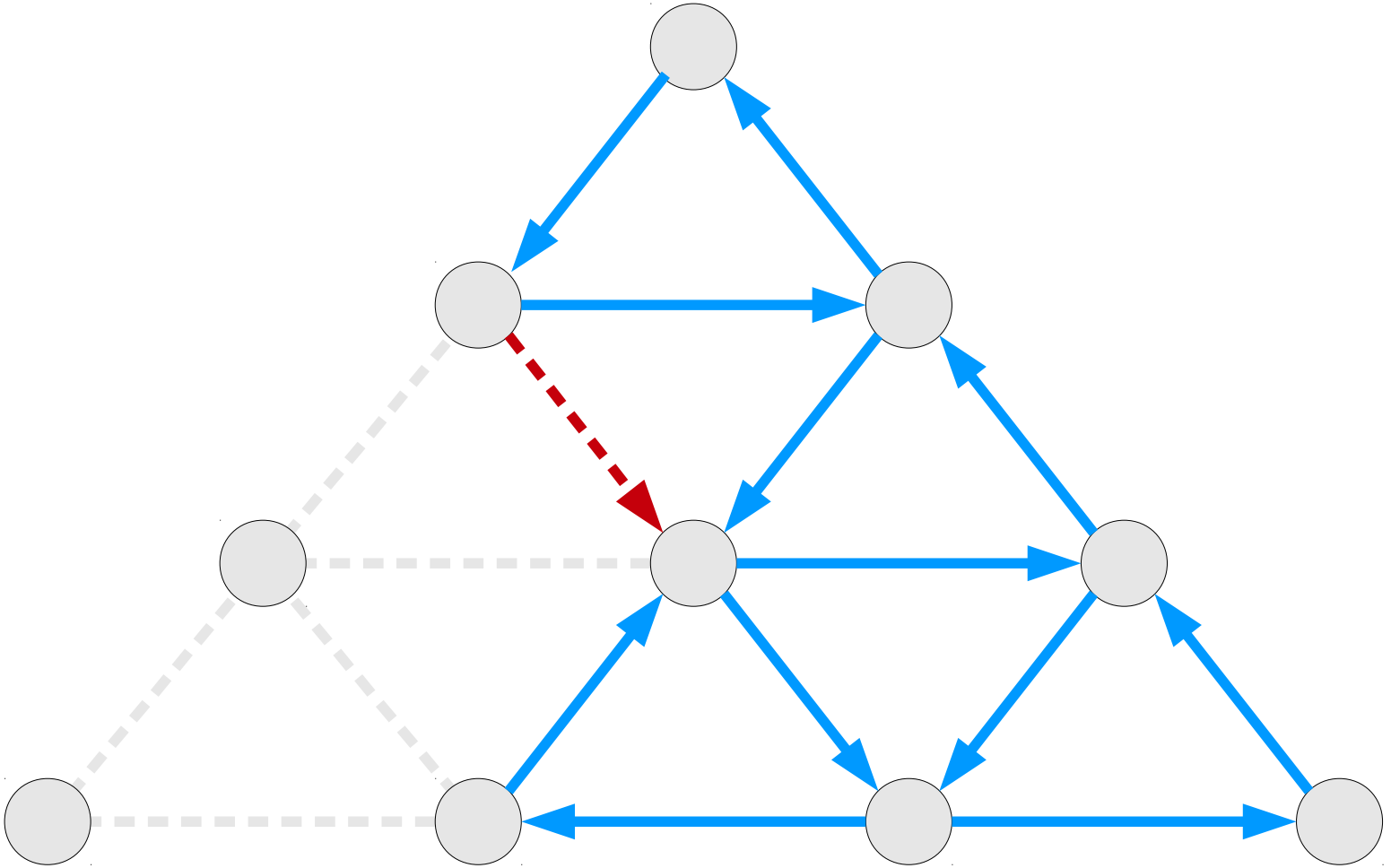


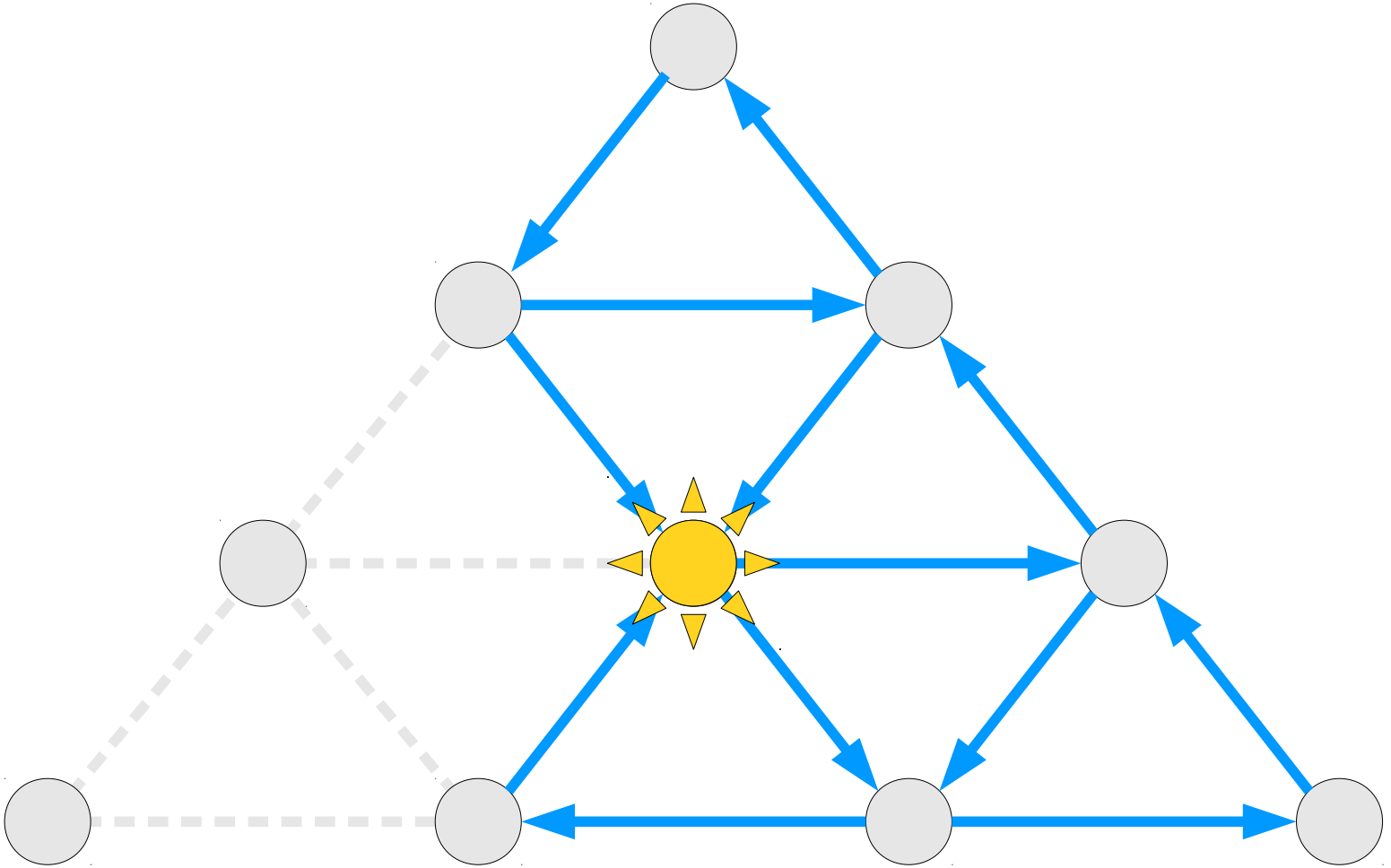


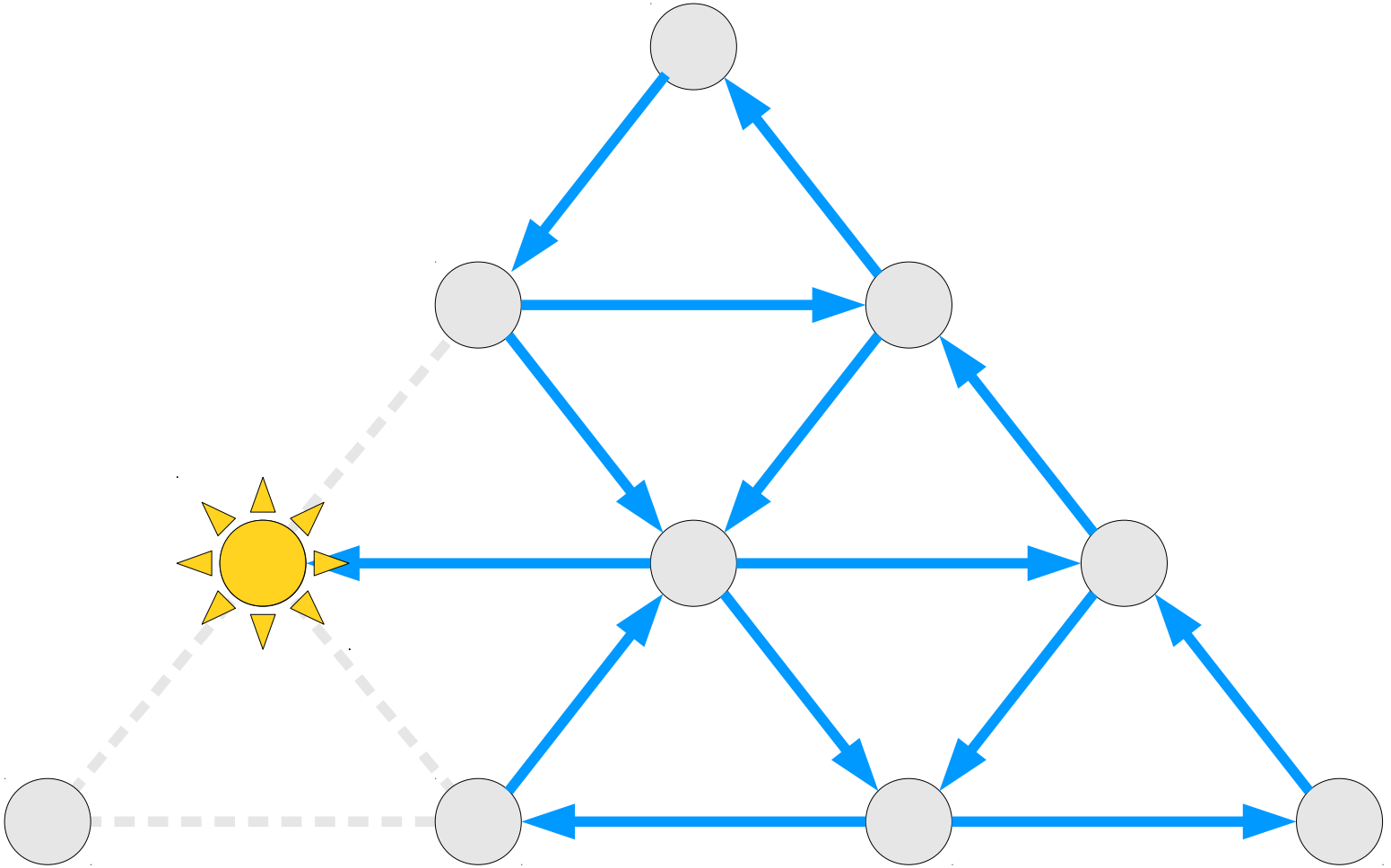


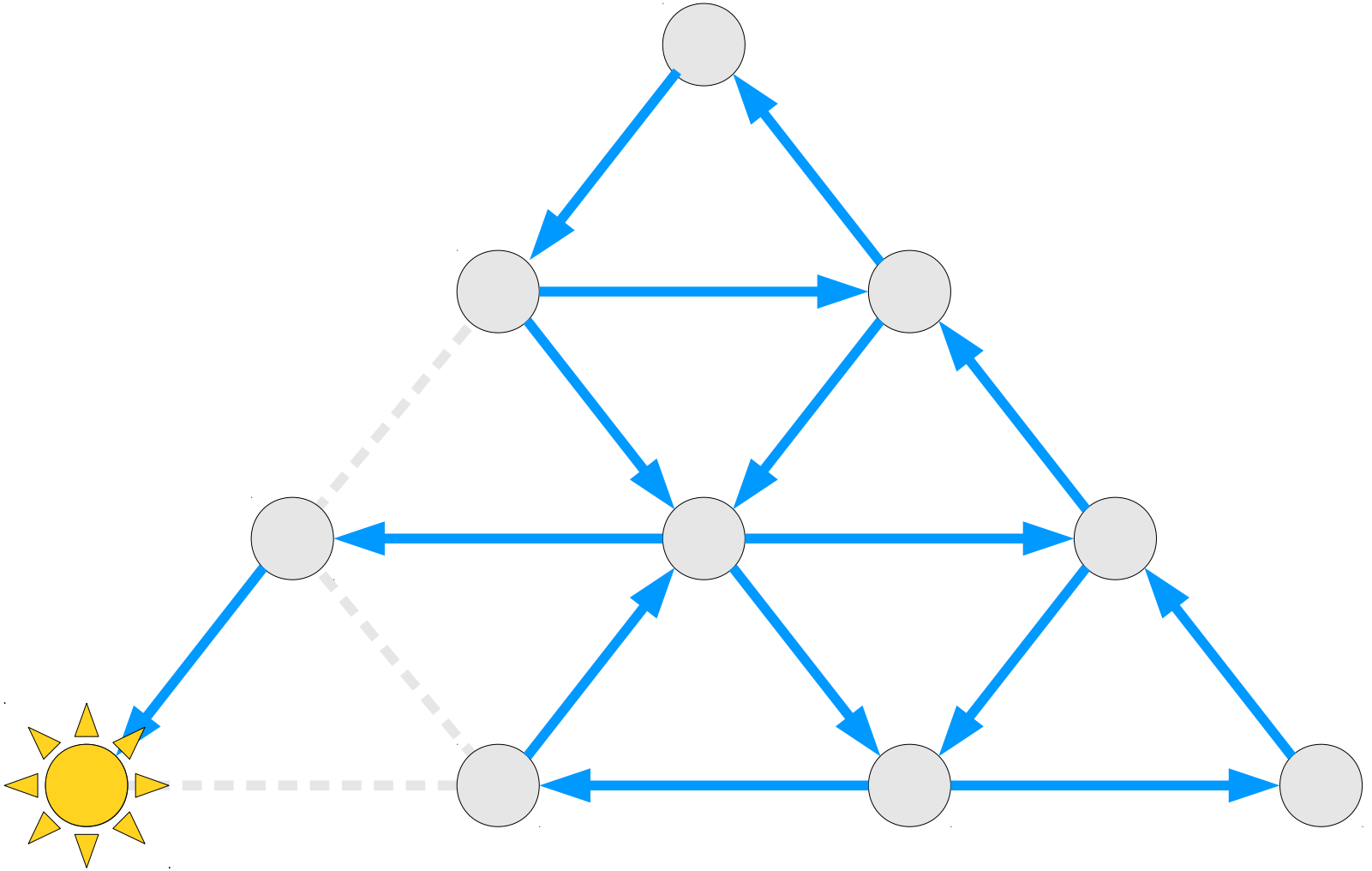


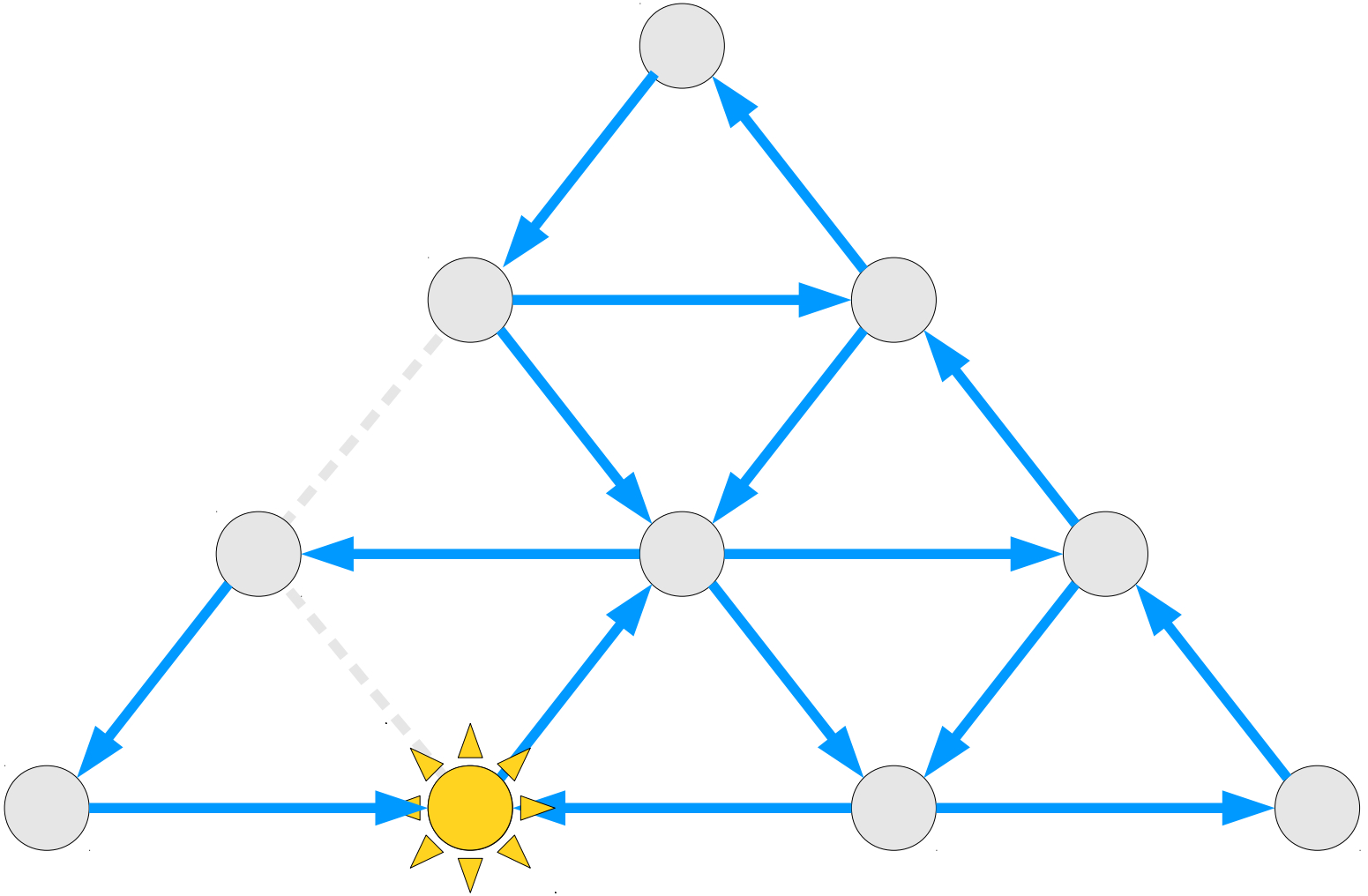


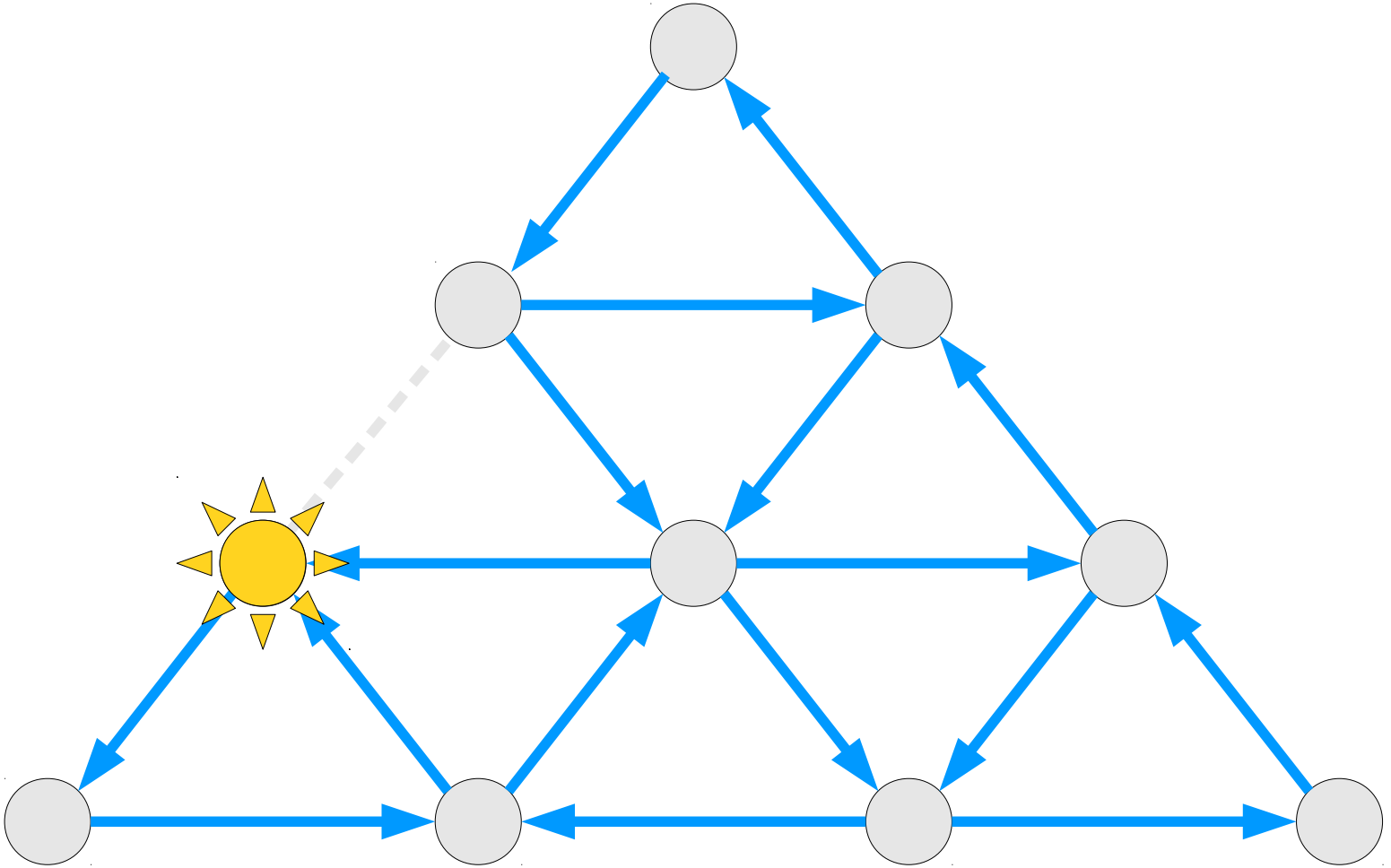


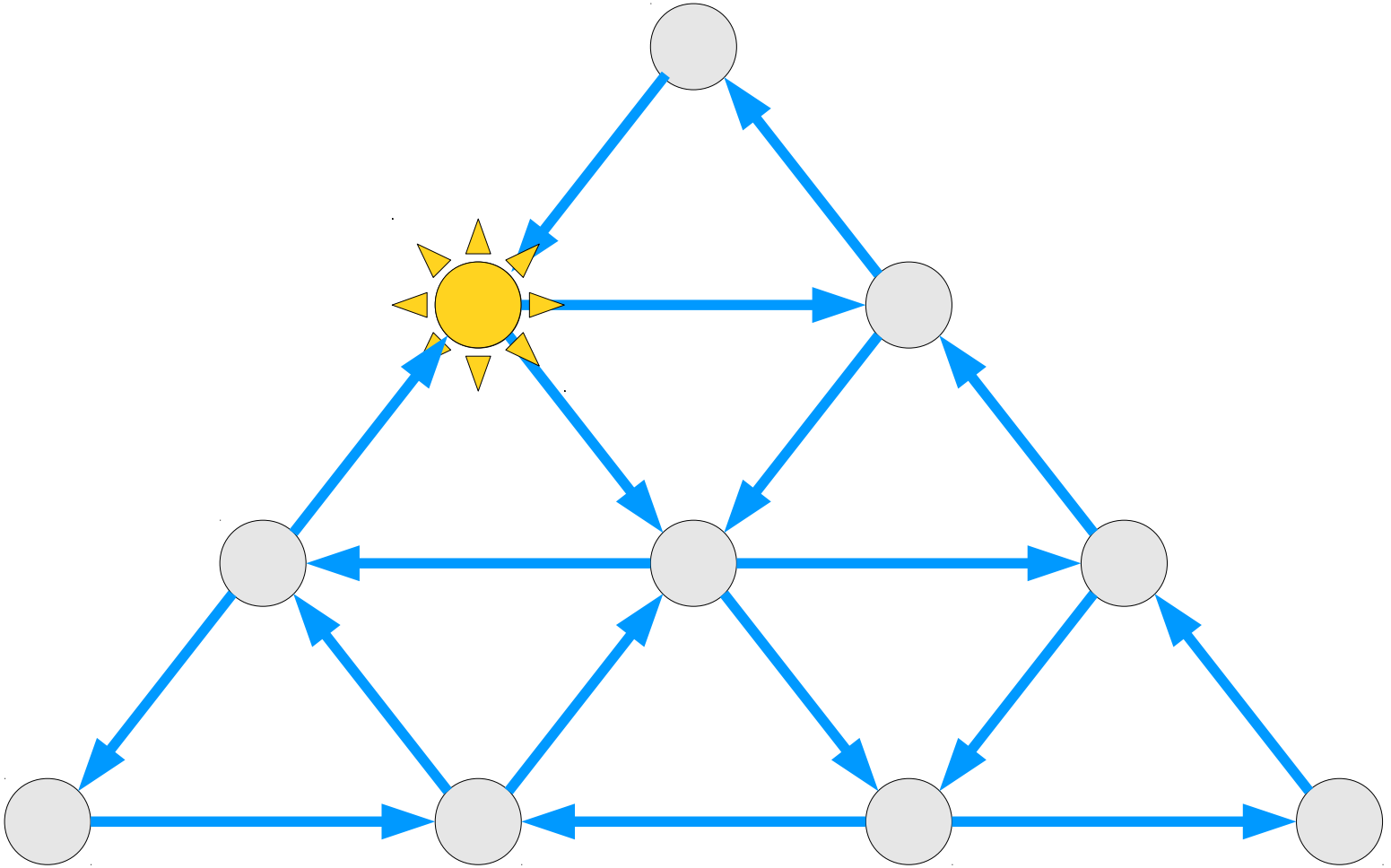


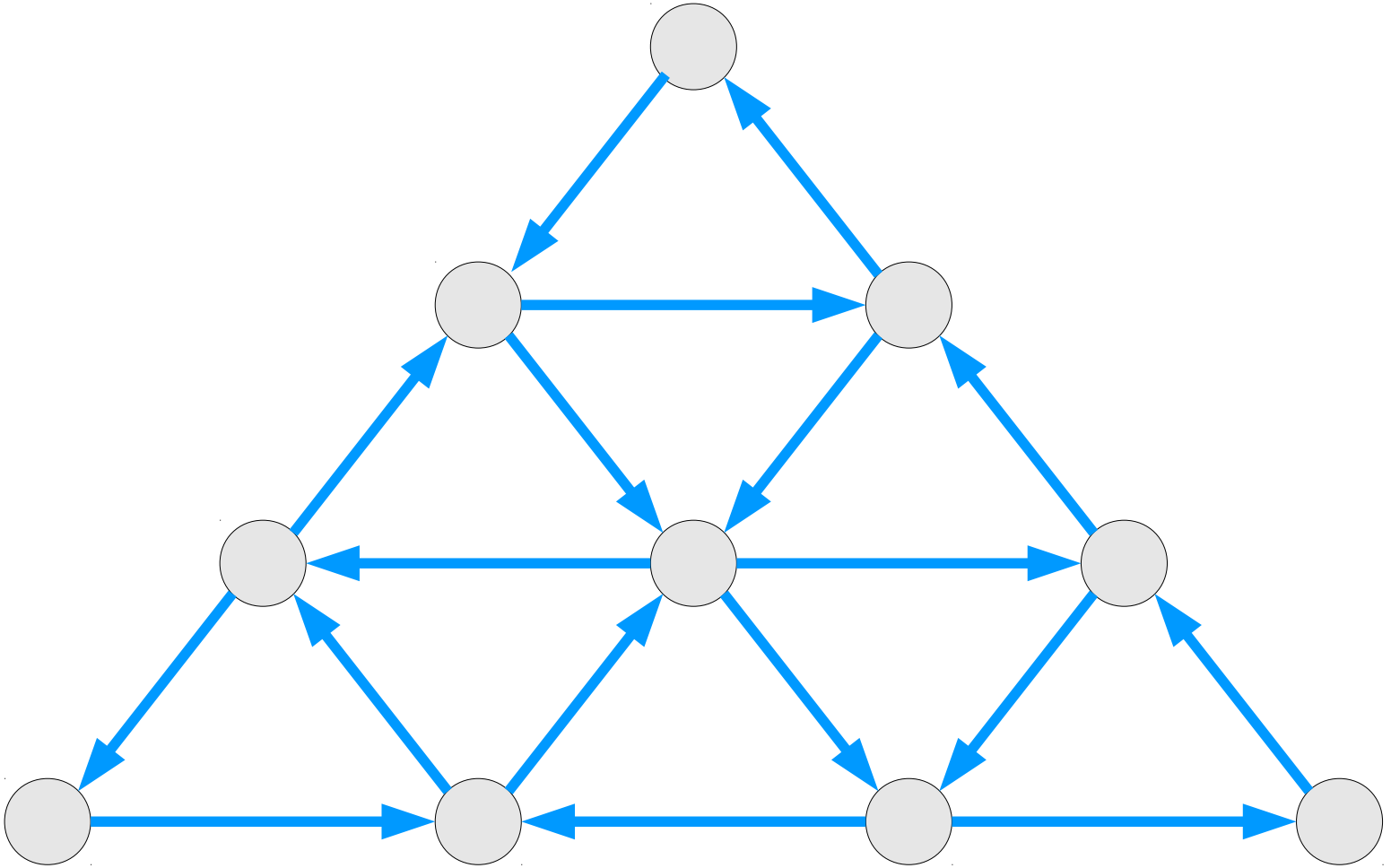












The Basic Idea

- Start with the empty path.
- If the path you have so far isn't a cycle, extend it by adding in an edge.
  - (It's always possible to do this; we'll see why in a second.)
- If the path is a cycle and you aren't done yet, extend the cycle by following an edge hanging off it.
  - (It's always possible to do this; we'll see why in a second).

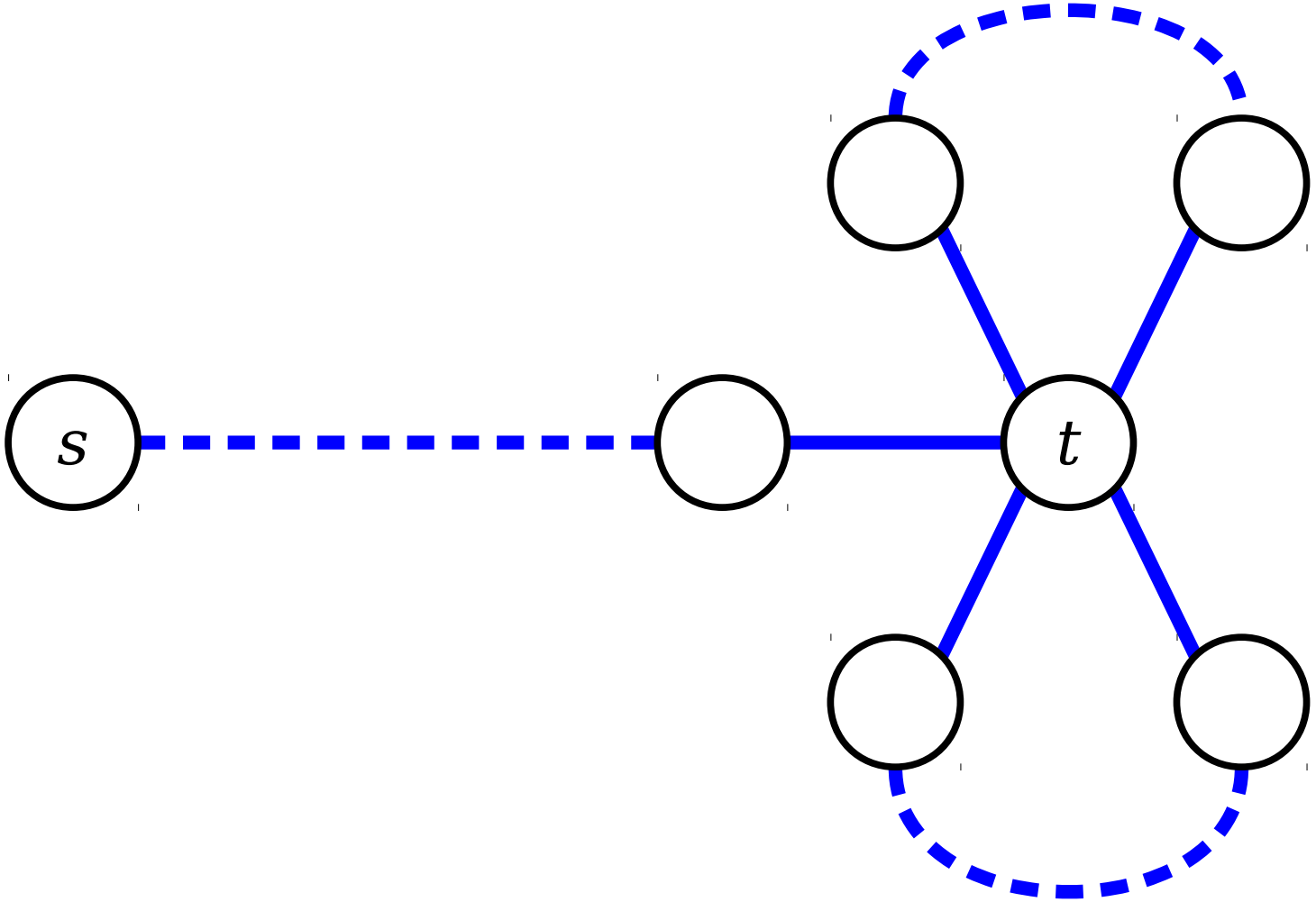
The Basic Idea

Start with the empty path.

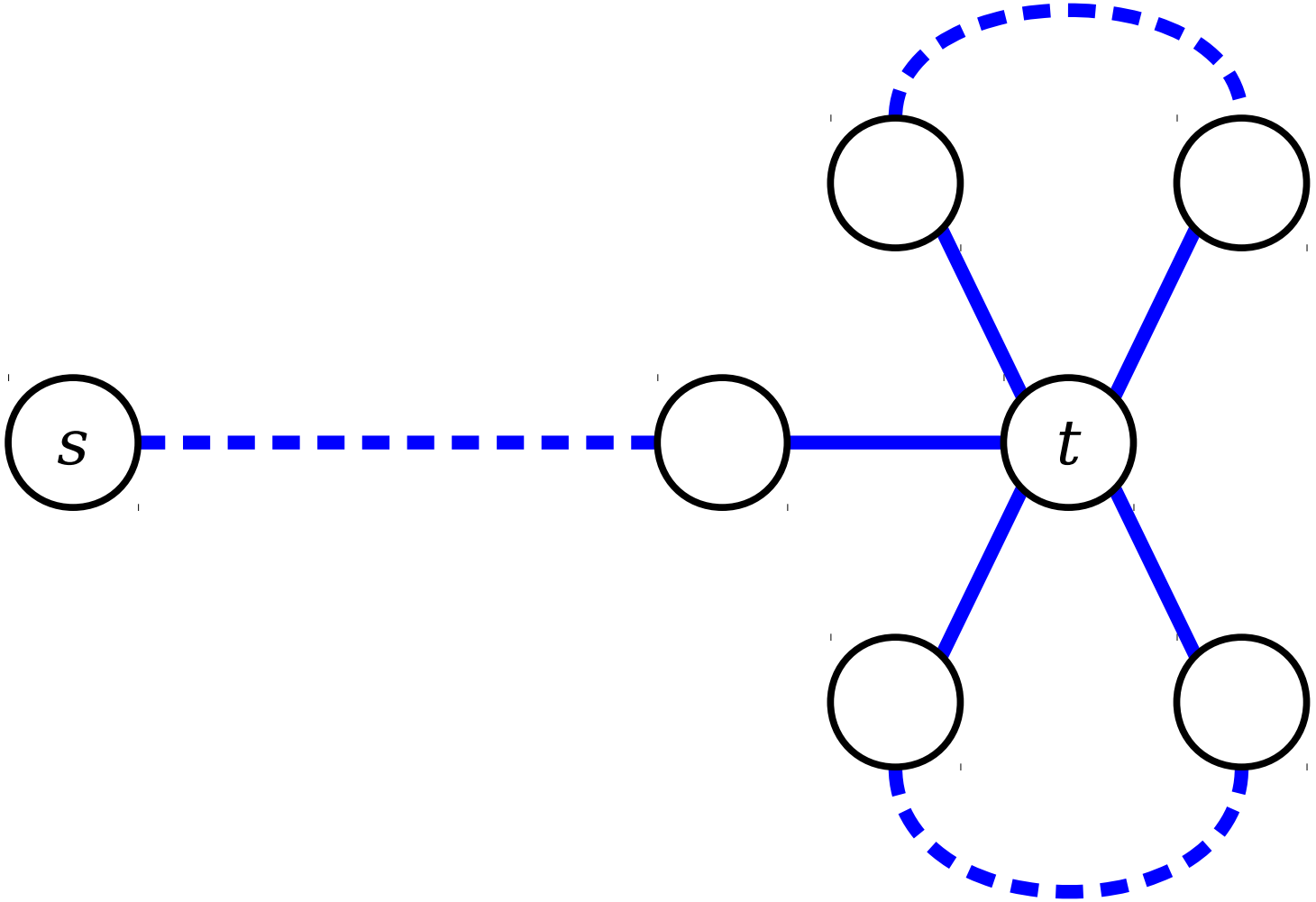
- If the path you have so far isn't a cycle, extend it by adding in an edge.
  - (It's always possible to do this; we'll see why in a second.)

If the path is a cycle and you aren't done yet, extend the cycle by following an edge hanging off it.

(It's always possible to do this; we'll see why in a second).

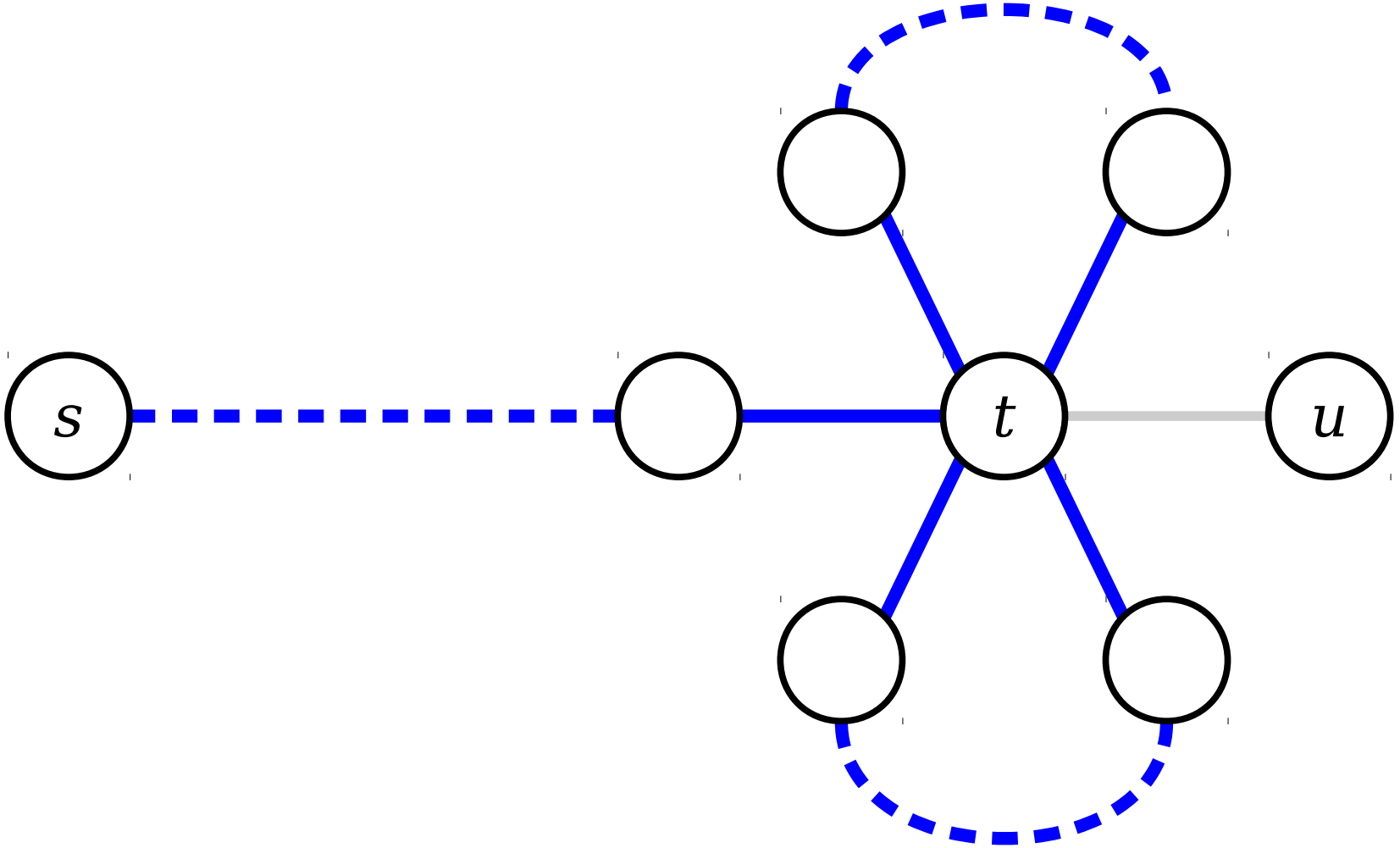


Suppose our path starts at s, ends at t, and that $s \neq t$.



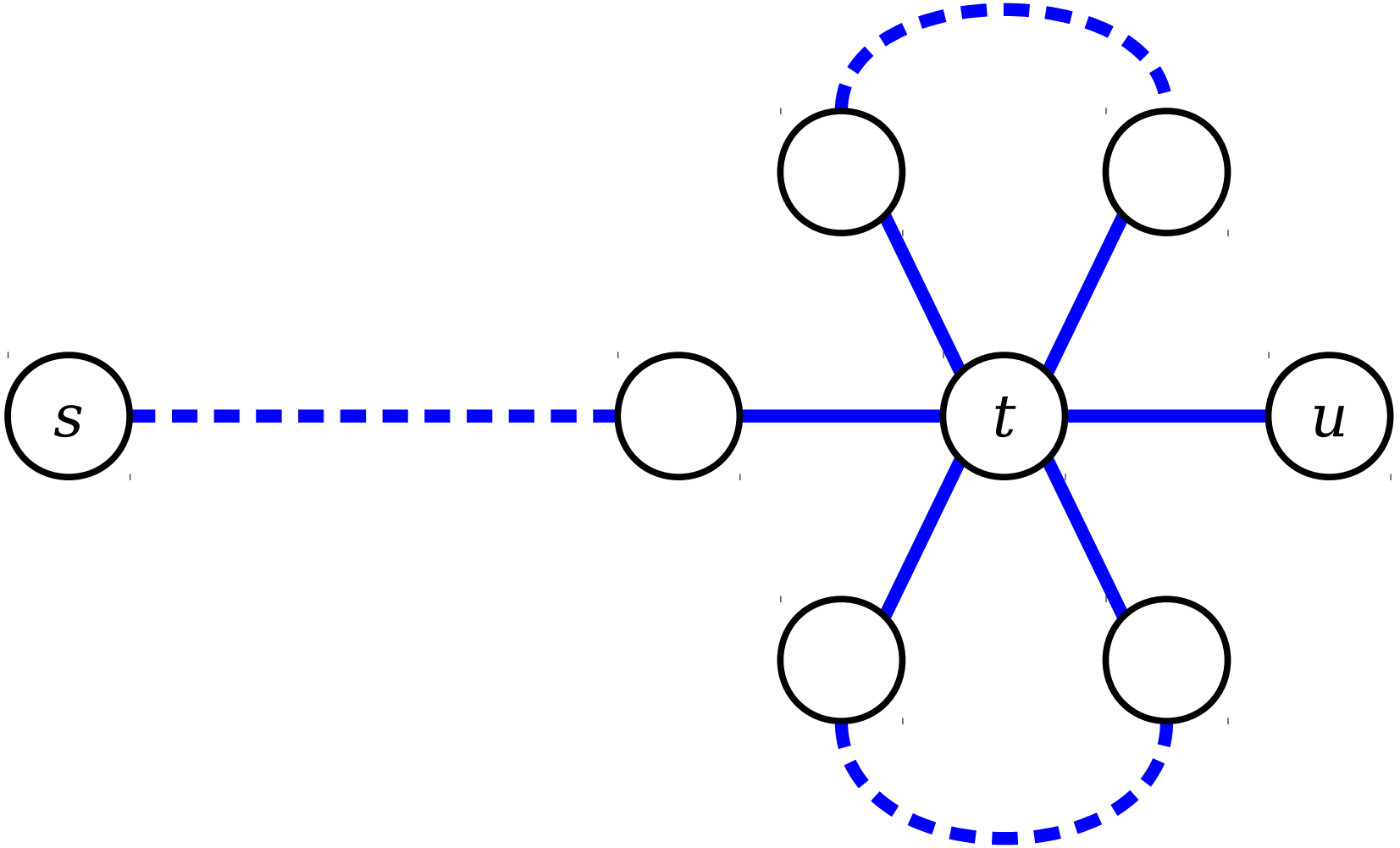
Suppose our path starts at s, ends at t, and that $s \neq t$.

Remember that all nodes in the graph have even degree.



Suppose our path starts at s, ends at t, and that $s \neq t$.

Remember that all nodes in the graph have even degree.



Suppose our path starts at s, ends at t, and that $s \neq t$.

Remember that all nodes in the graph have even degree.

Lemma 1: Let G be an undirected, connected graph where every node has even degree. If P is a path in G with no repeated edges that isn't a cycle, then P can be extended into a longer path P'.

Proof: Since P is not a cycle, it must start and end at different nodes. Let the end node be t. Since path P ends at node t, every time P passes through node t, it either ends, or it leaves t and then returns back to t later on.

Let's count how many edges adjacent to t are used by path P. One edge is used for the first time we enter t, and from that point forward, every time P exits and reenters t, we use up two more edges. In total, this means that P includes an odd number of edges incident to t. Since t has even degree, it is adjacent to an even number of nodes, there must be at least one more edge $\{t, u\}$ adjacent to t that is not included in P. Letting P' be path P extended by edge $\{t, u\}$ therefore produces a longer path than P. ■

The Basic Idea

- Start with the empty path.
- If the path you have so far isn't a cycle, extend it by adding in an edge.
  - (It's always possible to do this; we'll see why in a second.)
- If the path is a cycle and you aren't done yet, extend the cycle by following an edge hanging off it.
  - (It's always possible to do this; we'll see why in a second).

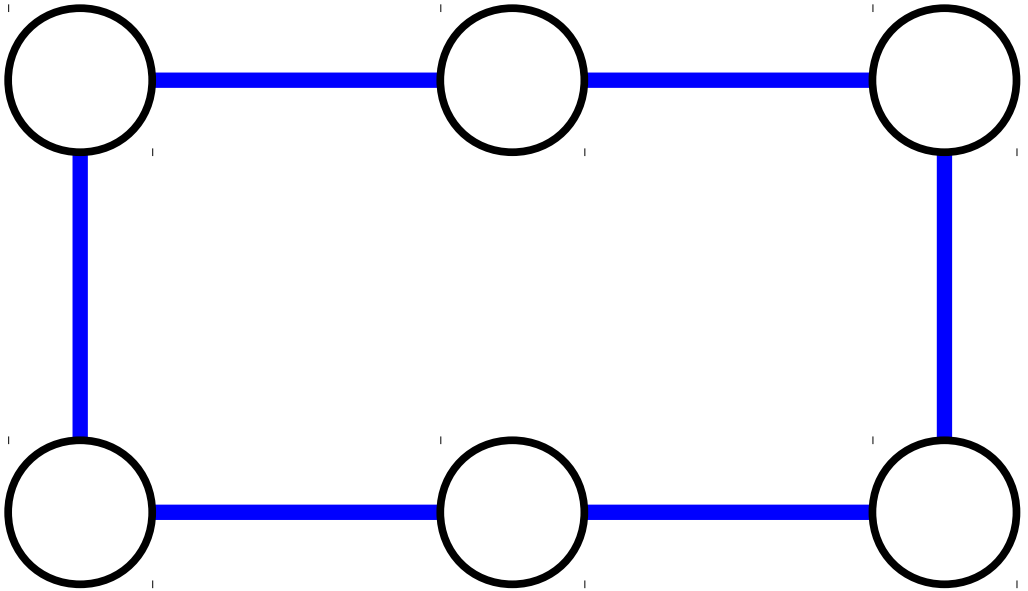
The Basic Idea

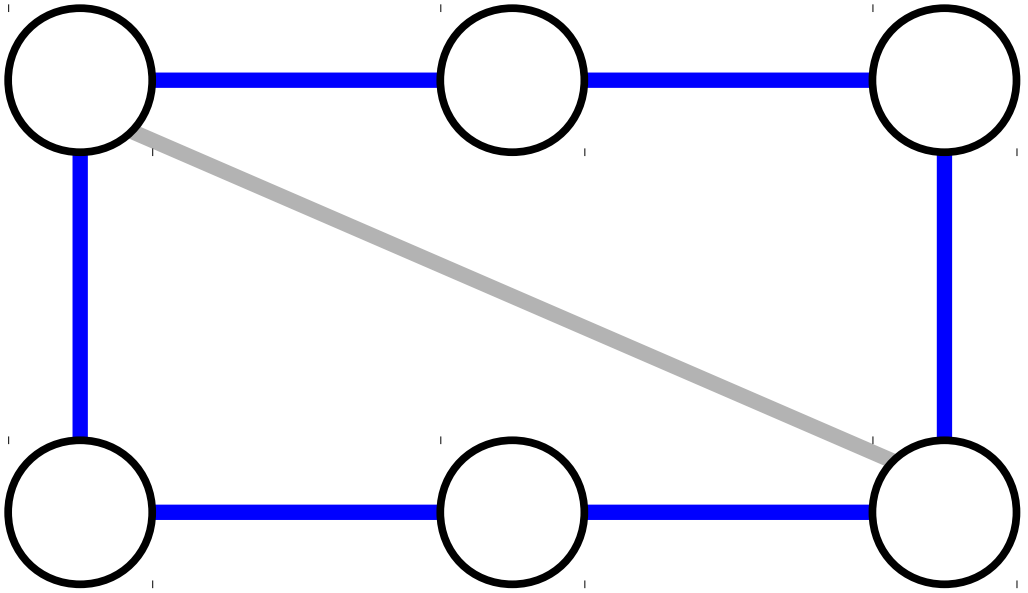
Start with the empty path.

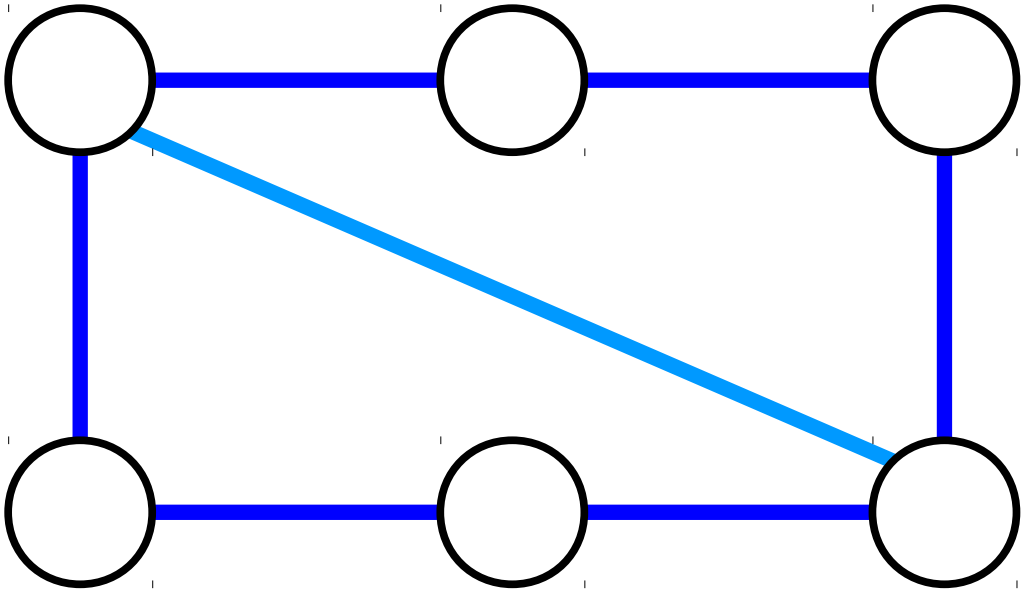
If the path you have so far isn't a cycle, extend it by adding in an edge.

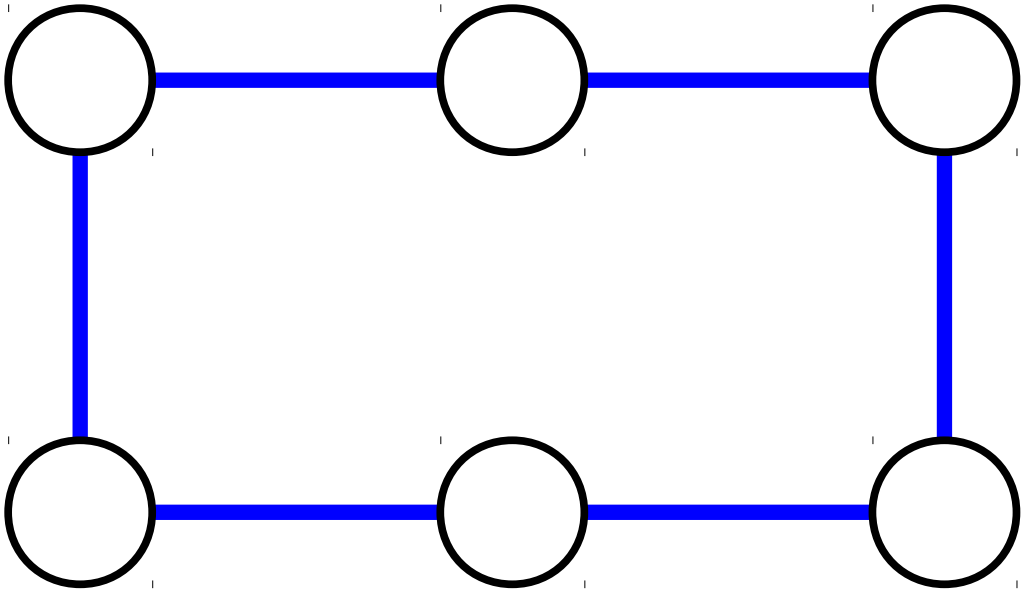
(It's always possible to do this; we'll see why in a second.)

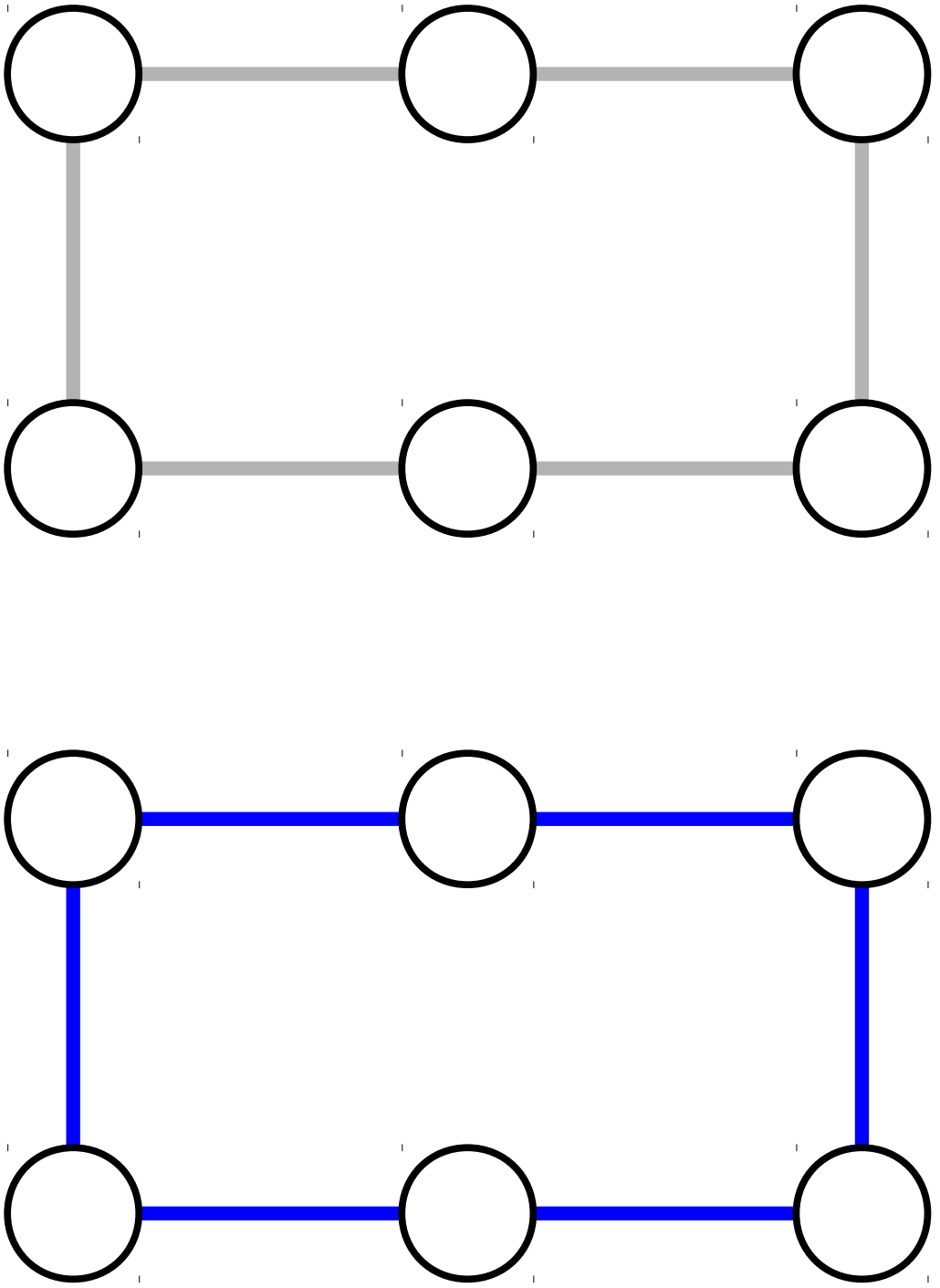
- If the path is a cycle and you aren't done yet, extend the cycle by following an edge hanging off it.
  - (It's always possible to do this; we'll see why in a second).

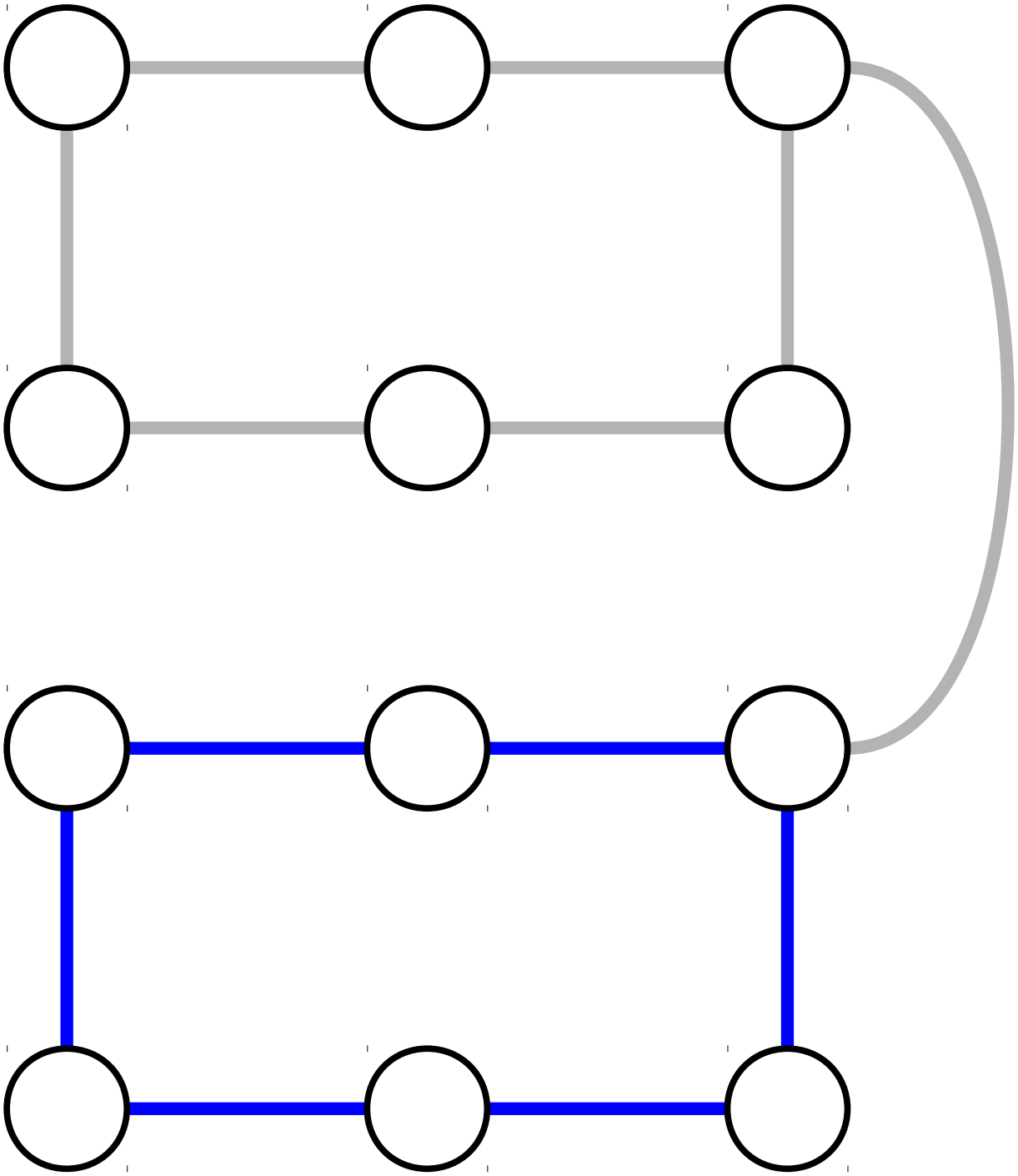


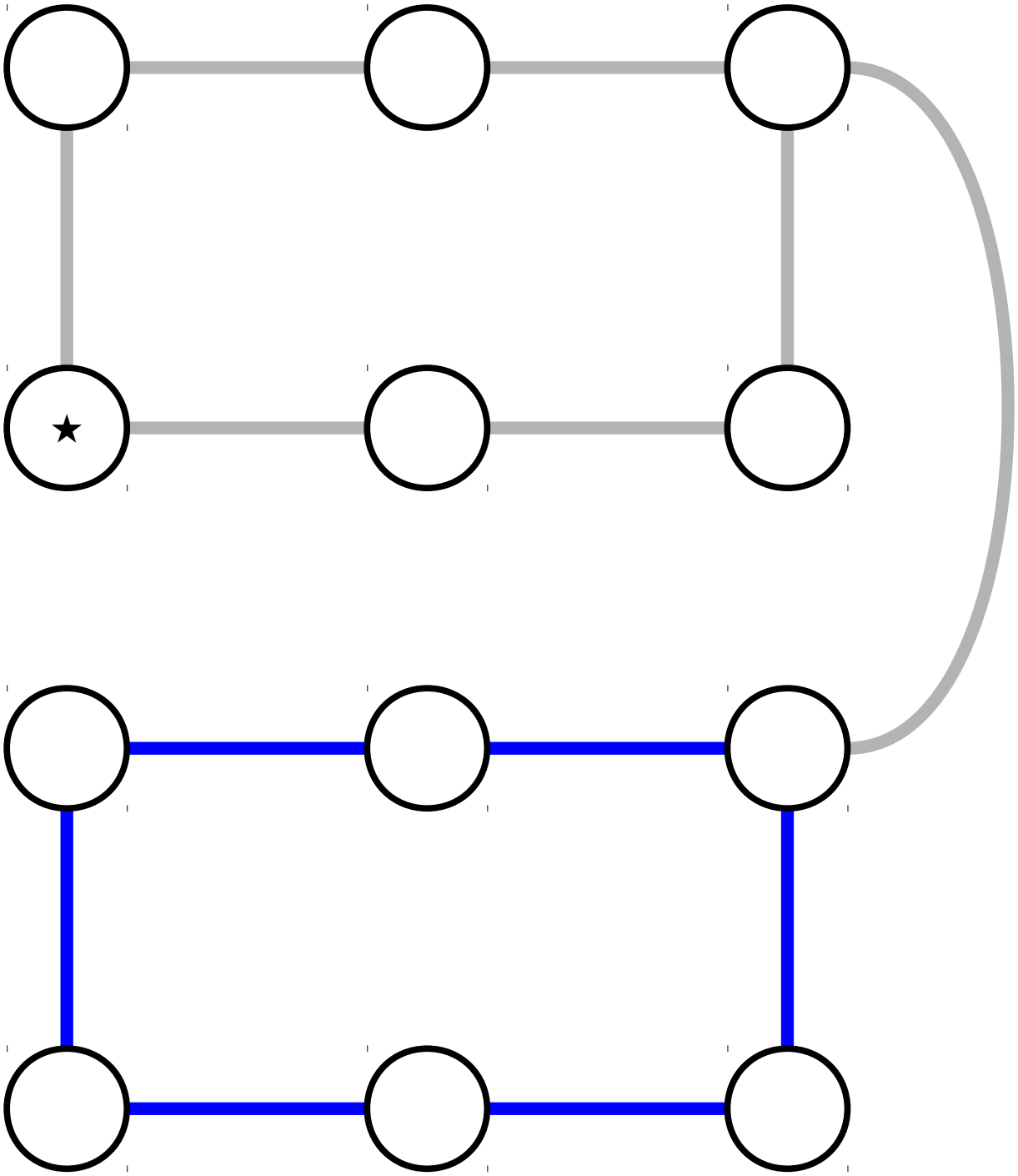


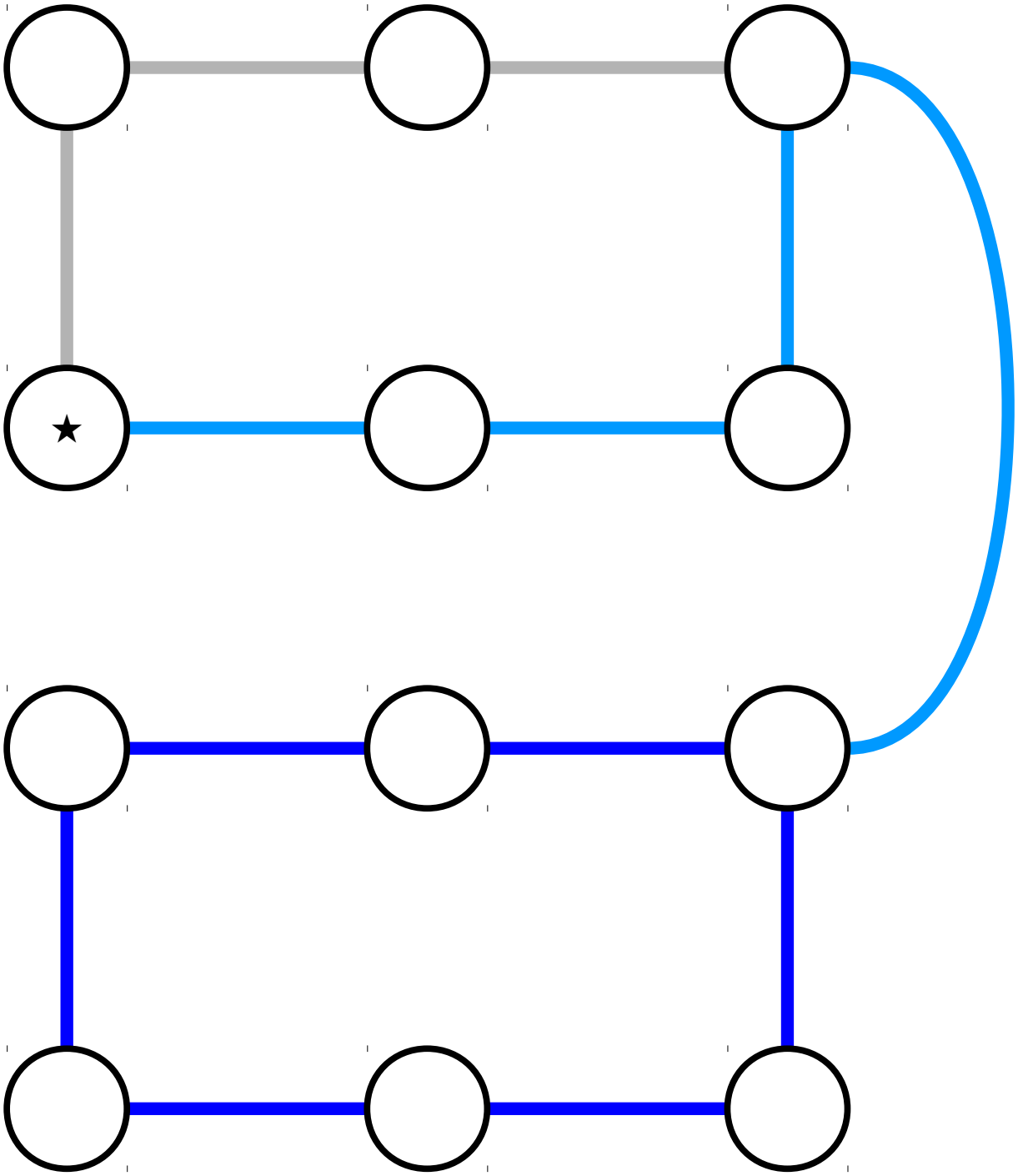












Lemma 2: Let G be an undirected, connected graph where every node has even degree. If C is a cycle in G that does not include every edge, then C can be extended into a longer path P' that includes all the edges of C.

Proof: Let M be the set of nodes included on cycle C. We will show there must be an unused edge $\{u, v\}$ in G such that $u \in M$. From there, we can form P' as follows: starting at node u, follow cycle C back to node u, then follow the edge $\{u, v\}$.

If there are any unused edges in G that connect two nodes in M, then we can pick any them and we're done.

If not, consider any nodes $x \in M$ and $y \in V - M$. Since G is connected, there must be a path from x to y; choose any one of them and call it P. Let u be the last node along path P that is in M. Path P ends at a node in $V - M$, so u can't be the last node in P. Let v be the node after u in path P.

Since u is the last node along P that's in M, we know that $v \notin M$. This means that the edge $\{u, v\}$ is not in cycle C, since otherwise we would have $v \in M$. Therefore, we've found our unused edge $\{u, v\}$ where $u \in M$, so we can extend C using the technique described earlier. ■

Theorem: If G is a connected graph where every node has even degree, then G is Eulerian.

Proof: Consider the longest path P in G with no repeated edges. We claim that P is an Eulerian cycle. To see why, suppose not. If P isn't a cycle, then by Lemma 1, we can extend P into a longer cycle P', contradicting that P is the longest path in the graph. If P doesn't include all edges, then by Lemma 2 we can extend P into a longer path P', contradicting that P is the longest path in the graph.

In both cases we reach a contradiction, so our assumption was wrong. Therefore, the longest path in G is an Eulerian circuit, so G is Eulerian, as required. ■

Time-Out for Announcements!

Midterm Logistics

- Midterm exam is Monday, October 26.
- Room assignments divvied up by last (family) name:
  - **`Aga - Ven`**: Go to Hewlett 200.
  - **`Ver - Zhe`**: Go to 370-370
- Remember that you get an 8.5” × 11” notes sheet, double-sided, during the exam.

Practice Problems

- Solutions to EPP2 have been released.
- We've released EPP3 and solutions today. (Try to solve the problems *before* you look a the solutions!)
- We've also released a set of challenge problems that build up to a cool result. Feel free to work on it if you'd like!

Problem Set Five

- Problem Set Five goes out now. It's due on Friday of next week.
- There is no checkpoint problem.
- We hope you have fun with this – it's all about induction in its many forms!
- As always, stop by office hours or ask on Piazza if you need help.

Your Questions

“How are you doing Keith?”

I'm doing a lot better now. Sorry for being a bit flustered on Wednesday. Everything's okay now!

“What other CS conferences are there besides Grace Hopper?”

There are tons of them! The ACM has a bunch of conferences (SIGGRAPH, SIGCOMM, SIGCSE, etc.) There are a lot of other conferences in areas like AI, HCI, etc.

There's also a lot of software engineering conferences, like PyCon, CppCon, etc.

“What are the medians for problem sets 2 and 3?”

PS2 has a median of 34/41, factoring in checkpoints.

PS3 has a median of 36/40, factoring in checkpoints.

Again, remember that problem set grades are not the major determining factor in your overall course grade. Use the feedback to improve your performance, but don't sweat the small stuff.

"Were you involved in research as an undergrad? How do you think undergrads should decide who to work with and what to work on?"

I did some research as an undergrad. It was fun, but I didn't really understand the expected dynamic and didn't get as much out of it as I should.

The best advice I have is to ask around to figure out who's a good person to work with and to find a mentor so that you have someone to get advice from. Also, strongly consideo trying to get started through CURIS - it's a super low-barrier way to get involved.

Back to CS103!

Theorem: An (undirected) graph G is Eulerian if and only if it is connected and every node has even degree.

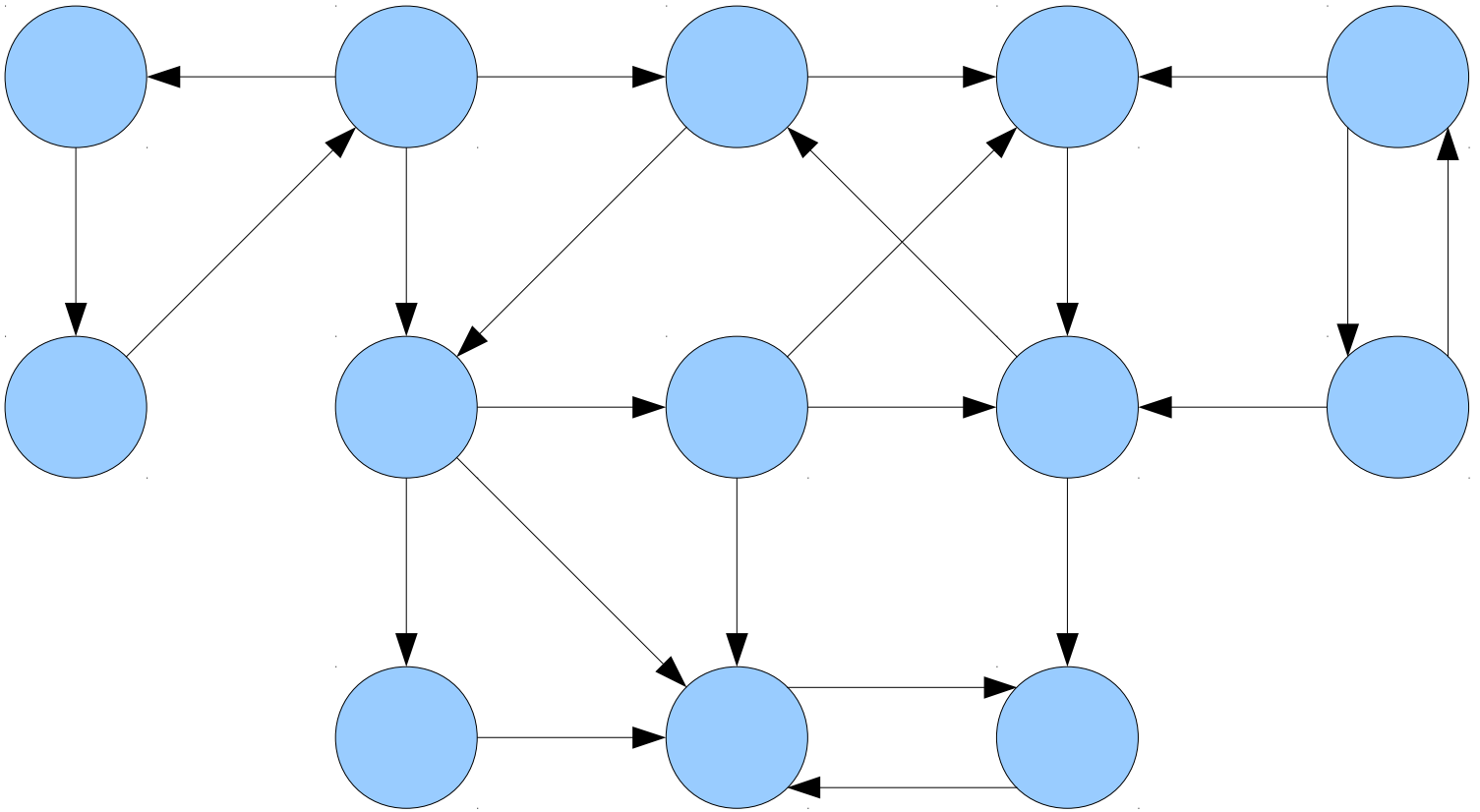
What about directed graphs?

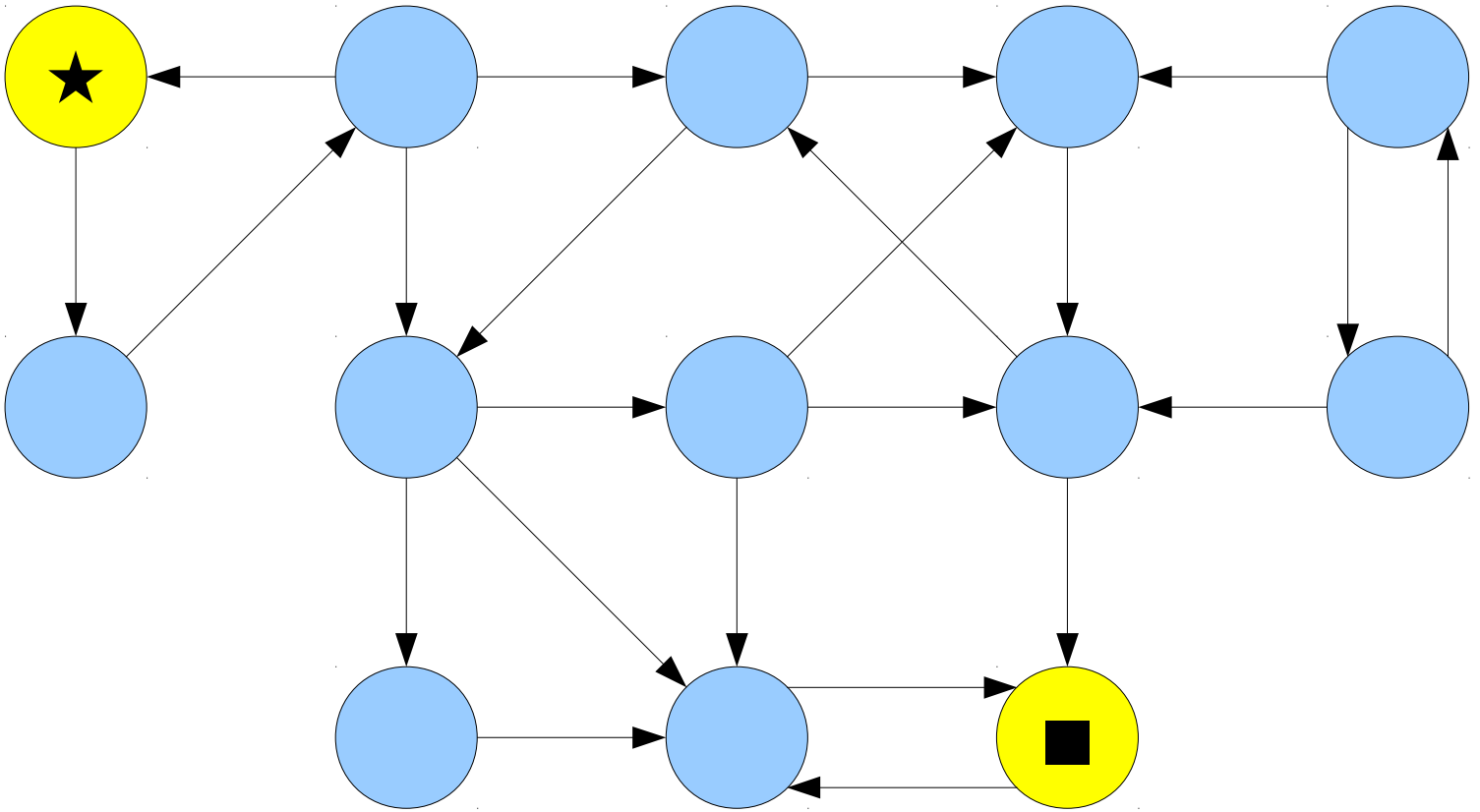
Theorem: An (undirected) graph G is Eulerian if and only if it is connected and every node has even degree.

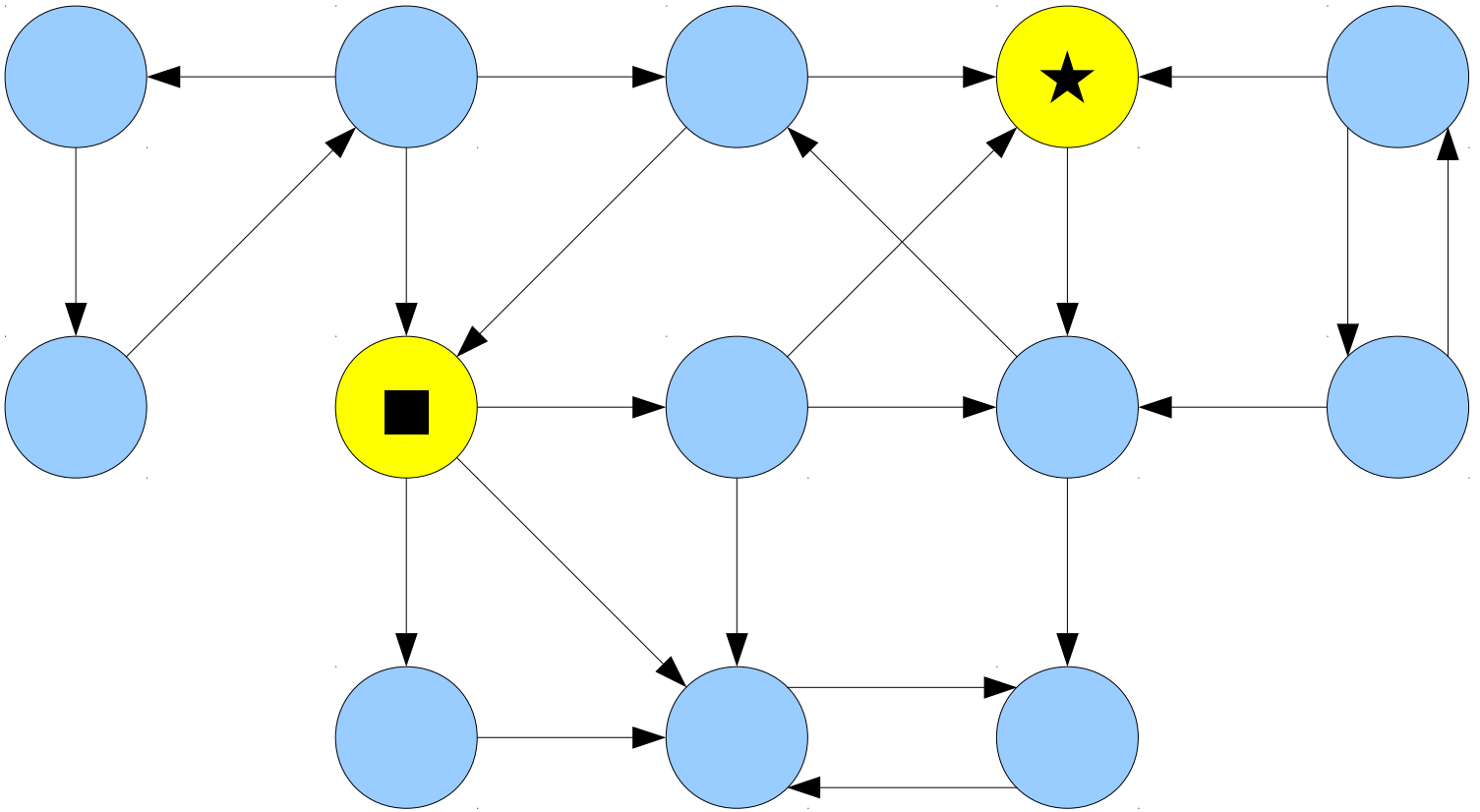
What about directed graphs?

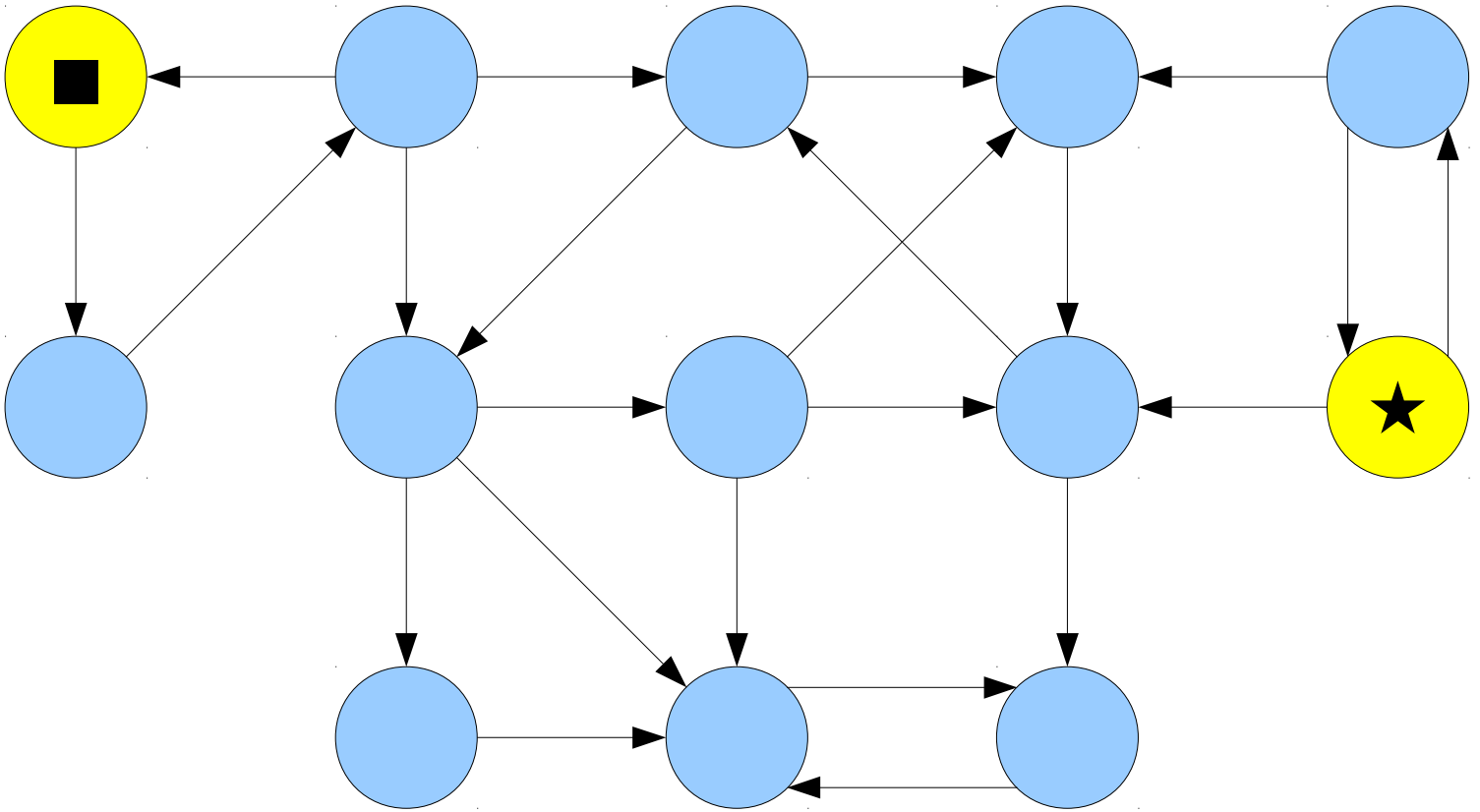
Directed Connectivity

- In a directed graph G, we say v is ***reachable*** from u if there is a path from u to v.
- In an undirected graph, if there is a path from u to v, there is also a path from v to u.
- In a directed graph, it is possible for there v to be reachable from u, but for u not to be reachable from v.









Strong Connectivity

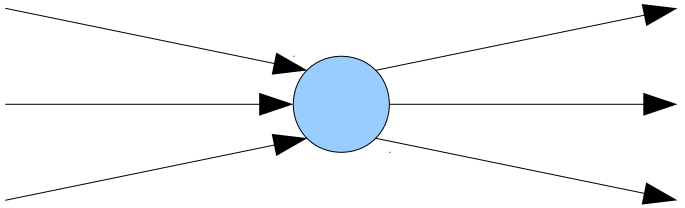
- Two nodes u and v are called ***strongly connected*** if v is reachable from u and u is reachable from v.
- A directed graph G as a whole is called ***strongly connected*** if any pair of nodes in G are strongly connected.
- This generalizes the idea of connectivity from undirected graphs to directed graphs.
- If you take CS161, you'll see some cool algorithms involving strongly connected components in a graph!

Theorem: An (undirected) graph G is Eulerian if and only if it is connected and every node has even degree.

Theorem: An (undirected) graph G is Eulerian if and only if it is connected and every node has even degree.

Directed Degrees

- In a directed graph, the ***indegree*** of a node v is the number of edges entering v. The ***outdegree*** of a node v is the number of edges leaving v.
- In the undirected case, we wanted an even degree at each node so that we could leave any node we enter.
- In the directed case, we accomplish this by requiring each node's indegree to equal its outdegree.

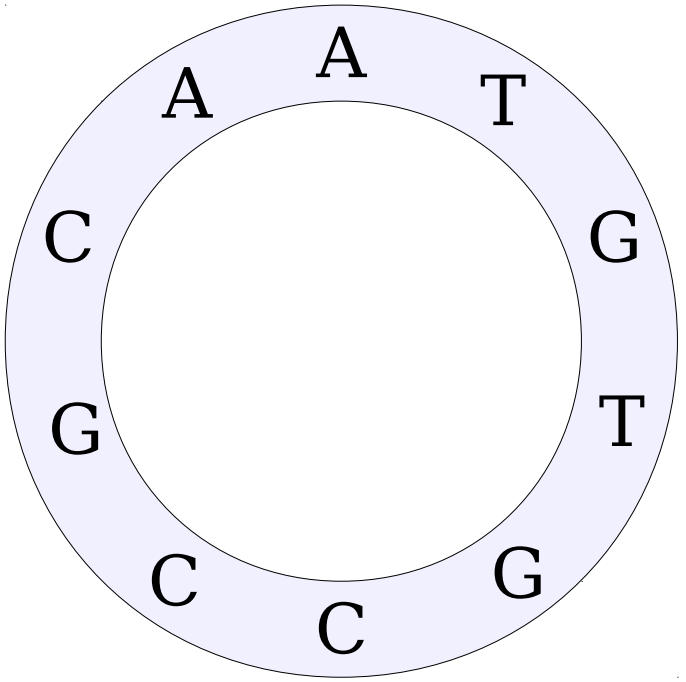


Theorem: A directed graph G is Eulerian if and only if it is strongly connected and every node's indegree equals its outdegree.

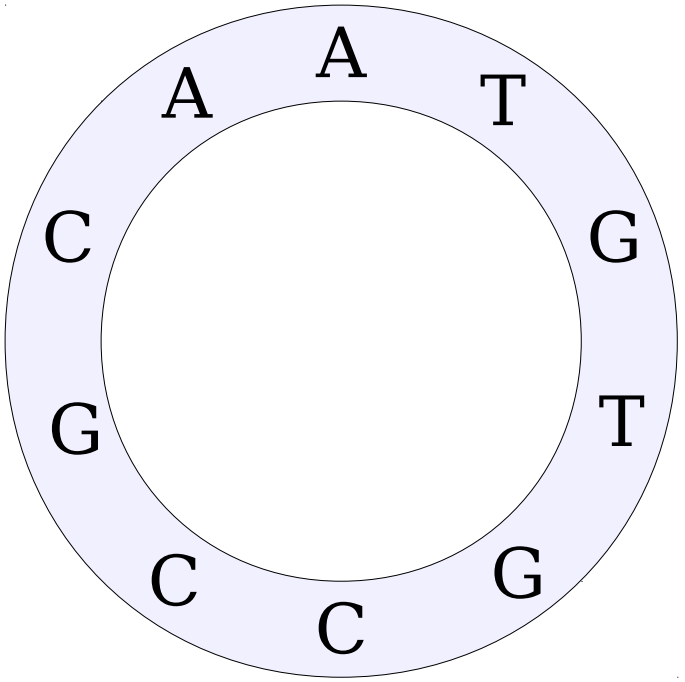
Proof Idea: Repeat the proof of the undirected case with some minor modifications.

An Application: ***Plasmid Reassembly***

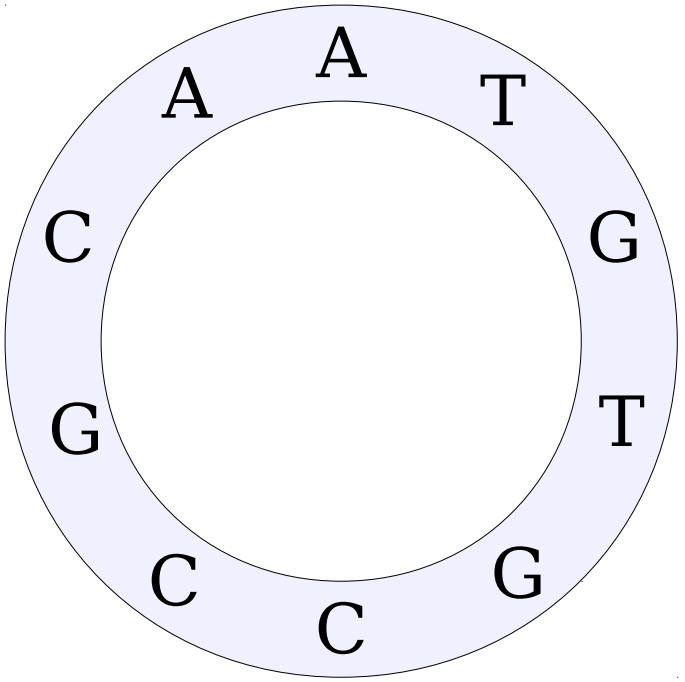
Plasmid Reassembly



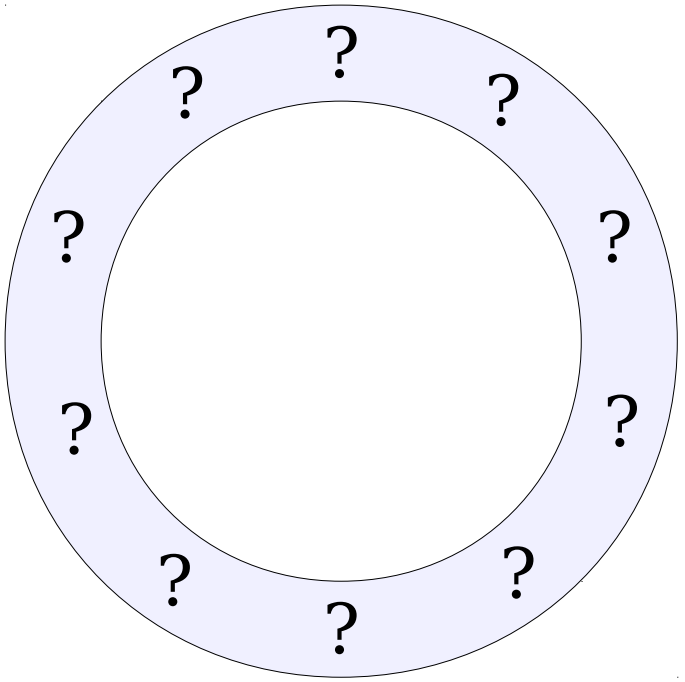
Plasmid Reassembly



Plasmid Reassembly



Plasmid Reassembly



A T G T

C A A T G

G T G C

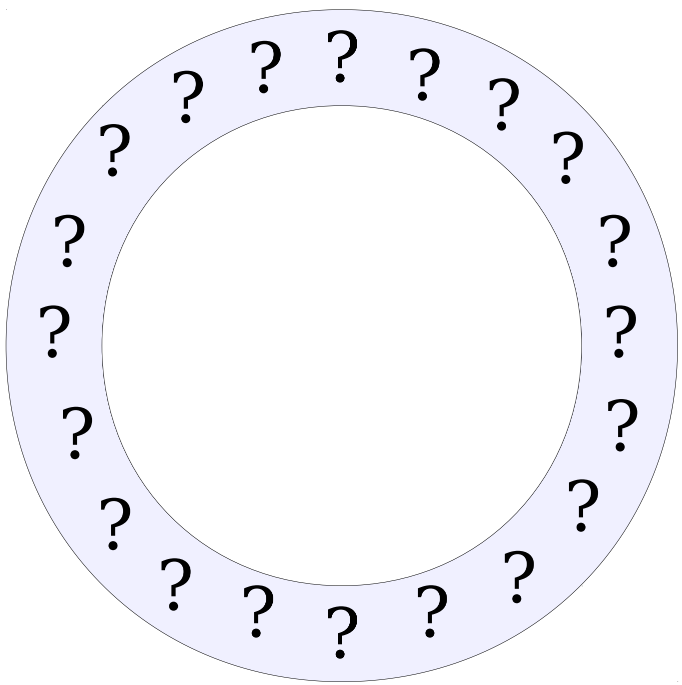
C C G C A A

C G C A

A A T G

T G T G C C

Plasmid Reassembly



ATGT

CAATG

GTGC

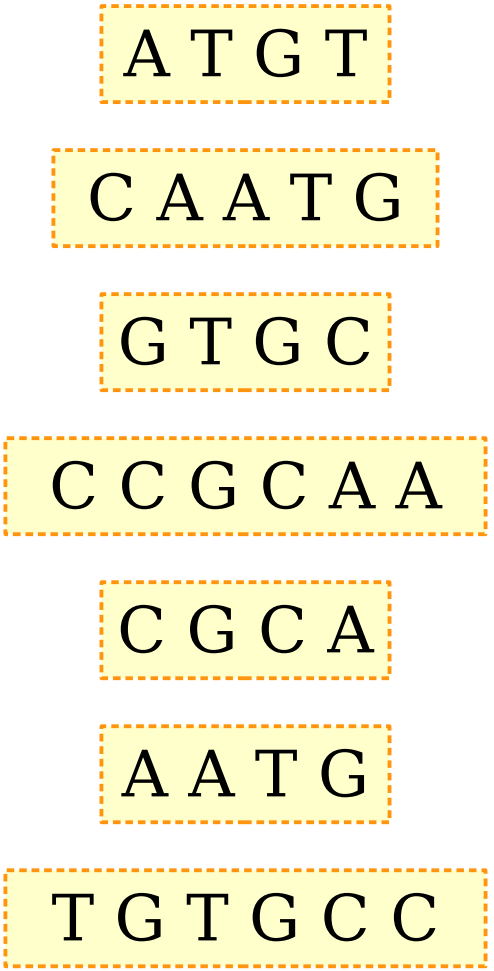
CCGCAA

CGCA

AATG

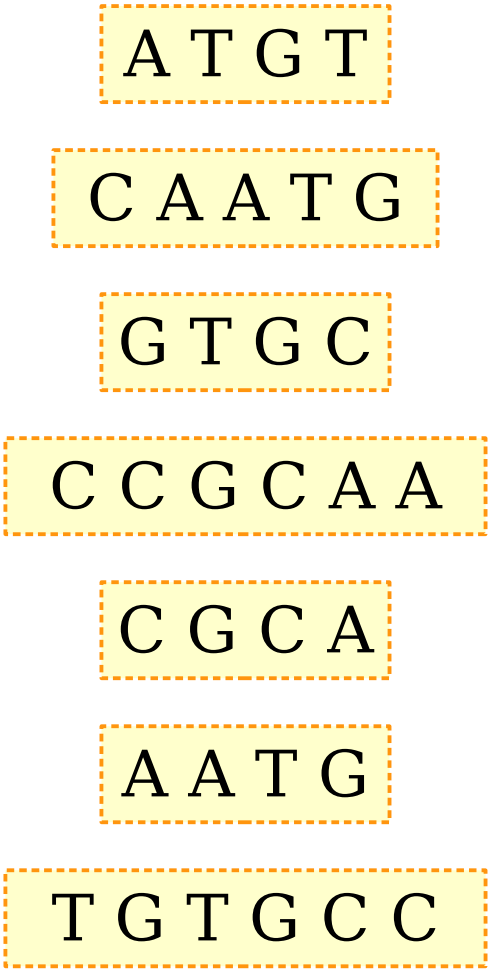
TGTGCC

Plasmid Reassembly

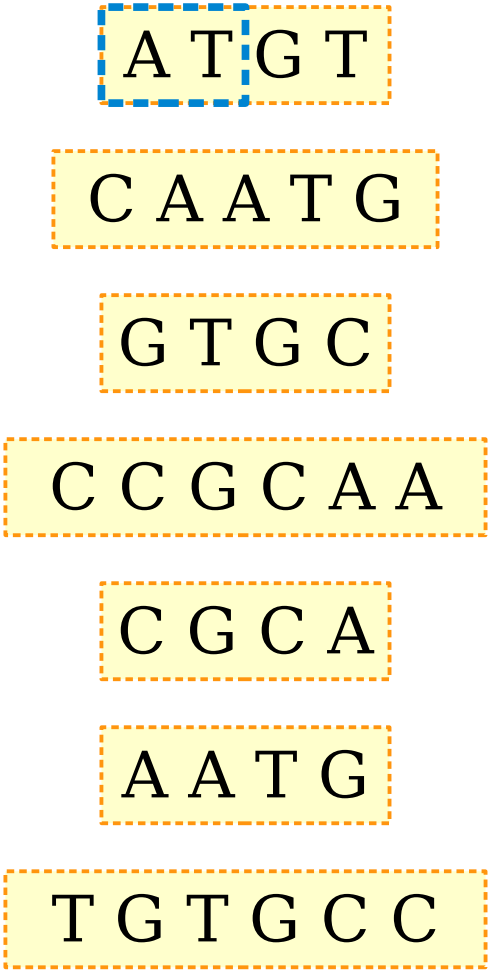


Plasmid Reassembly

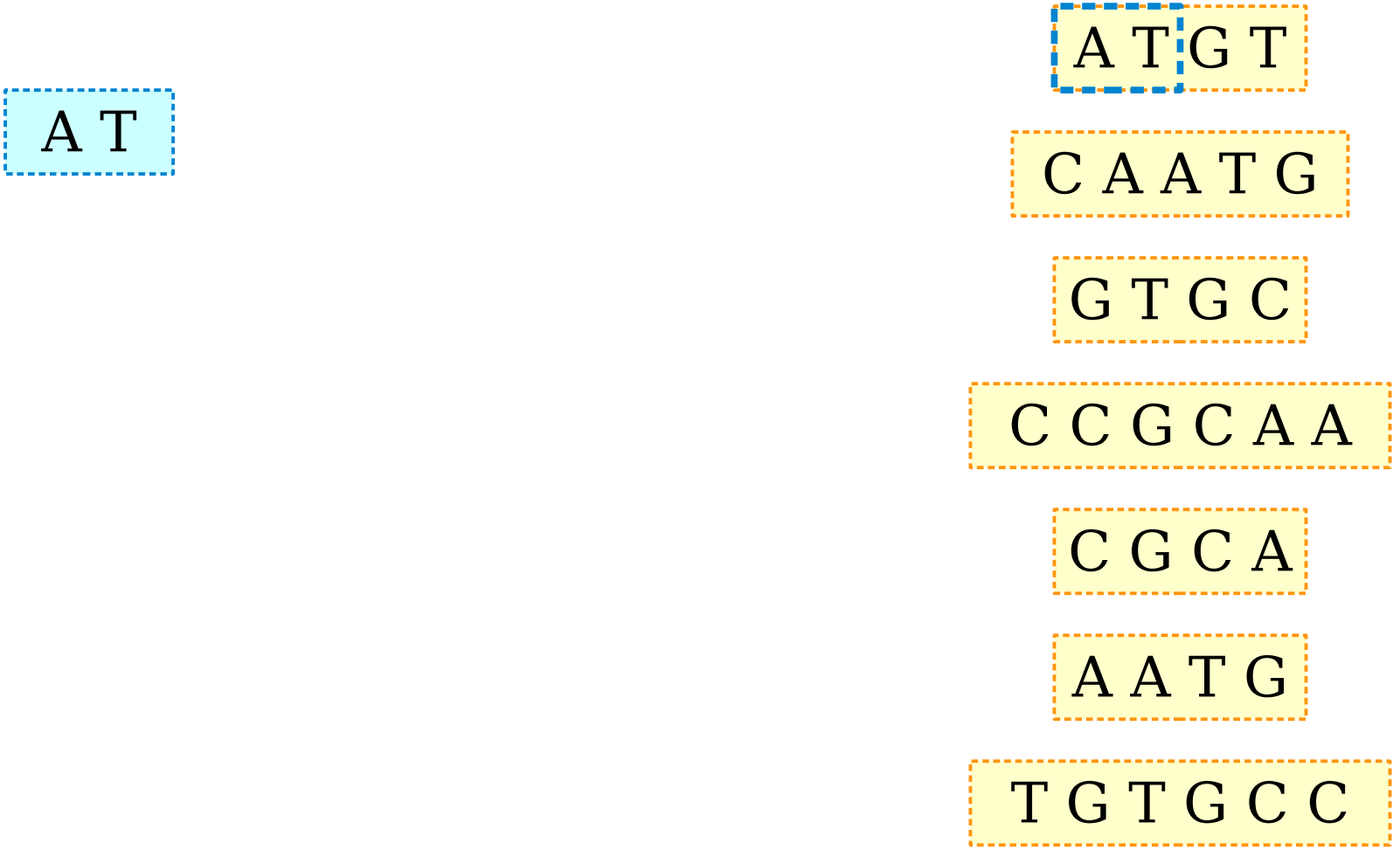
Assumption: The DNA reads provide high coverage of the entire plasmid.



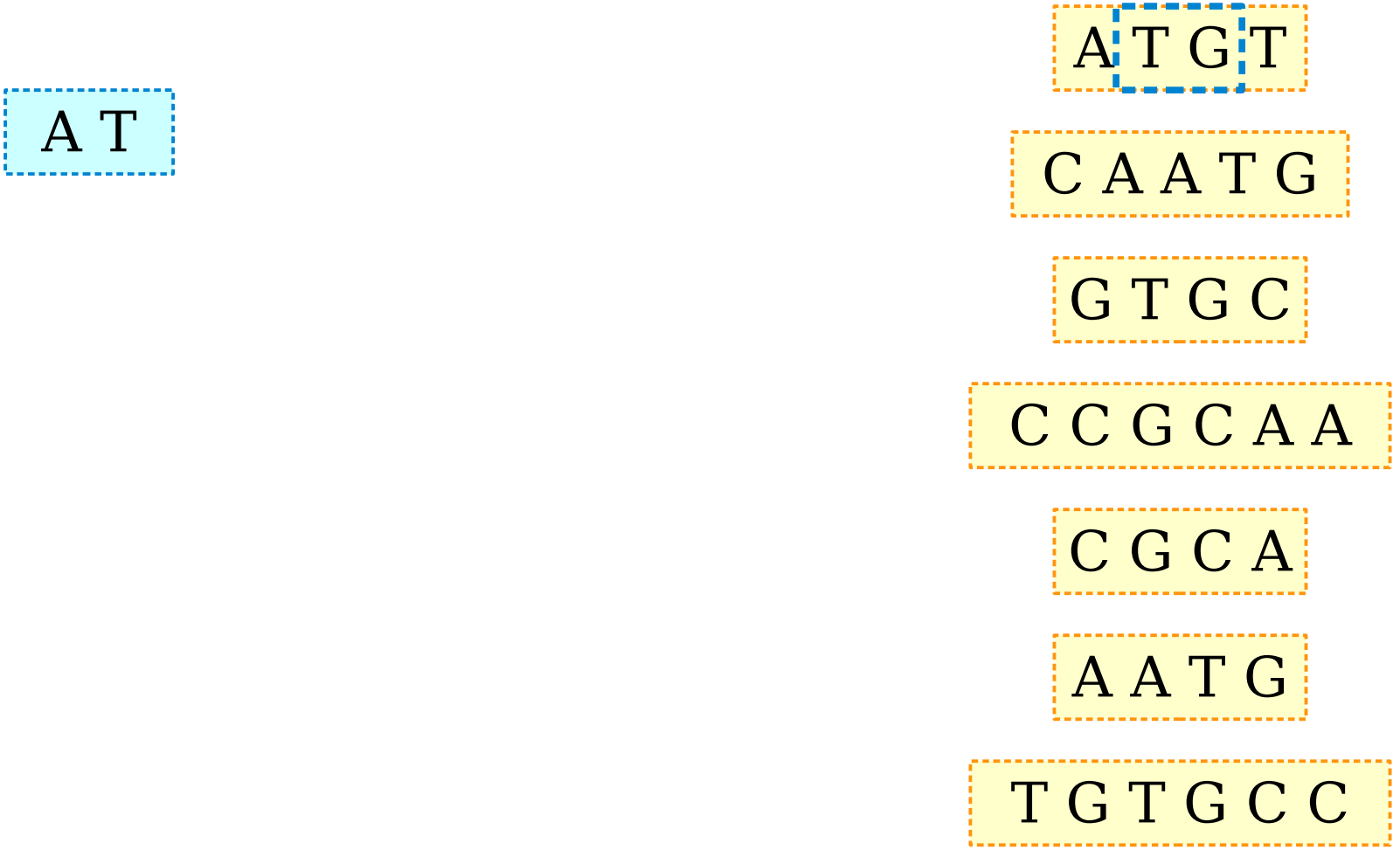
Plasmid Reassembly



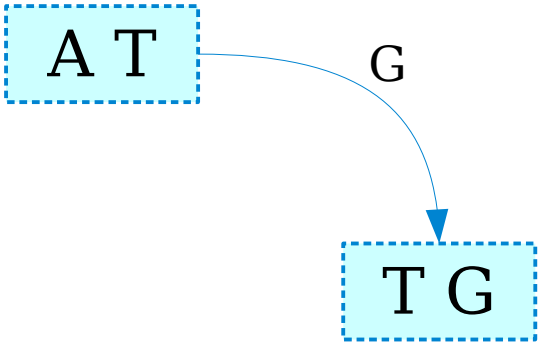
Plasmid Reassembly

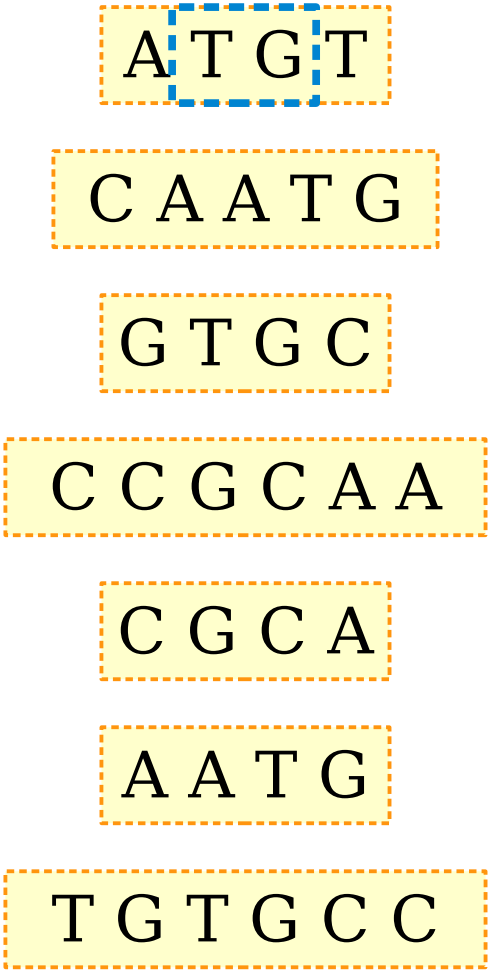


Plasmid Reassembly

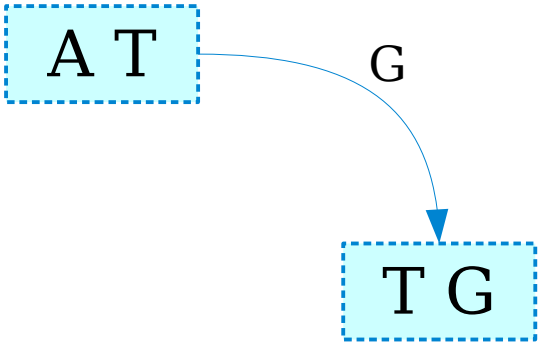


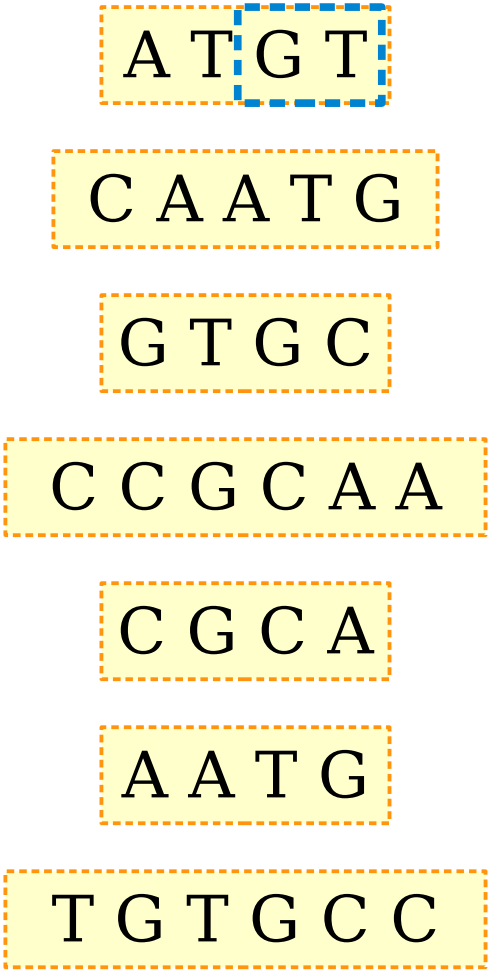
Plasmid Reassembly



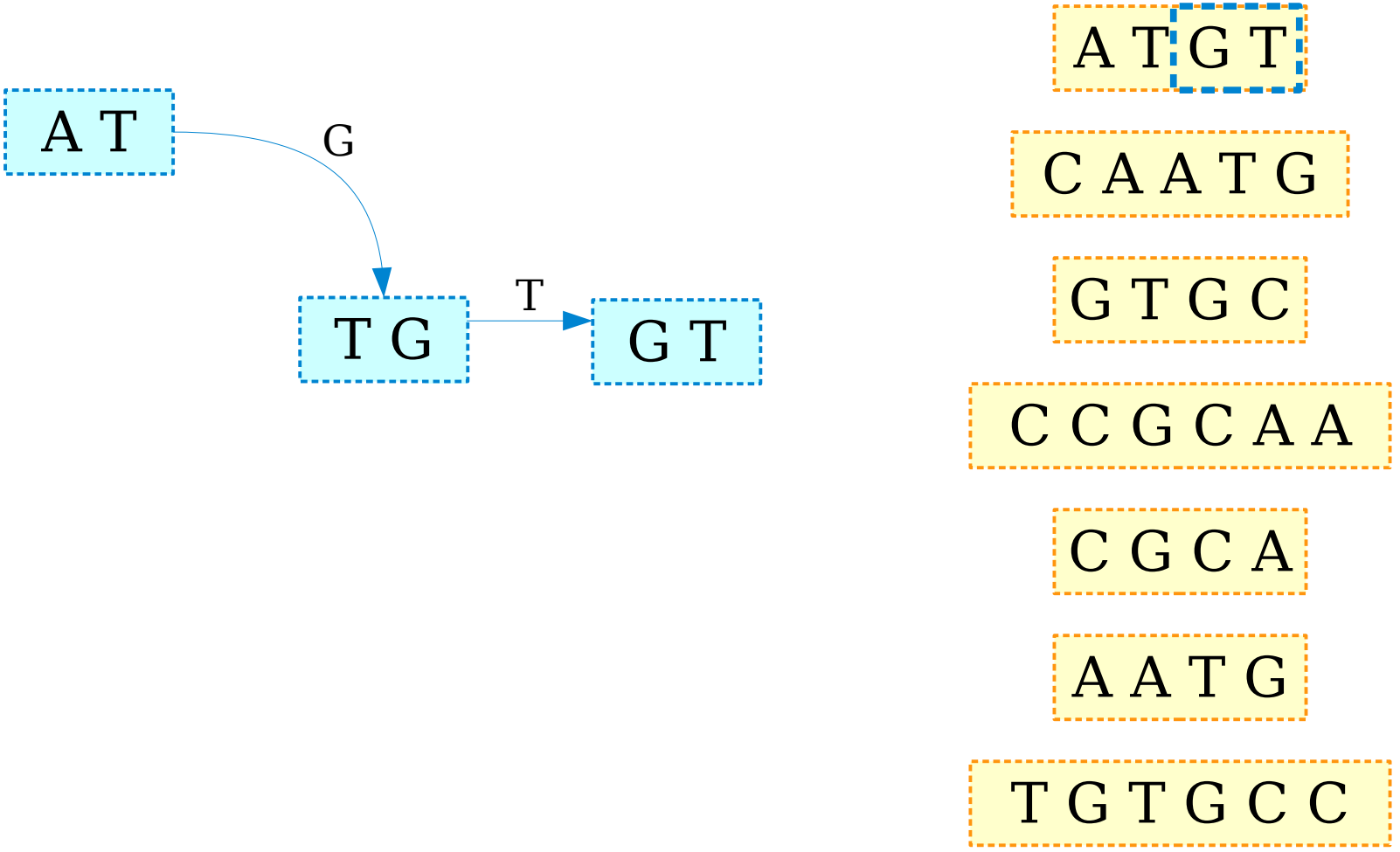


Plasmid Reassembly

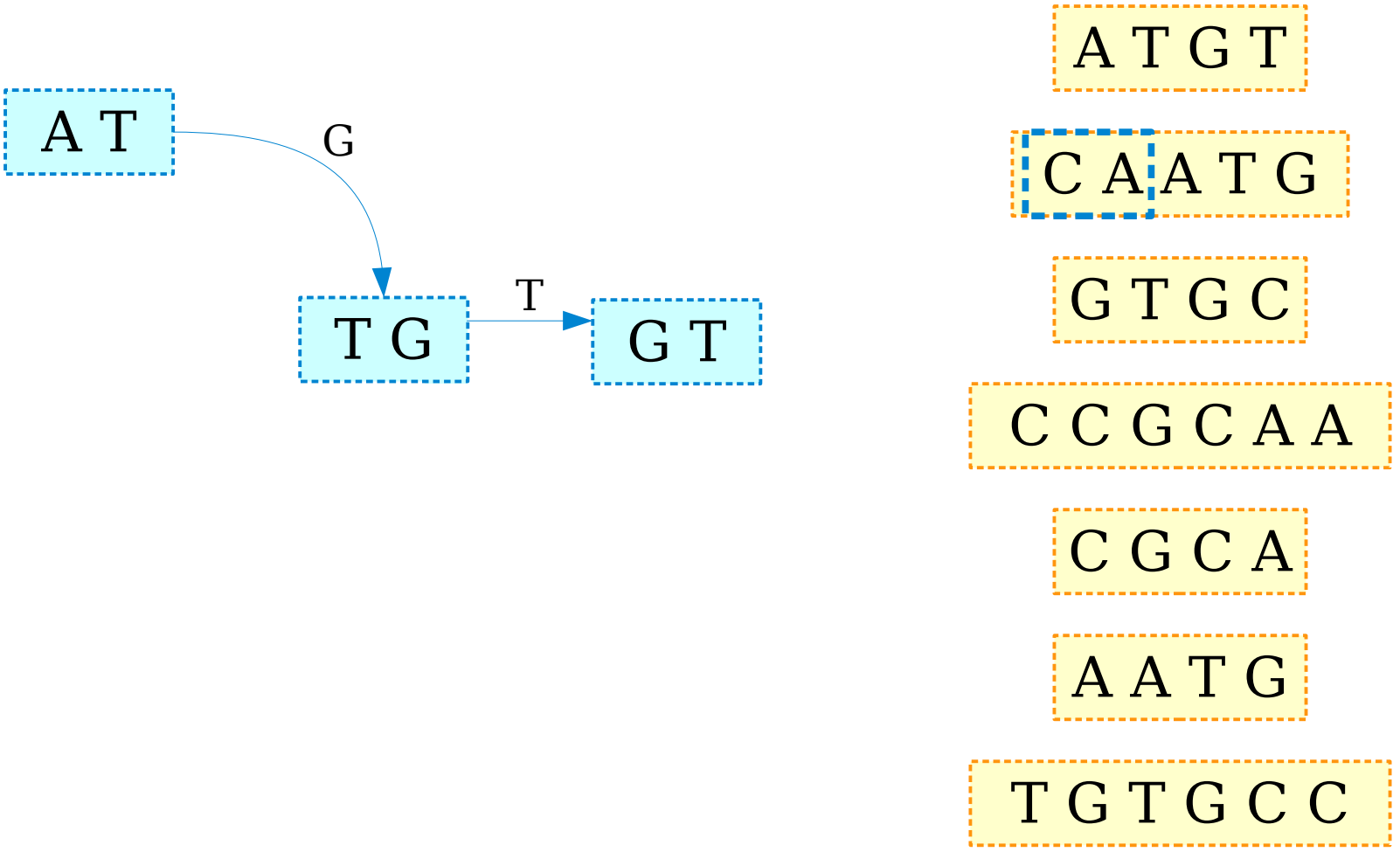




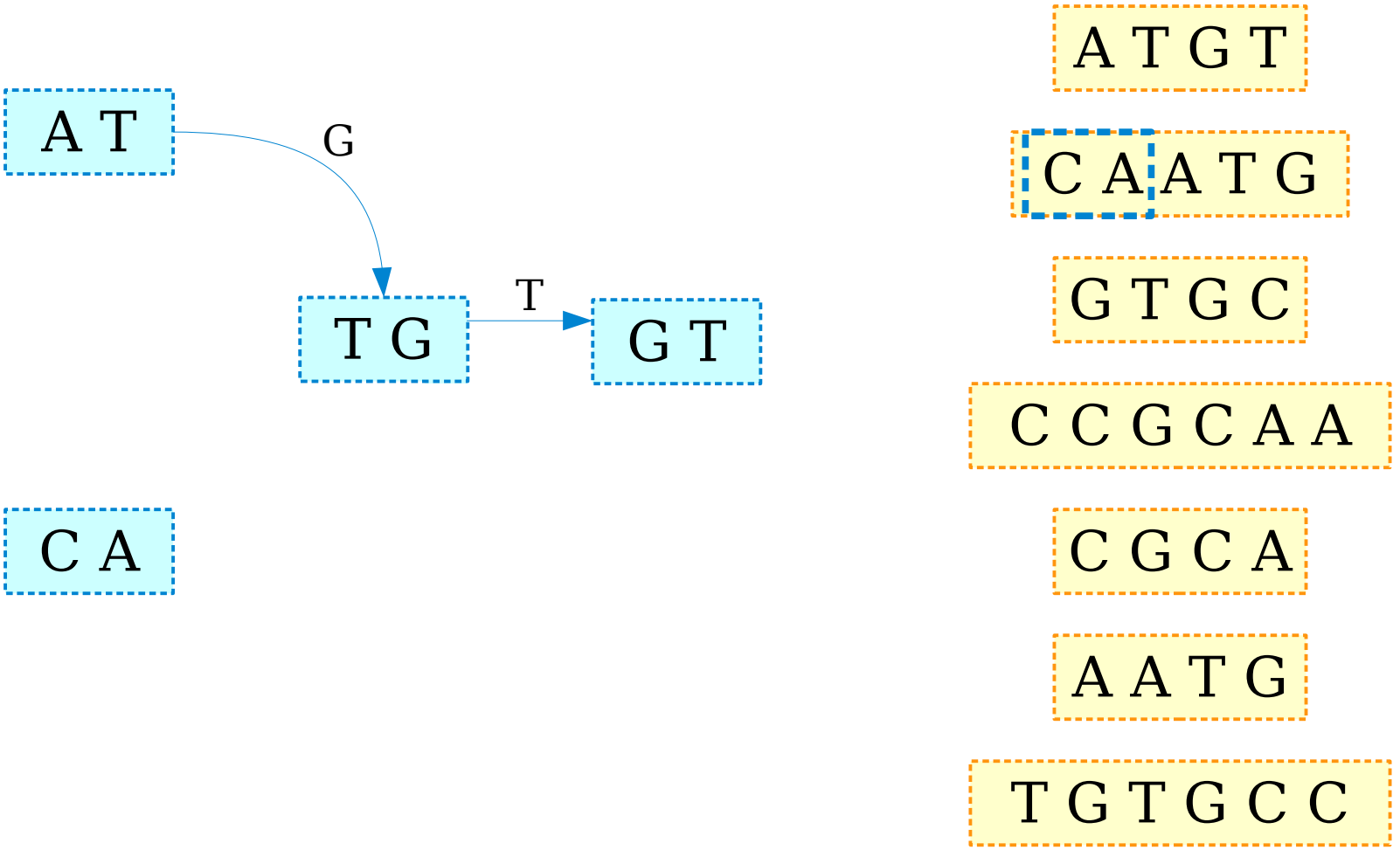
Plasmid Reassembly



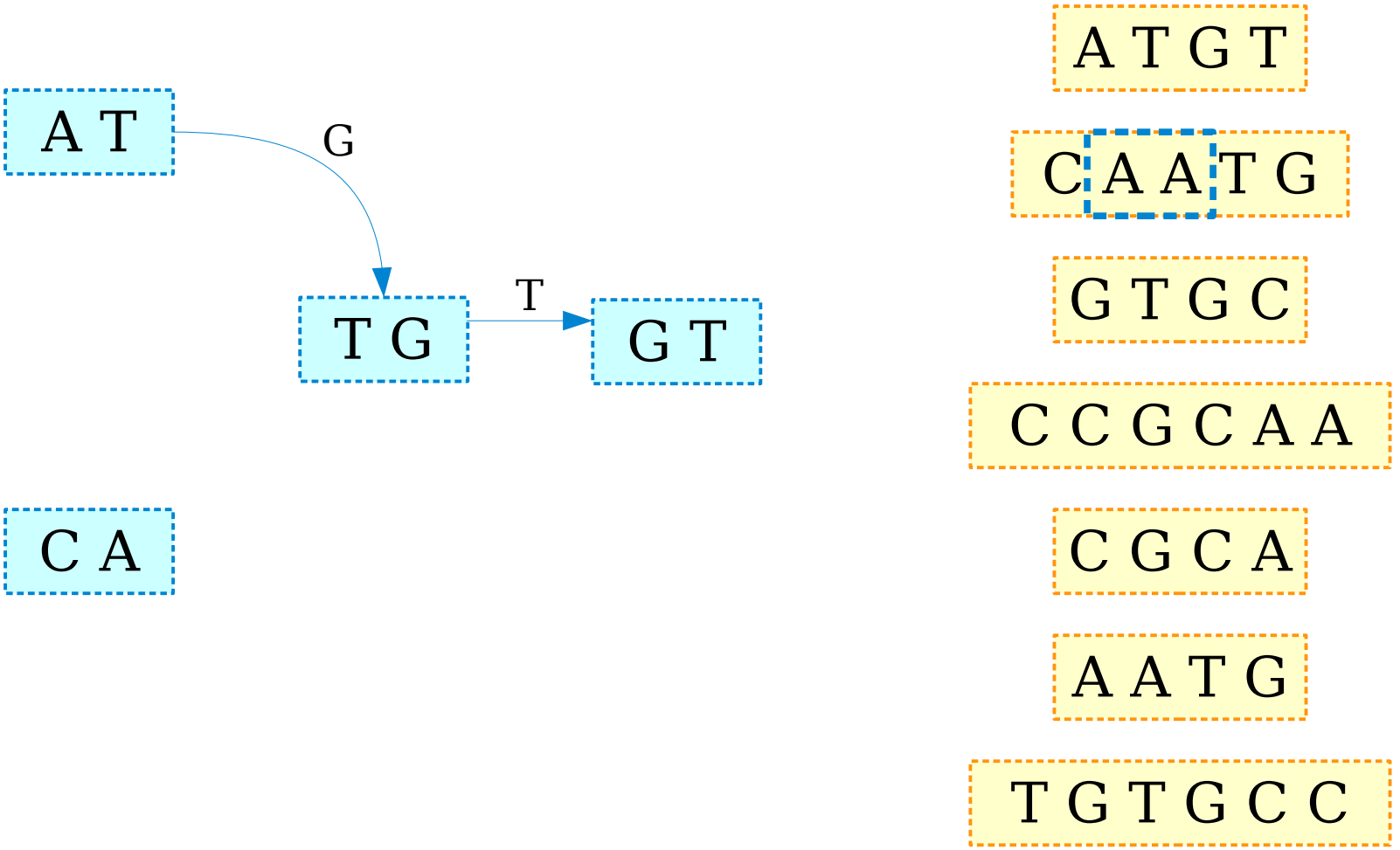
Plasmid Reassembly



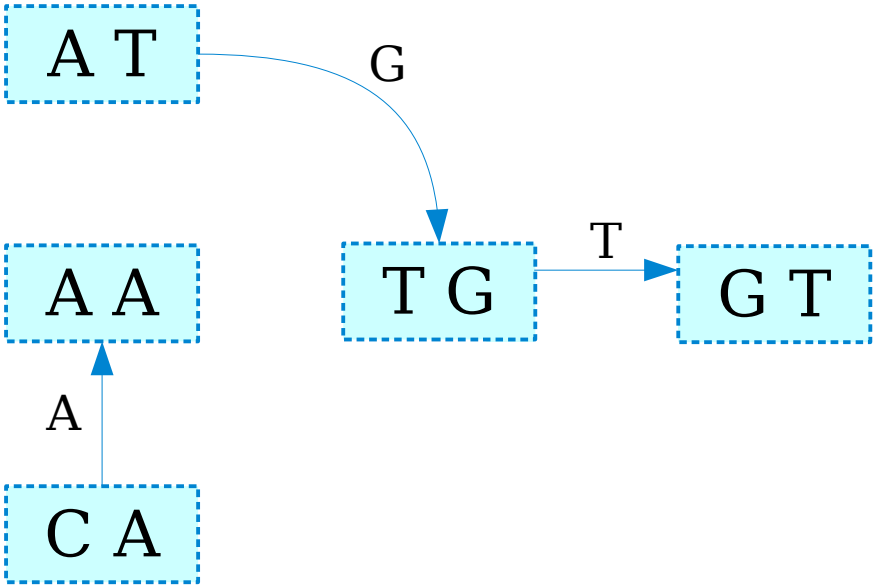
Plasmid Reassembly

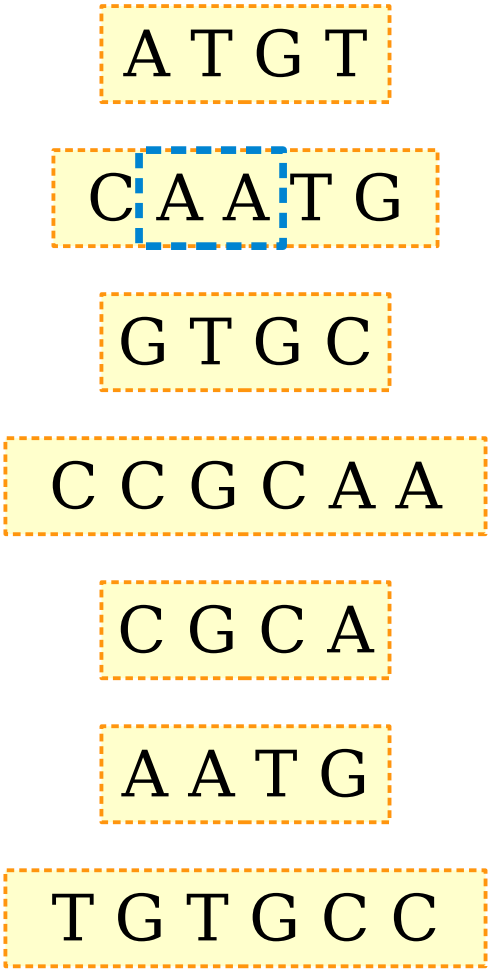


Plasmid Reassembly

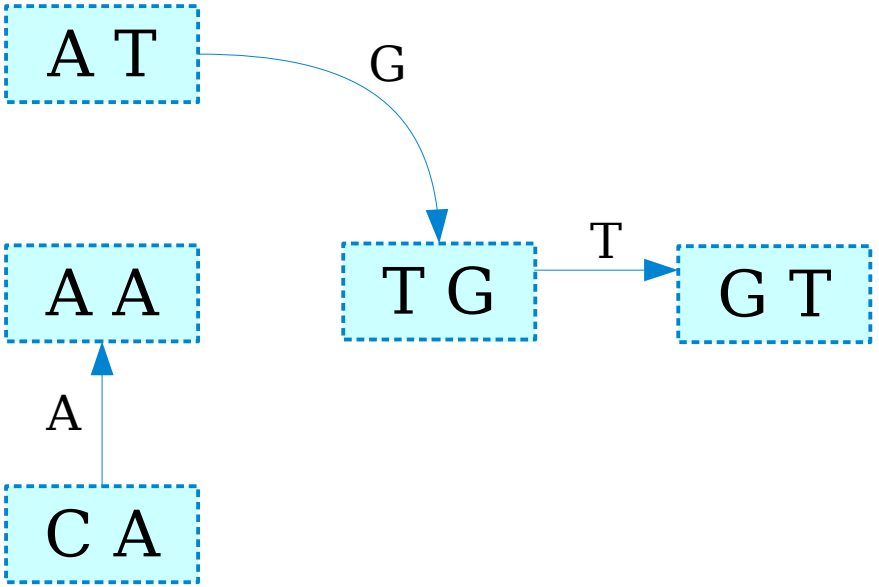


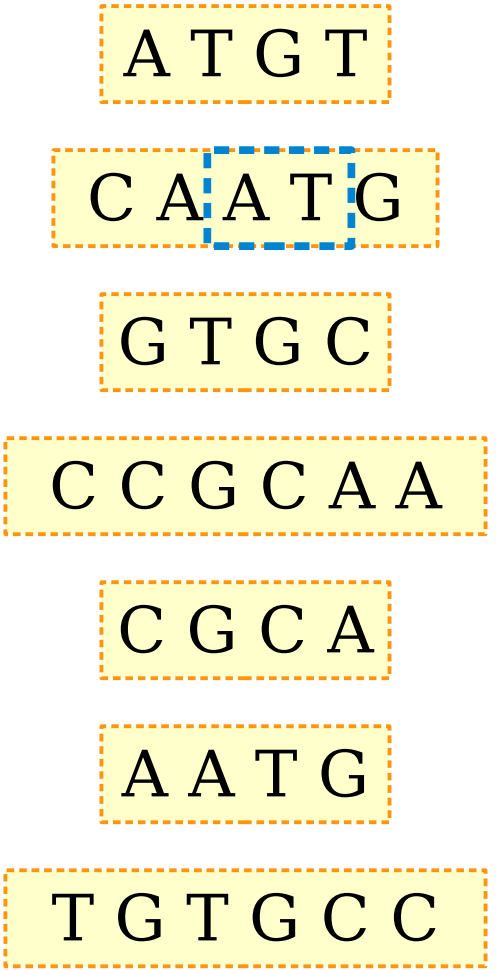
Plasmid Reassembly



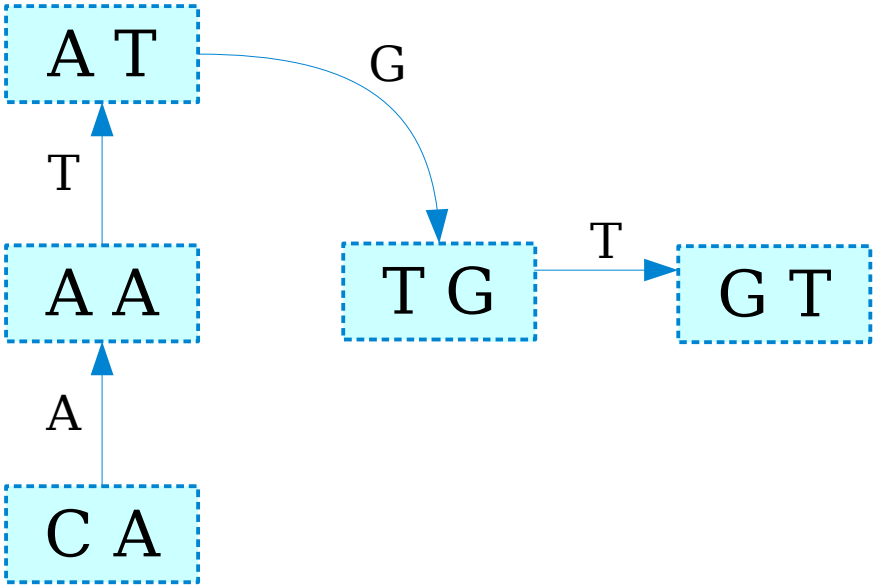


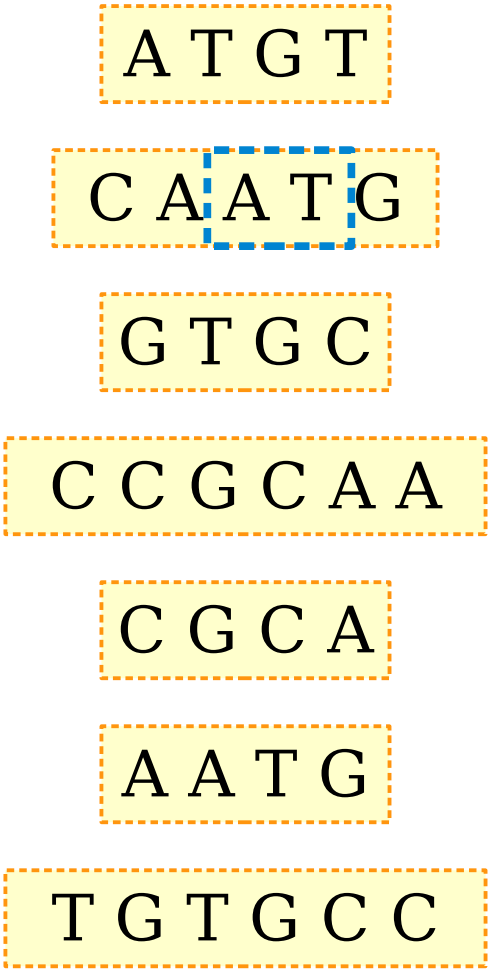
Plasmid Reassembly



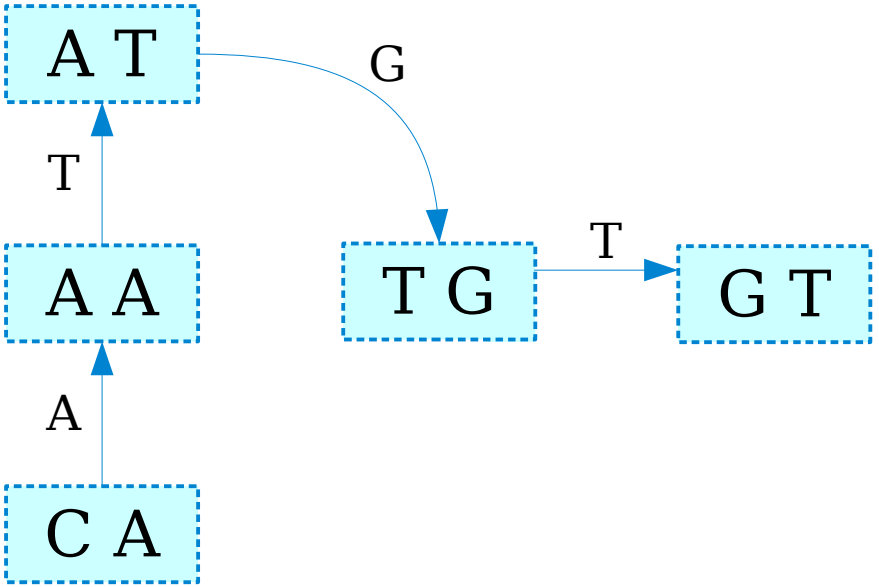


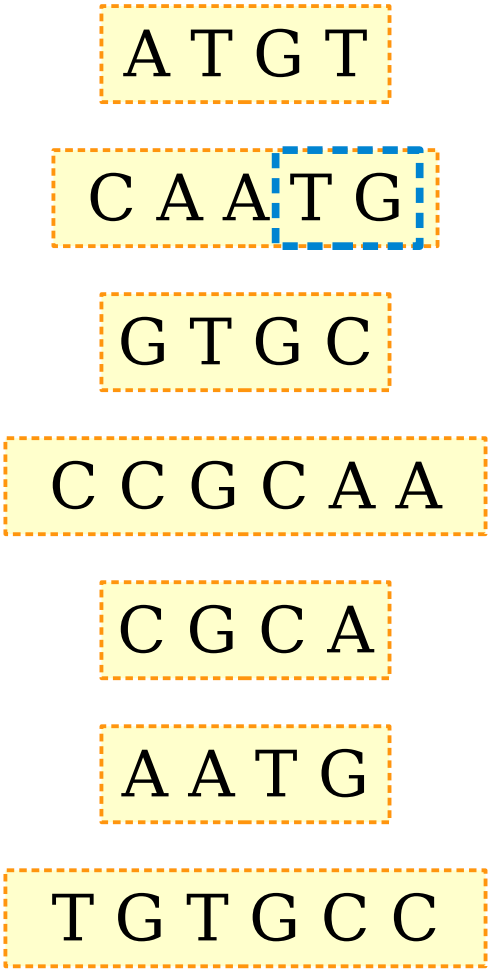
Plasmid Reassembly



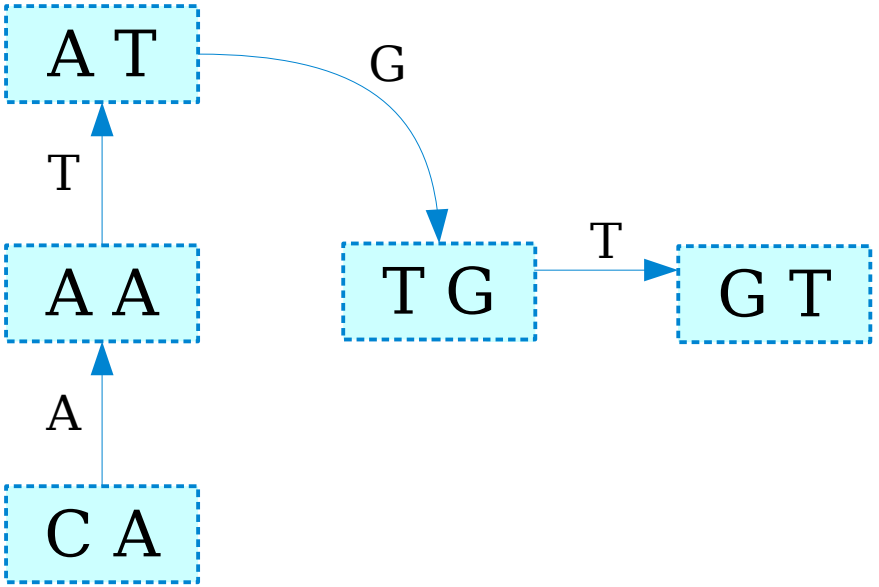


Plasmid Reassembly





Plasmid Reassembly



A T G T

C A A T G

G T G C

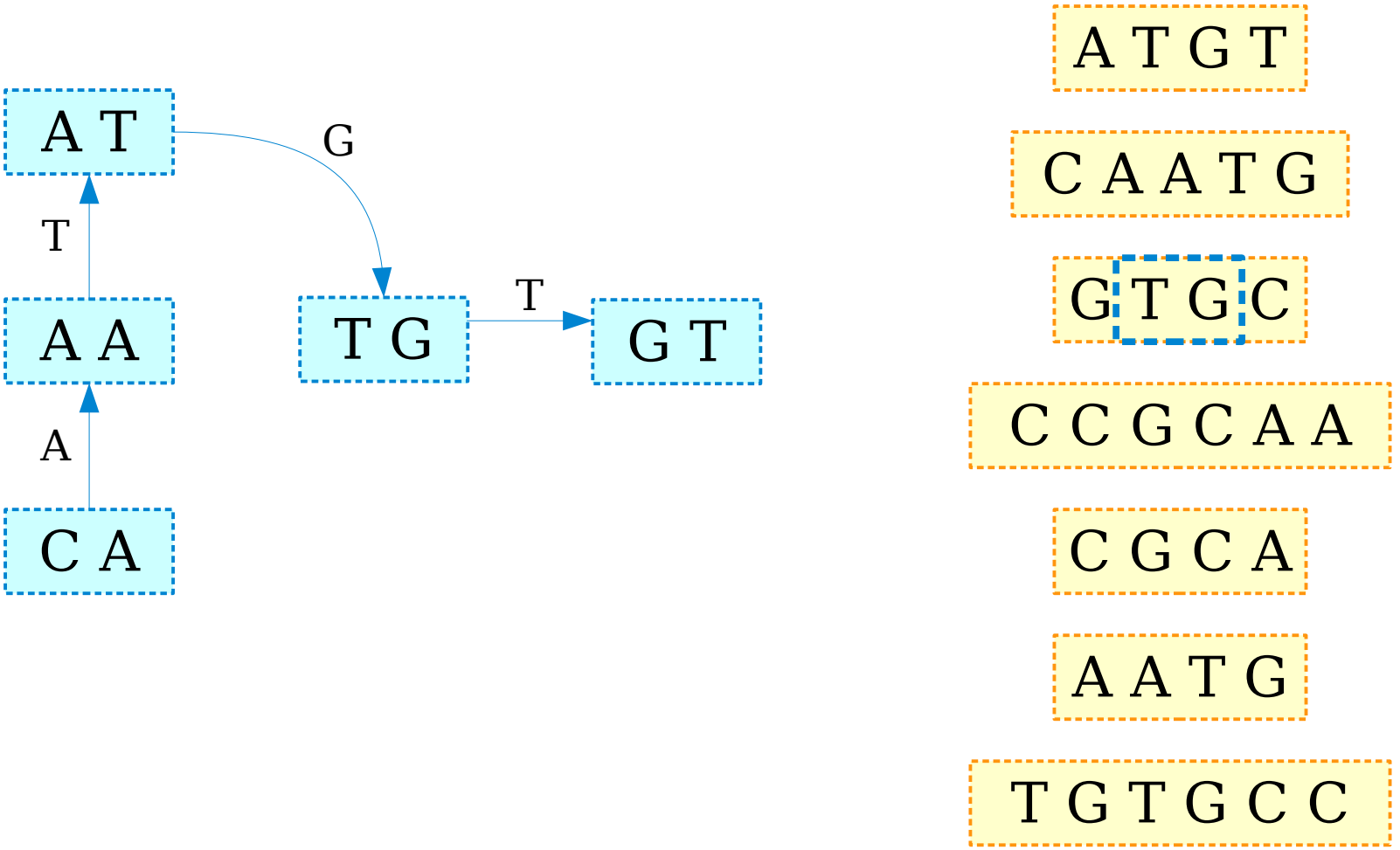
C C G C A A

C G C A

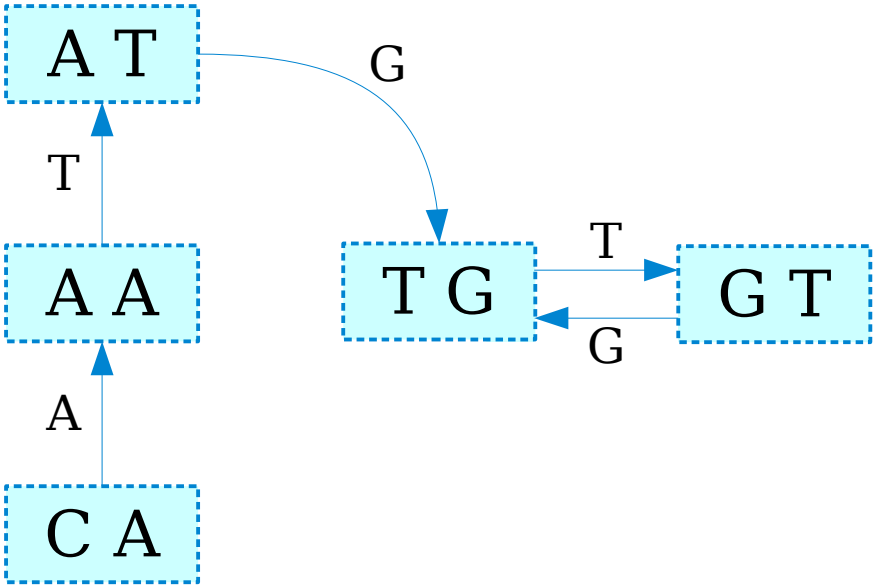
A A T G

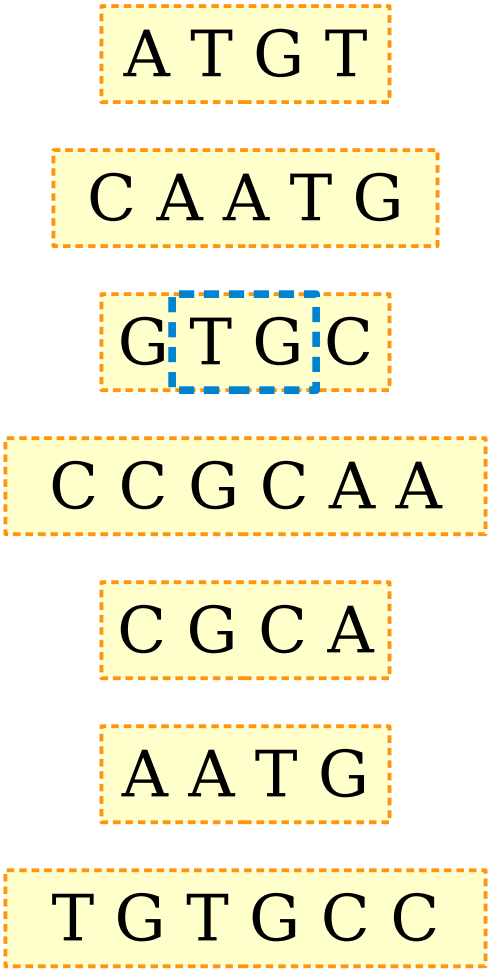
T G T G C C

Plasmid Reassembly

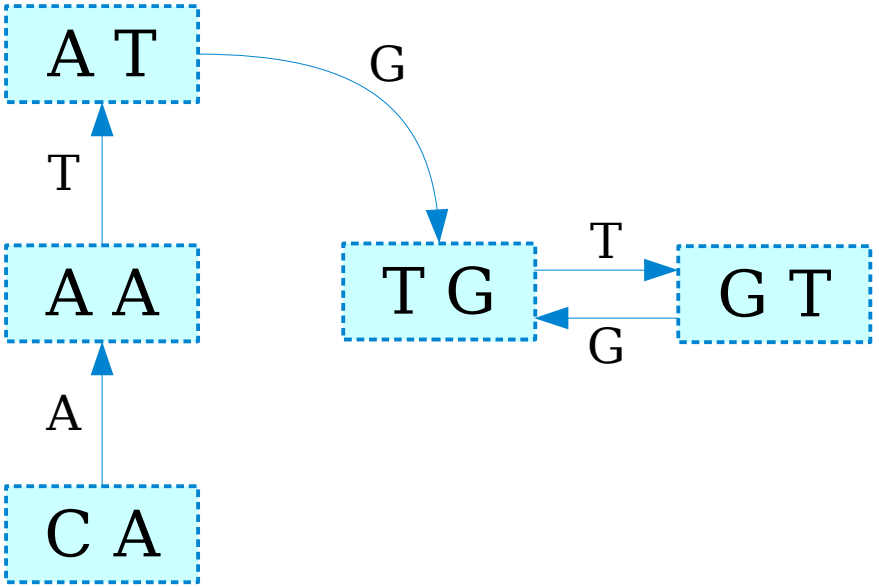


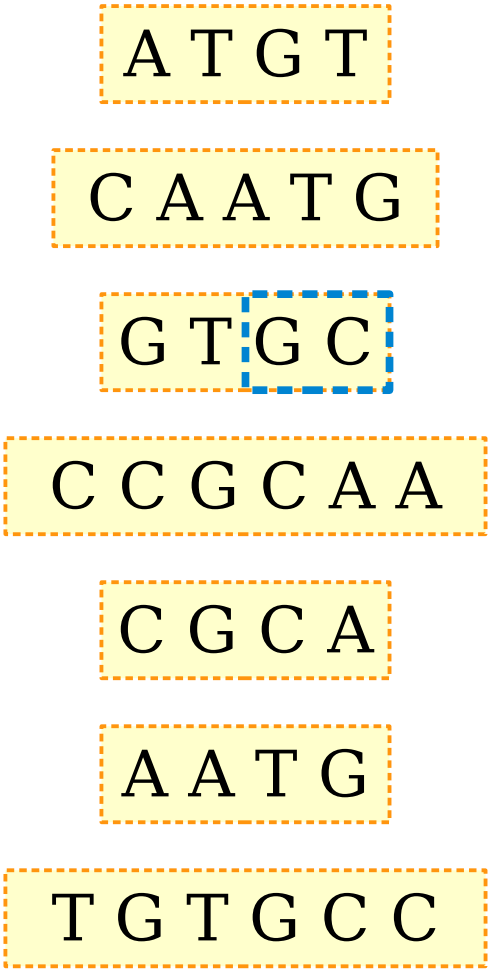
Plasmid Reassembly



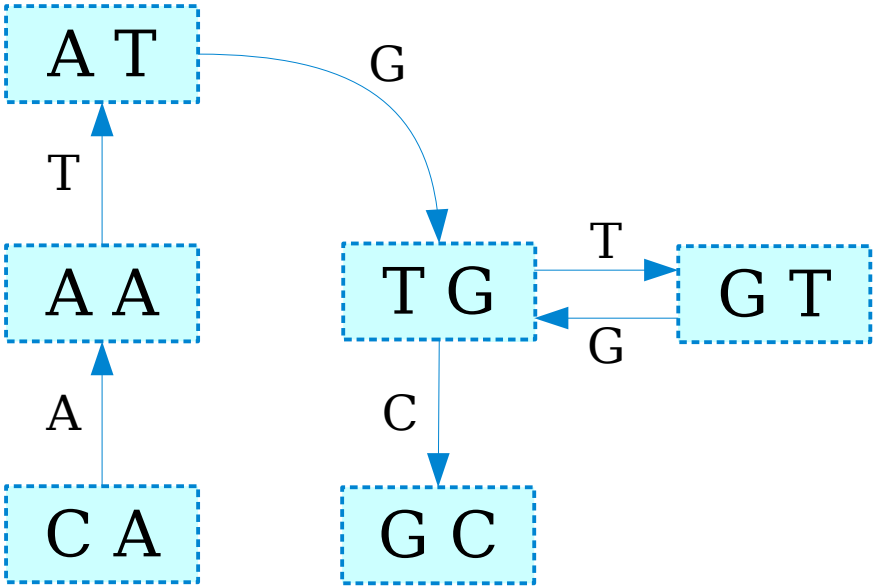


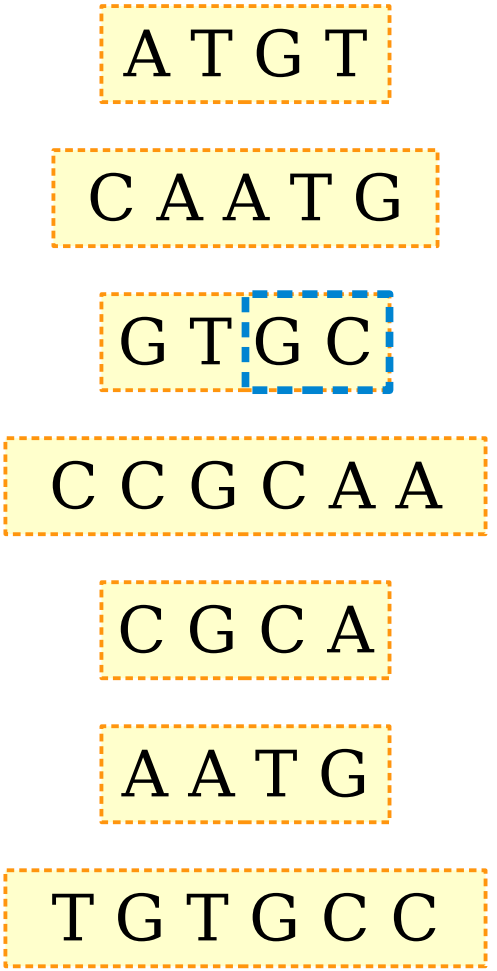
Plasmid Reassembly



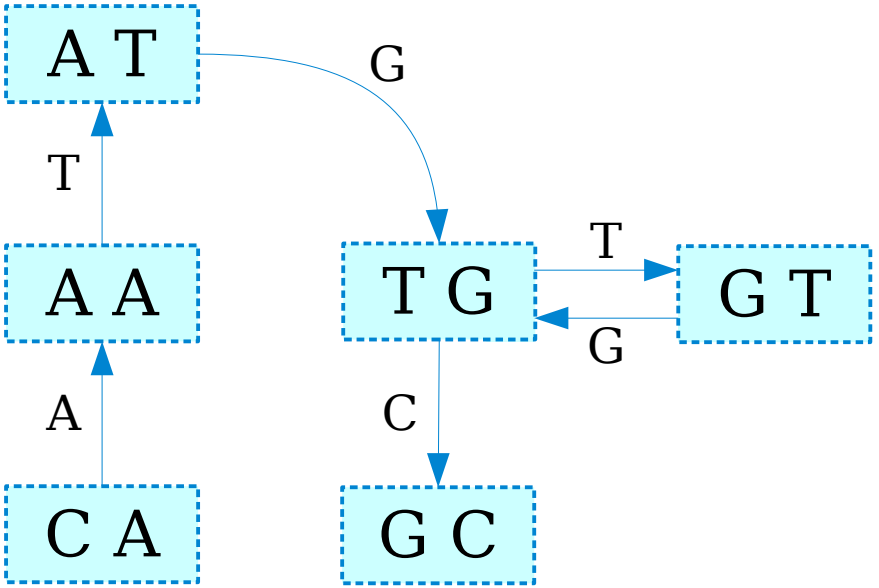


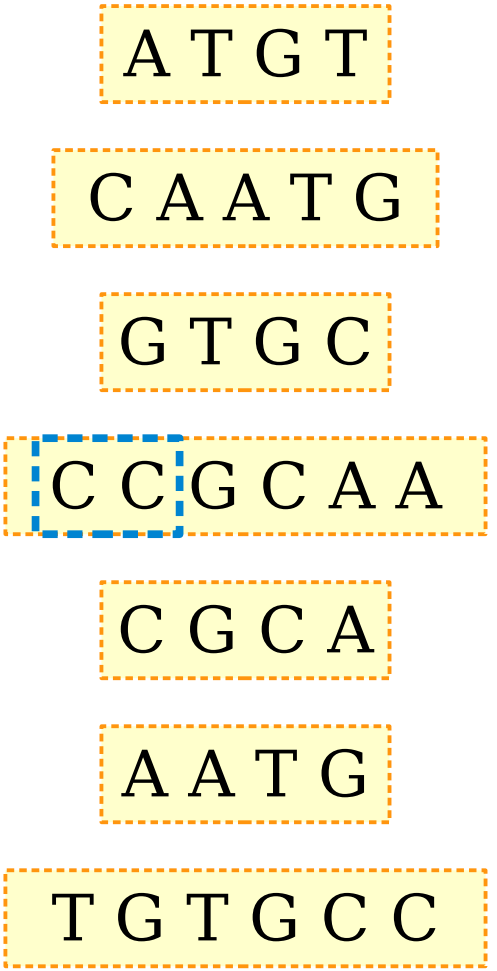
Plasmid Reassembly



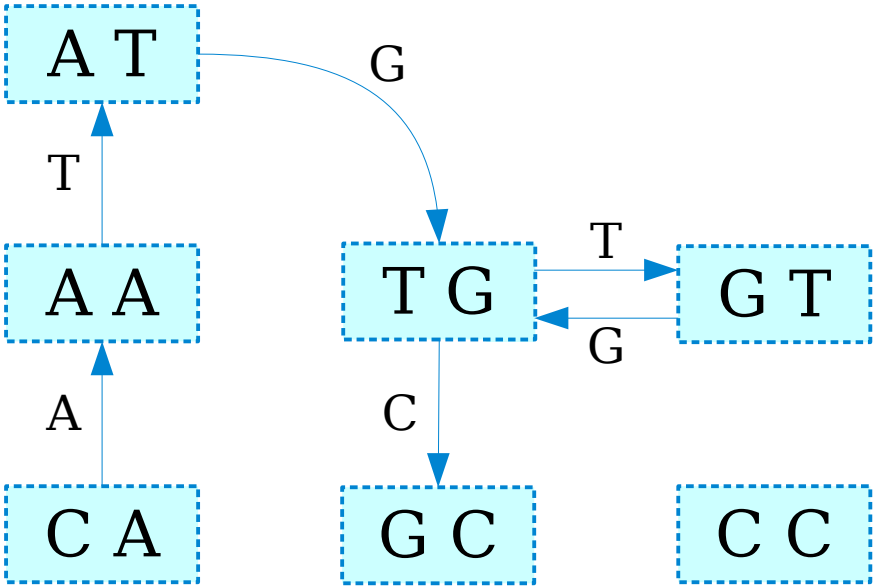


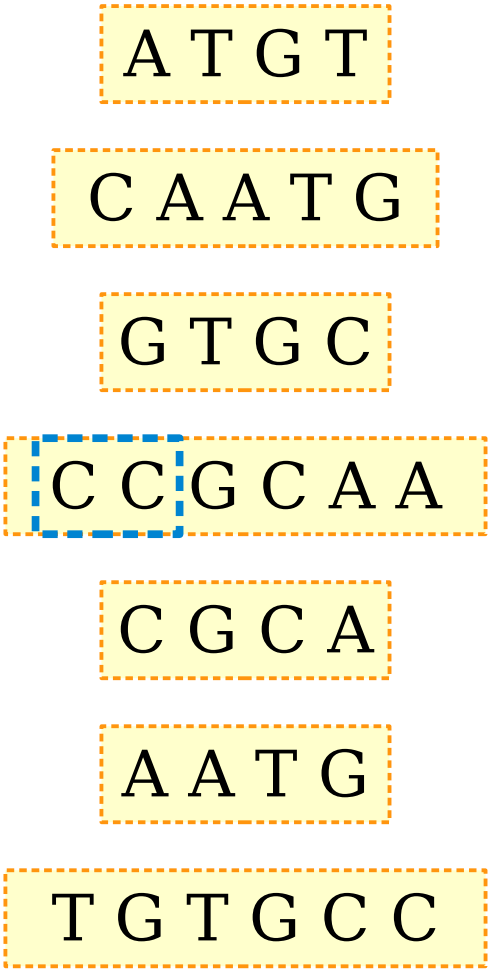
Plasmid Reassembly



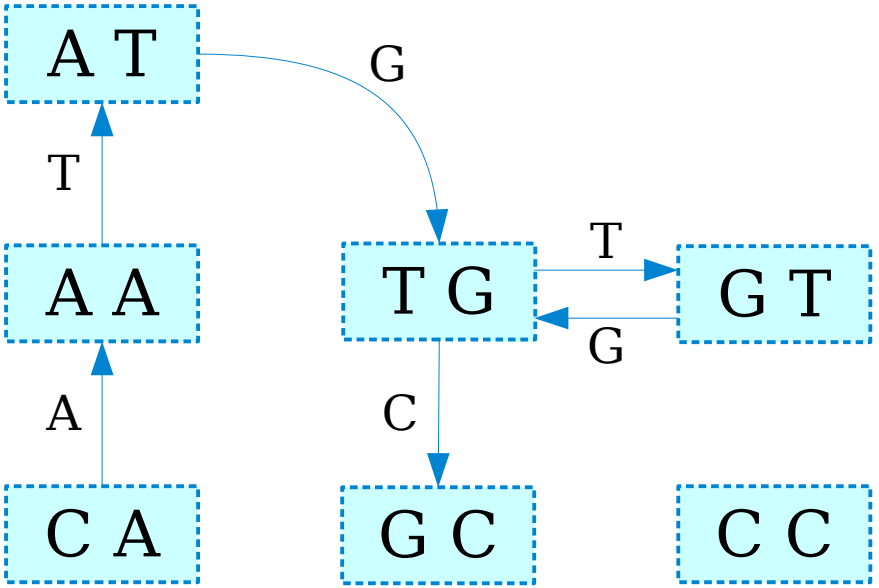


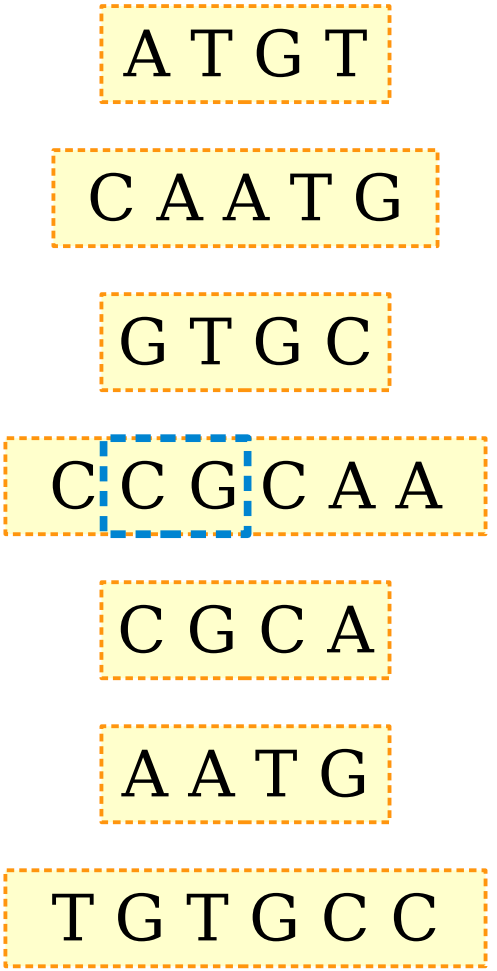
Plasmid Reassembly



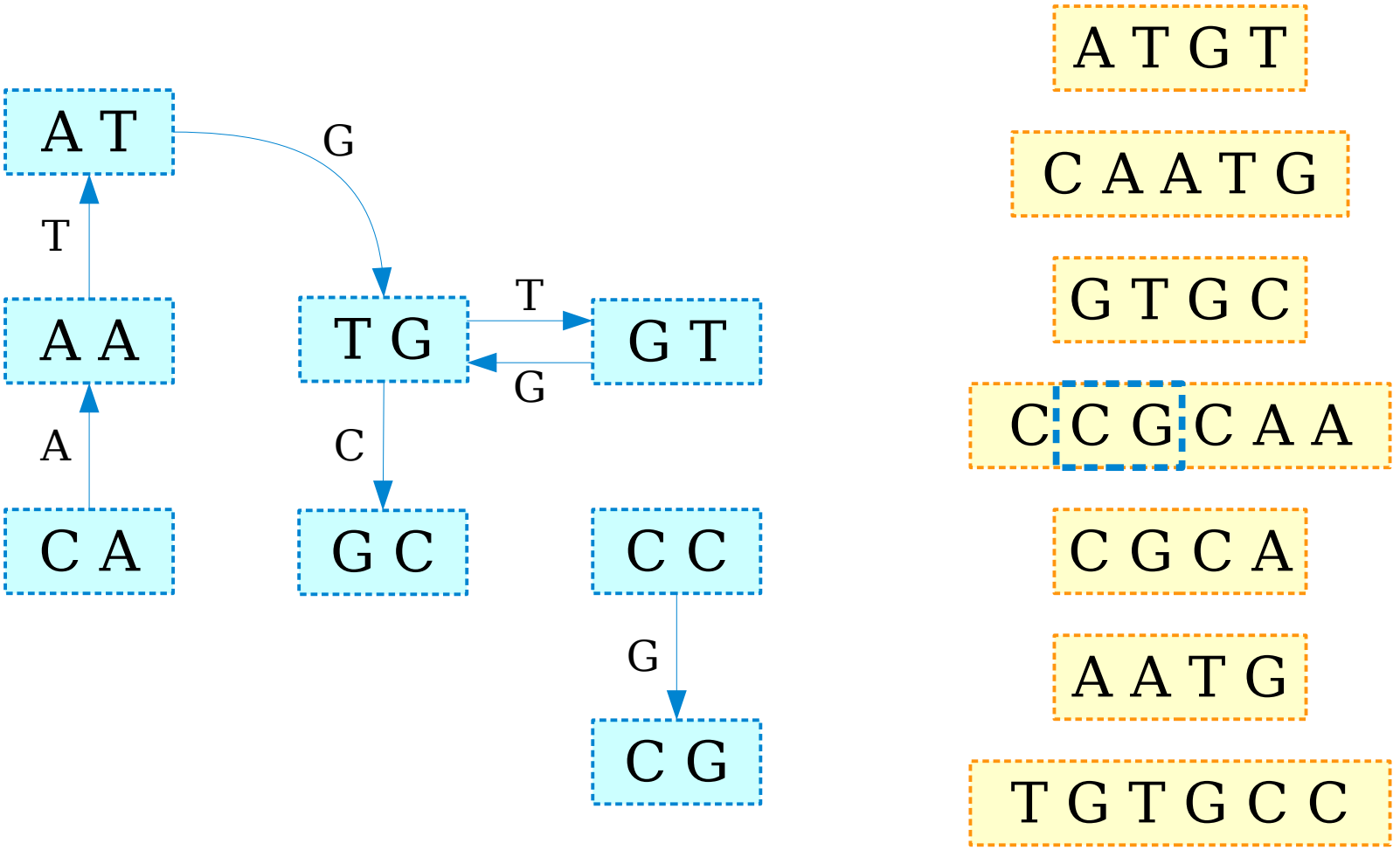


Plasmid Reassembly

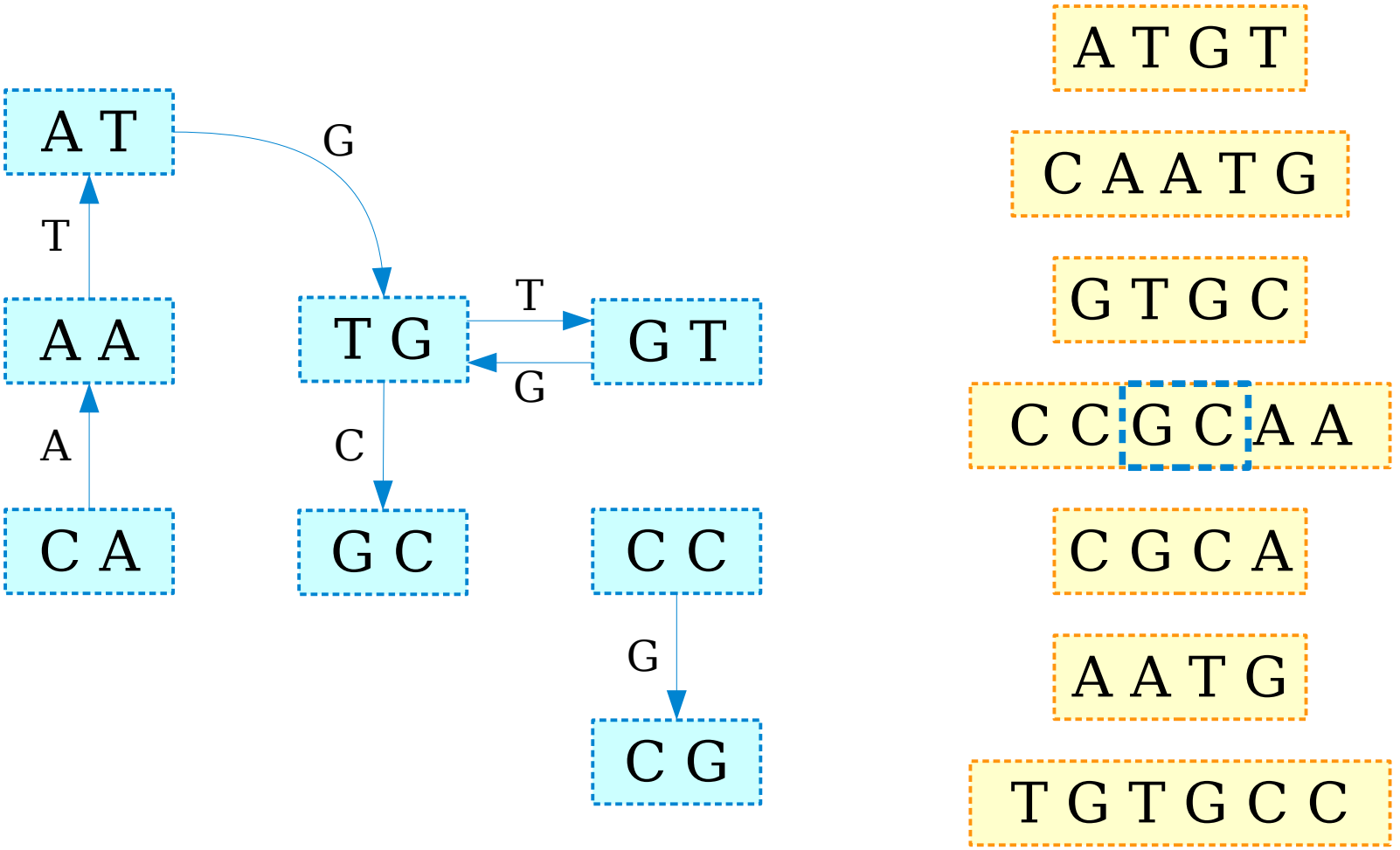




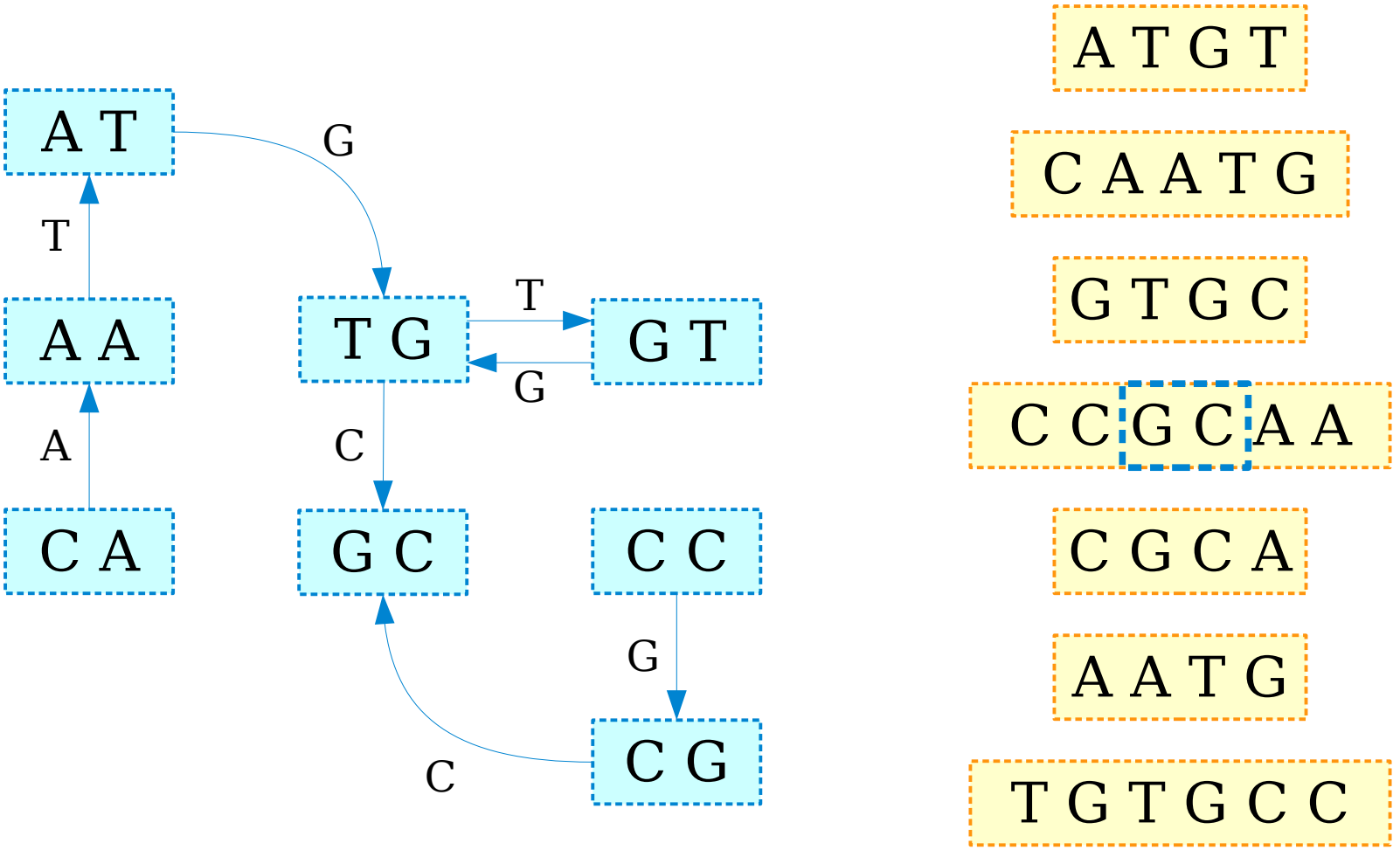
Plasmid Reassembly



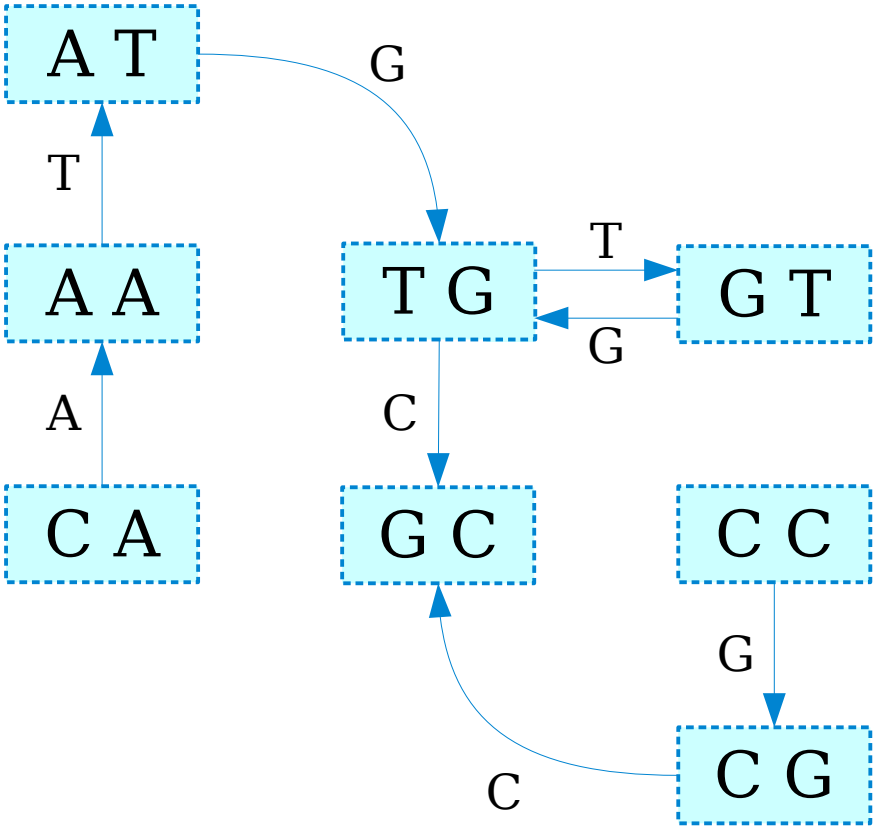
Plasmid Reassembly

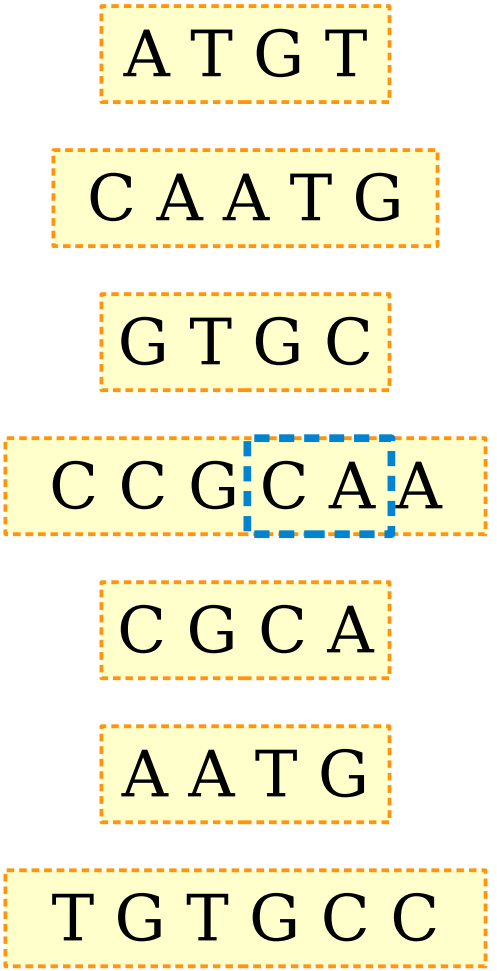


Plasmid Reassembly

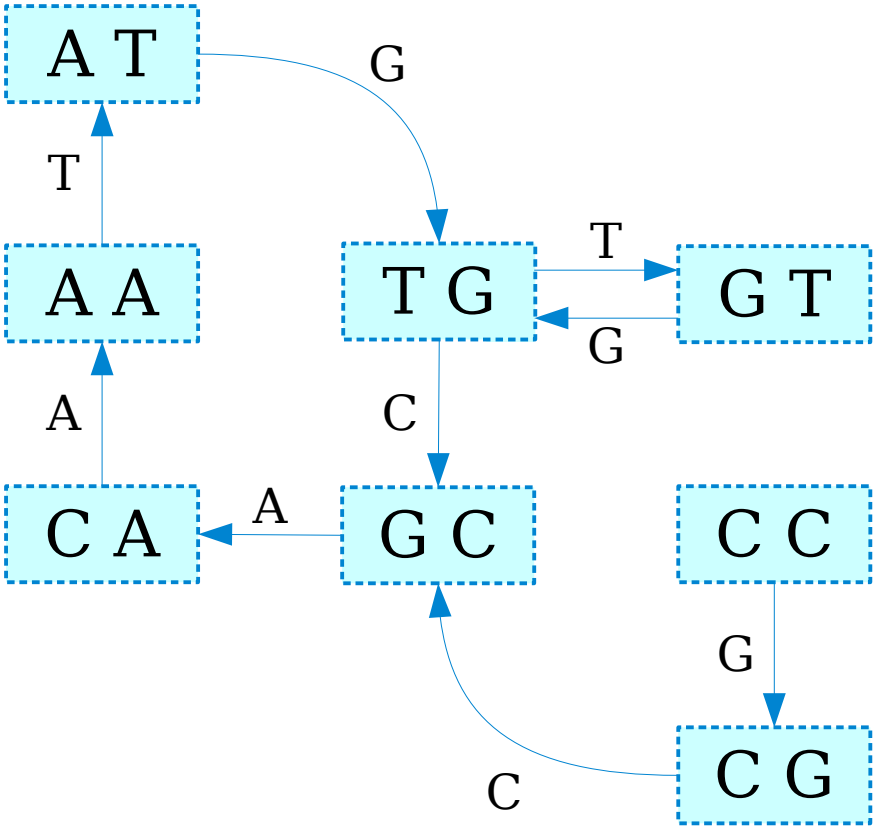


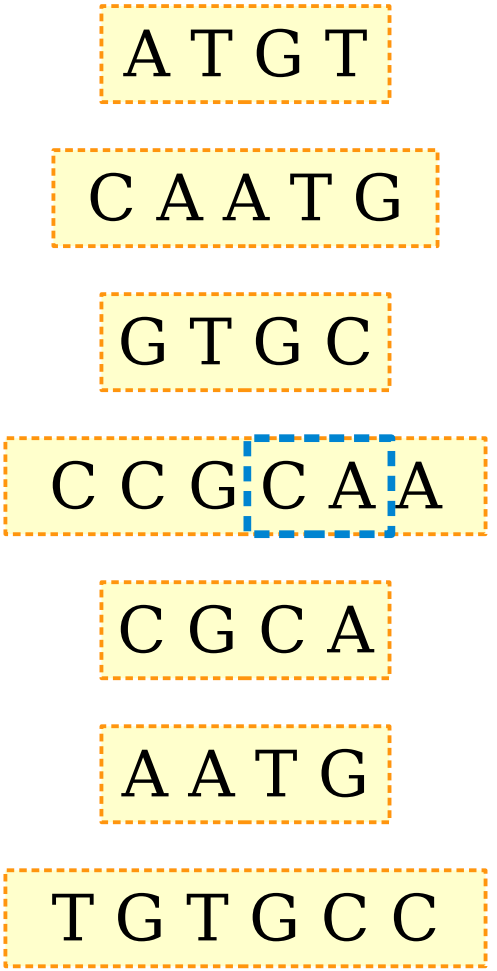
Plasmid Reassembly



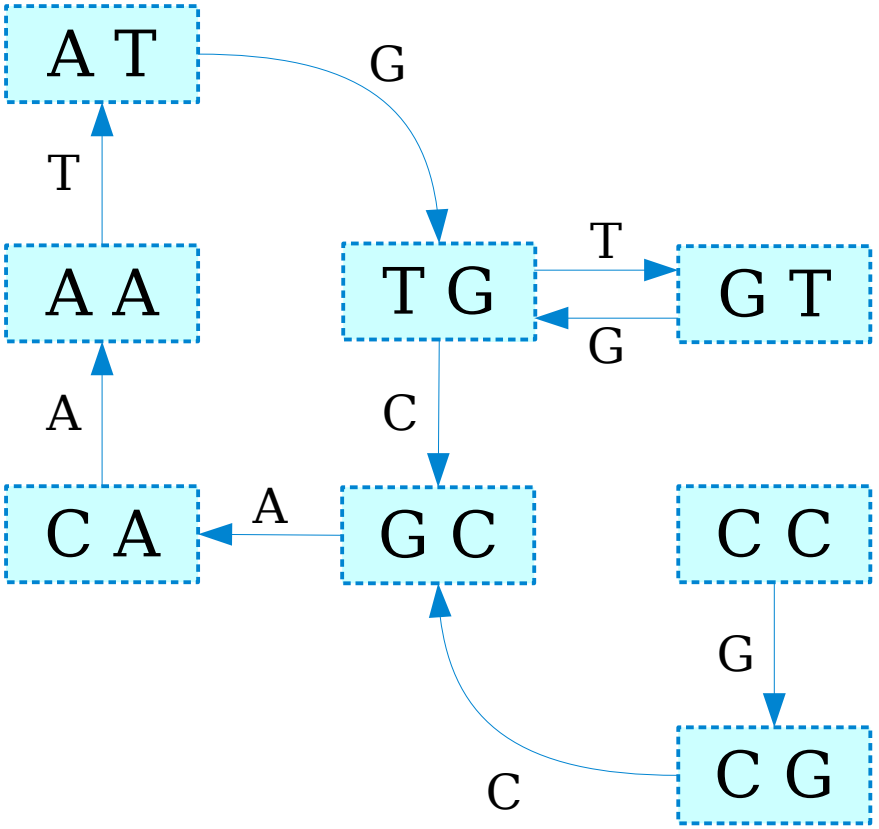


Plasmid Reassembly





Plasmid Reassembly



A T G T

C A A T G

G T G C

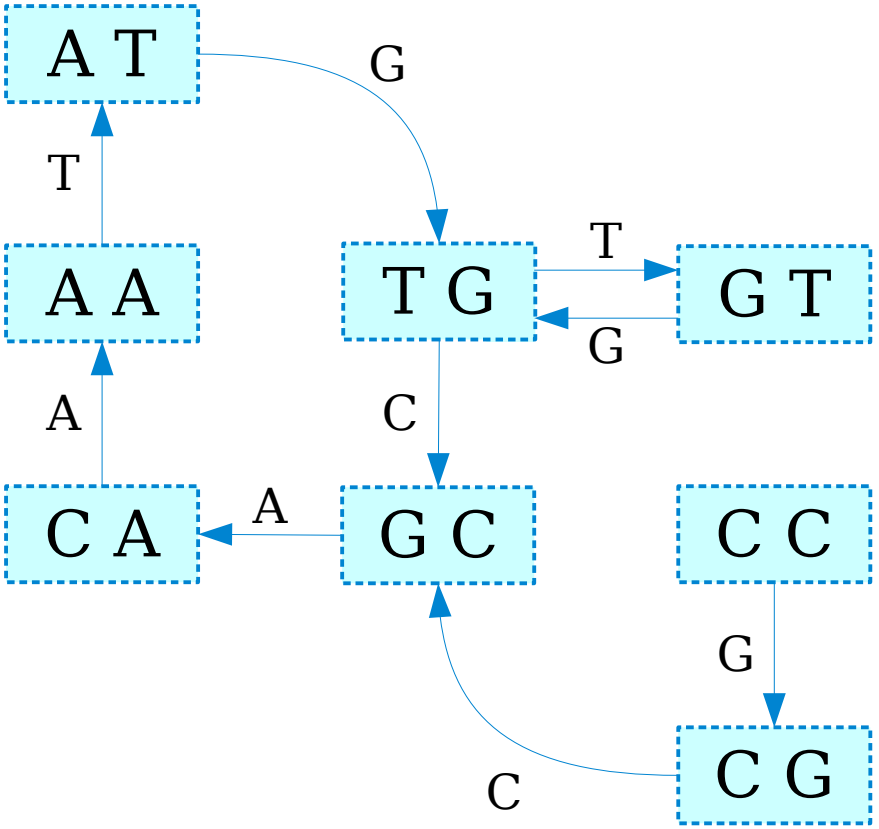
C C G C A A

C G C A

A A T G

T G T G C C

Plasmid Reassembly



A T G T

C A A T G

G T G C

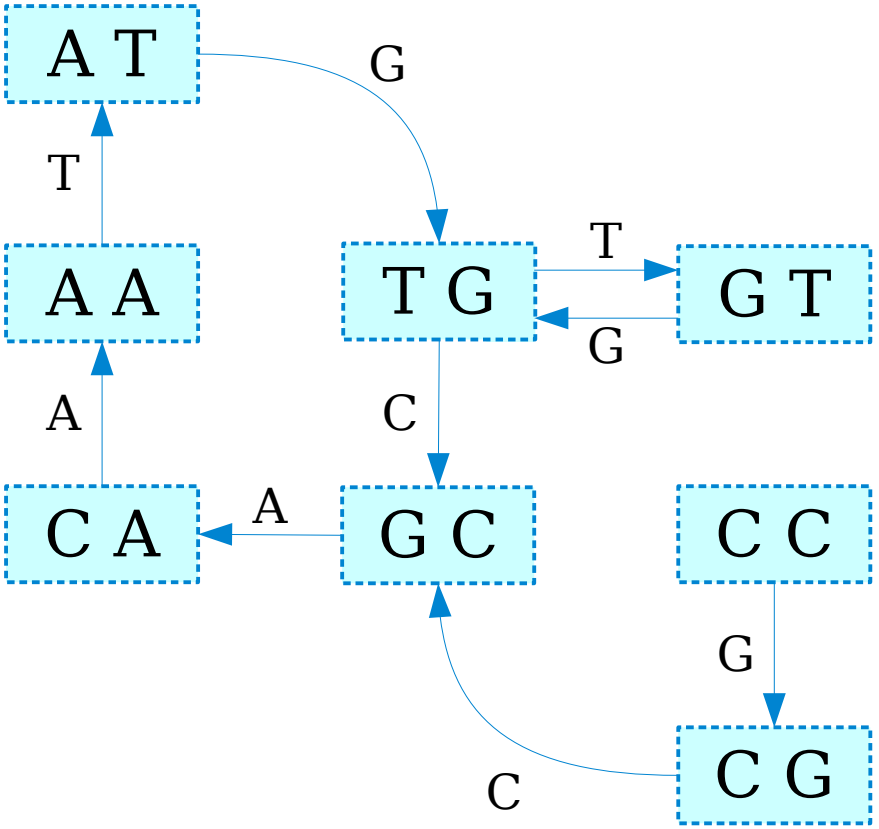
C C G C A A

C G C A

A A T G

T G T G C C

Plasmid Reassembly



A T G T

C A A T G

G T G C

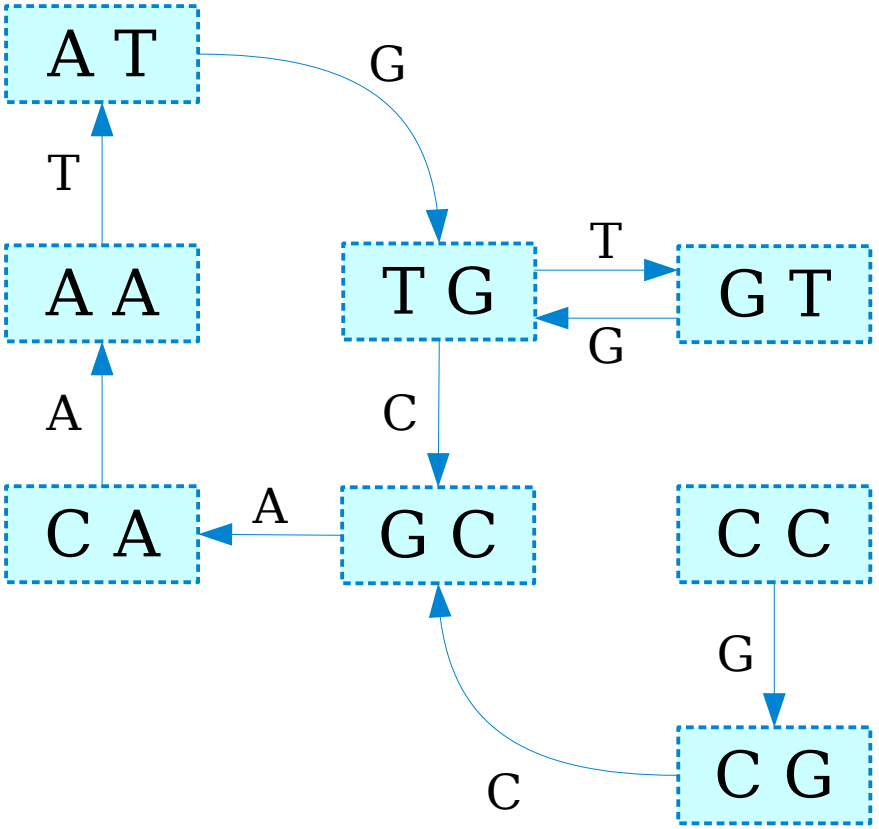
C C G C A A

C G C A

A A T G

T G T G C C

Plasmid Reassembly



A T G T

C A A T G

G T G C

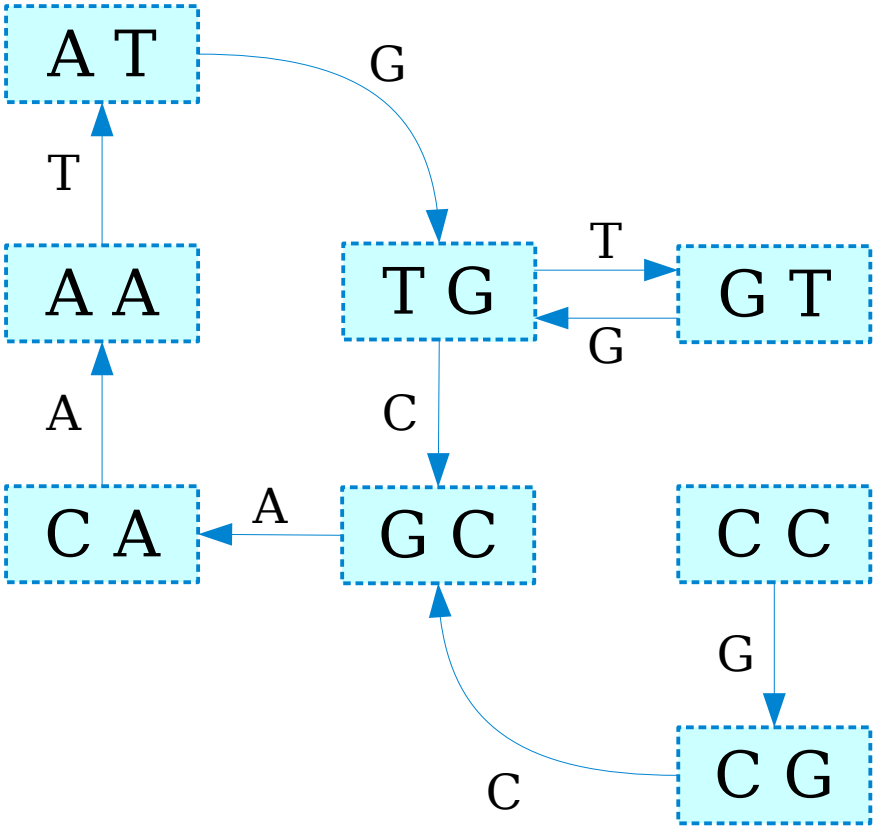
C C G C A A

C G C A

A A T G

T G T G C C

Plasmid Reassembly



A T G T

C A A T G

G T G C

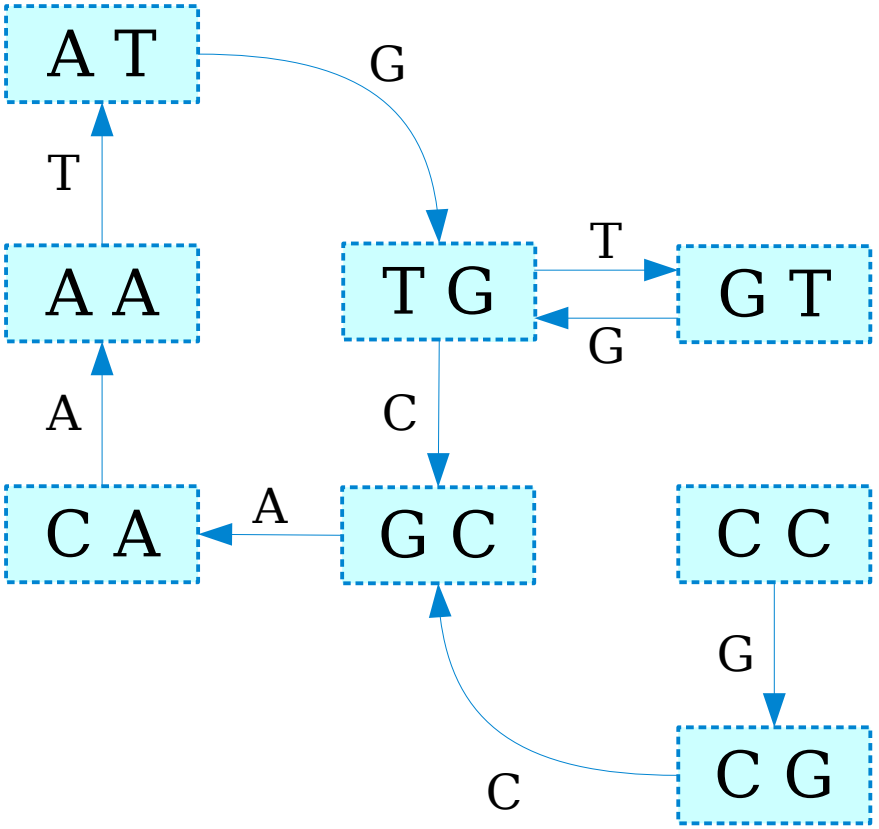
C C G C A A

C G C A

A A T G

T G T G C C

Plasmid Reassembly



A T G T

C A A T G

G T G C

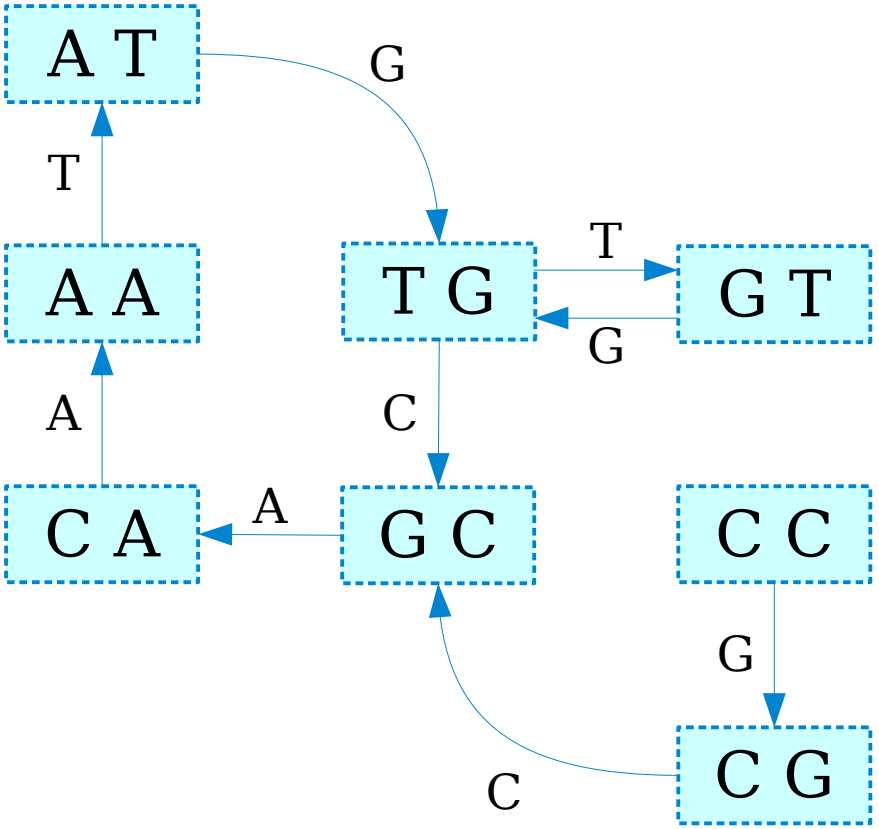
C C G C A A

C G C A

A A T G

T G T G C C

Plasmid Reassembly



A T G T

C A A T G

G T G C

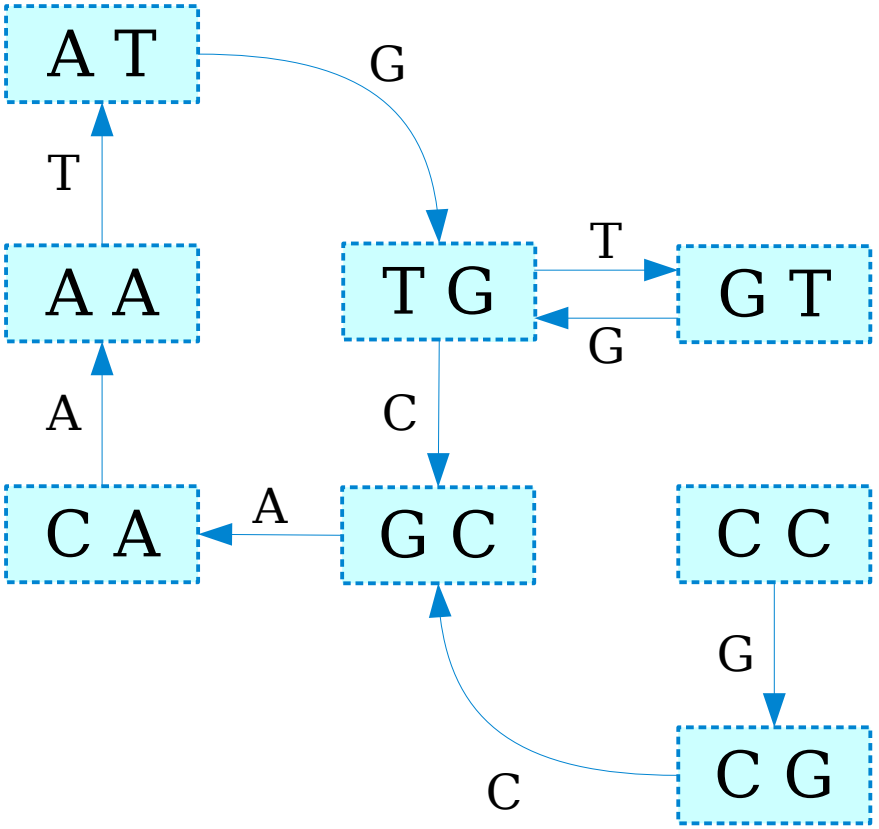
C C G C A A

C G C A

A A T G

T G T G C C

Plasmid Reassembly



A T G T

C A A T G

G T G C

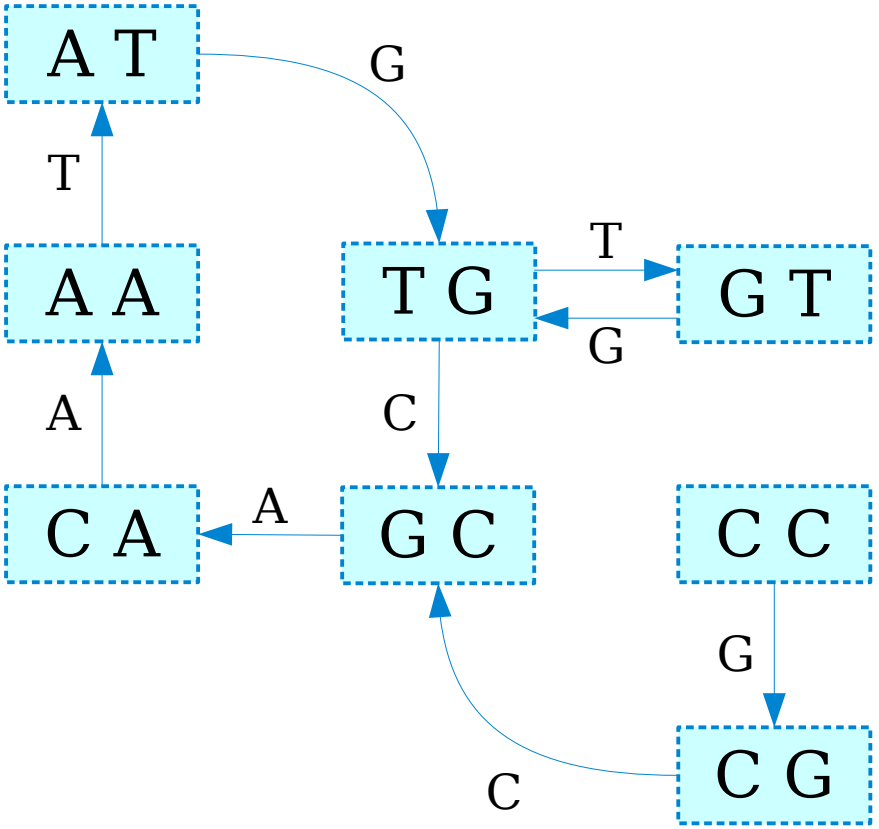
C C G C A A

C G C A

A A T G

T G T G C C

Plasmid Reassembly



A T G T

C A A T G

G T G C

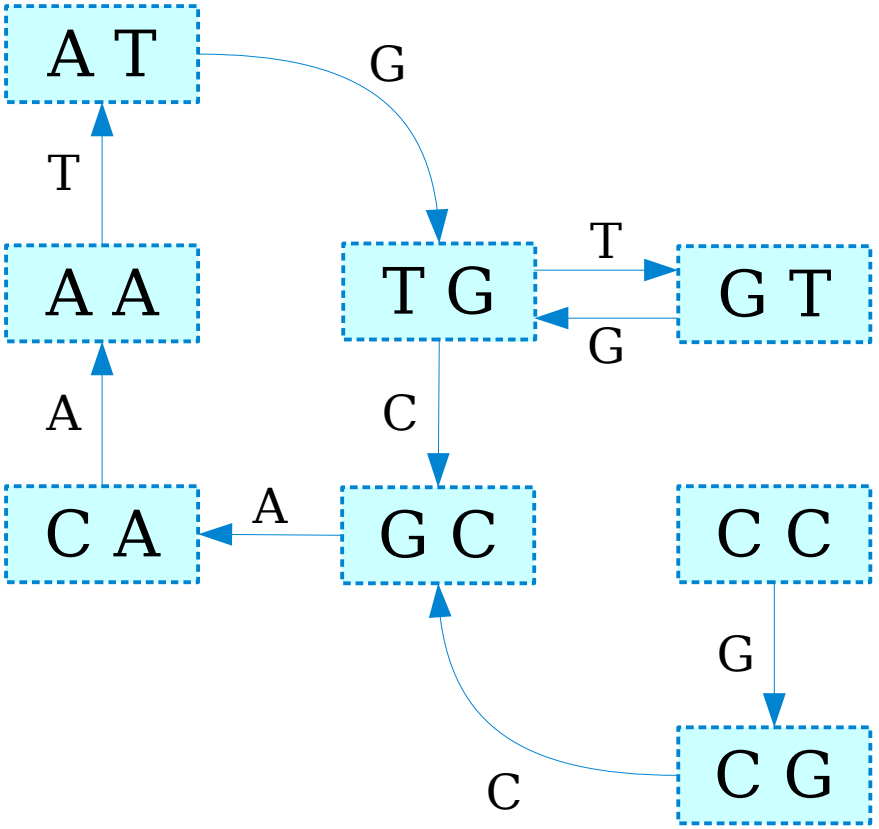
C C G C A A

C G C A

A A T G

T G T G C C

Plasmid Reassembly



A T G T

C A A T G

G T G C

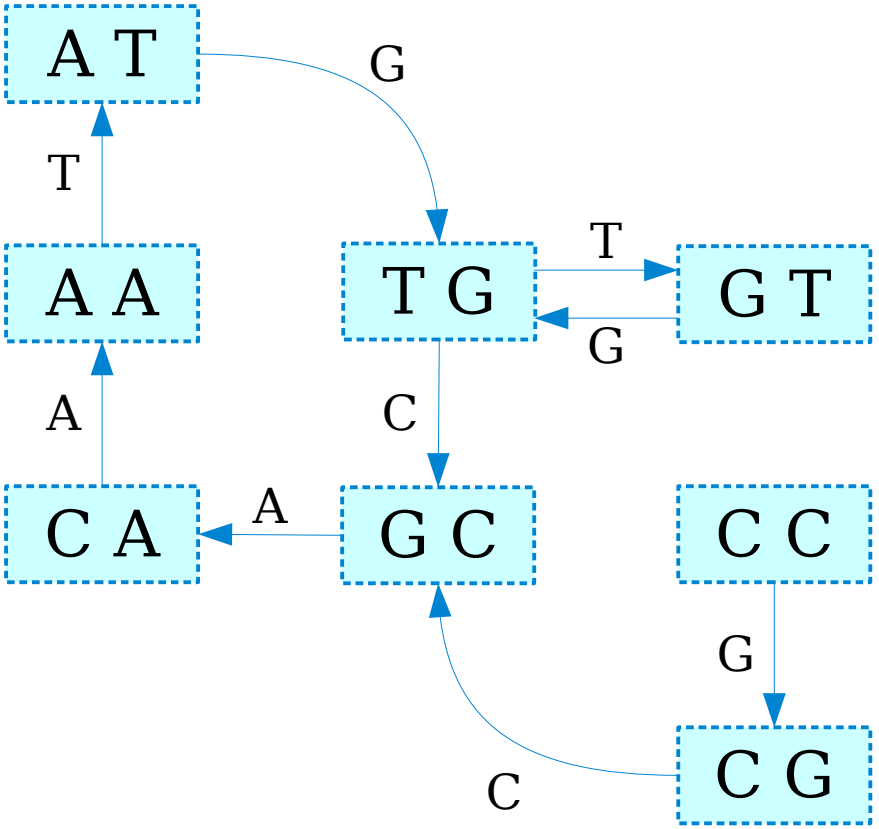
C C G C A A

C G C A

A A T G

T G T G C C

Plasmid Reassembly



A T G T

C A A T G

G T G C

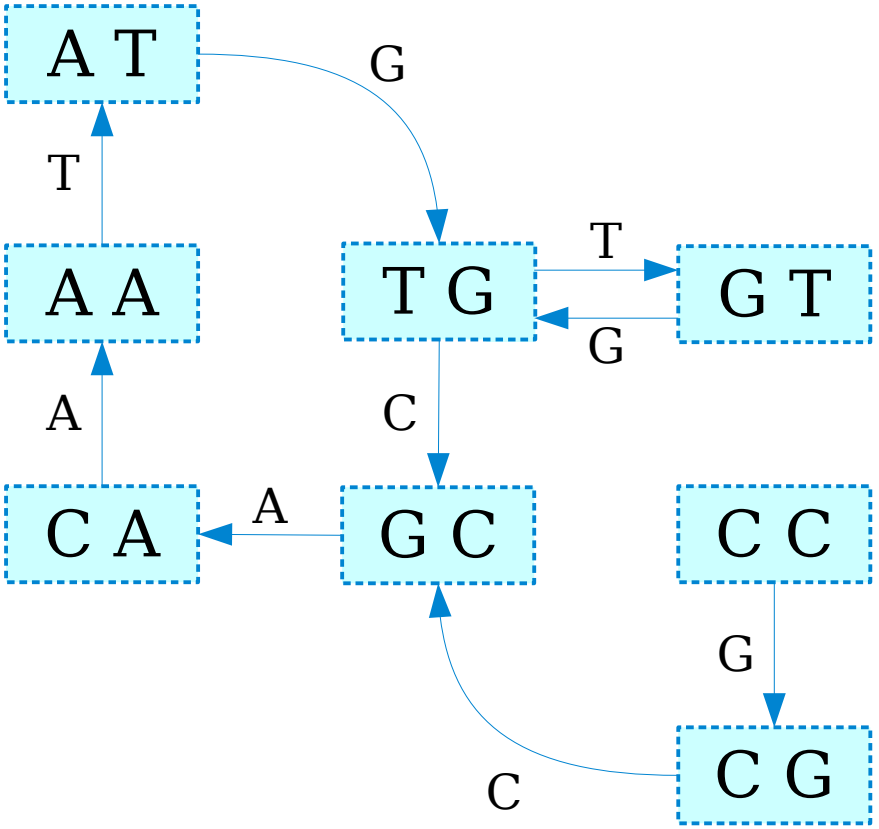
C C G C A A

C G C A

A A T G

T G T G C C

Plasmid Reassembly



A T G T

C A A T G

G T G C

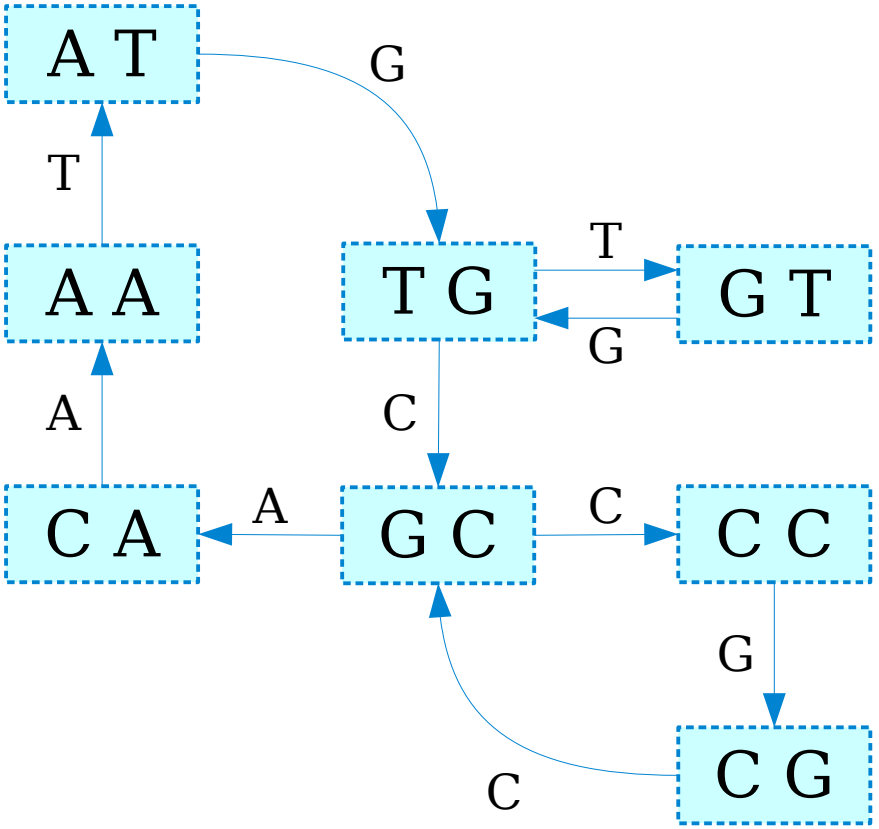
C C G C A A

C G C A

A A T G

T G T G C C

Plasmid Reassembly



A T G T

C A A T G

G T G C

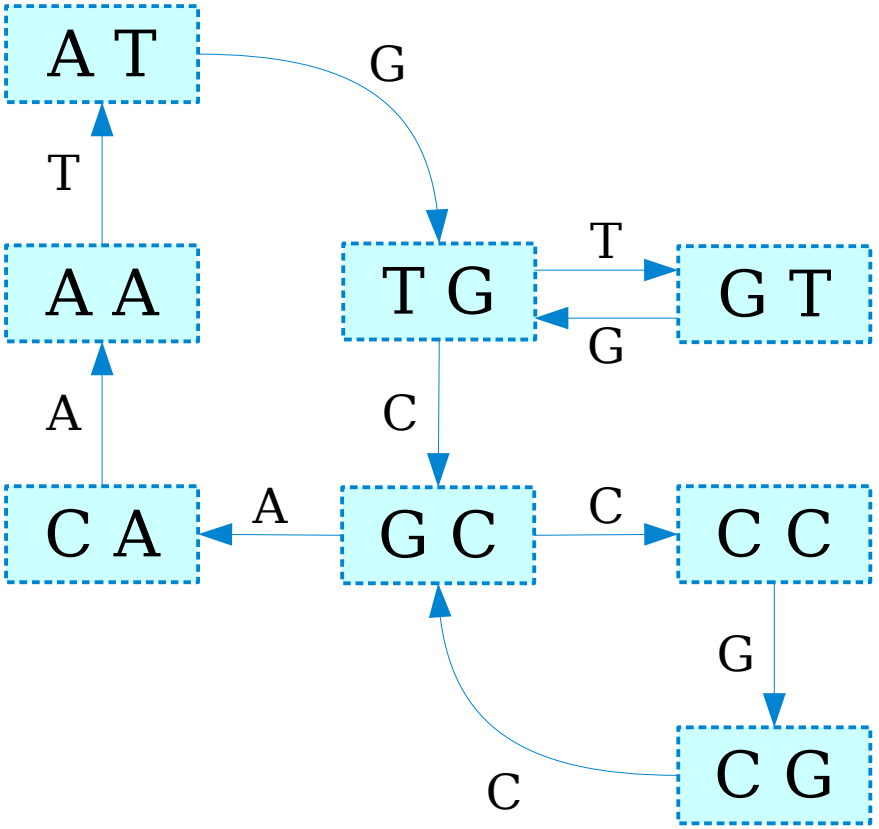
C C G C A A

C G C A

A A T G

T G T G C C

Plasmid Reassembly



A T G T

C A A T G

G T G C

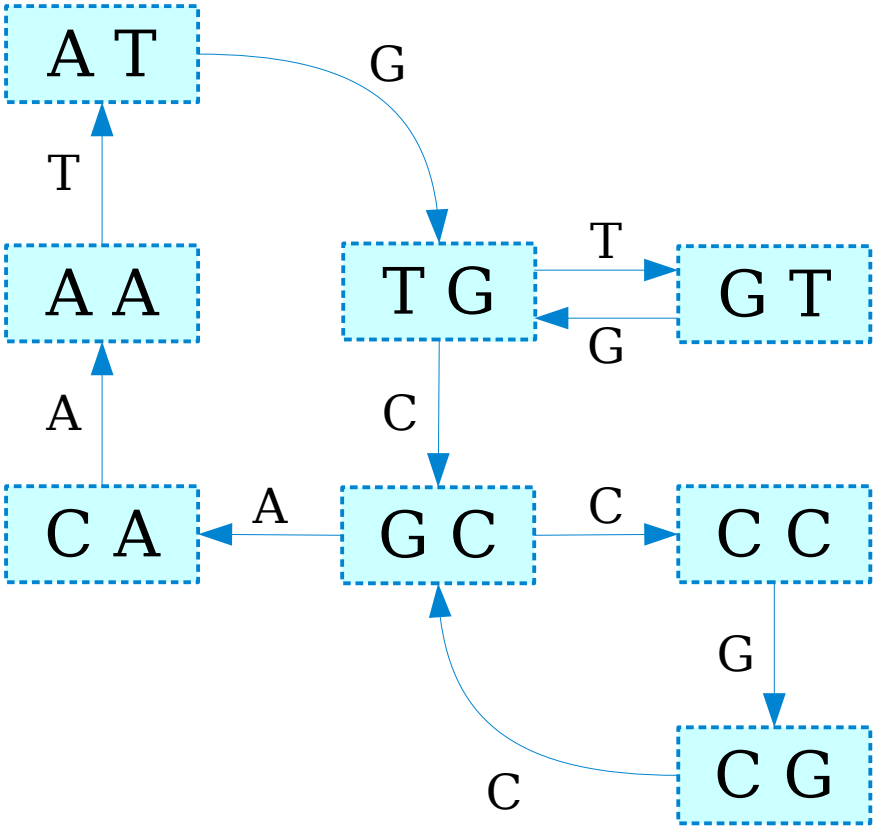
C C G C A A

C G C A

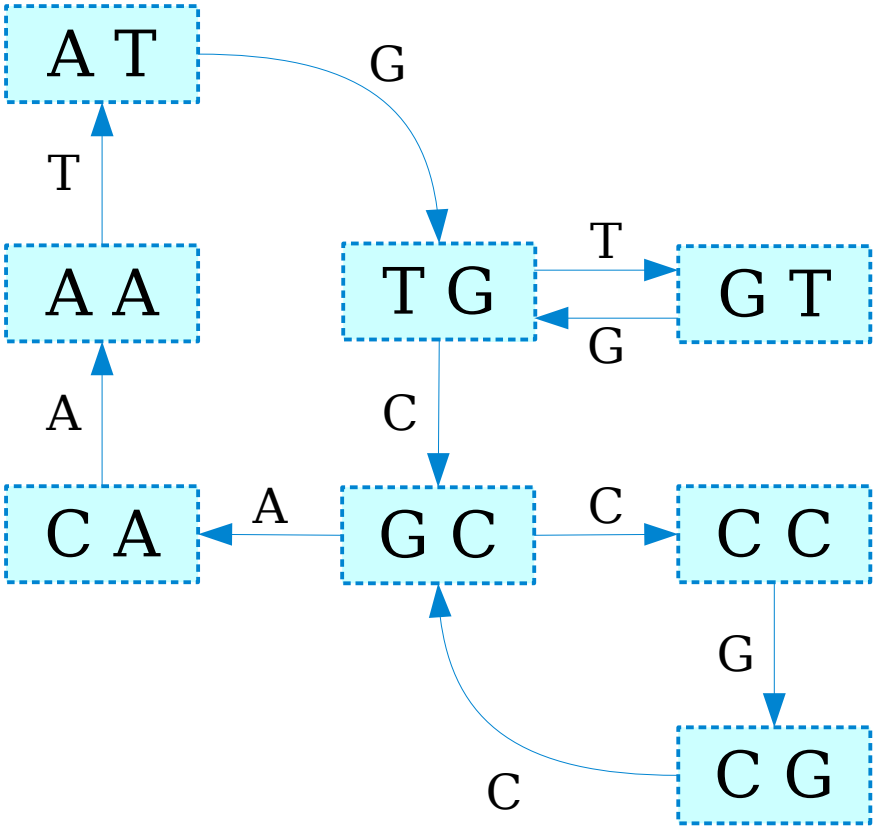
A A T G

T G T G C C

Plasmid Reassembly



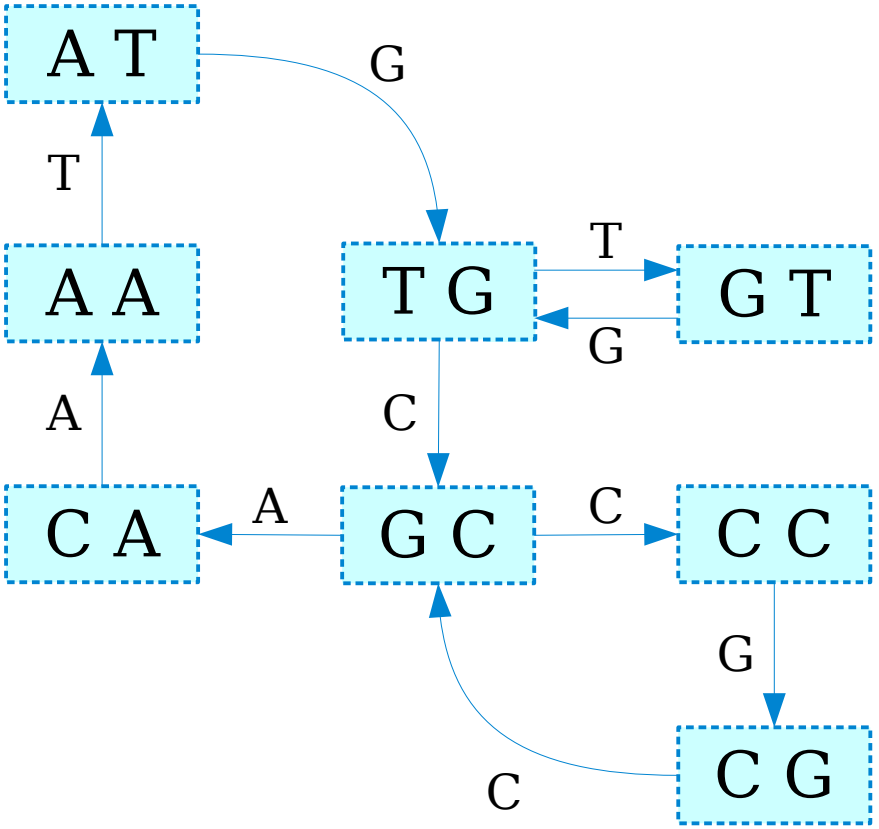
Plasmid Reassembly



Notice that every node's indegree equals its outdegree.

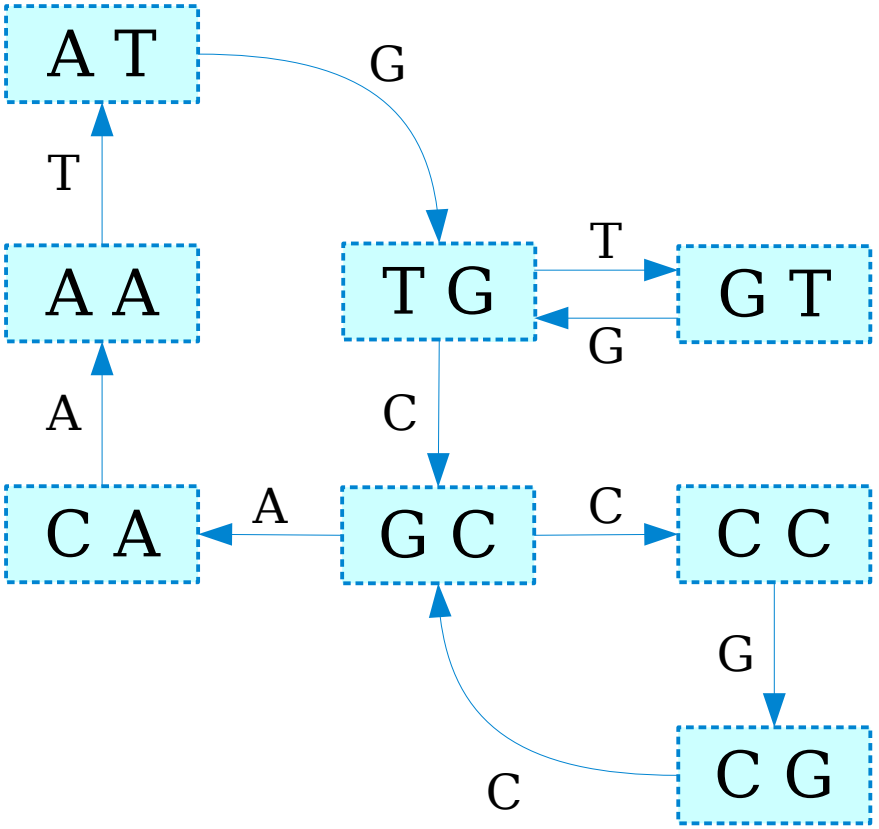
(This happens because the block size is at least as large as the largest repeated part of the plasmid.)

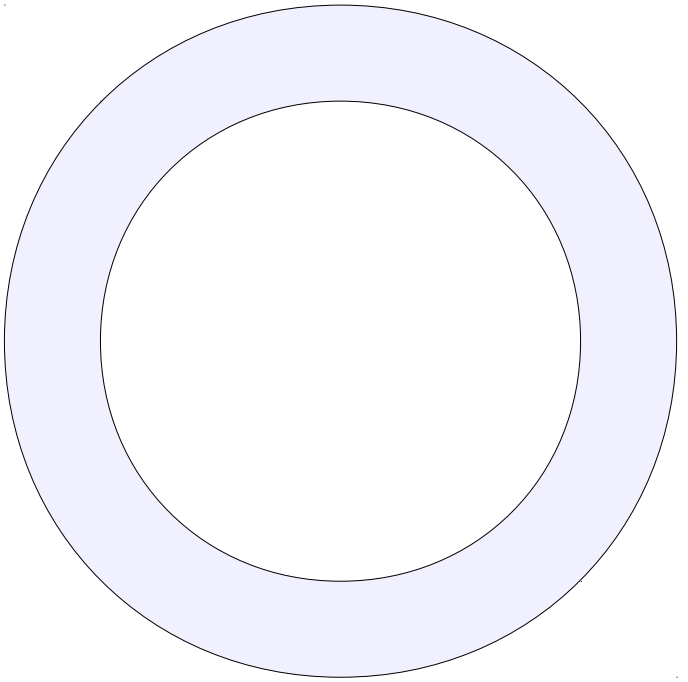
Plasmid Reassembly



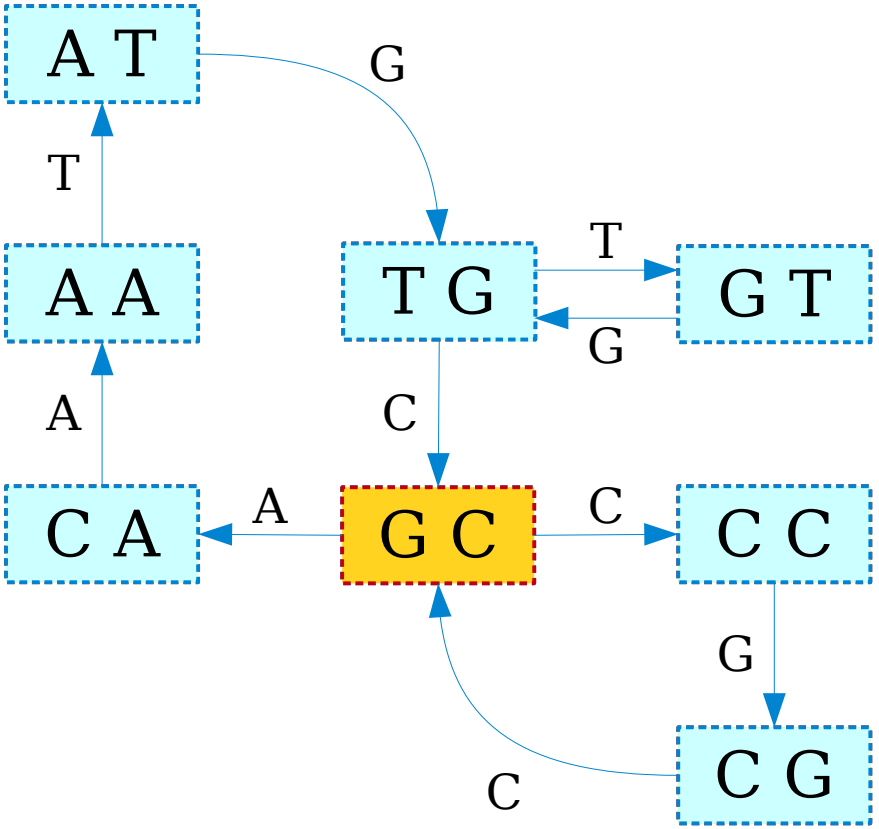
This means that we can find an Eulerian circuit in this graph!

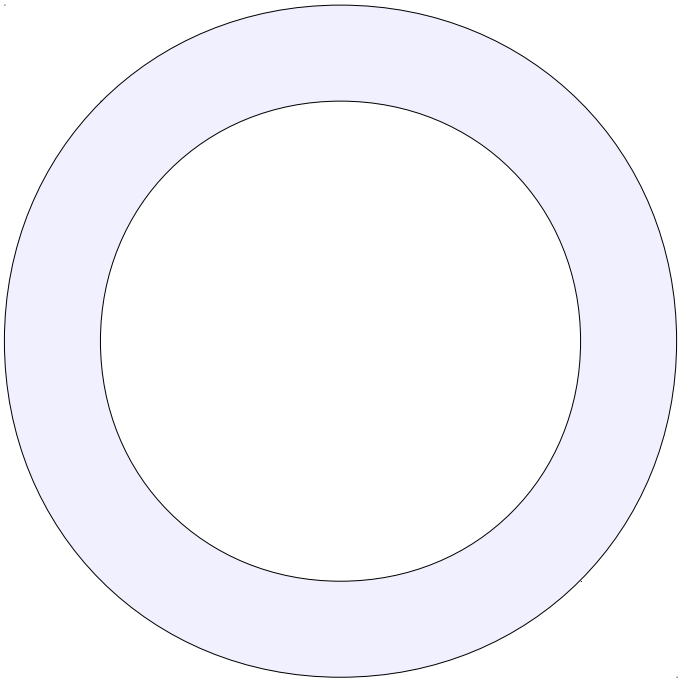
Plasmid Reassembly



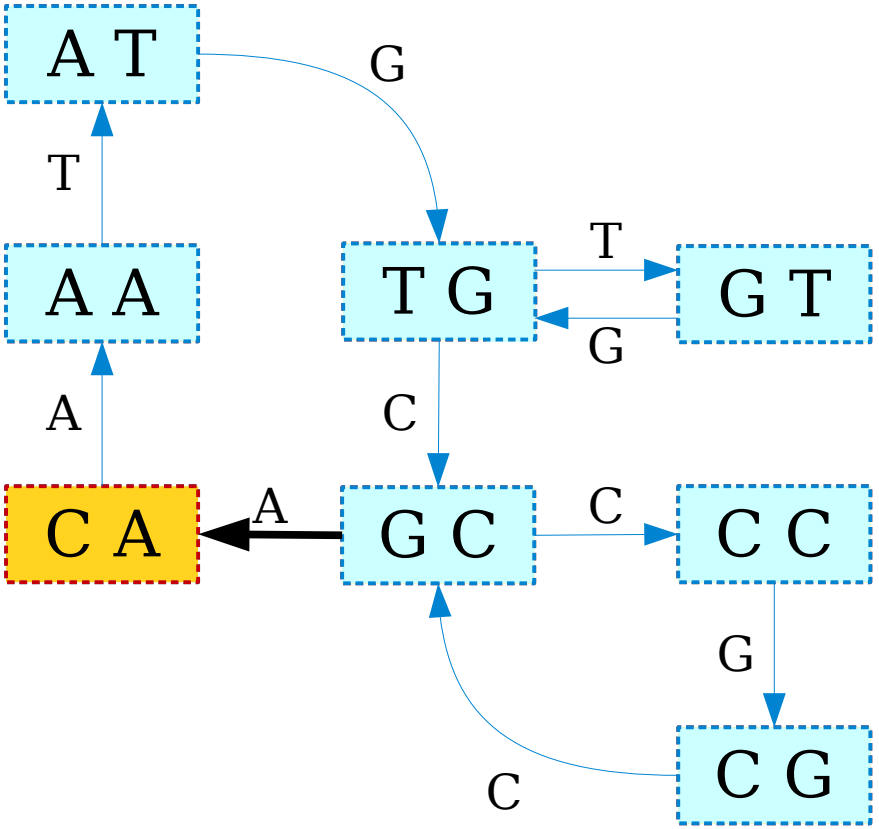


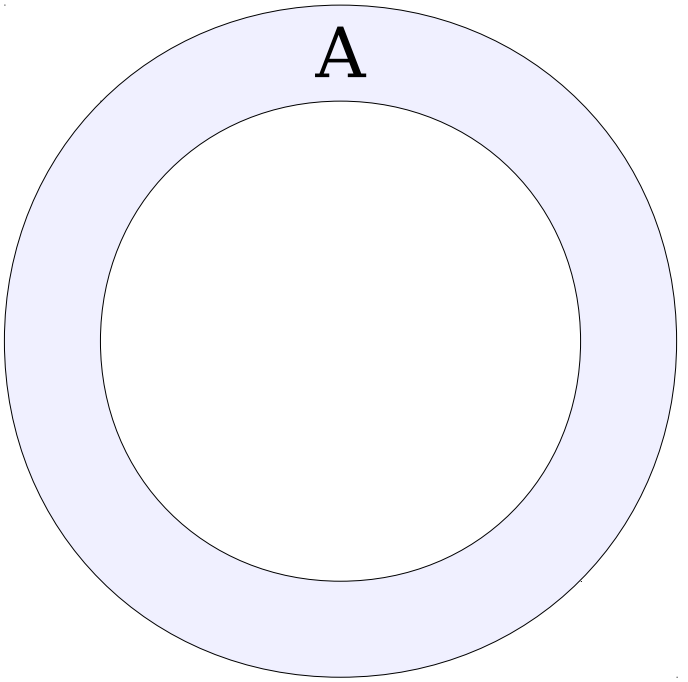
Plasmid Reassembly



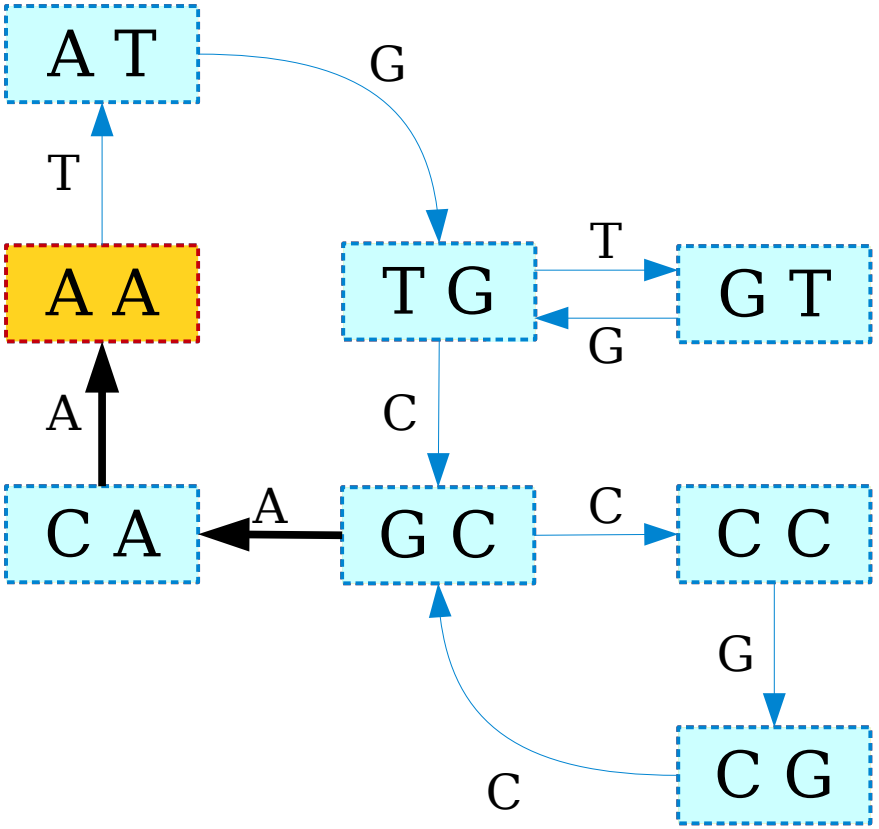


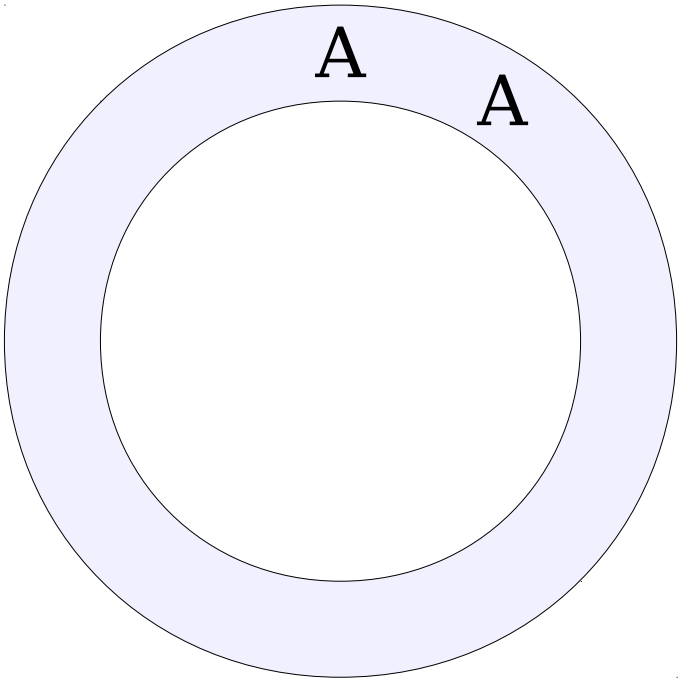
Plasmid Reassembly



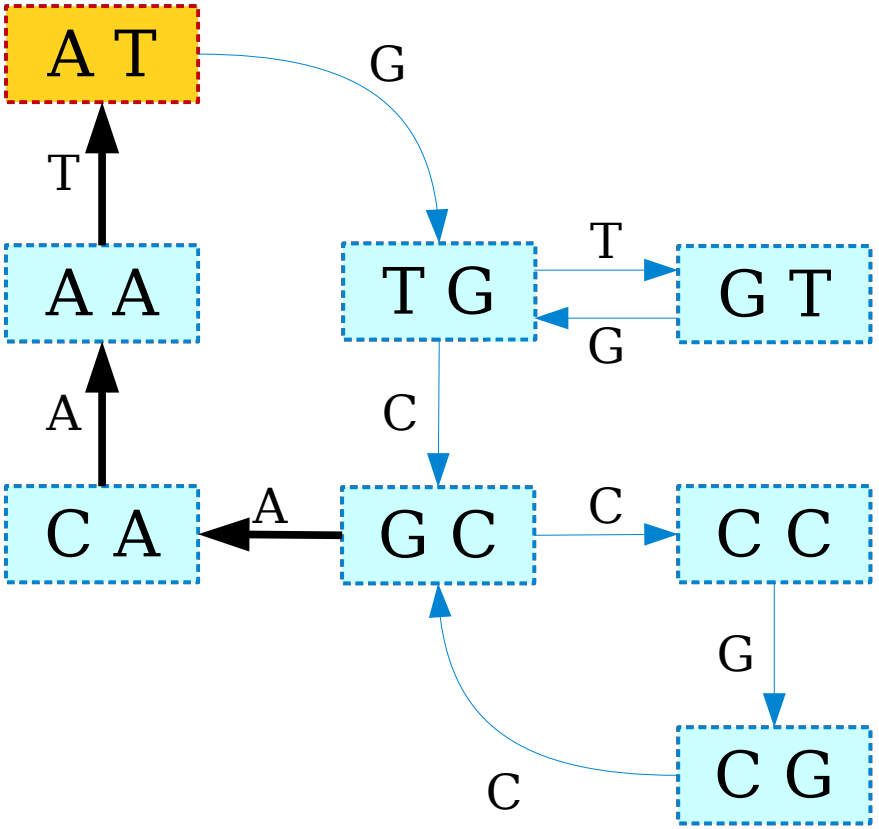


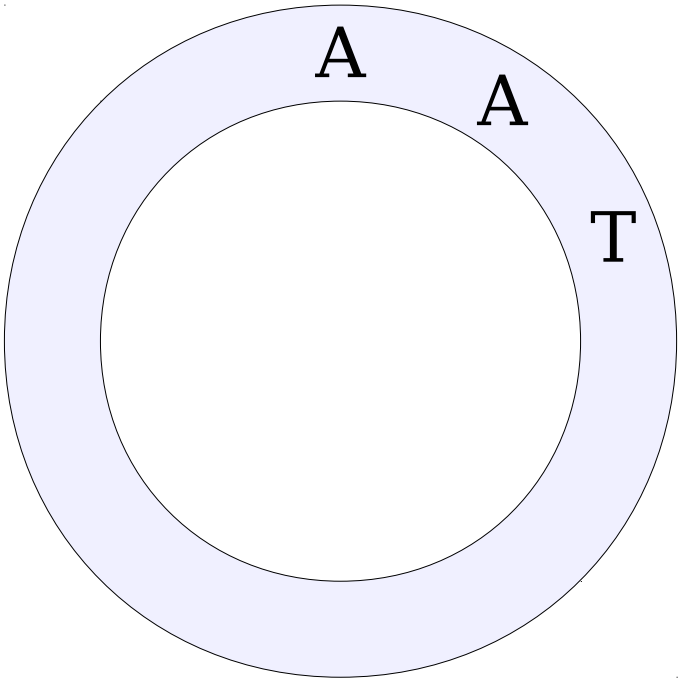
Plasmid Reassembly



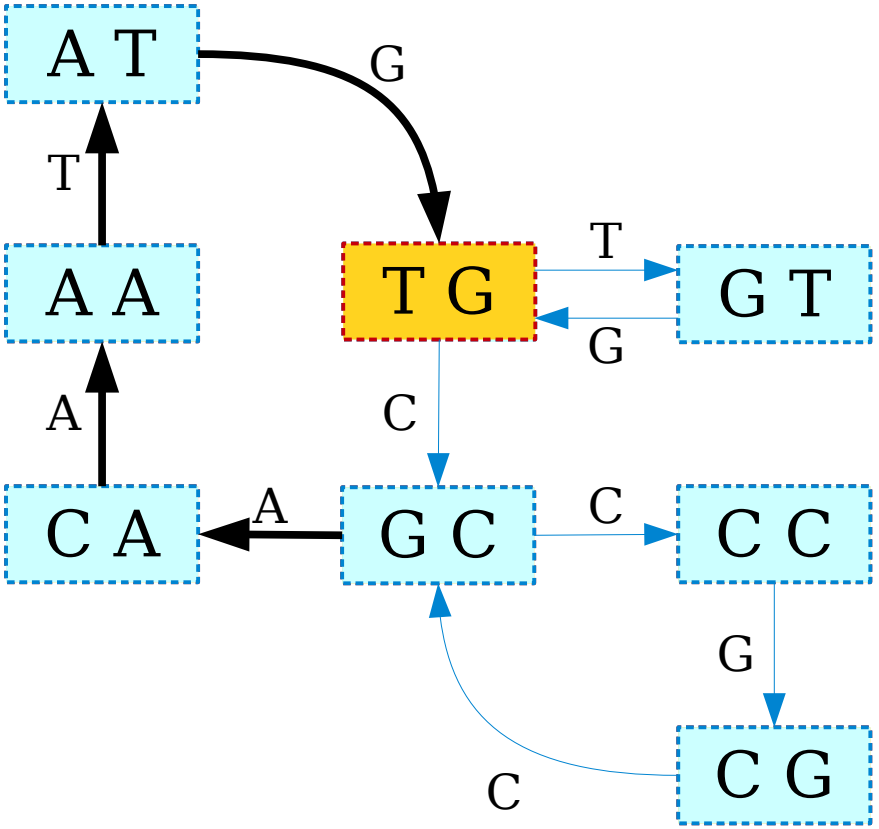


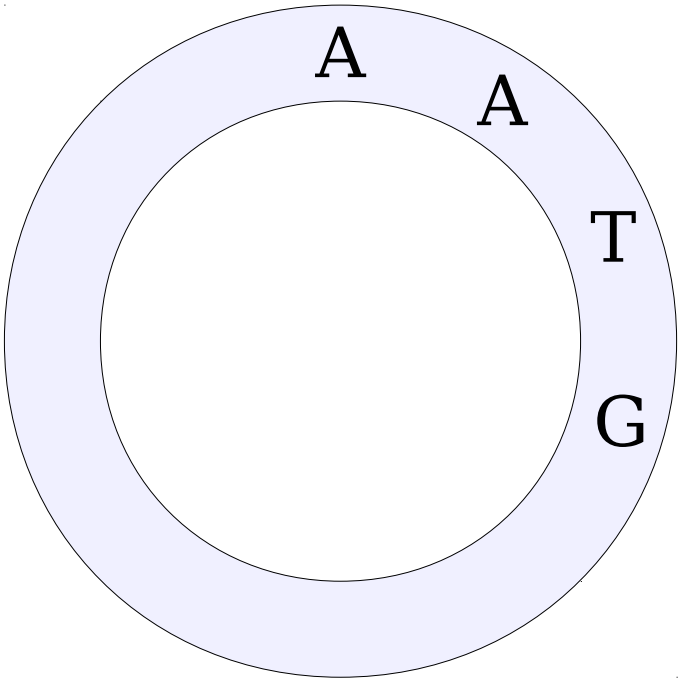
Plasmid Reassembly



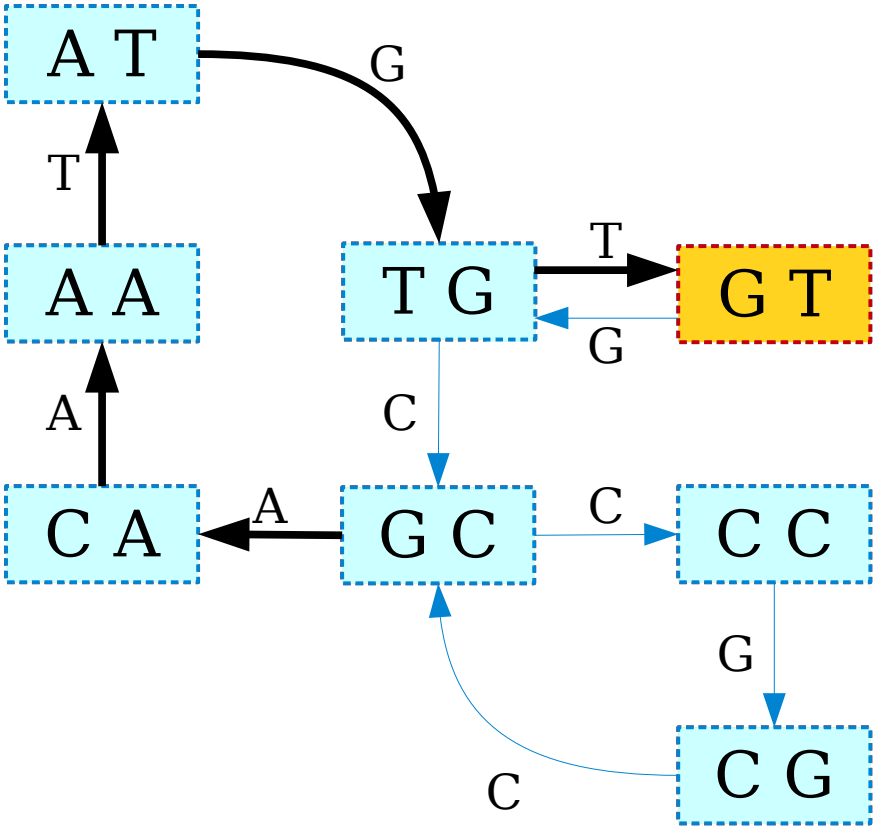


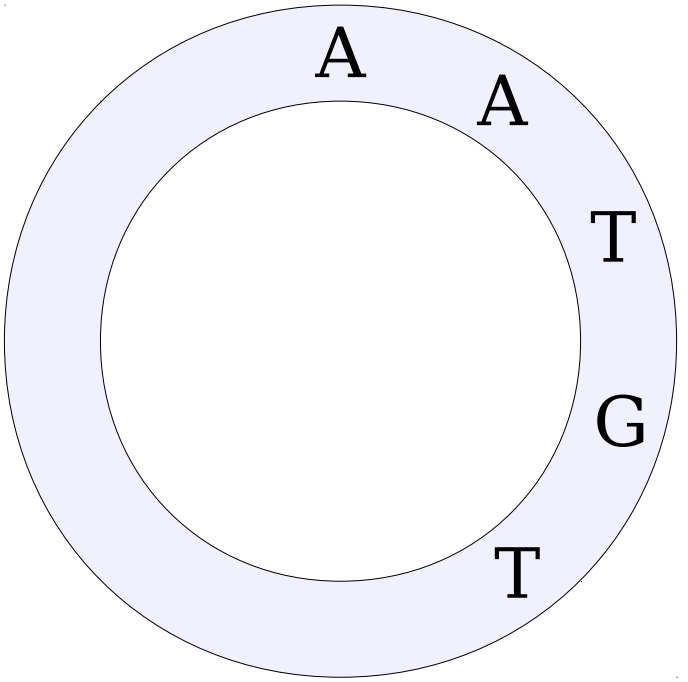
Plasmid Reassembly



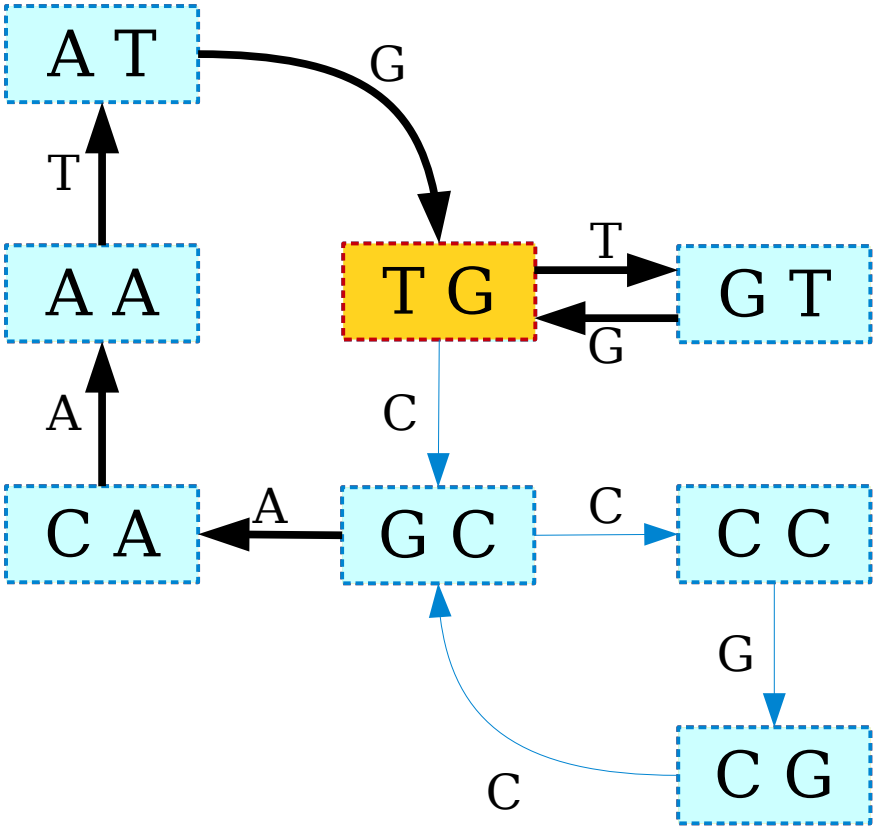


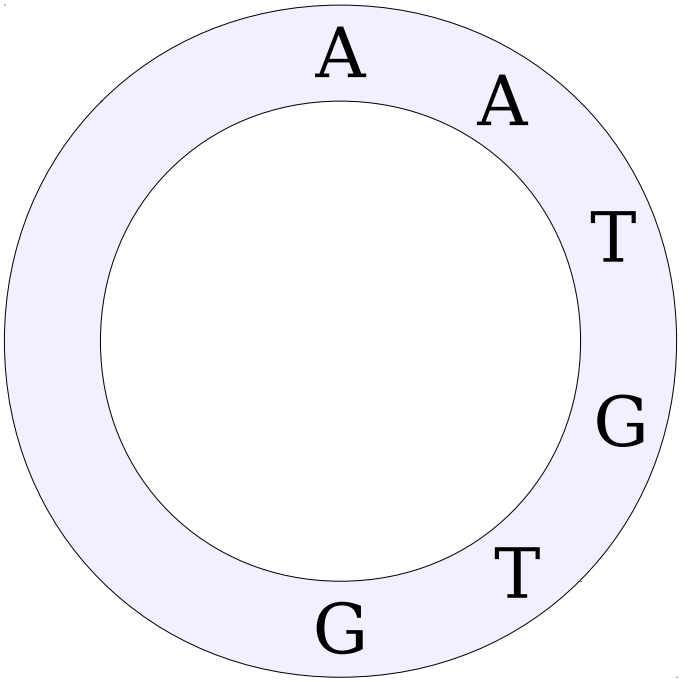
Plasmid Reassembly



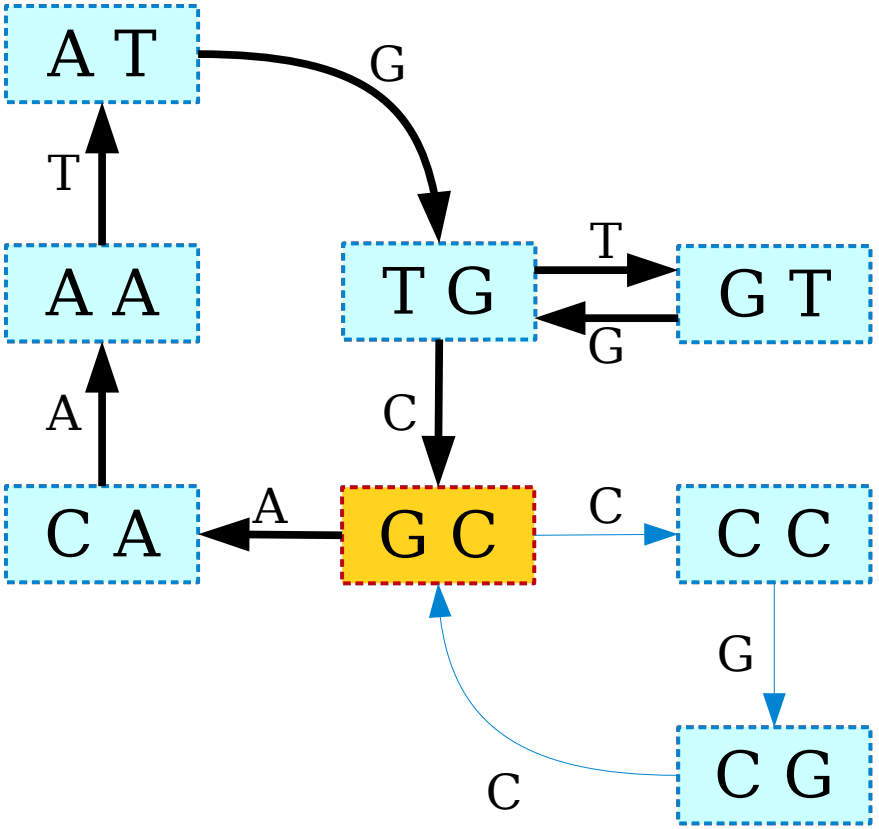


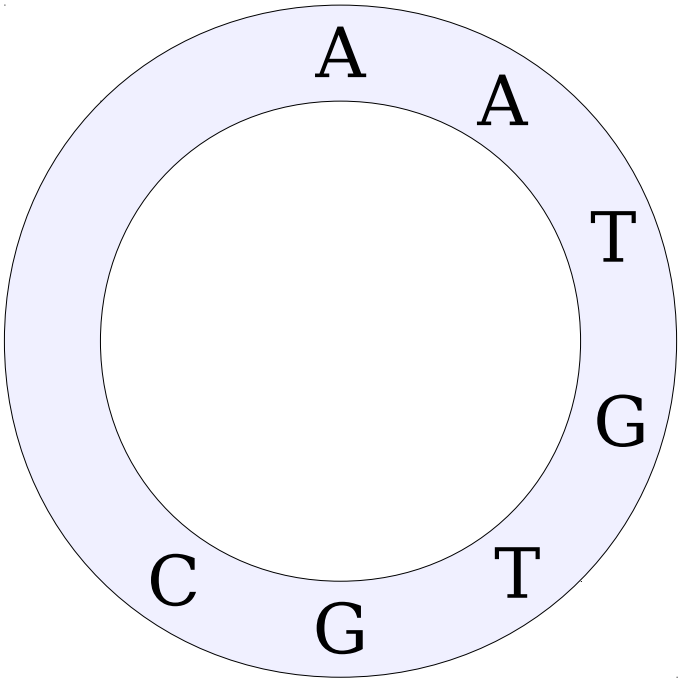
Plasmid Reassembly



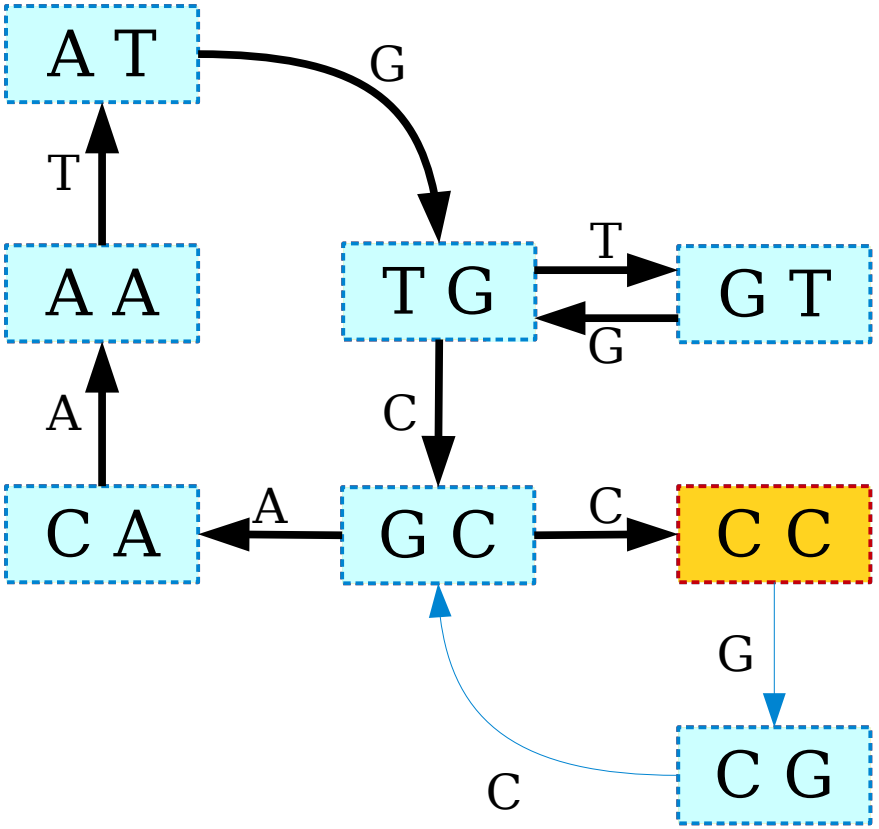


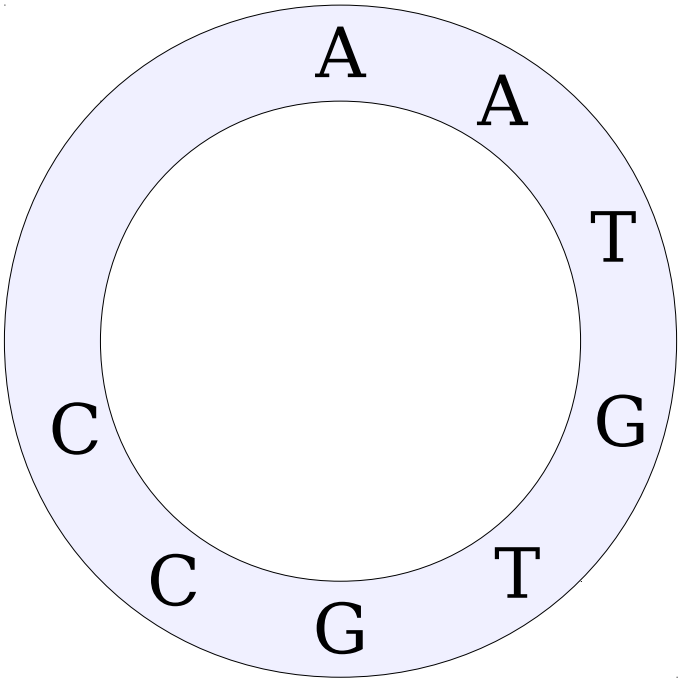
Plasmid Reassembly



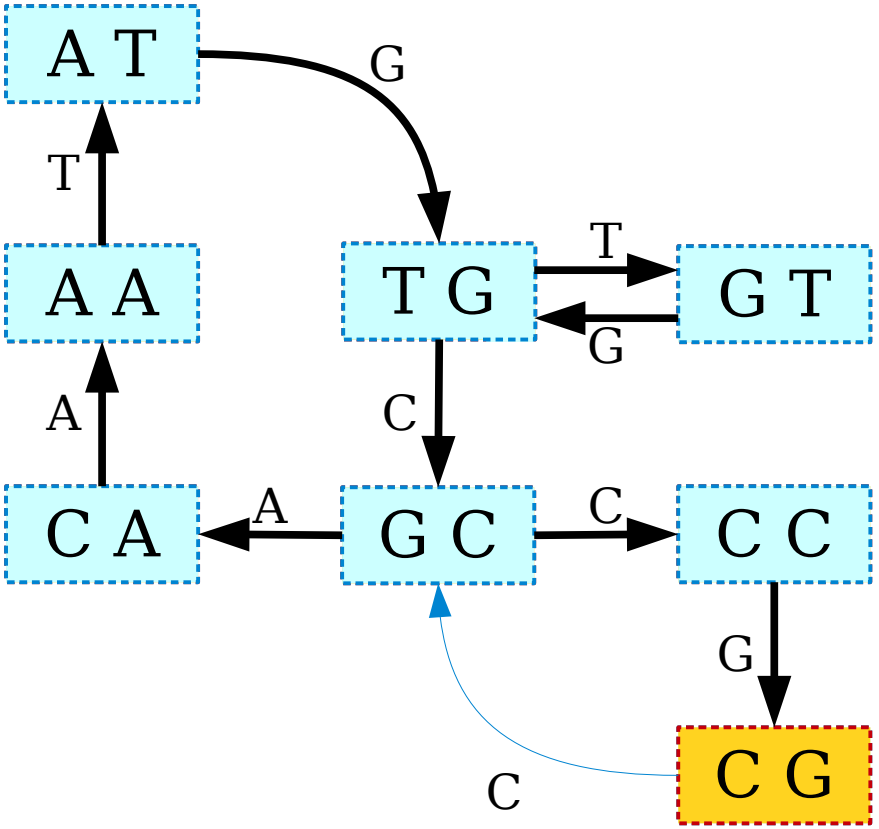


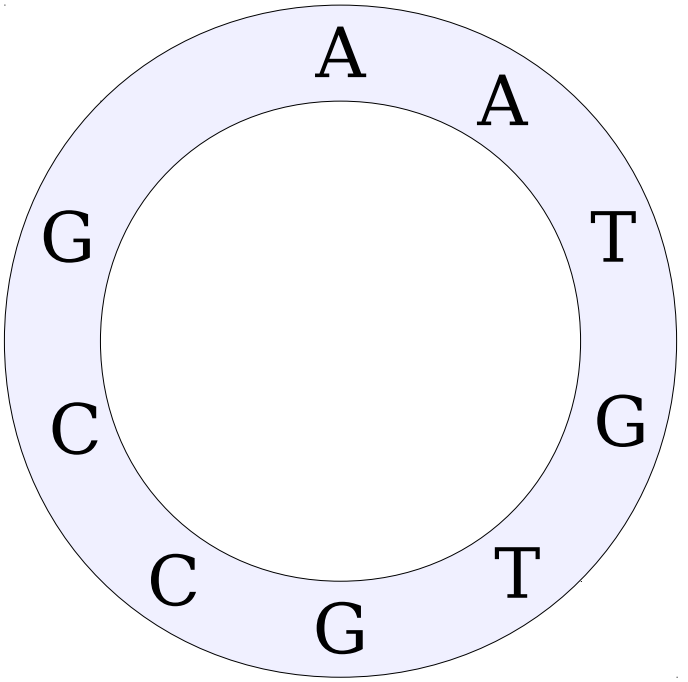
Plasmid Reassembly



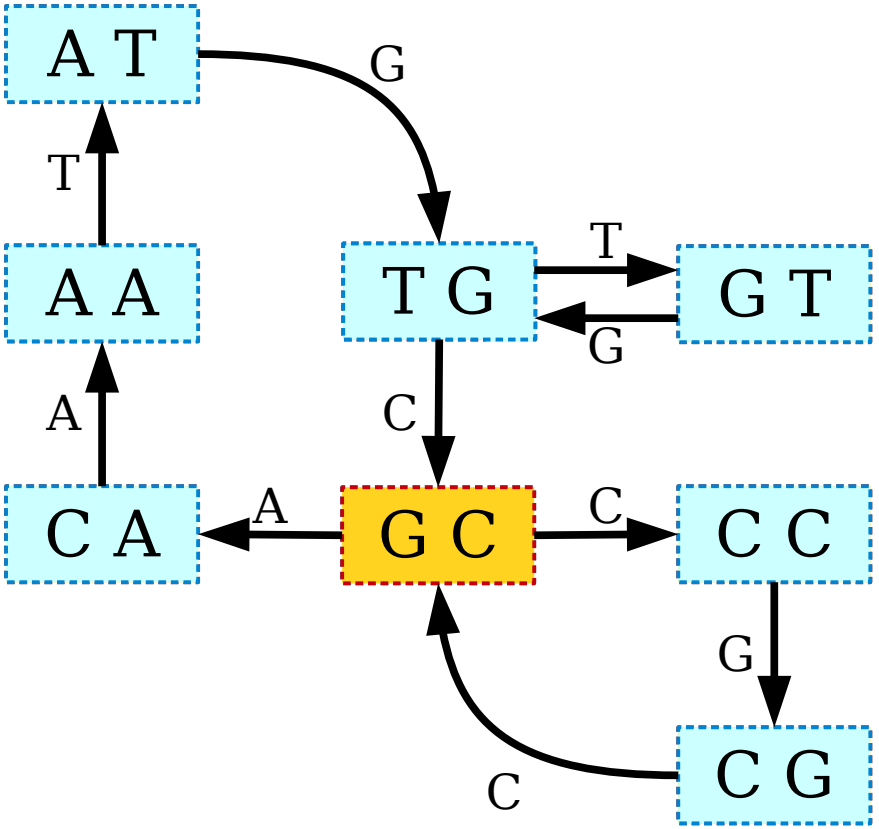


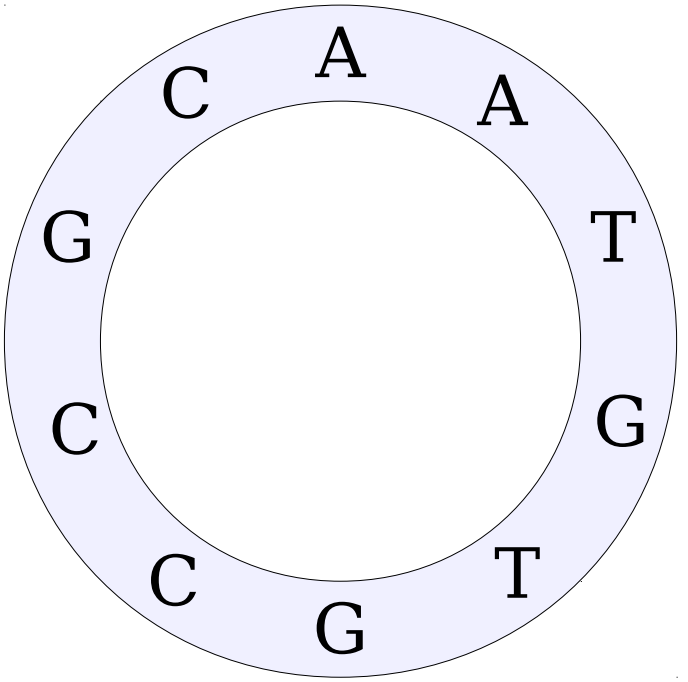
Plasmid Reassembly



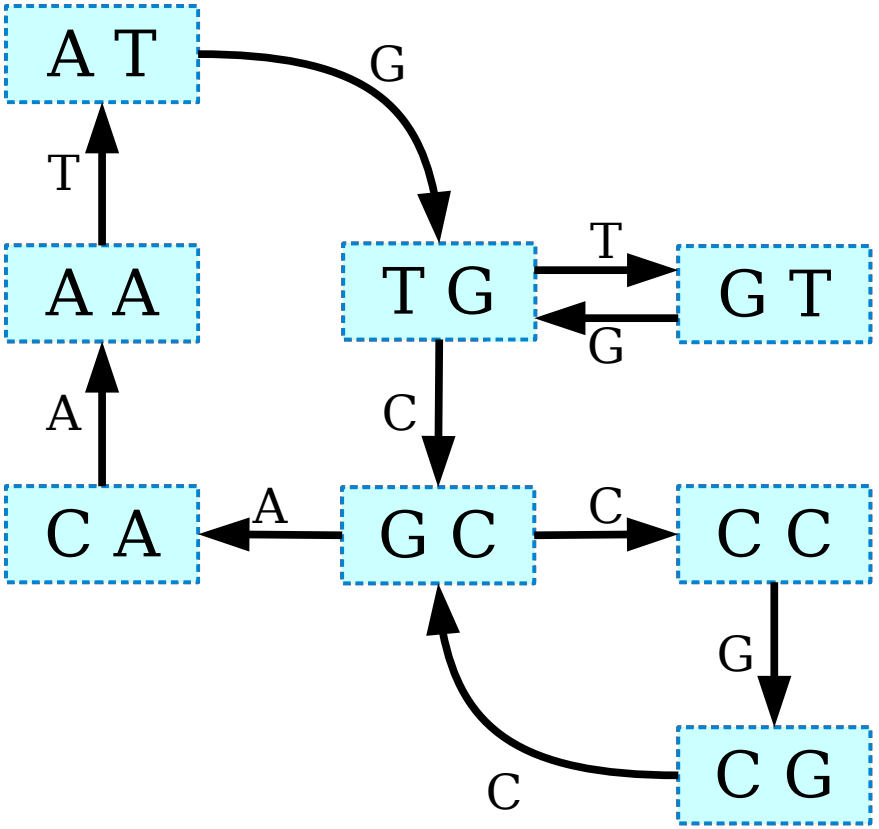


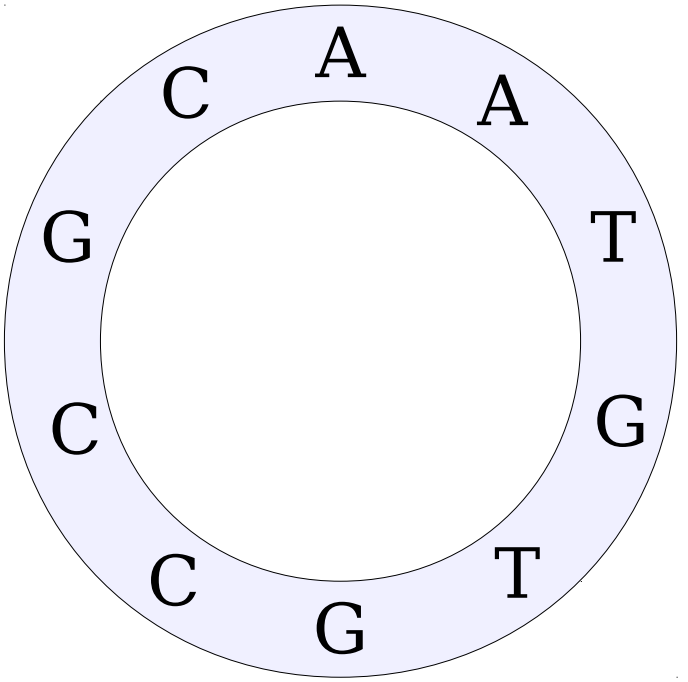
Plasmid Reassembly



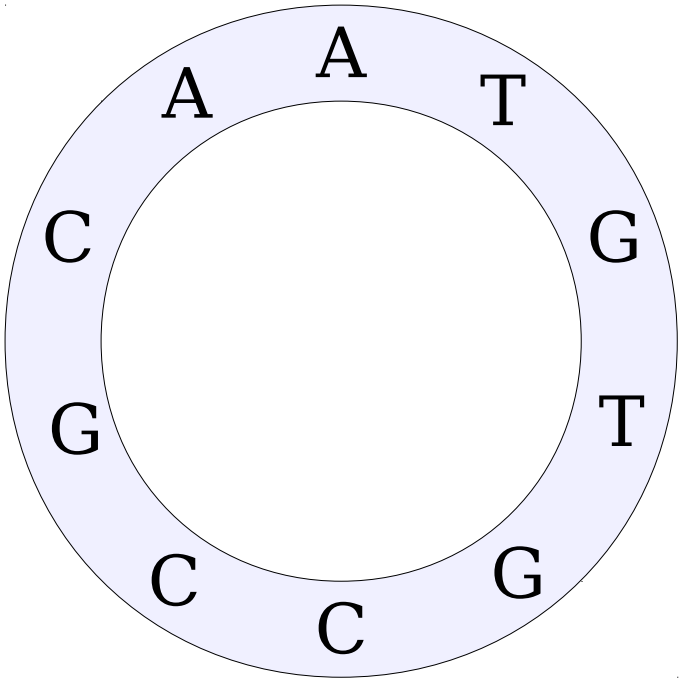


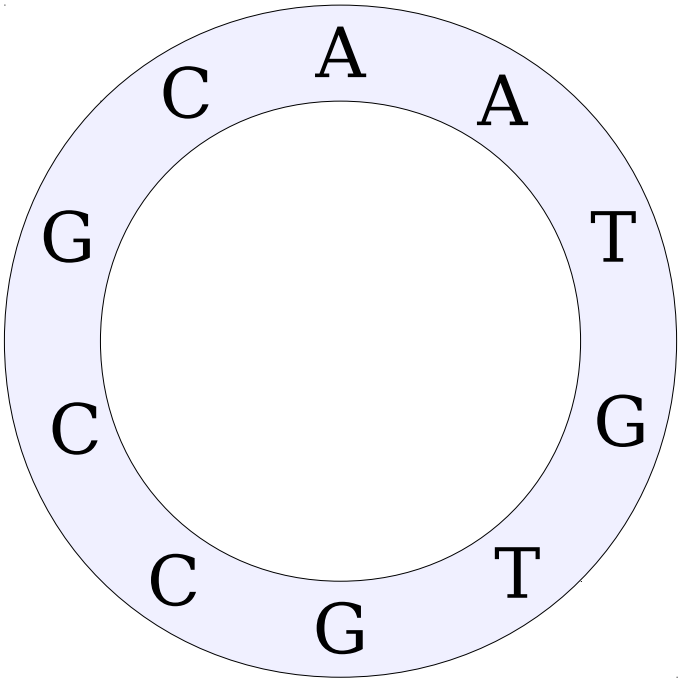
Plasmid Reassembly



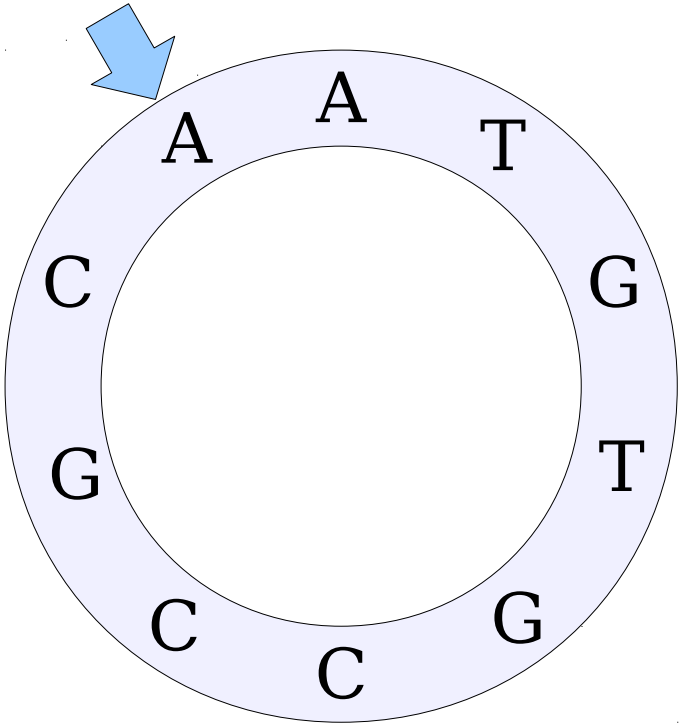


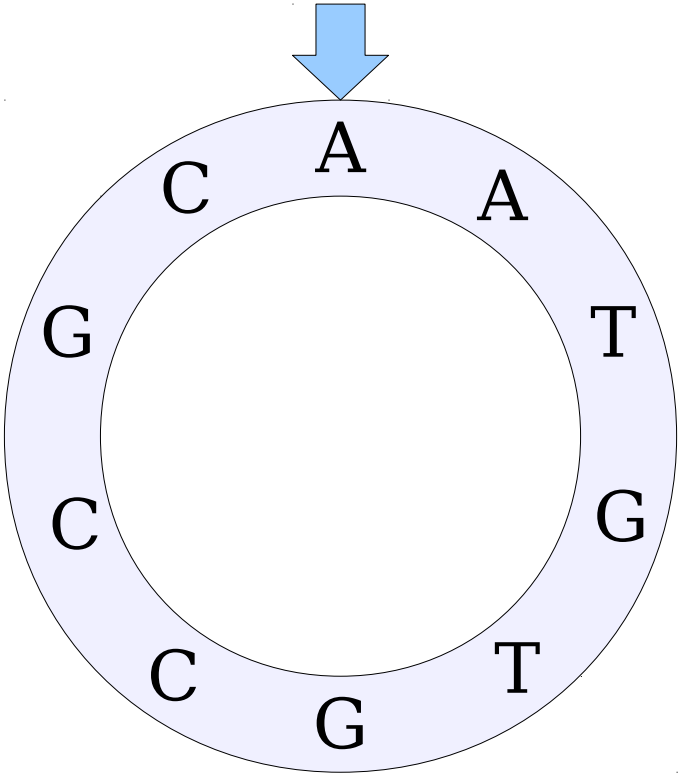
Plasmid Reassembly





Plasmid Reassembly





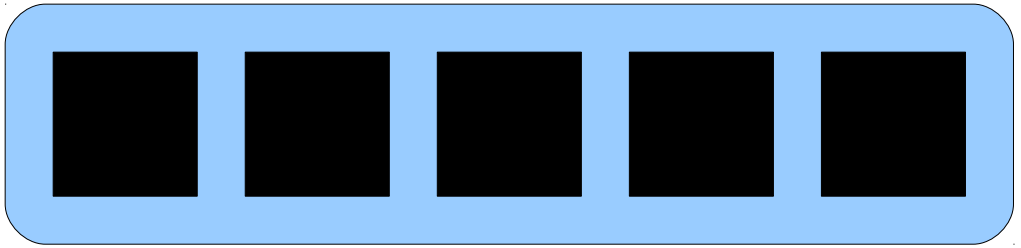
Plasmid Reassembly

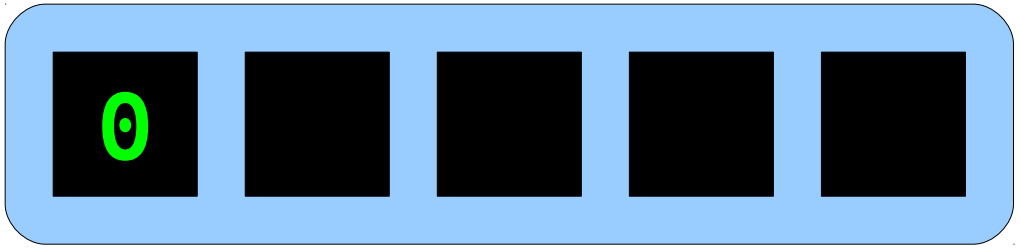
- The general algorithm works like this:
  - Choose a sliding window size. The larger, the better.
  - Slide that window across the reads. For each window, create a node in the graph. Form edges linking each window to the window that comes after it.
  - Assuming the window size is bigger than the largest repeated string, and assuming you have good coverage, the graph will be Eulerian.
  - Each Eulerian circuit corresponds to one possible plasmid.
- Want to learn more? See “A New Algorithm for DNA Sequence Assembly” by Idury and Waterman or “An Eulerian Path Approach to DNA Fragment Assembly” by Pevzner, Tang, and Waterman.

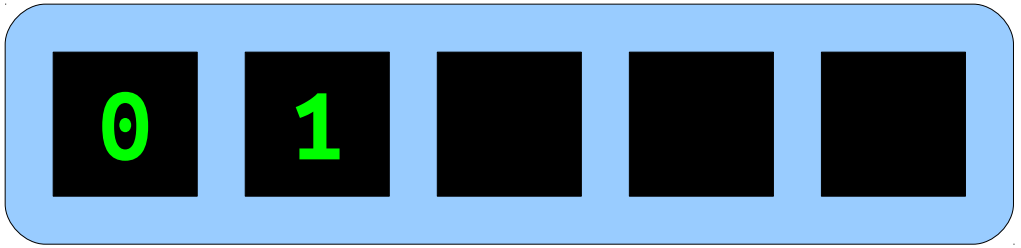
Changing Gears: ***Stealing Things***

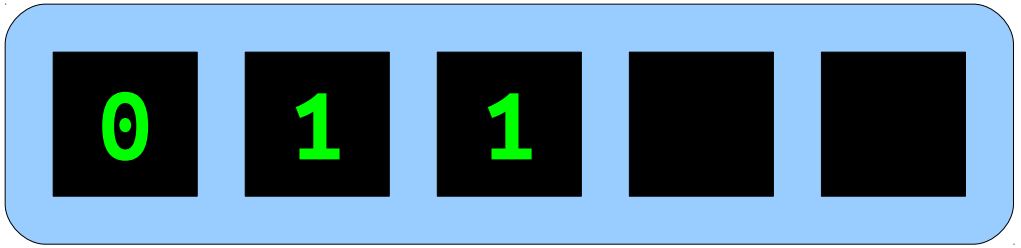
The Security System

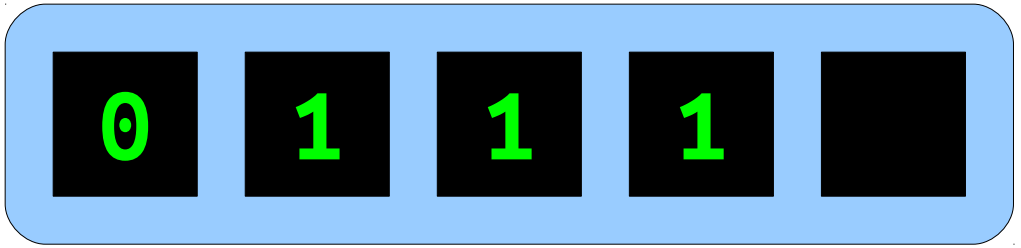
- Suppose there's a security system that asks for an n-bit password.
  - Recall that there are 2^n possible bitstrings of length n.
- You can try to break this system by entering all 2^n possible passwords one at a time.
- How many symbols do you have to punch in to do this?
  - **Answer:** $n \cdot 2^n$.

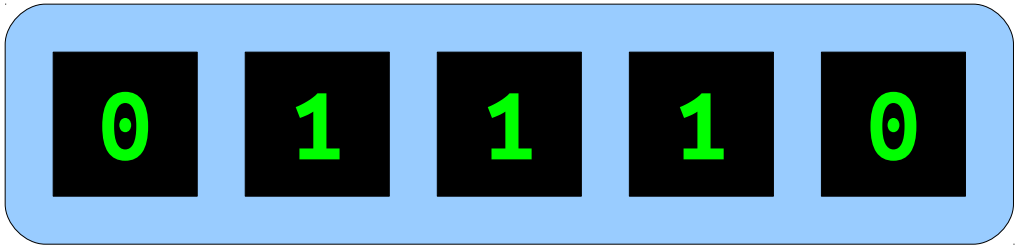


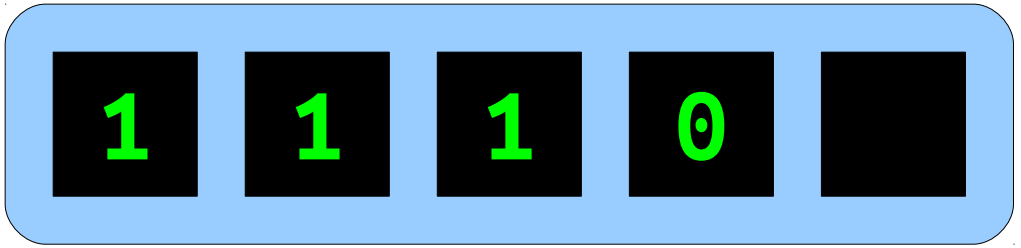

0

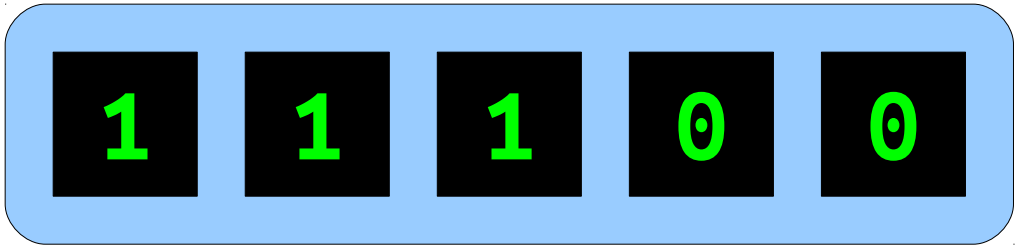

0
1

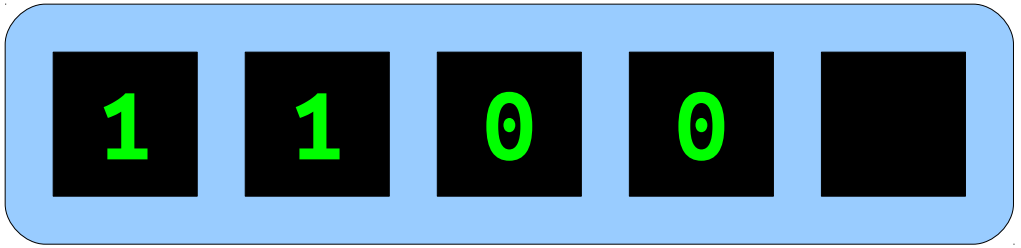

0
1
1

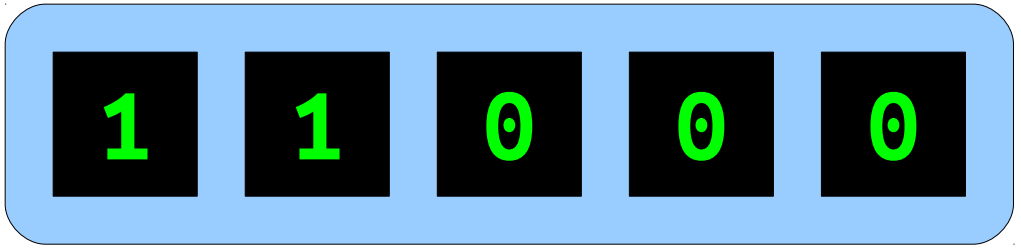

0
1
1
1


0 1 1 1 0


1
1
1
0

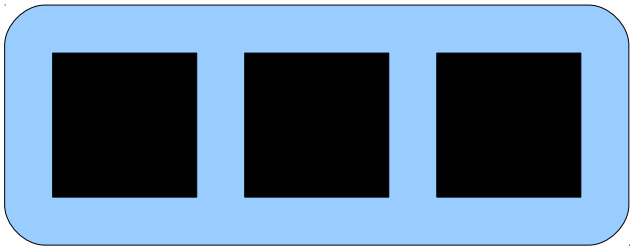

1 1 1 0 0


1
1
0
0


1 1 0 0 0

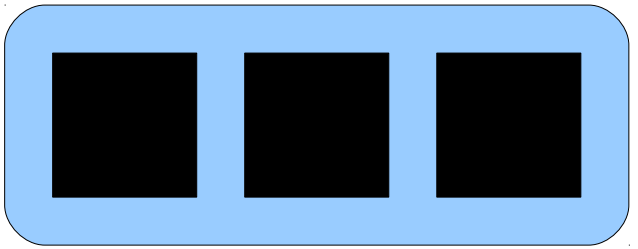
The Security System

- Suppose the security system has a design flaw: it always listens to the last n bits that have been read (where n is the length of the password) and opens as soon as the correct n bits are entered.
- Can you open the lock faster now?



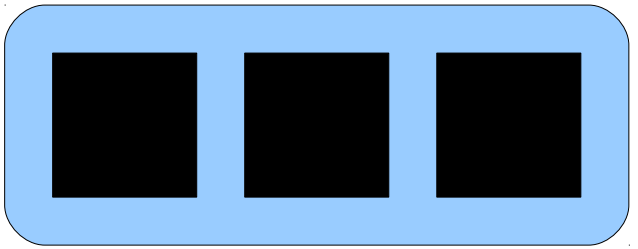
All Possible
3-Bit Codes

0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1



All Possible
3-Bit Codes

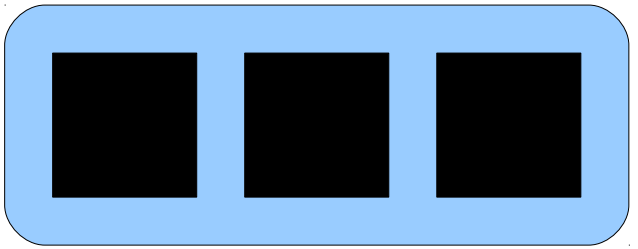
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

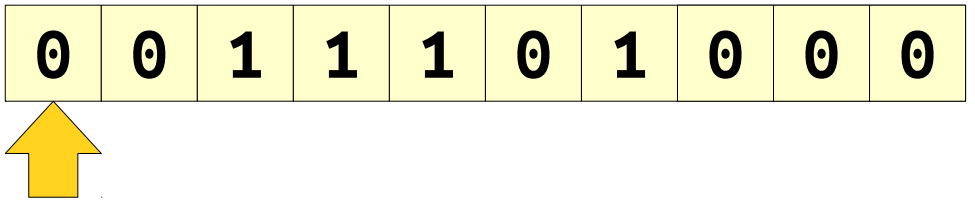


0	0	1	1	1	0	1	0	0	0

All Possible
3-Bit Codes

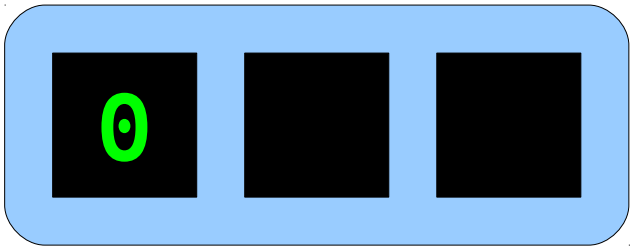
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

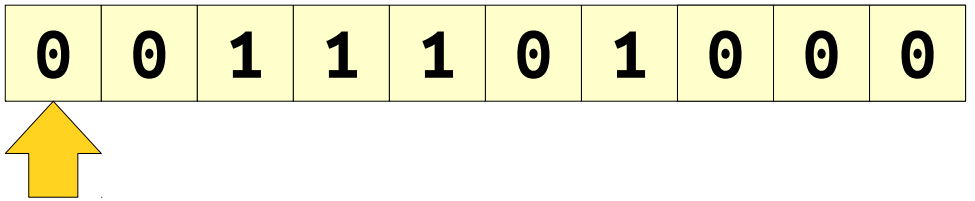




All Possible
3-Bit Codes

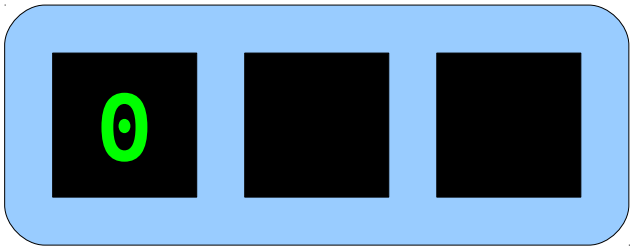
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

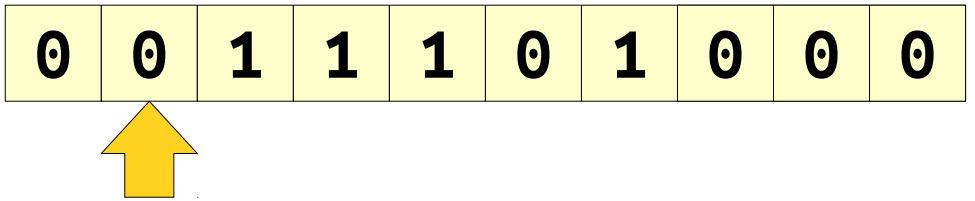




All Possible
3-Bit Codes

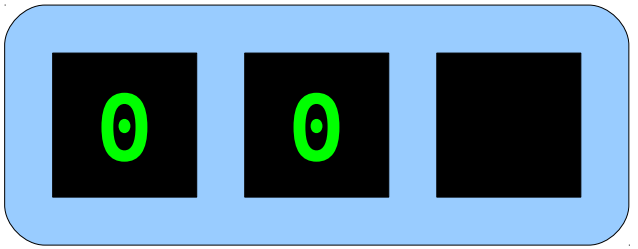
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

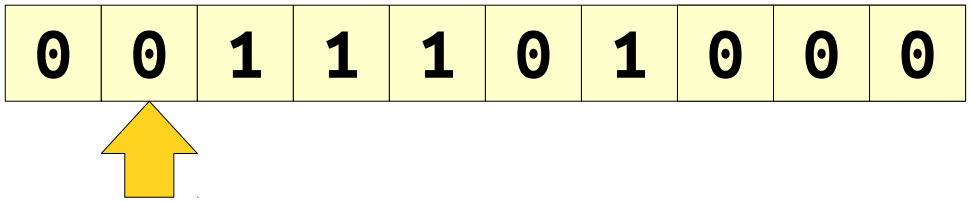




All Possible
3-Bit Codes

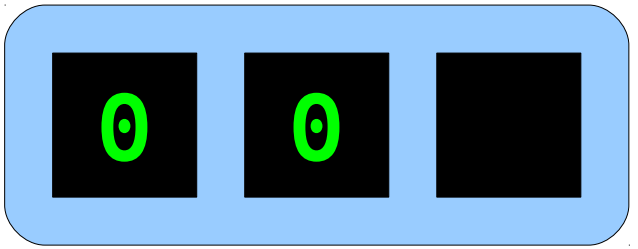
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

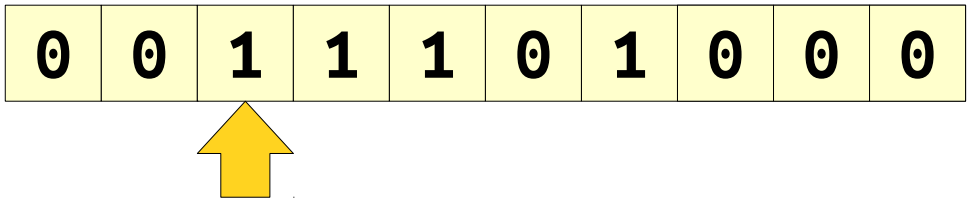




All Possible
3-Bit Codes

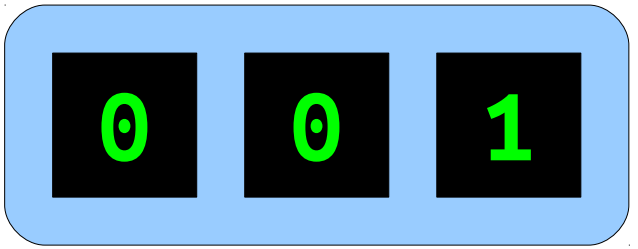
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

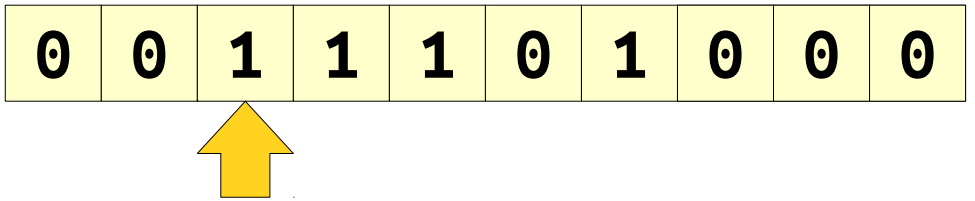




All Possible
3-Bit Codes

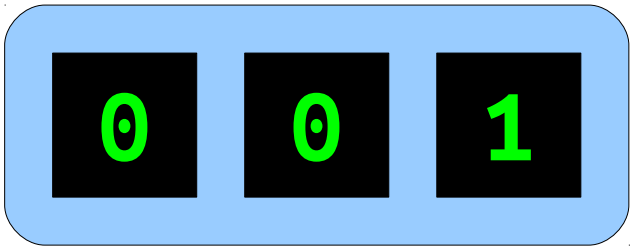
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

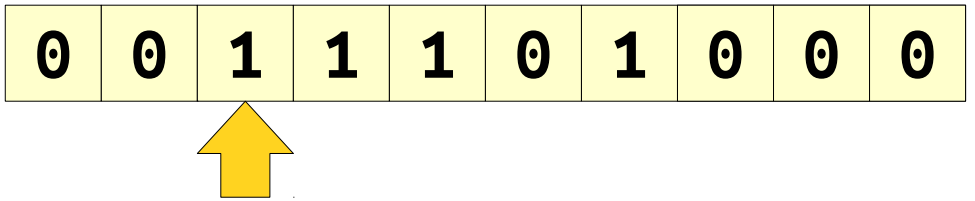




All Possible
3-Bit Codes

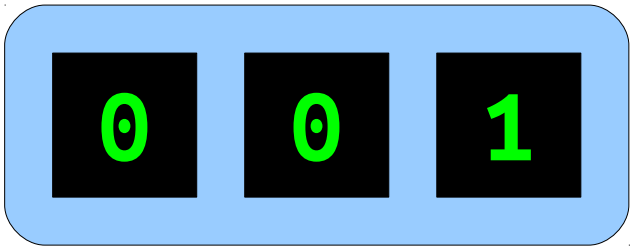
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

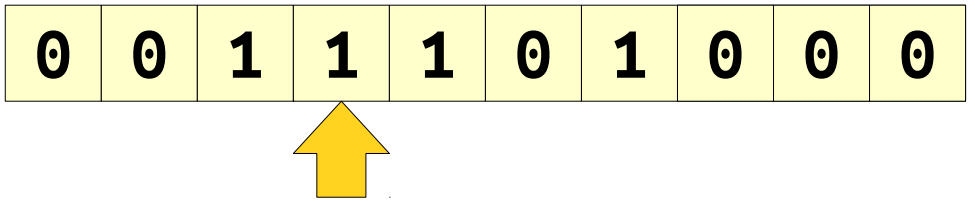




All Possible
3-Bit Codes

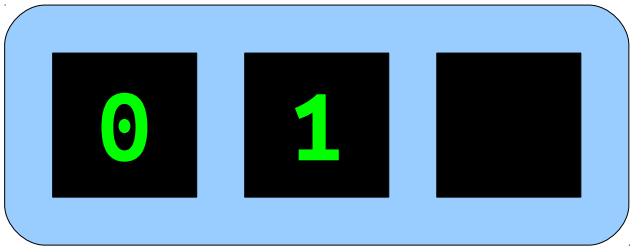
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

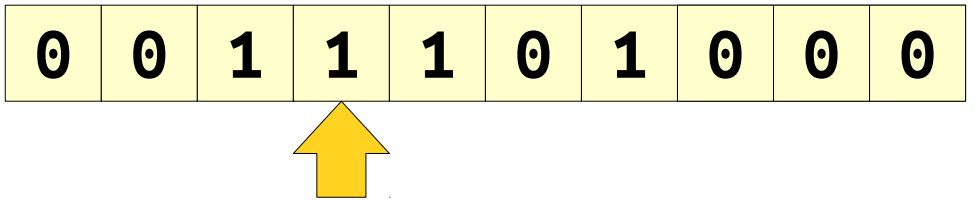




All Possible
3-Bit Codes

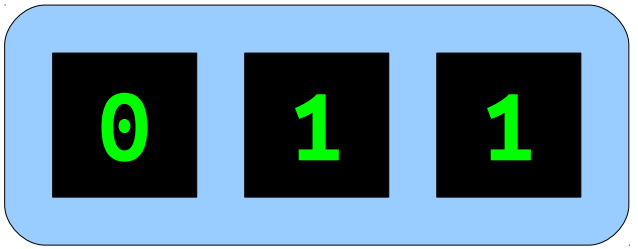
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

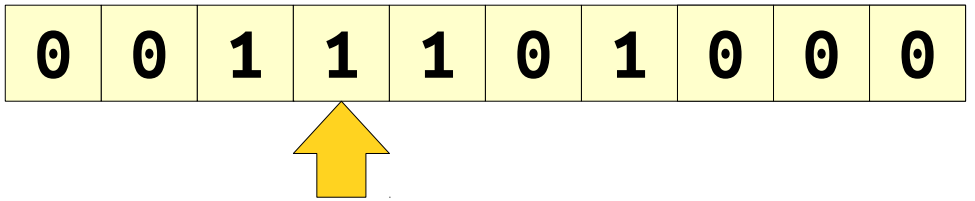




All Possible
3-Bit Codes

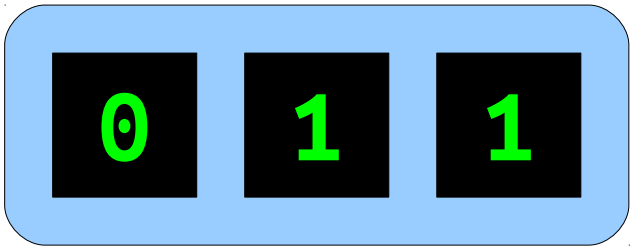
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

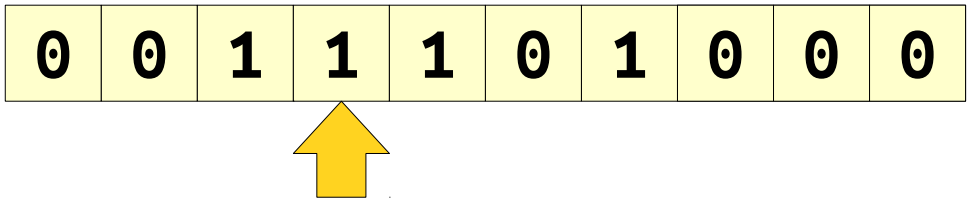




All Possible
3-Bit Codes

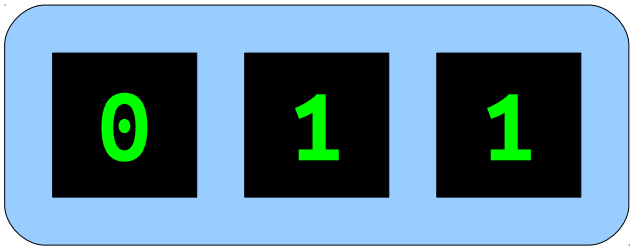
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

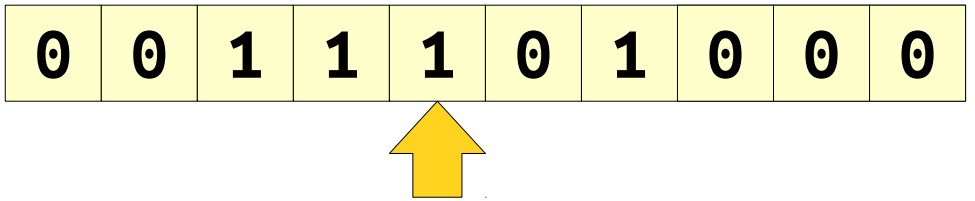




All Possible
3-Bit Codes

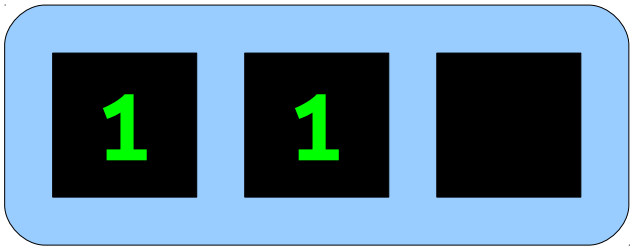
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

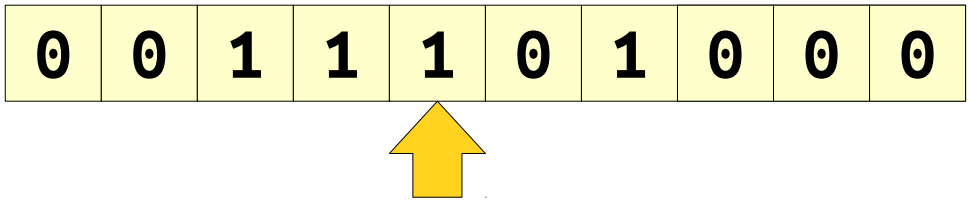




All Possible
3-Bit Codes

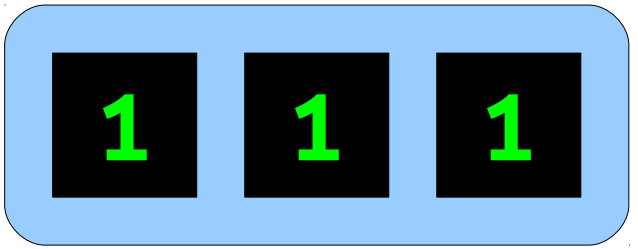
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

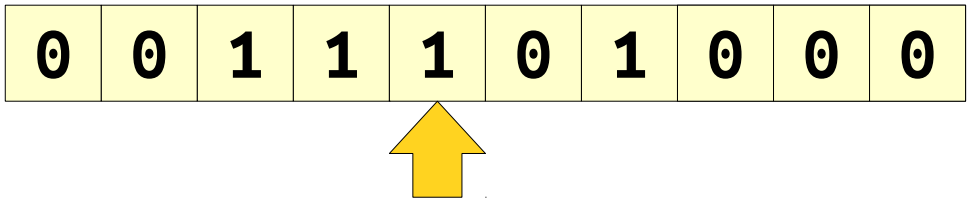




All Possible
3-Bit Codes

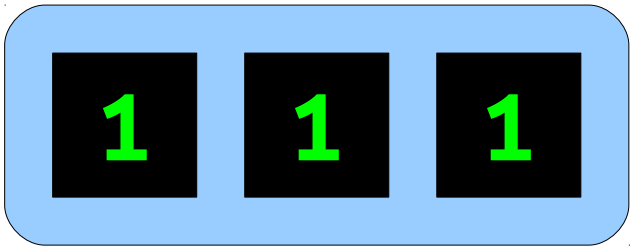
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

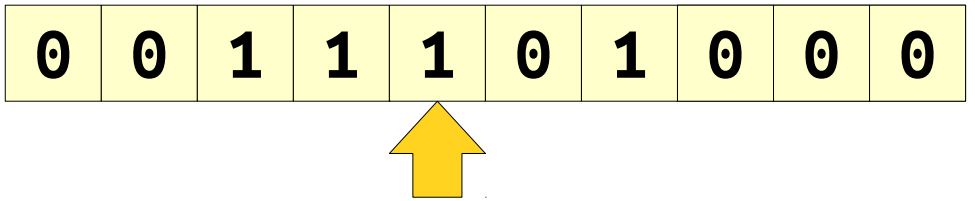




All Possible
3-Bit Codes

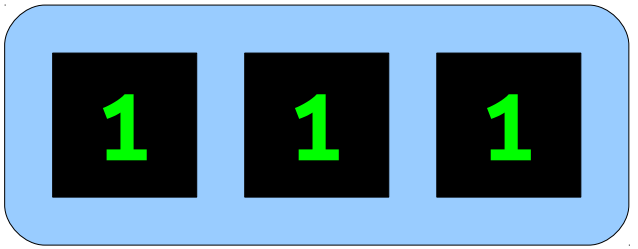
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

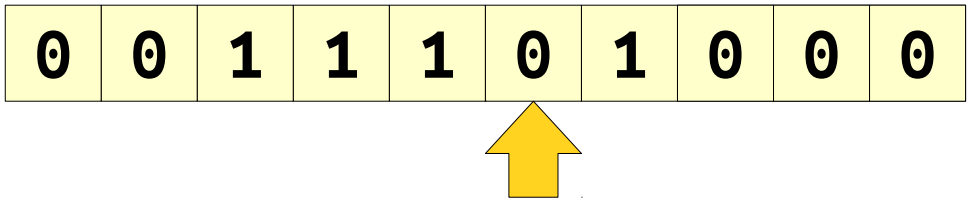




All Possible
3-Bit Codes

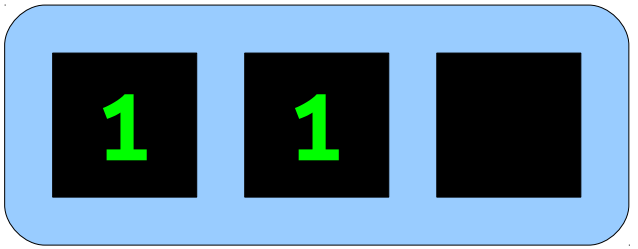
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

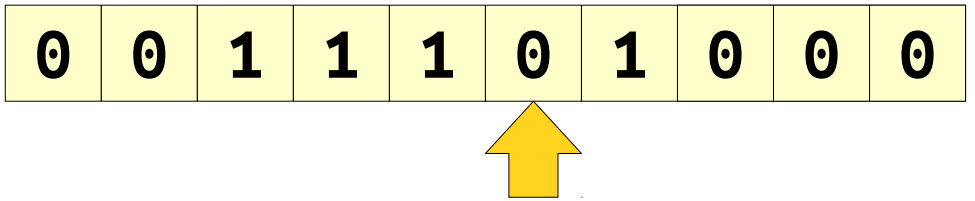




All Possible
3-Bit Codes

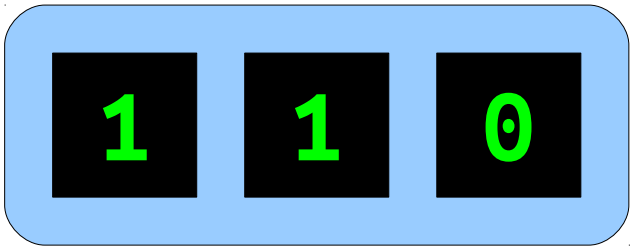
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

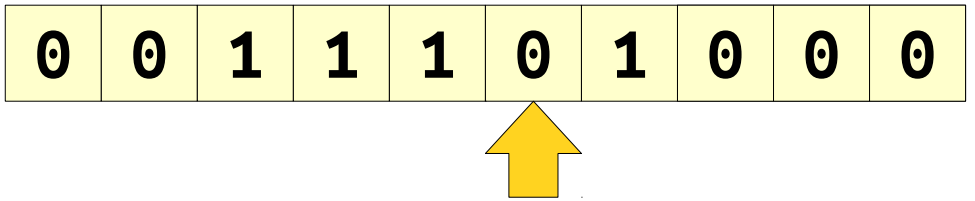




All Possible
3-Bit Codes

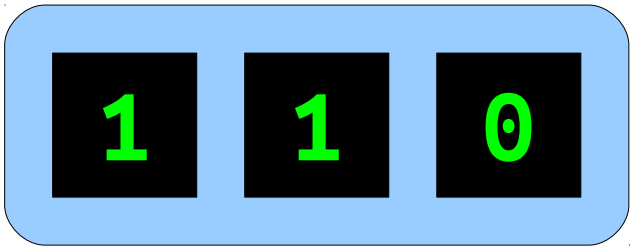
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

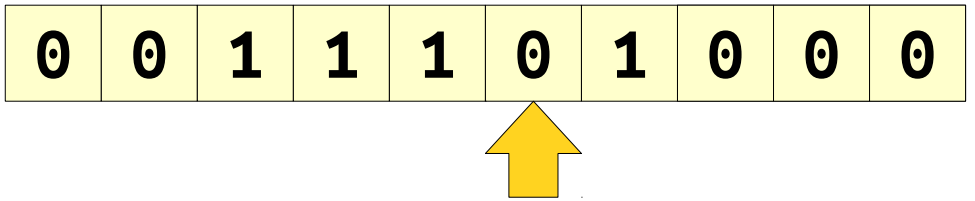




All Possible
3-Bit Codes

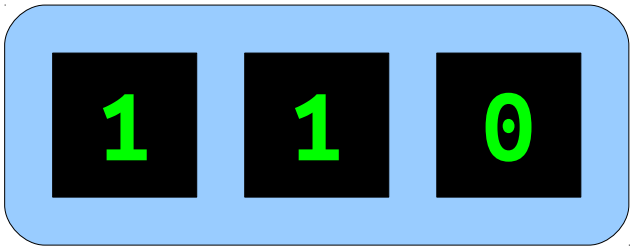
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

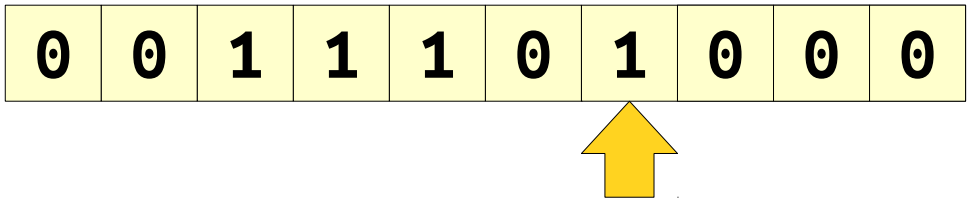




All Possible
3-Bit Codes

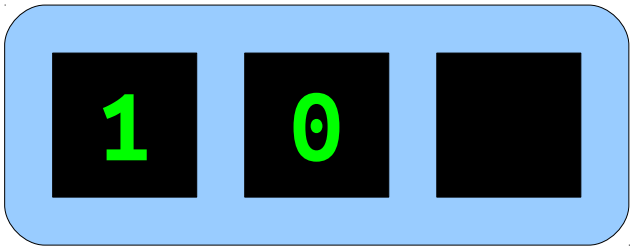
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

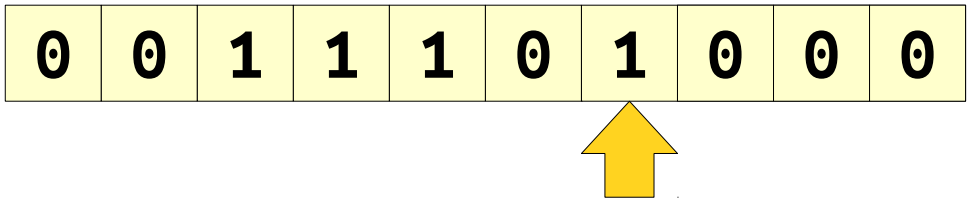




All Possible
3-Bit Codes

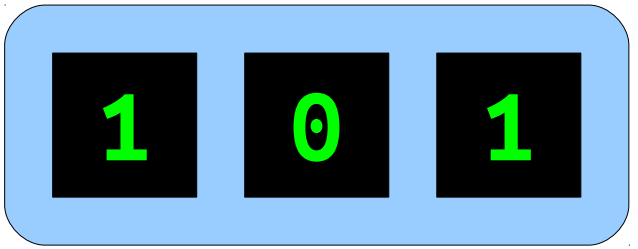
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

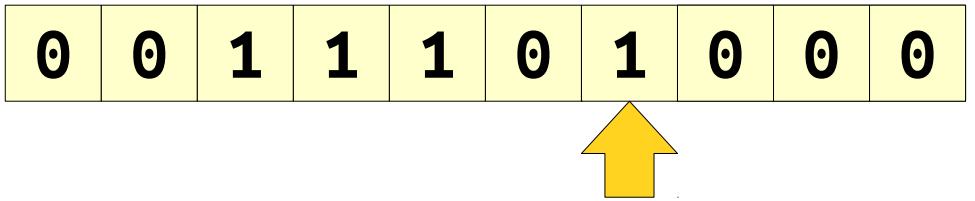




All Possible
3-Bit Codes

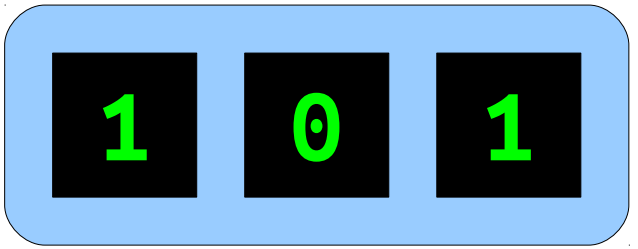
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

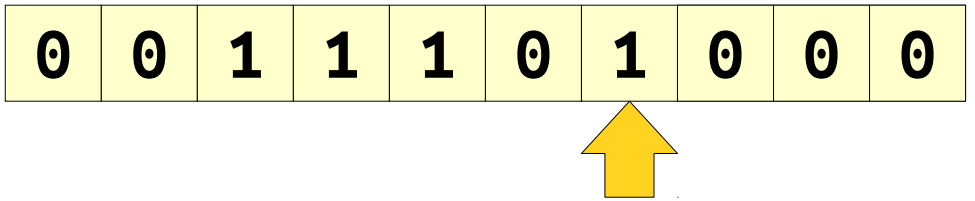




All Possible
3-Bit Codes

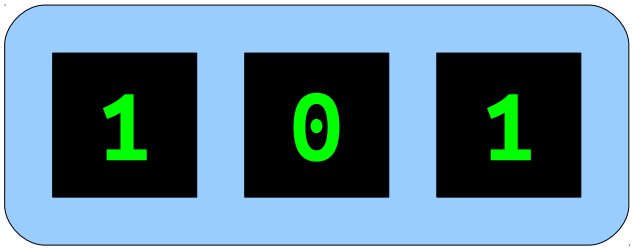
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

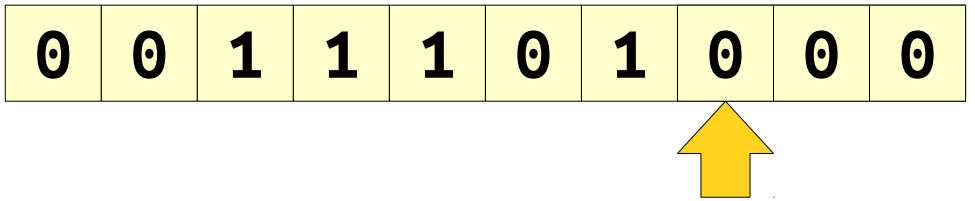




All Possible
3-Bit Codes

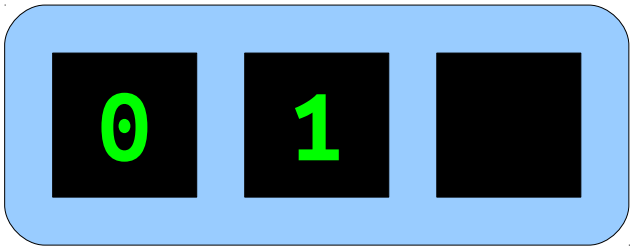
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

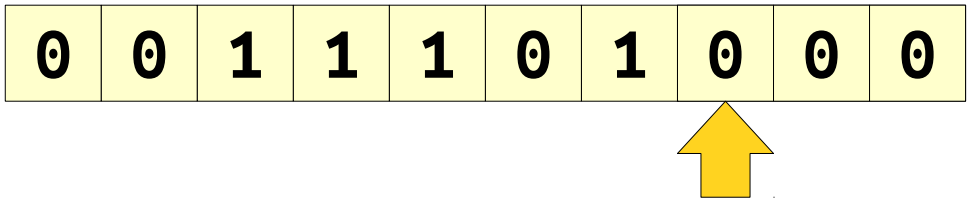




All Possible
3-Bit Codes

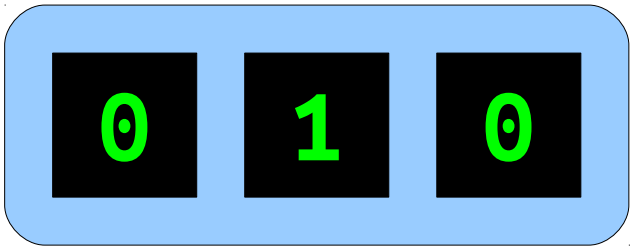
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

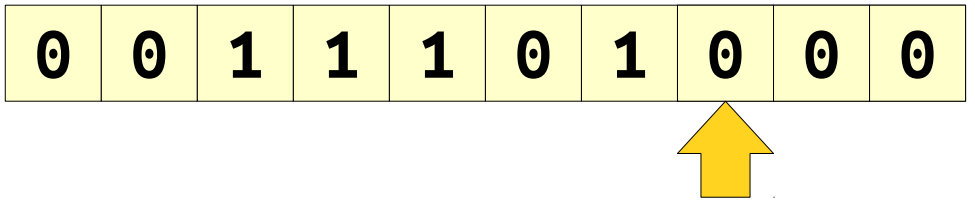




All Possible
3-Bit Codes

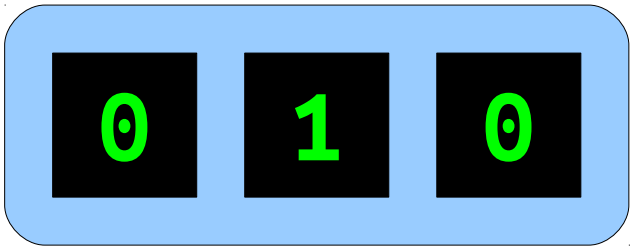
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

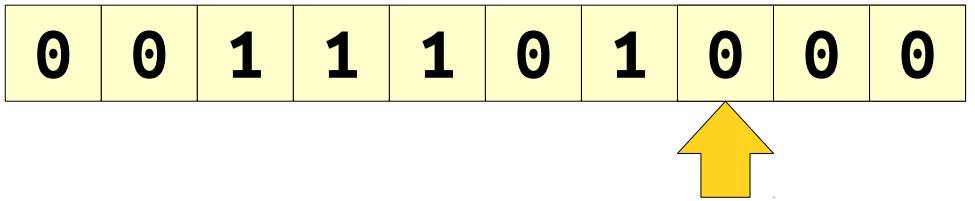




All Possible
3-Bit Codes

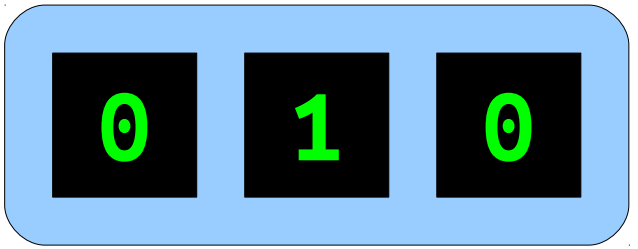
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

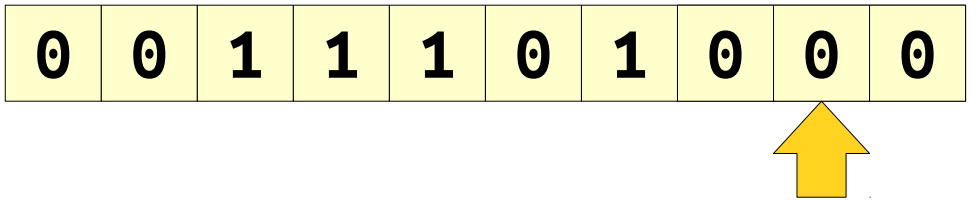




All Possible
3-Bit Codes

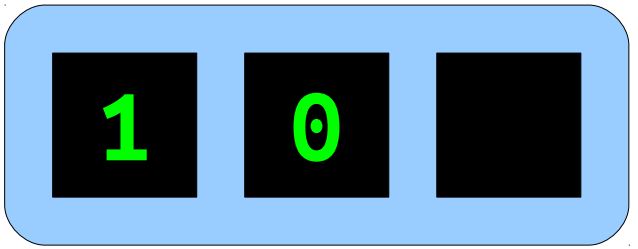
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

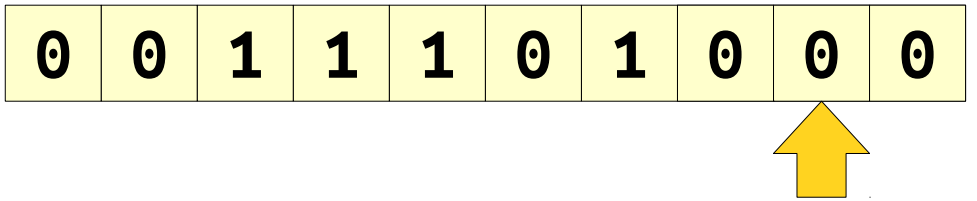




All Possible
3-Bit Codes

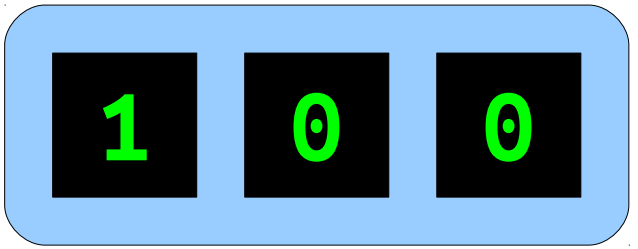
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

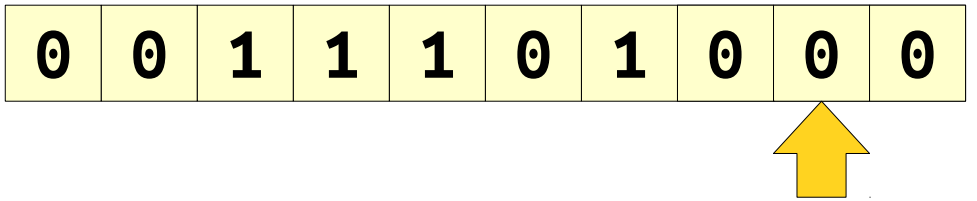




All Possible
3-Bit Codes

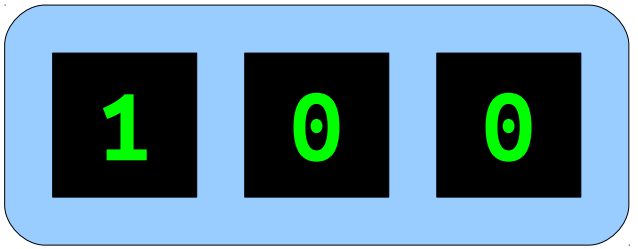
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

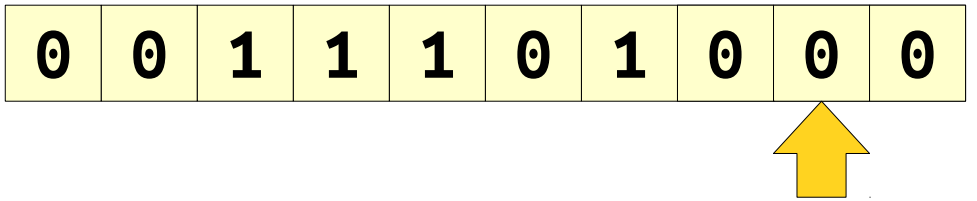




All Possible
3-Bit Codes

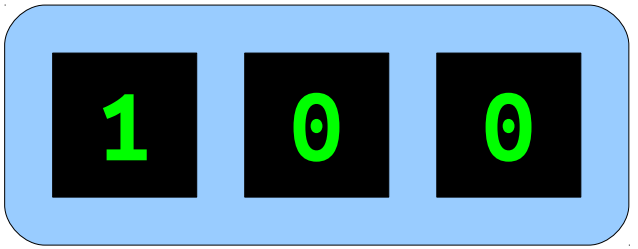
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

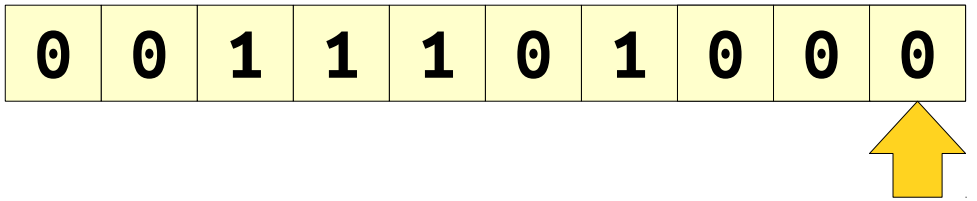




All Possible
3-Bit Codes

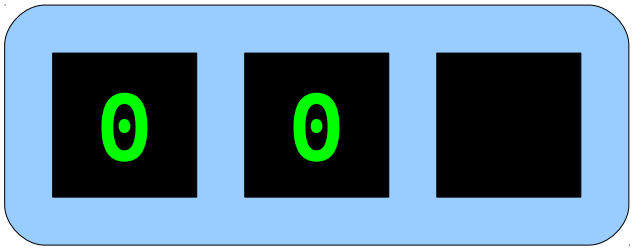
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

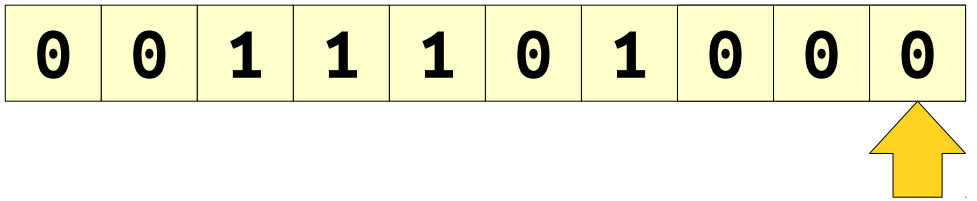




All Possible
3-Bit Codes

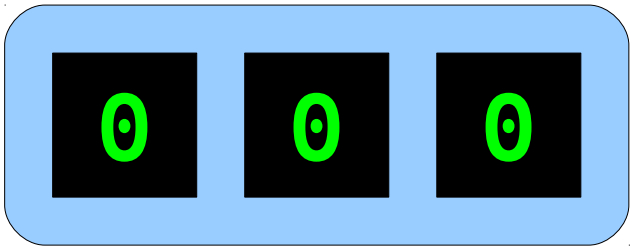
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

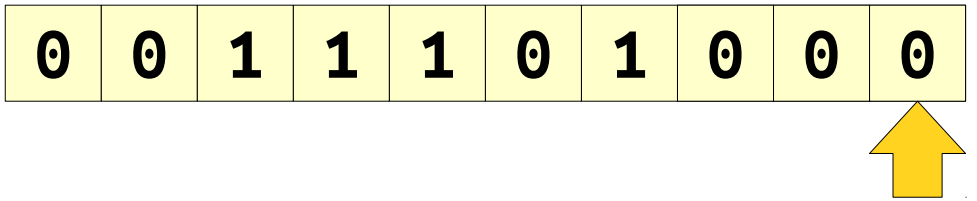




All Possible
3-Bit Codes

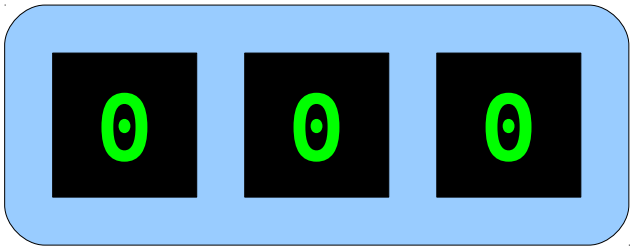
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

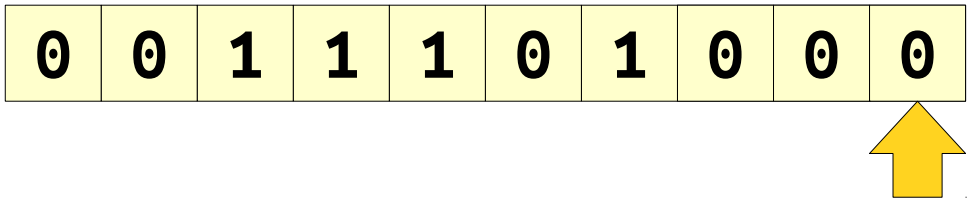




All Possible
3-Bit Codes

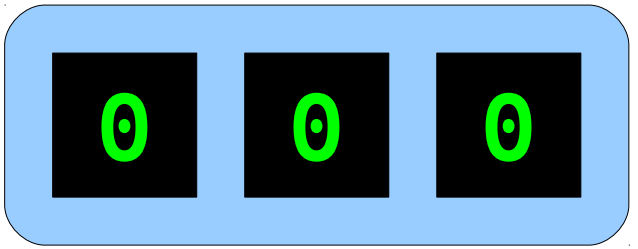
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1





All Possible
3-Bit Codes

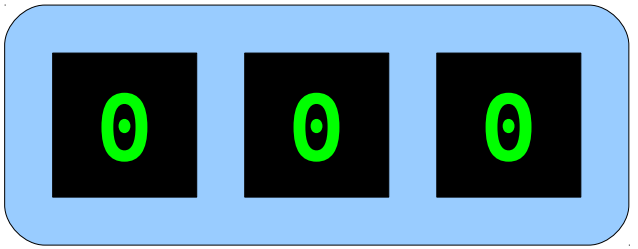
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1



0	0	1	1	1	0	1	0	0	0

All Possible
3-Bit Codes

0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1



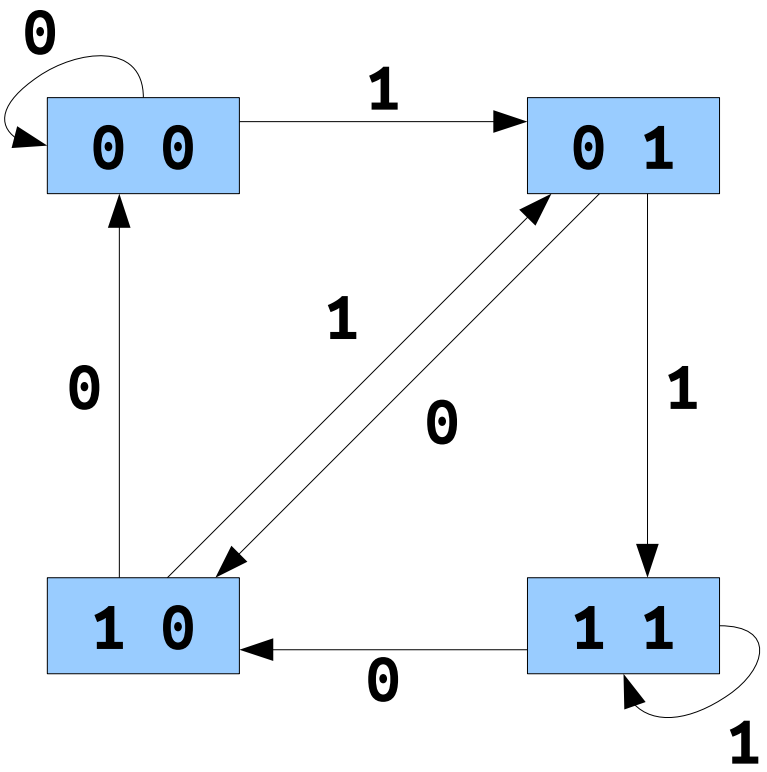
0	0	1	1	1	0	1	0	0	0

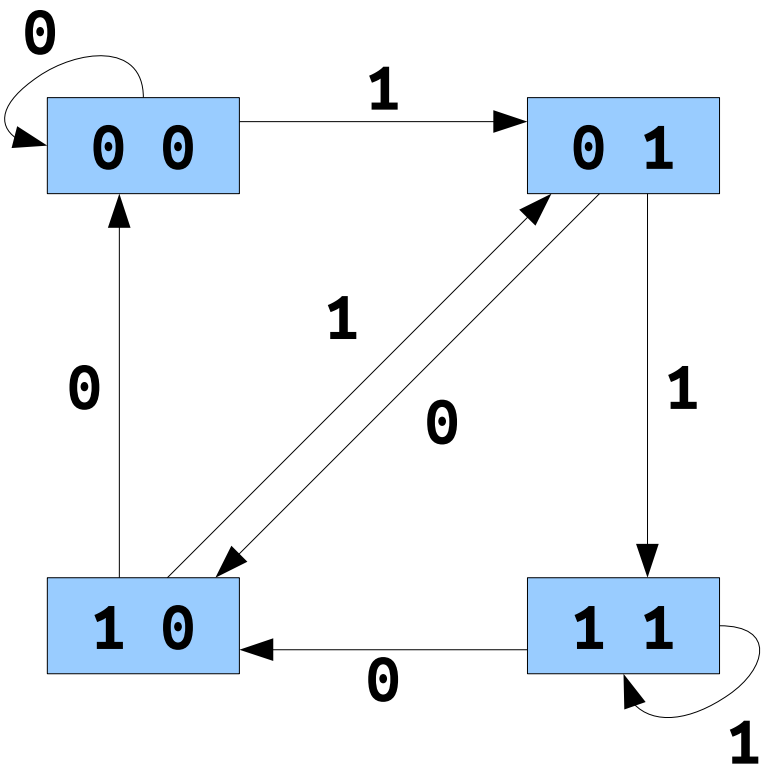
Bits required for naïve approach:
$3 \cdot 2^3 = \mathbf{24}$

Bits in this string: **10**.

This sequence had 10 symbols in it. That's fewer than half of what we would have needed in a brute-force search.

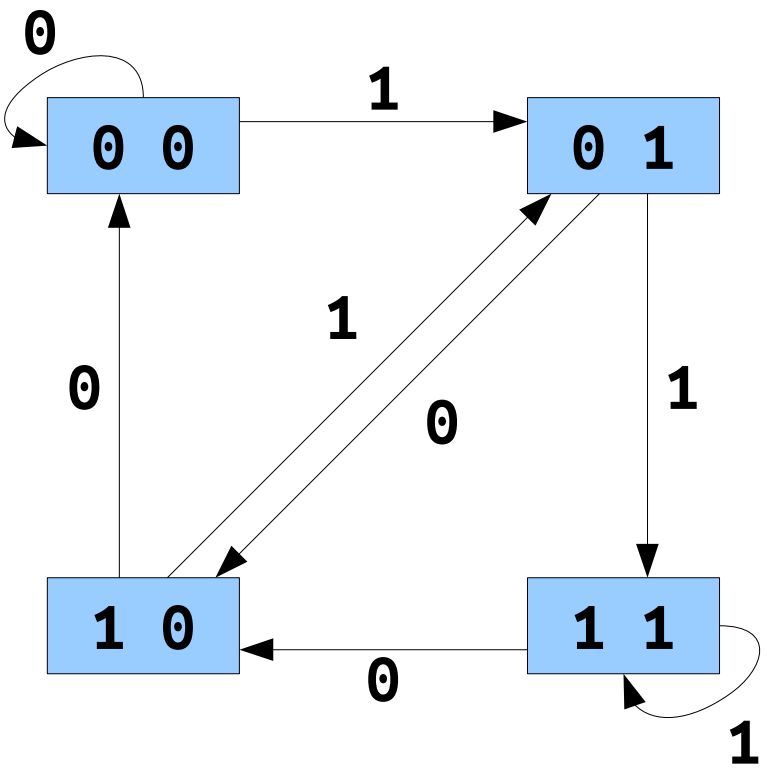
Where did it come from?


0
0 0
1
0 1
1
0
0
1
1 0
1 1
0
1

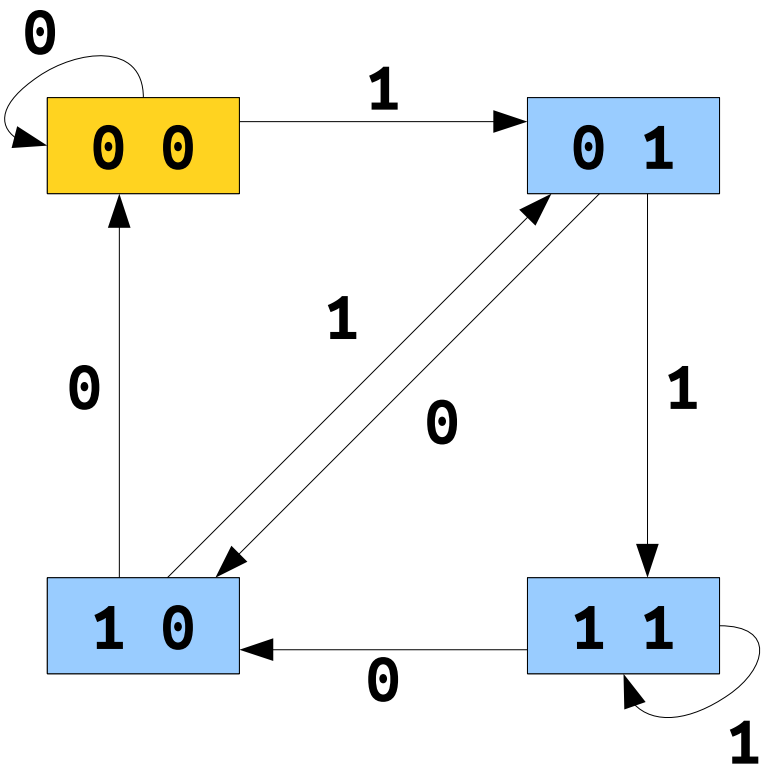


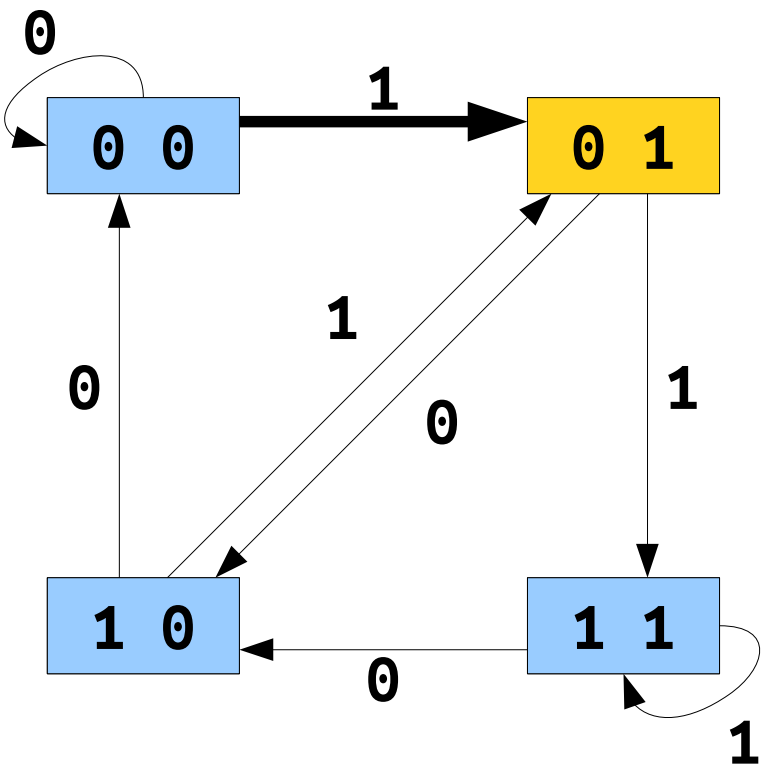
Observation: Every node's indegree equals its outdegree.

(There's one incoming edge for each bit that was shifted out.)

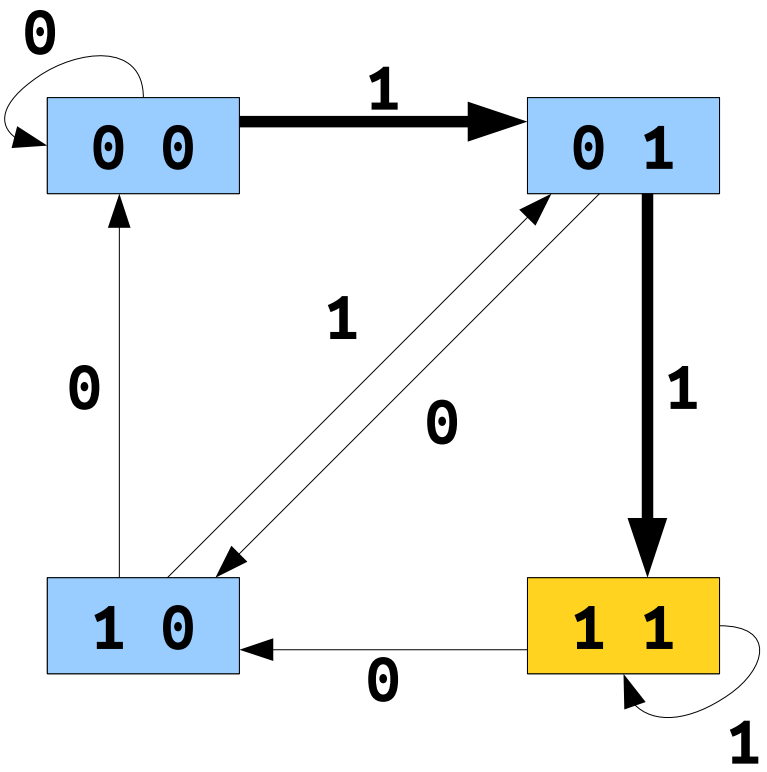


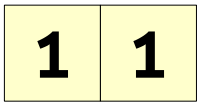
What does an Eulerian circuit in this graph look like?

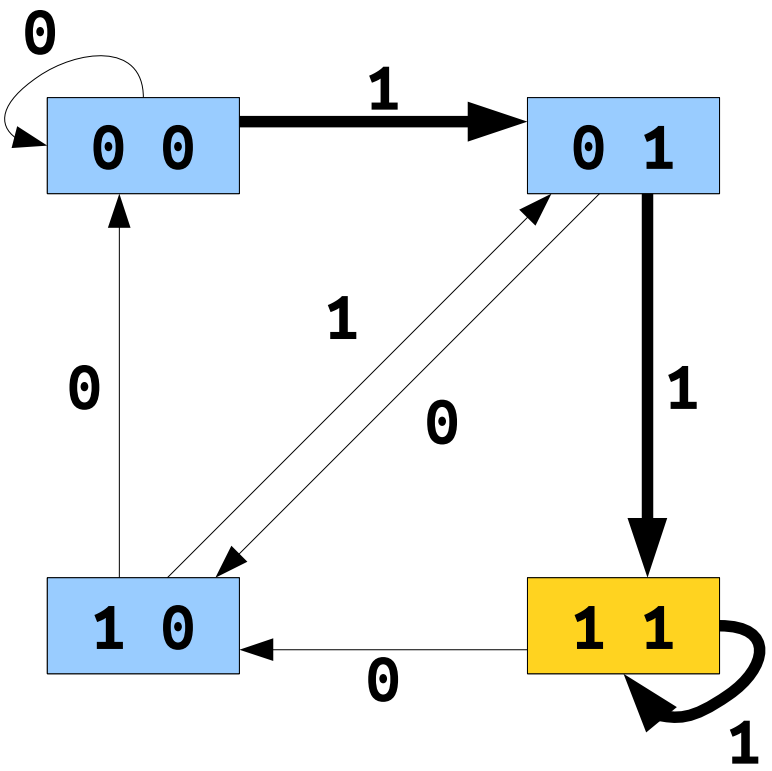

0
0 0
1
0 1
1
0
0
1
1 0
0
1 1
1



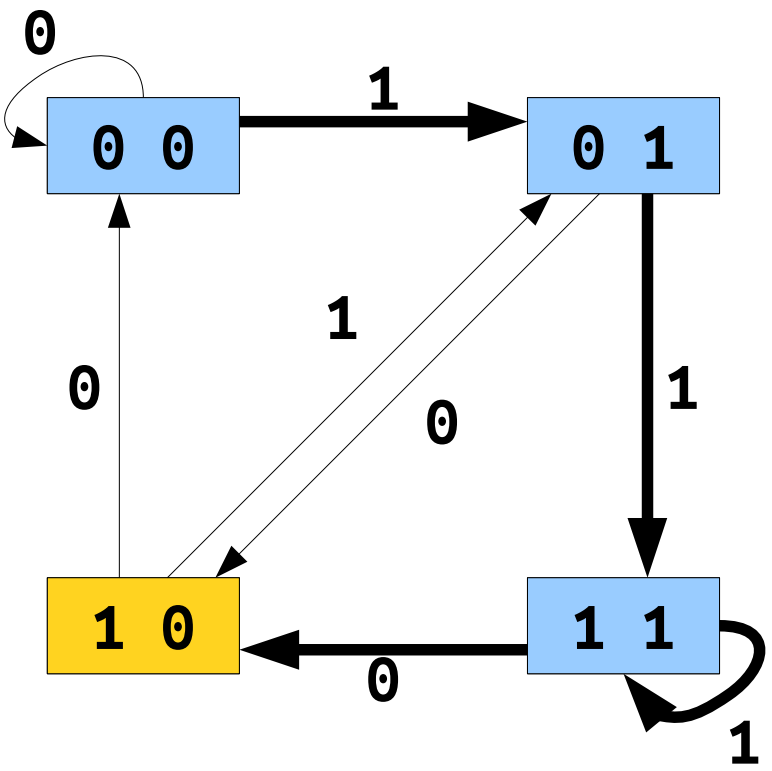
1


0
1
0 0
0 1
1
0
1
0
1 0
1 1
0
1

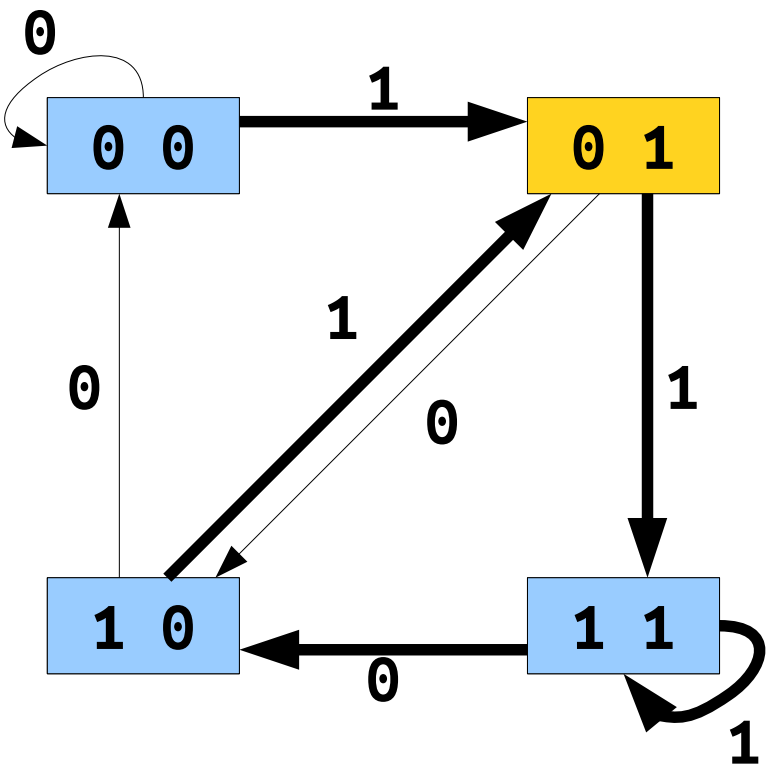

1 1



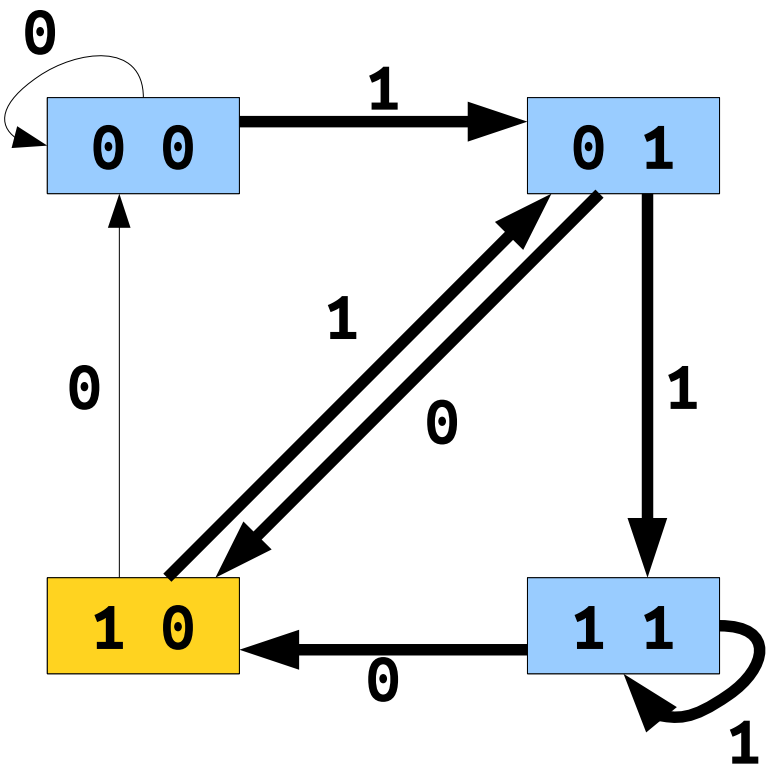
1	1	1



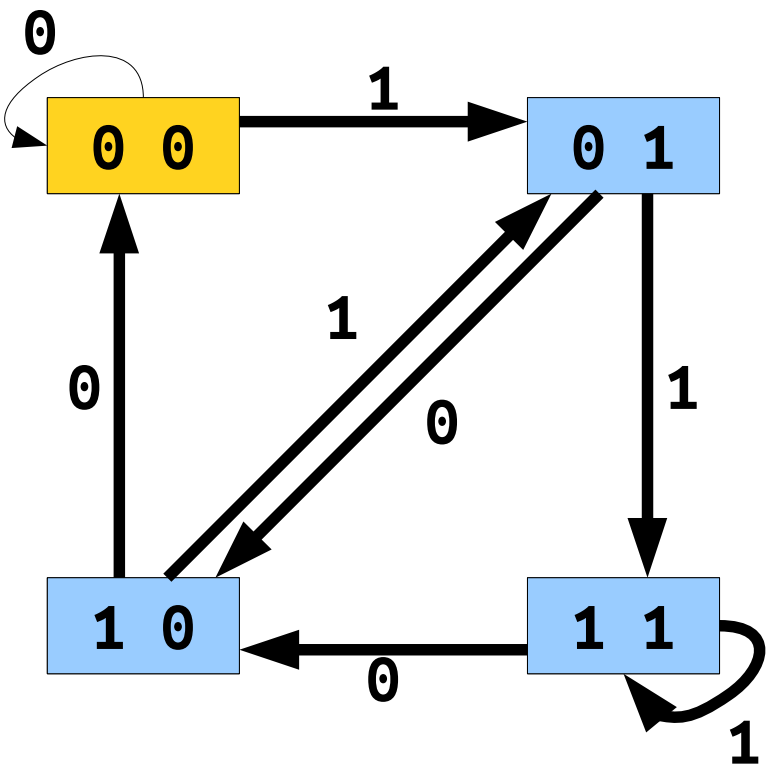
1	1	1	0



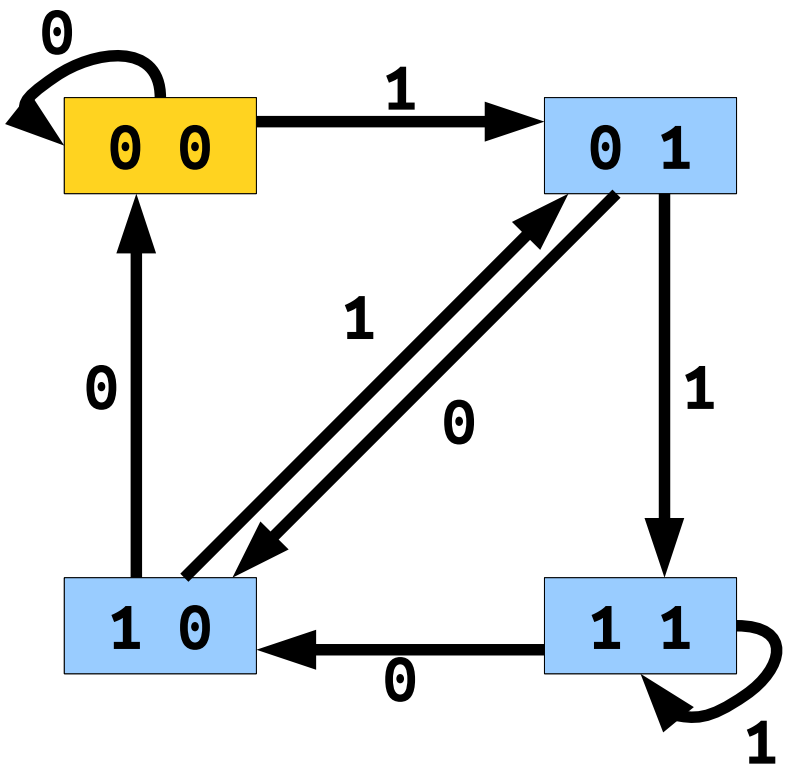
1	1	1	0	1



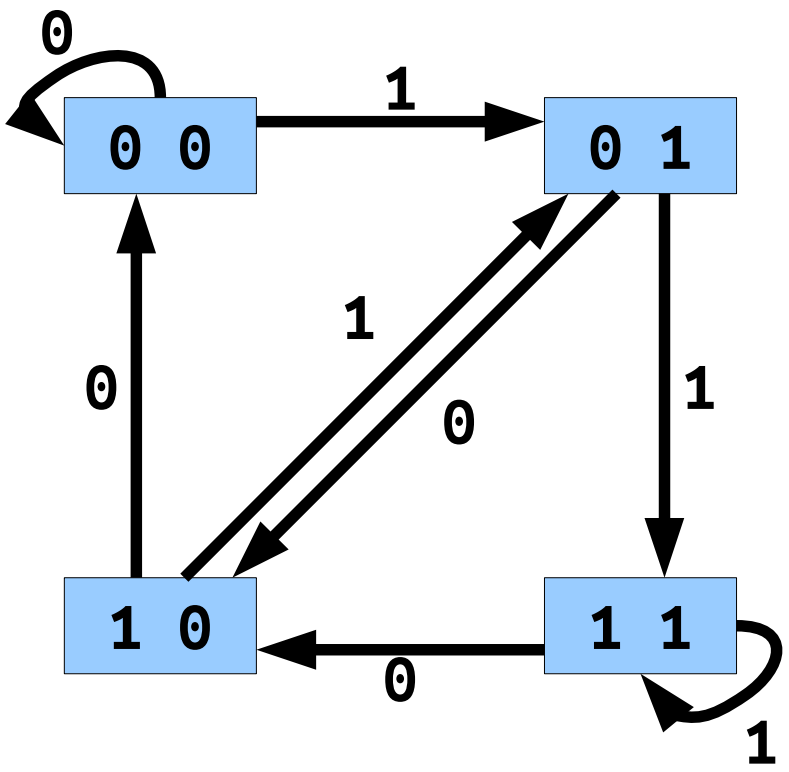
1	1	1	0	1	0



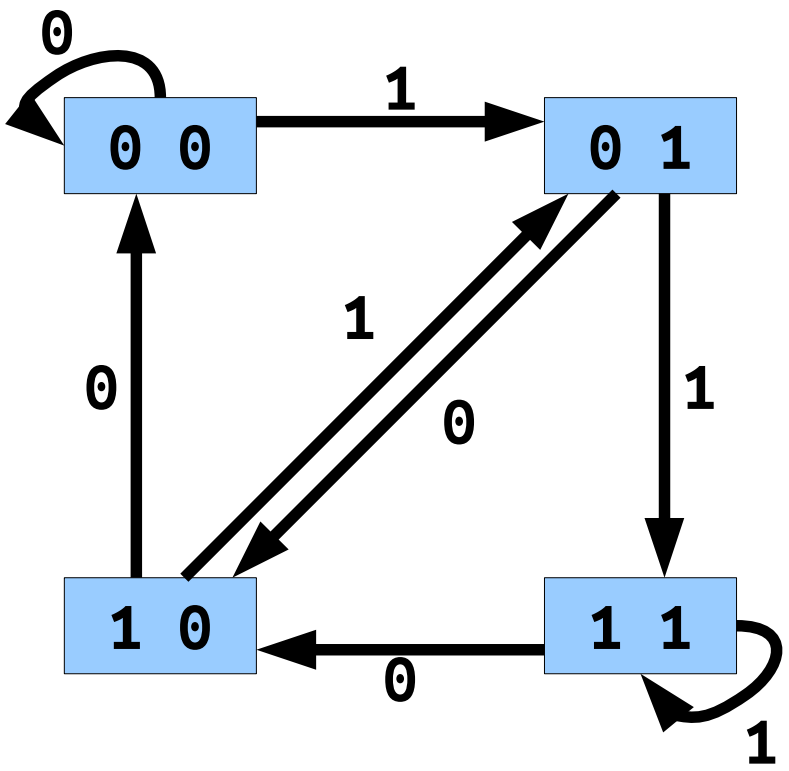
1	1	1	0	1	0	0



1	1	1	0	1	0	0	0

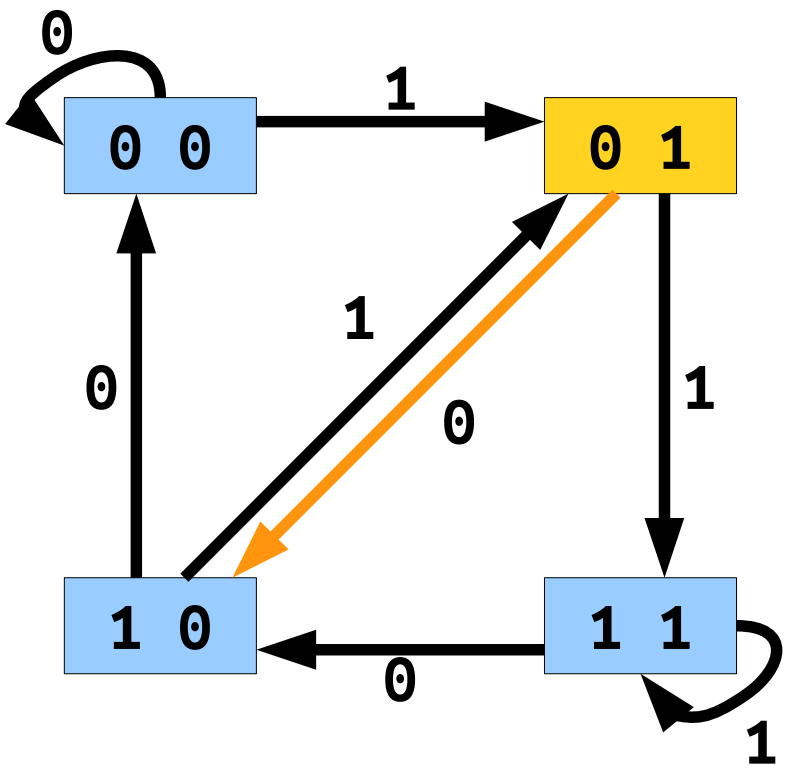


1	1	1	0	1	0	0	0



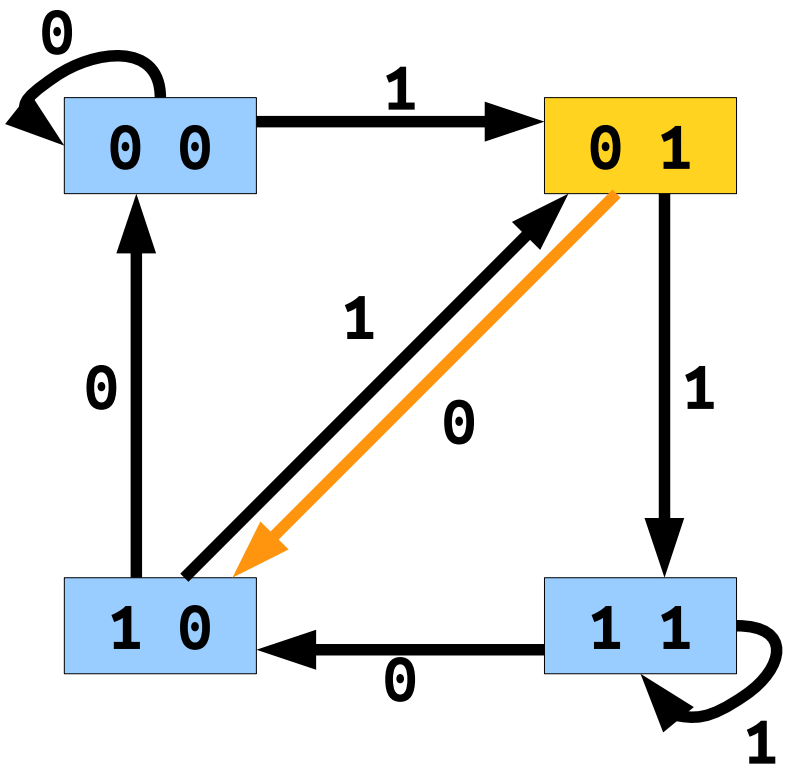
1	1	1	0	1	0	0	0

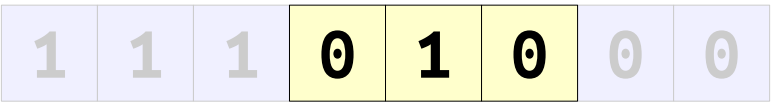
0	1	0

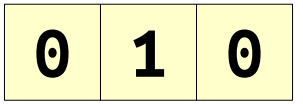


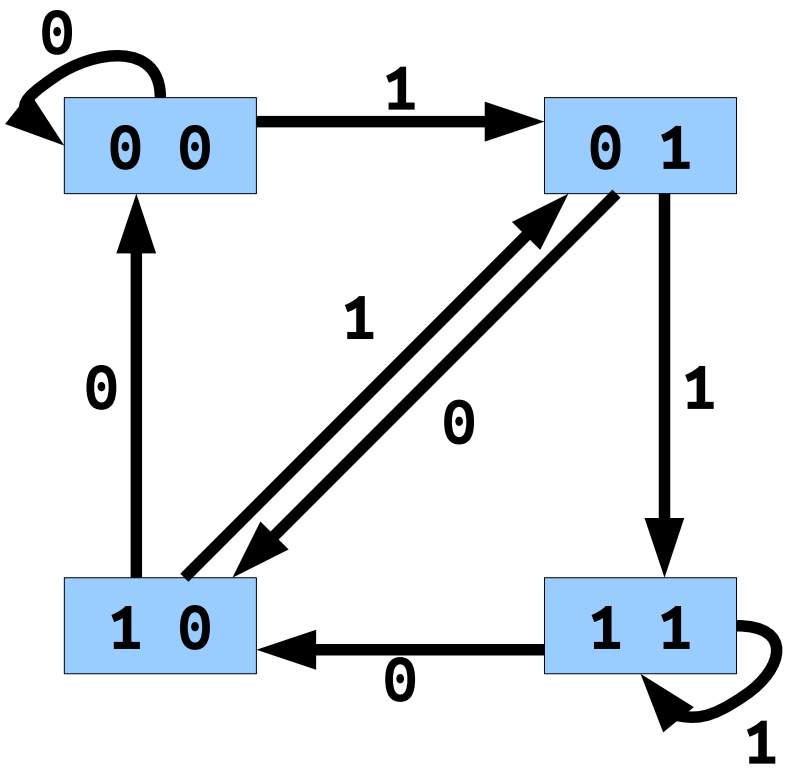
1	1	1	0	1	0	0	0

0	1	0


0
1
0 0
0 1
1
0
1
0
1 0
1 1
0
1

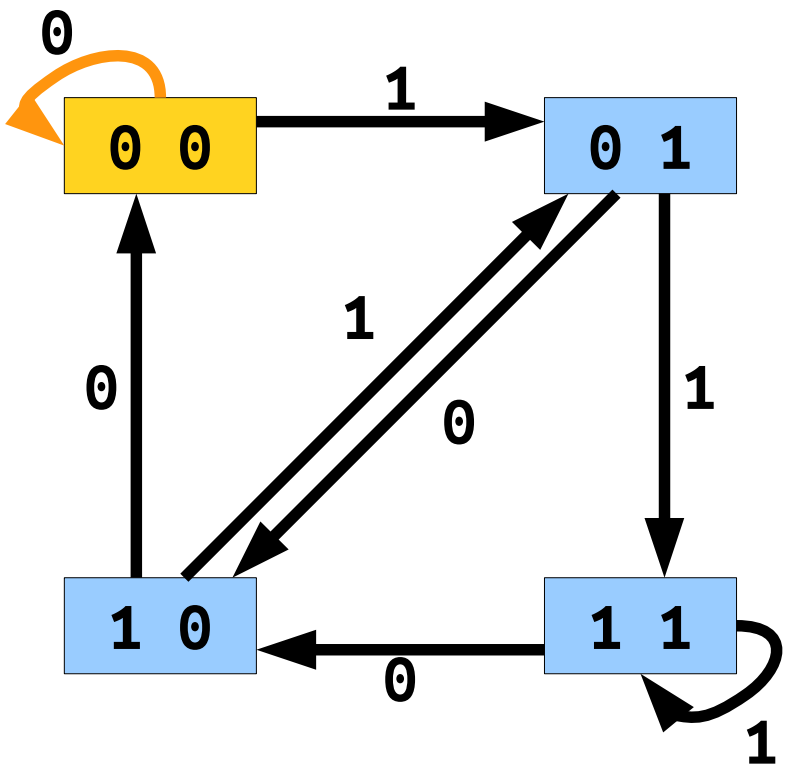

1 1 1 0 1 0 0 0


0 1 0



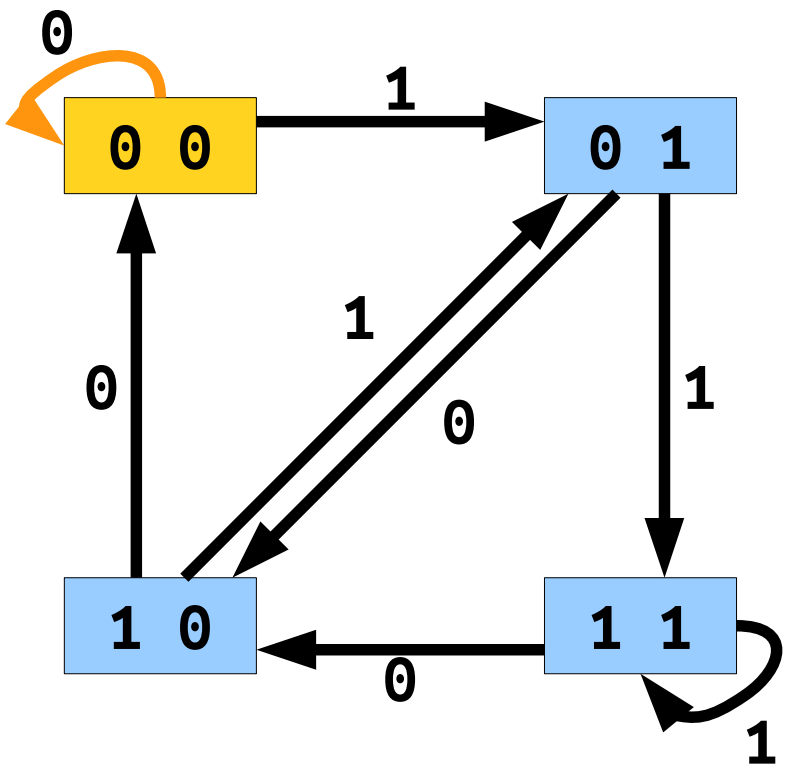
1	1	1	0	1	0	0	0

0	0	0



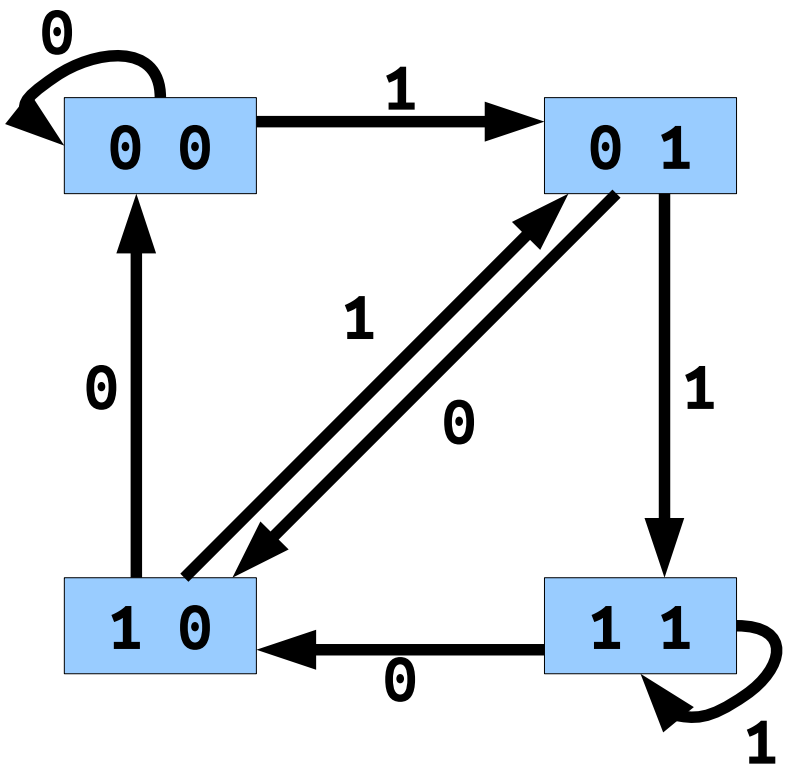
1	1	1	0	1	0	0	0

0	0	0

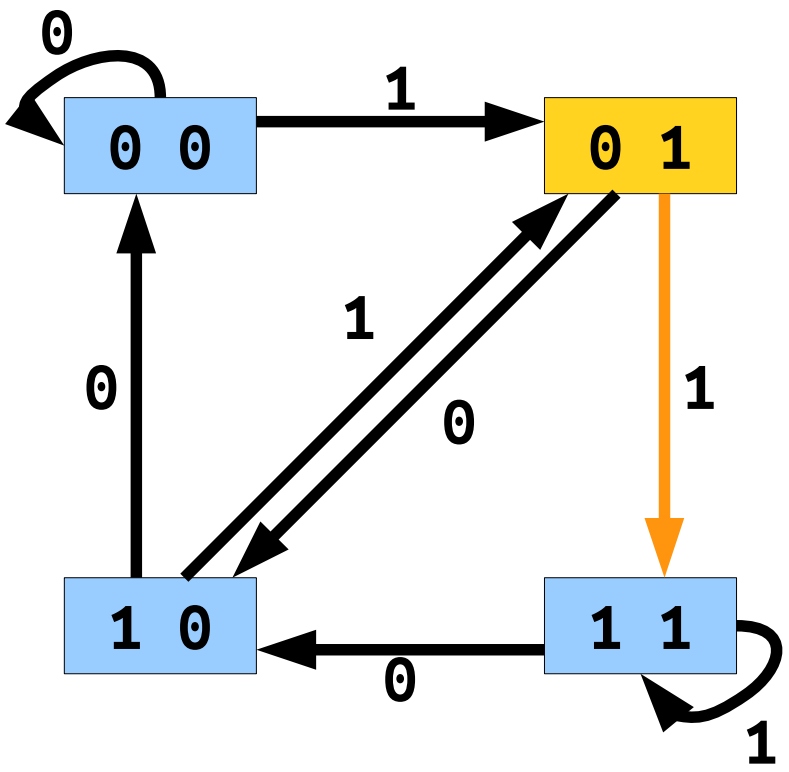


1	1	1	0	1	0	0	0

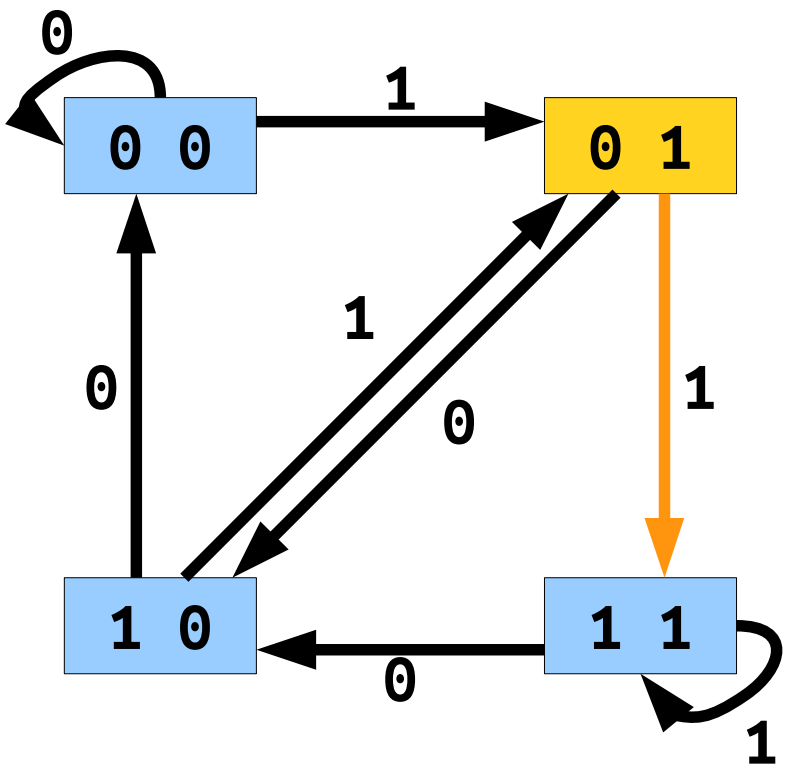
0	0	0



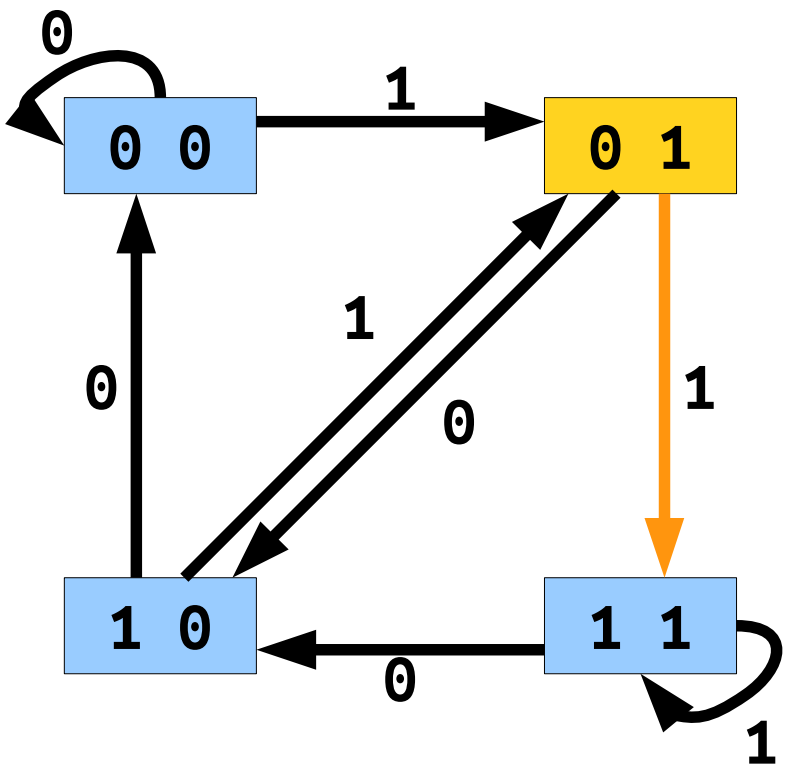
1	1	1	0	1	0	0	0



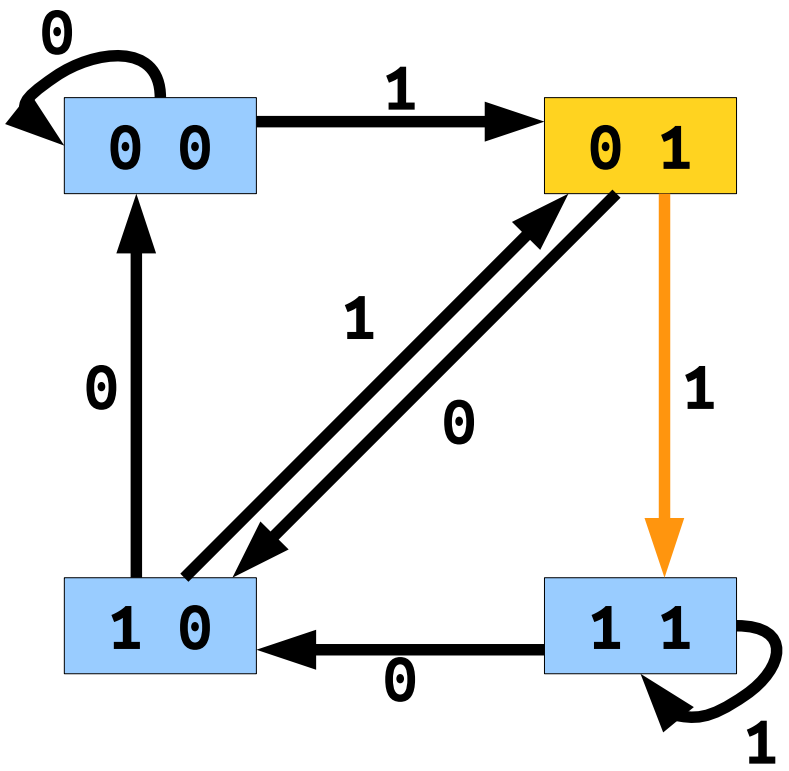
1	1	1	0	1	0	0	0


0
1
0 0
0 1
1
0
0
1
1
1 0
1 1
0
1

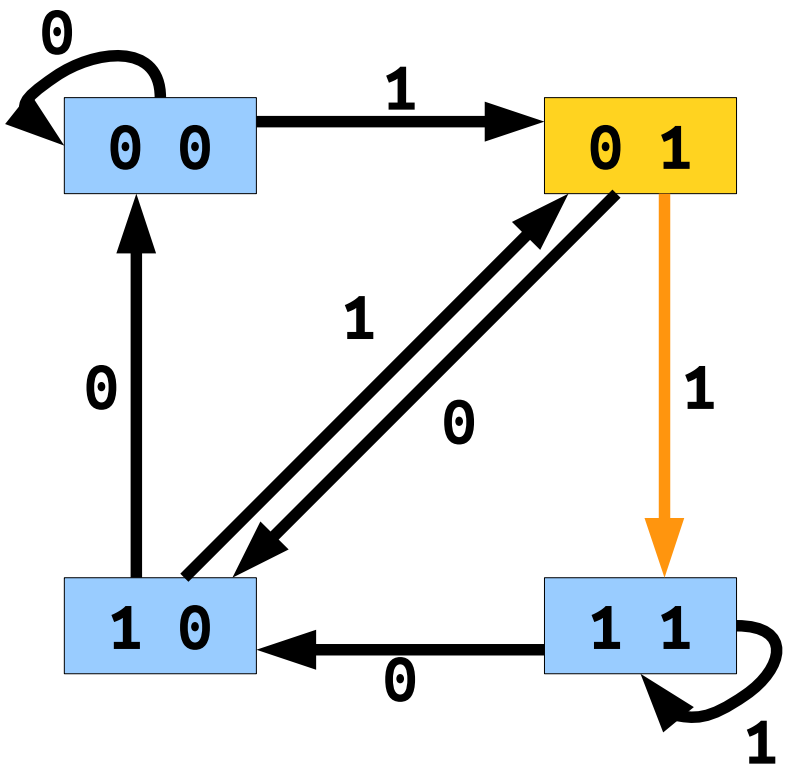

1 1 1 0 1 0 0 0



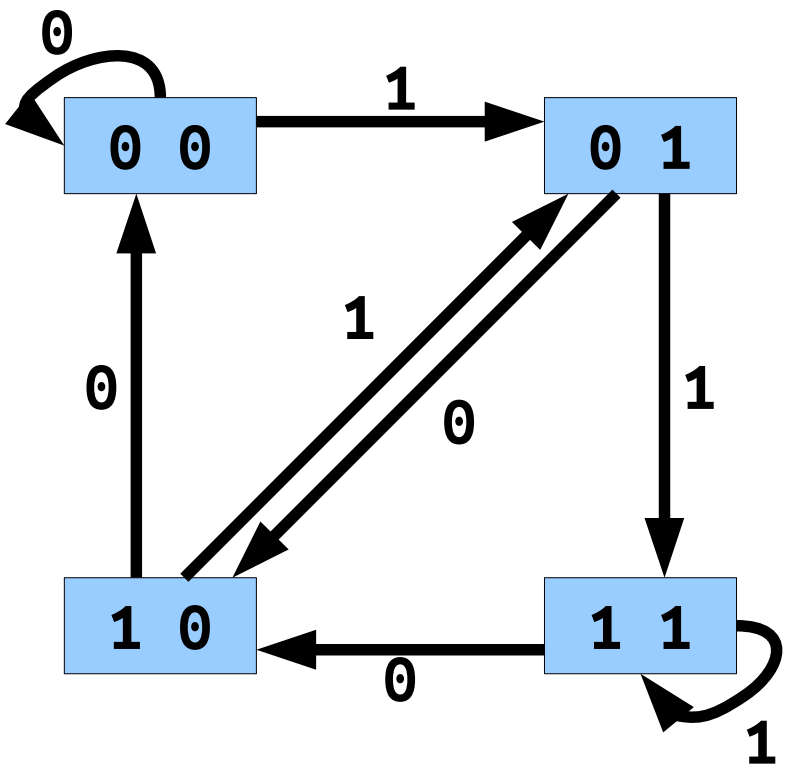
1	1	1	0	1	0	0	0



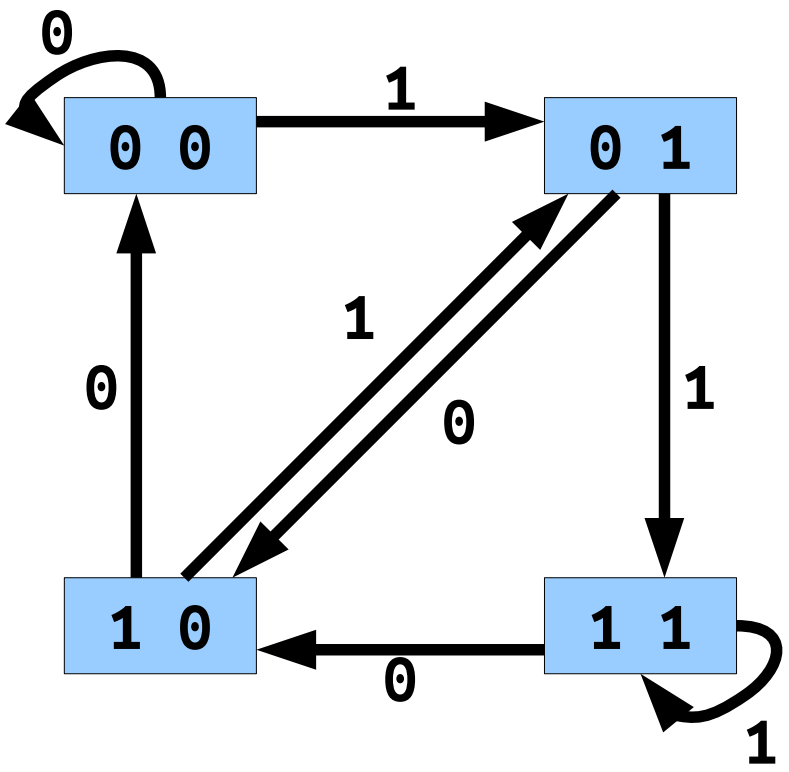
0	0	1	1	1	0	1	0	0	0


0
1
0 0
0 1
1
0
1
0
1
1 0
1 1
0
1


0 0 1 1 1 0 1 0 0 0

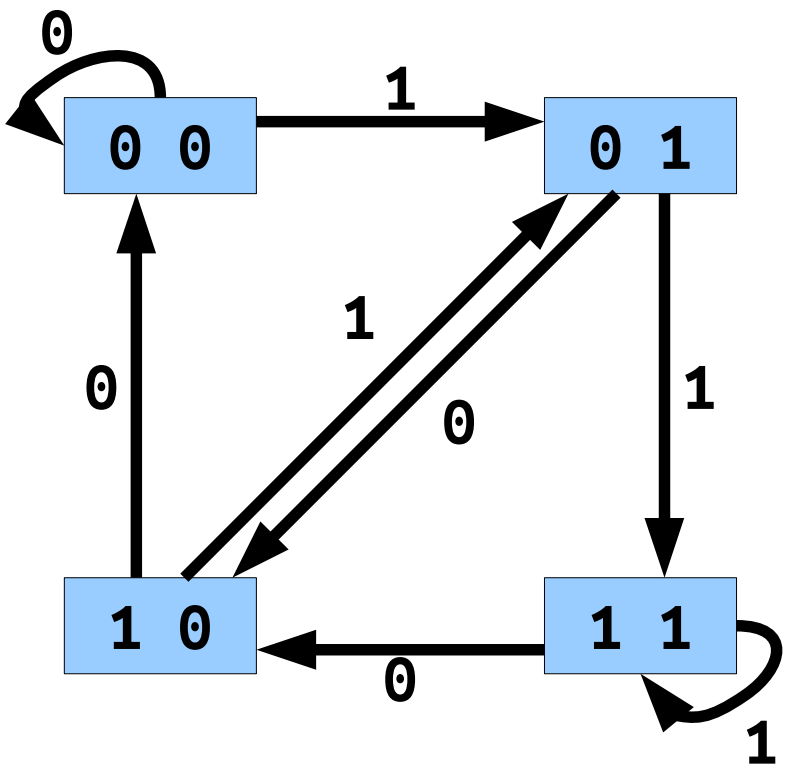


0	0	1	1	1	0	1	0	0	0



0	0	1	1	1	0	1	0	0	0

Every edge in the graph corresponds to a 3-bit code.



0	0	1	1	1	0	1	0	0	0

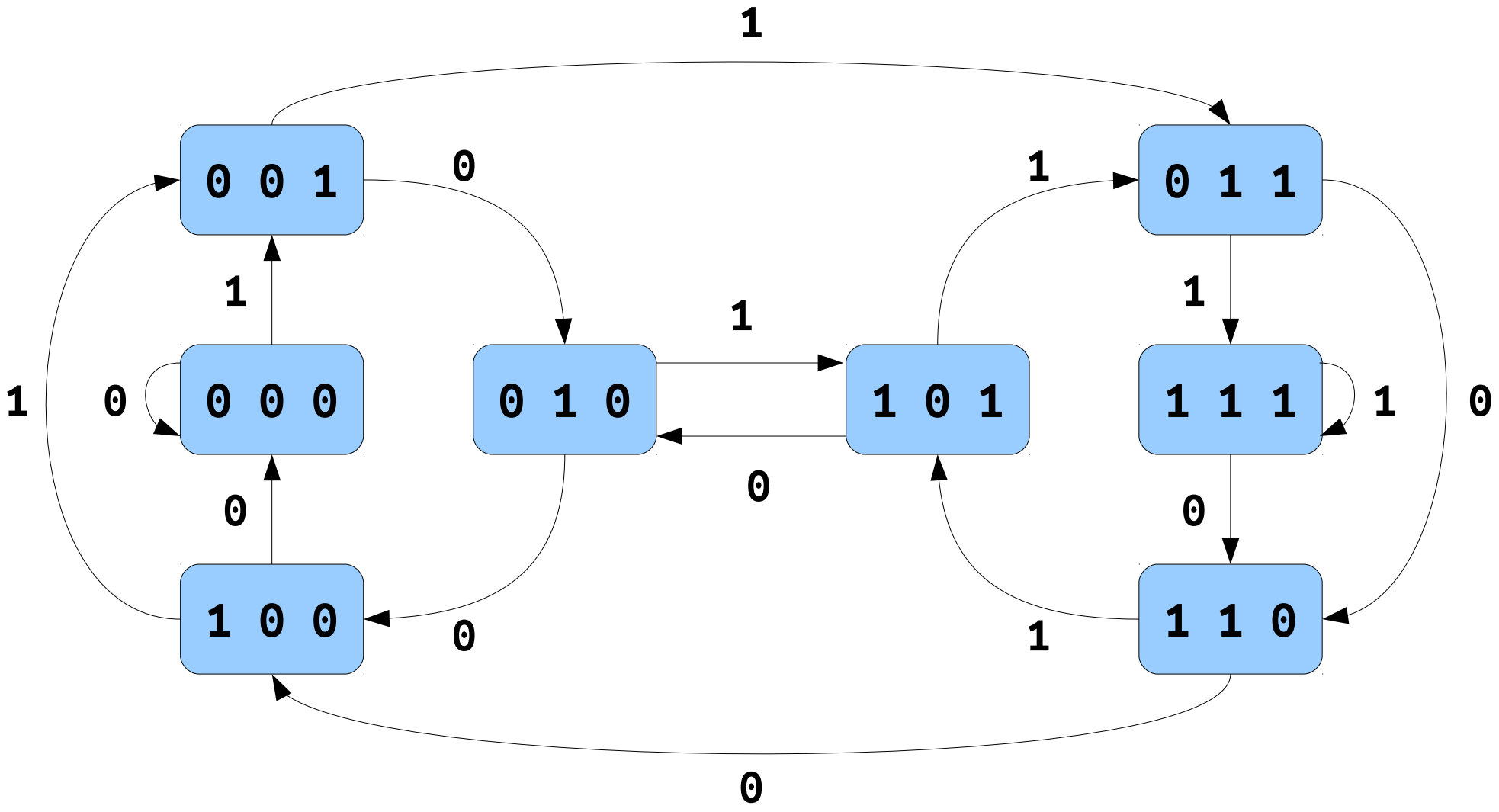
Every edge in the graph corresponds to a 3-bit code.

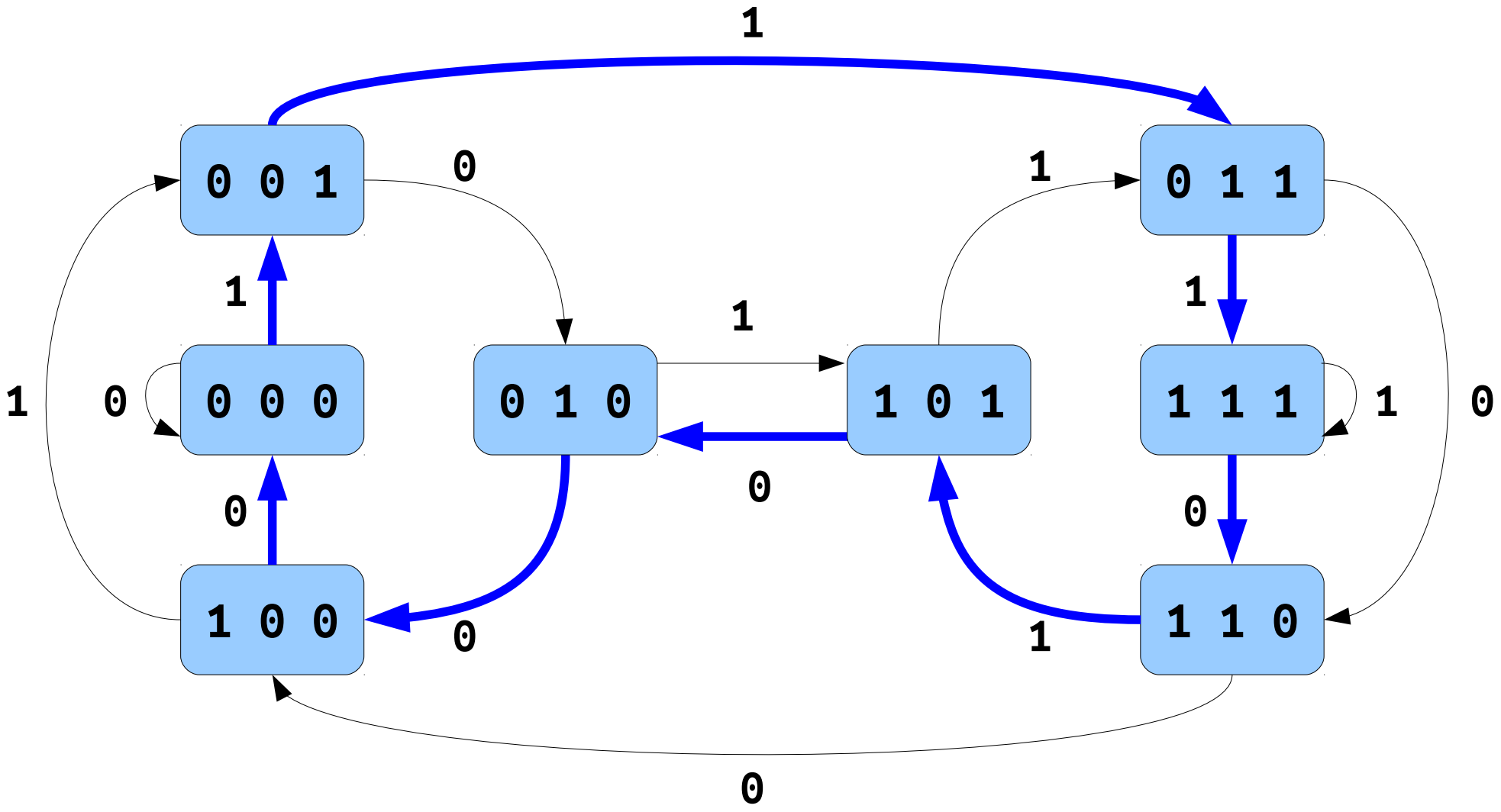
An Eulerian circuit lists all 3-bit codes in the most efficient way possible!

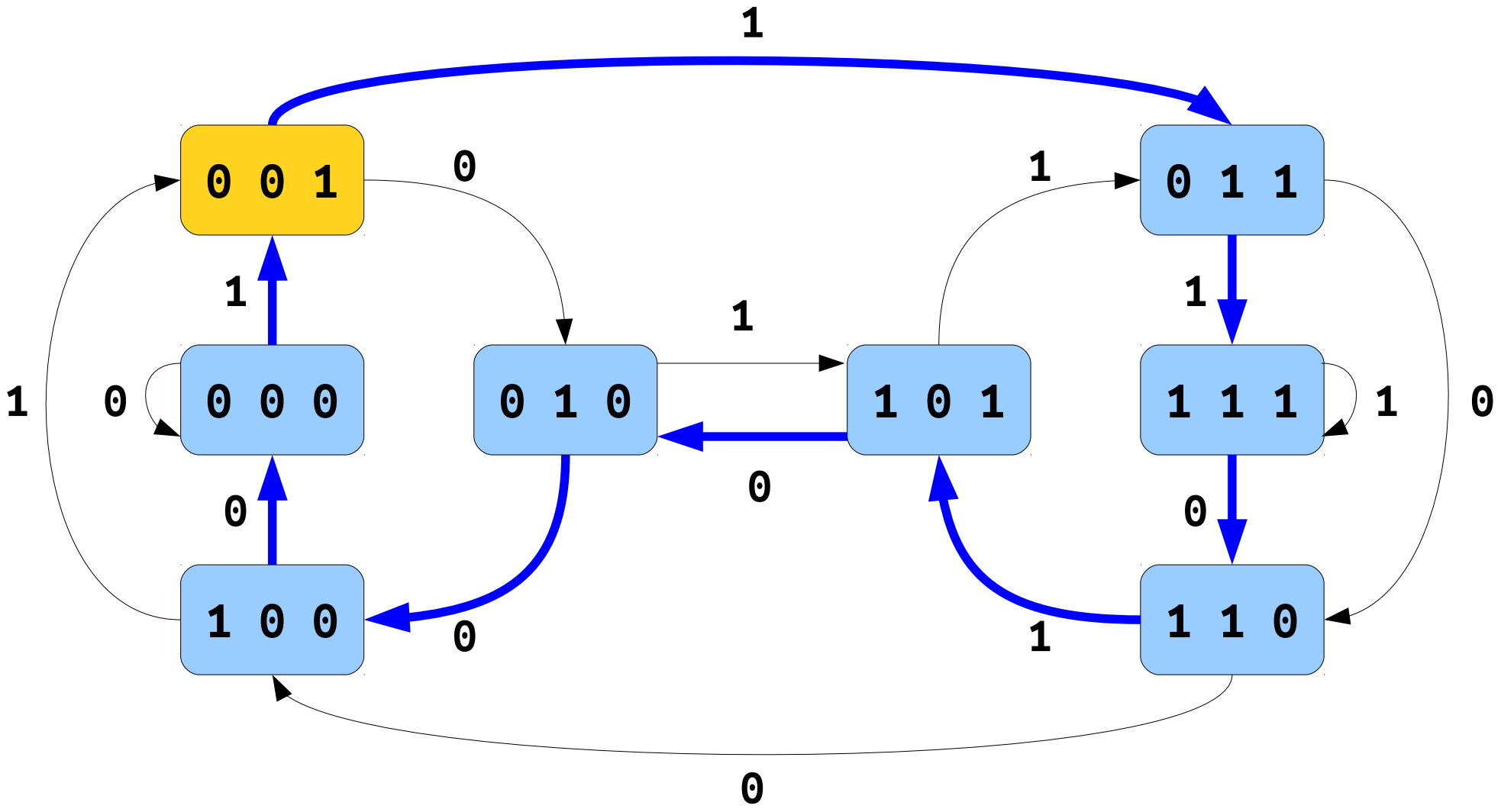
Our Solution

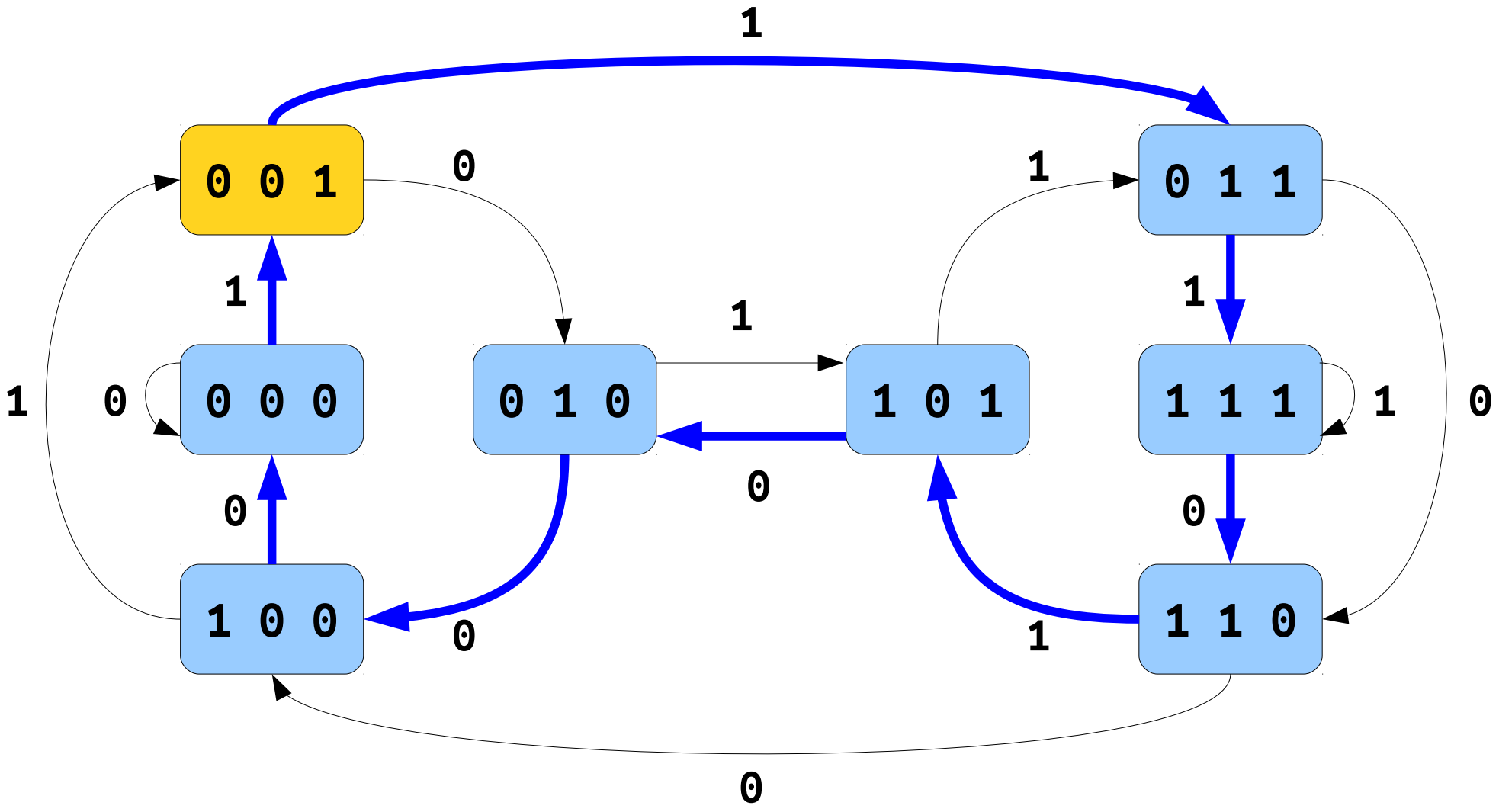
- This graph has one node for each combination of n – 1 bits, where n is the number of bits in the code.
- This means there are 2^{n-1} nodes.
- Each node has outgoing edges: one for 0 and one for 1.
- Length of an Eulerian circuit in this graph: 2^n.
- We need n padding bits (since the first bits assume we've read the first node.)
- Total bits needed: $\mathbf{2^n + n - 1}$.

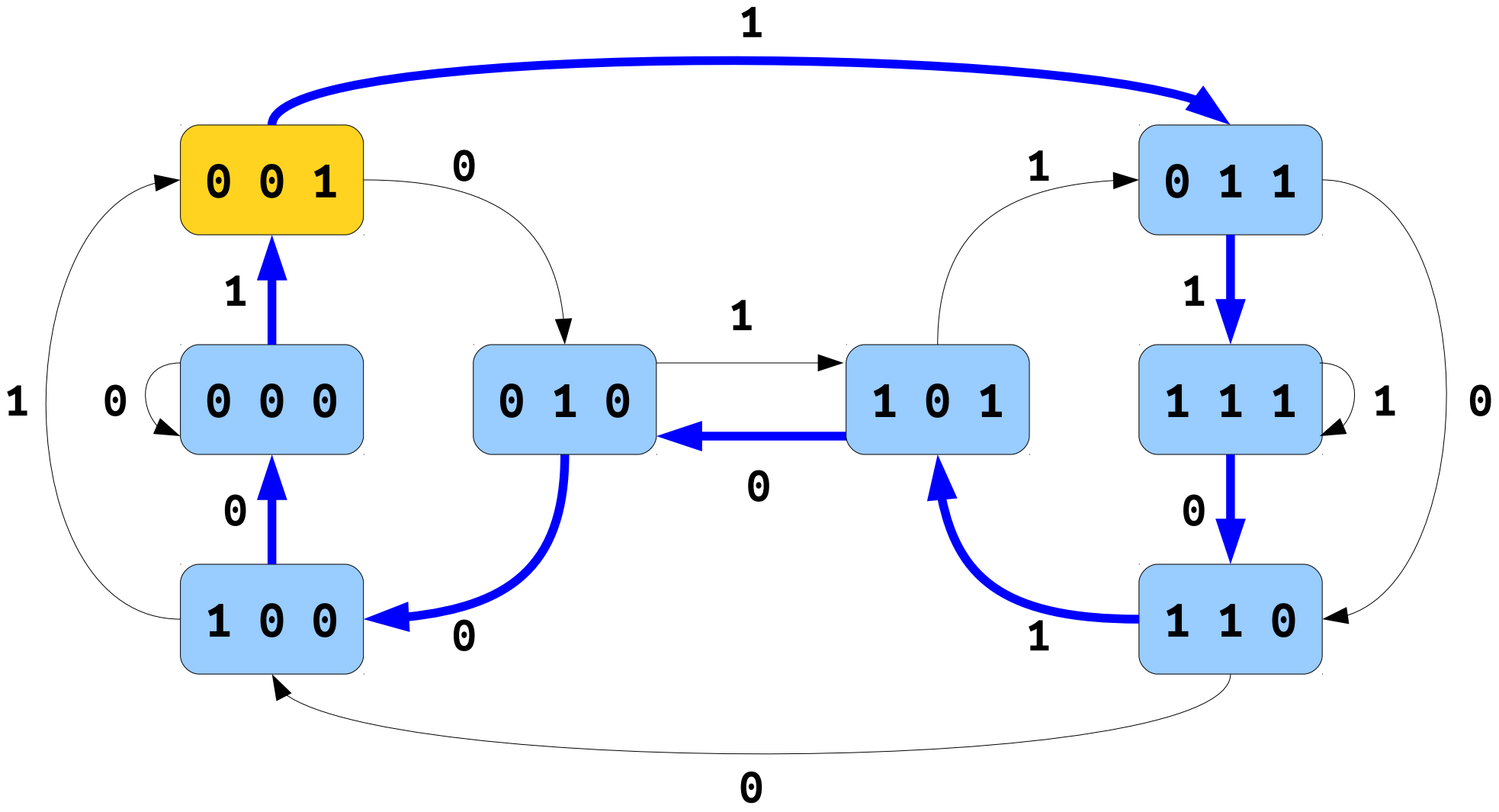
A Different Solution


0 0 1
0 1 1
0 0 0
0 1 0
1 0 1
1 1 1
1 0 0
1 1 0
1
0
1
0
1
1
0
1
1
0
0
1
0
1
0
0


0 0 1
0 1 1
0 0 0
0 1 0
1 0 1
1 1 1
1 0 0
1 1 0
1
0
1
0
1
1
0
1
1
0
0
1
0
0
1
0
0


0 0 1
0 1 1
0 0 0
0 1 0
1 0 1
1 1 1
1 0 0
1 1 0
1
0
1
0
1
1
0
1
1
0
0
1
0
1
0
0


1
0 0 1
0
1
0 1 1
1
1
1
0 0 0
0 1 0
1 0 1
1 1 1
1
0
1
0
0
0
1 0 0
0
1
1 1 0
0
1 1 0 1 0 0 0



0	0	1	1	1	0	1	0	0	0

Hamiltonian Cycles

- A ***Hamiltonian cycle*** is a simple cycle that passes through every node in a graph exactly once.
- A graph is called ***Hamiltonian*** if it contains a Hamiltonian cycle.
- The cycle we saw a second ago is a Hamiltonian cycle, but not an Eulerian circuit.

de Bruijn Graphs

- The graphs we've shown here are called ***de Bruijn Graphs***.
- ***Theorem:*** Every de Bruijn graph has both a Hamiltonian cycle and an Eulerian circuit.
  - An Eulerian circuit of an n-bit de Bruijn graph gives an unlock code for an $(n+1)$-bit lock.
  - A Hamiltonian cycle of an n-bit de Bruijn graph gives an unlock code for an n-bit lock.
- ***Theorem:*** Any n-bit lock that works by shifting can be unlocked with $2^n + n - 1$ bits of input.
  - This is much faster than the $n \cdot 2^n$ bits needed in the naïve solution!

Interesting Fact:

There is a known, simple, efficiently-checkable criterion we can use to test for Eulerian cycles.

But no one knows an simple, efficiently-checkable criterion to test for Hamiltonian cycles!

There is a $1,000,000 prize for solving this problem in general graphs.*

* For a specific definition of “efficiently-checkable.”

Where We're Going

- **Finite Automata**
  - We just modeled the state of the password system as a graph with nodes and edges labeled with characters. Turns out, this is a *great* way to define computers!
- **Complexity Theory**
  - Why would anyone want to pay $1,000,000 for an efficient way to check if a graph has a Hamiltonian cycle?