

Logic and Computability

Lecture 5



Introduction to Z3

Bettina Könighofer

bettina.koenighofer@iaik.tugraz.at

Stefan Pranger

stefan.pranger@iaik.tugraz.at

What is Z3?

- Solver for Satisfiability Modulo Theories

What is Z3?

- Solver for **Satisfiability Modulo Theories**
 - we know how to check satisfiability ✓
 - ... until now: Only propositional logic!
- Z3 allows us to efficiently answer decision problems including
 - Integers, Reals, Arithmetic
 - BitVectors, uninterpreted Functions, Arrays,
 - etc.
- More on Theories starting from next week 🕒
- Today: Basics Principles of Z3 and First Problems

Background

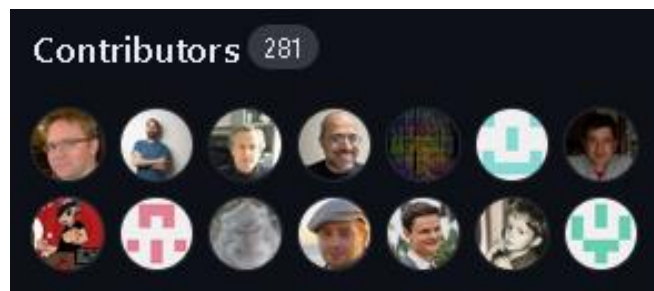
- Developed by Microsoft Research
 - <https://github.com/Z3Prover/z3>

Christoph
Wintersteiger

Lev
Nachmanson

Leonardo
de Moura

Nikolaj
Bjørner



- SMT-LIB2 – A standardized language for Problems in SMT

Principles

- Is $\neg a \wedge (a \vee b)$ satisfiable?
- What do we need to describe a problem for the solver?
 - Variables (of a specific Sort),
`(declare-const a Bool)`
`(declare-const b Bool)`
 - Constraints, and
`(assert (not a))`
`(assert (or a b))`
 - Checking for Satisfiability
`(check-sat)`

A Simple Example in SMT-LIB2

```
(declare-const a Bool)
(declare-const b Bool)
(assert (not a) )
(assert (or a b) )
(check-sat)
(get-model)
```

Background

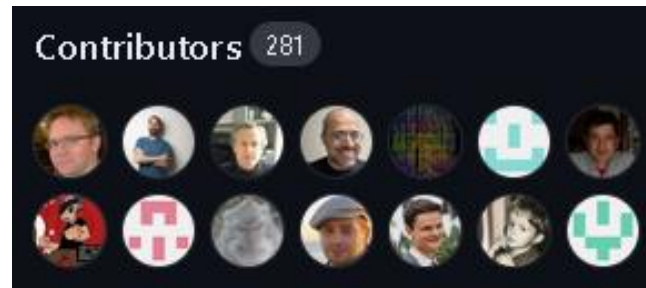
- Developed by Microsoft Research
 - <https://github.com/Z3Prover/z3>

Christoph
Wintersteiger

Lev
Nachmanson

Leonardo
de Moura

Nikolaj
Bjørner



- SMT-LIB2 – A standardized language for Problems in SMT
- API for C++, Python, Julia, etc.

Installing

- We will use the Python API:
 - `pip install z3-solver`
- Optionally, you may install z3 natively:
 - `sudo apt-get install z3` (Via aptitude for Ubuntu, etc.)
 - <https://www.nuget.org/packages/Microsoft.Z3/> (Windows)
 - <https://jfmc.github.io/z3-play> (online)

Python API

- User-friendly interface for `SMT-LIB2`
- Used in the Programming Assignment

- Variables (of a specific Sort),

```
(declare-const a Bool)  
(declare-const b Bool)
```



```
a = Bool("a")  
b = Bool("b")
```

- Constraints, and

```
(assert (not a) )  
(assert (or a b) )
```



```
solver = Solver()  
solver.add(Not(a) )  
solver.add(Or(a,b) )
```

- Checking for Satisfiability

```
(check-sat)
```



```
solver.check()
```

Python API

```
from z3 import *

a, b = Bools("a b")

solver = Solver()
solver.add(Not(b))
solver.add(Or(a,b))

print(solver.sexpr())
result = solver.check()
model = solver.model()
print(result)
print(model)
```

Python API

- Constraints

```
(assert (not a) )
```

```
(assert (or a b) )
```



```
solver.add(Not(a))
```

```
solver.add(Or(a,b))
```

- Provides Methods for Connectives:

- **And()**, **Or()**, **Not()**, **Implies()**, **==**, **^**, etc.

- Method to check whether two statements can be distinct:

- **Distinct(a,b)**

- Operator overloading:

- **+**, **-**, **>>**, **<<**, etc.

- Reference: <https://z3prover.github.io/api/html/namespacez3py.html>

A First Example

- We want to show that the following statements are equal:
 - $p \rightarrow q$
 - $\neg p \vee q$

A First Example

- $p \rightarrow q \iff \neg p \vee q$?

```
from z3 import *
```

```
solver = Solver()
```

```
a, b = Bools("a b")
```

```
l, r = Bools("l r")
```

```
solver.add(l == Implies(a, b))
```

```
solver.add(r == Or(Not(a), b))
```

```
solver.add(Distinct(r, l) )
```

```
result = solver.check()
```

```
print(result)
```

Back to SMT-LIB2

- $p \rightarrow q \iff \neg p \vee q$?

```
from z3 import *
```

```
solver = Solver()
```

```
a, b = Booleans("a b")
```

```
l, r = Booleans("l r")
```

```
solver.add(l == Implies(a, b))
```

```
solver.add(r == Or(Not(a), b))
```

```
solver.add(Distinct(r, l))
```

```
print(solver.sexpr())
```

```
result = solver.check()
```

```
print(result)
```

BitVectors

- Z3 allows us to use so-called *theories*
- We have a first look at bitvectors
- Syntax:
 - `bv = BitVector("bv", <size>)`
- BitVectors respect under-/overflow behaviour!
 - In contrast to Z3's integers

Operations on BitVectors

- The BitVector Sort respects overloaded operators:
 - $<$, $>$, $<=$, $+$, $-$, $<<$, $>>$, $/$, etc.
 - Caution: These are signed interpretations
 - Use `ULT`, `UGT`, `ULE` for unsigned interpretations

Equivalence Checking for BitVectors

- We want to prove the equivalence of the following
 - $((y \& x) * -2) + (y + x)$
 - $x \oplus y$

Weird XOR

```
from z3 import *

x = BitVec('x', 32)
y = BitVec('y', 32)

output = BitVec('output ', 32)

s = Solver()
s.add(x^y==output)
s.add(Distinct(((y & x) * -2) + (y + x), output))

print(s.check())
```

Operations on BitVectors

- The BitVector Sort respects overloaded operators:
 - `<`, `>`, `<=`, `+`, `-`, etc.
 - Caution: These are signed interpretations
 - Use `ULT`, `UGT`, `ULE` for unsigned interpretations
- Overflow and Underflow
 - `BVAddNoOverflow`, `BVAddNoUnderflow`
 - `BVMulNoOverflow`, `BVMulNoUnderflow`

Overflow Behaviour

- We want to check whether the statement TODO
 - $(x + 1 < x - 1)$

Variables in a Satisfying Model

- Variables and Expressions are stored in z3-specific classes
- We can use `solver.model().decls()` to iterate through all declared variables
 - Use `.as_long()` to convert a BitVector to a Python Integer

```
model = solver.model()
for var in solver.model.decls():
    print(f"{var}: {model[var]} (: {type(model[var])})")
```

Overflow Behaviour

- We want to check whether the statement `TODO`
 - `(x + 1 < x - 1)`
- We need to add
 - `BVNoOverflow(x, 1, True)`
 - `BVNoUnderflow(x, 1, True)`
- Functions that evaluate to `False` when Over-/Underflow would occur in the model

Assignment Sheet

- 4 Exercises + 1 Bonus Exercise
- You are allowed to work in groups of 2
 - If you do so, please add your information into the README
- Deadline: 05. 06. 2024

Outline – Part II

- IntSort + Z3 Built-in Sorts
- Quantifiers
- Custom Sorts
- Uninterpreted Functions

Working with Integers

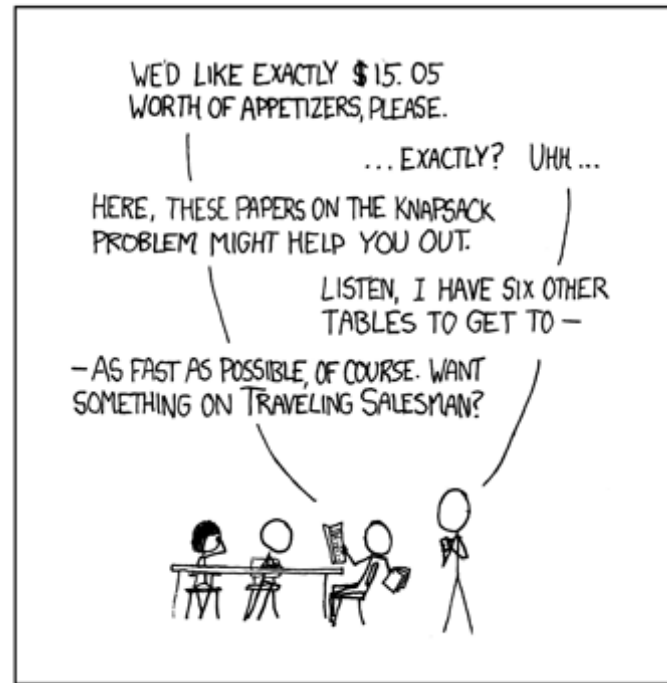
- IntSort
 - <, >, <=, ==, +, -, etc.

Working with Integers

- IntSort
 - $<$, $>$, $<=$, $=$, $+$, $-$, etc.

MY HOBBY:
EMBEDDING NP-COMPLETE PROBLEMS IN RESTAURANT ORDERS

CHOTCHKIES RESTAURANT	
~ APPETIZERS ~	
MIXED FRUIT	2.15
FRENCH FRIES	2.75
SIDE SALAD	3.35
HOT WINGS	3.55
MOZZARELLA STICKS	4.20
SAMPLER PLATE	5.80
~ SANDWICHES ~	
BARBECUE	6.55



Example

```
#!/usr/bin/python3

from z3 import *

a,b,c,d,e,f = Ints('a b c d e f')
s = Solver()
s.add(215*a + 275*b + 335*c + 355*d + 420*e + 580*f == 1505,
a>=0, b>=0, c>=0, d>=0, e>=0, f>=0)
result = s.check()
if result == sat:
    print(s.model())
```

Variables in a Satisfying Model

- Variables and Expressions are stored in z3-specific classes
- We can use `solver.model().decls()` to iterate through all declared variables
 - Use `.as_long()` to convert a BitVector, Int, Real, etc. to a Python Integer

Example contd.

```
results=[]
while True:
    if s.check() == sat:
        m = s.model()
        print(m)
        results.append(m)
        block = [a != m[a].as_long(), b != m[b].as_long(), c != m[c].as_long(), d !=
m[d].as_long(), e != m[e].as_long(), f != m[f].as_long()]
        """
        #Different approach: Iterate over all entries in the model
        block = []
        for d in m.decls():
            print(d, type(d), d(), type(d()), m[d], type(m[d]))
            c = d()
            block.append(c != m[d].as_long())
        """
        s.add(Or(block))
    else:
        print ("All results enumerated, total=", len(results))
        break
```

Z3 Built-in Sorts

- BoolSort, BitVecSort, IntSort, RealSort
- Sequences, Strings
- Arrays

Assignment Sheet

- 4 Exercises + 1 Bonus Exercise
- You are allowed to work in groups of 2
 - If you do so, please add your information into the README
- Deadline: 05. 06. 2024 – **12 days**

Assignment Sheet - Interviews

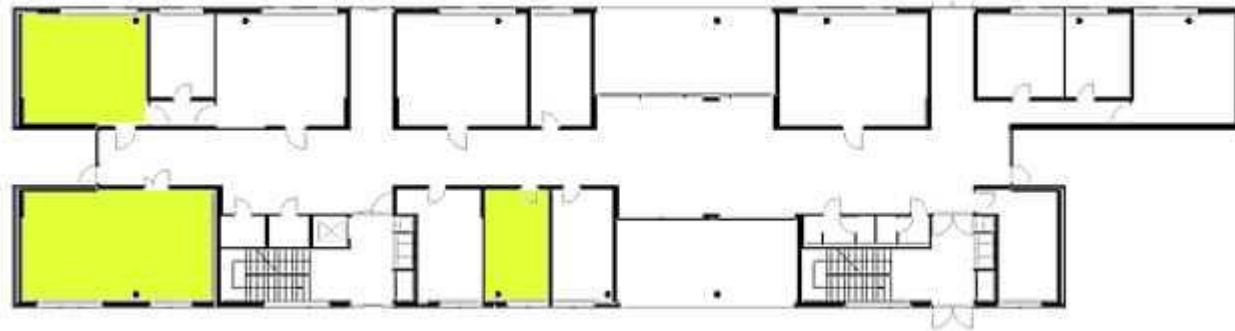
- 10 Min Interview Slots
- When? 06. 06. 12:00 – 17:00 – Slots via TeachCenter

Assignment Sheet - Interviews

- 10 Min Interview Slots
- When? 06. 06. 12:00 – 17:00 – Slots via TeachCenter
- What? Discussion with (*one of your*) tutor or Stefan – Bring your Laptop!

Assignment Sheet - Interviews

- 10 Min Interview Slots
- When? 06. 06. 12:00 – 17:00 – Slots via TeachCenter
- What? Discussion with (*one of your*) tutor or Stefan – Bring your Laptop!
- Where? IAIK Ground Floor



Quantifiers

- Z3 offers `ForAll()` and `Exists()`
- Usage: `ForAll(<vars>, <formula>)`

Example

► 15. [M26] J. H. Quick noticed that $((x + 2) \oplus 3) - 2 = ((x - 2) \oplus 3) + 2$ for all x . Find all constants a and b such that $((x + a) \oplus b) - a = ((x - a) \oplus b) + a$ is an identity.

Example

► 15. [M26] J. H. Quick noticed that $((x + 2) \oplus 3) - 2 = ((x - 2) \oplus 3) + 2$ for all x . Find all constants a and b such that $((x + a) \oplus b) - a = ((x - a) \oplus b) + a$ is an identity.

- We want to use Z3 to find all constants , s.t.
 - $\forall x ((x + a) \oplus b) - a = ((x - a) \oplus b) + a$

Example

- $$\forall x \left((x + a) \oplus b \right) - a = \left((x - a) \oplus b \right) + a$$

```

from z3 import *
s = Solver()
a, b = BitVecs('a b', 4)
x = BitVec('x', 4)
lhs, rhs = BitVecs('lhs rhs', 4)
s.add(lhs == ((x+a)^b)-a)
s.add(rhs == ((x-a)^b)+a)
s.push()
s.add(ForAll(x, lhs == rhs))
results=[]
while True:
    if s.check() == sat:
        m = s.model(); results.append(m)
        block = [a != m[a].as_long(), b != m[b].as_long()]
        s.add(Or(block))
    else:
        print ("results total=", len(results))
        break

```

Example

- $\forall x ((x + a) \oplus b) - a = ((x - a) \oplus b) + a$
 - Let's also use Z3 to find constants such that the equality does not hold

Example

- $\forall x ((x + a) \oplus b) - a = ((x - a) \oplus b) + a$
 - Let's also use Z3 to find constants such that the equality does not hold
- Use `solver.push()` and `solver.pop()` to store and restore solver states

Example

- $\forall x \left((x + a) \oplus b \right) - a = \left((x - a) \oplus b \right) + a$
 - Let's also use Z3 to find constants such that the equality does not hold

```

from z3 import *
s = Solver()
a, b = BitVecs('a b', 4)
x = BitVec('x', 4)
s.add(lhs == ((x+a)^b)-a)
s.add(rhs == ((x-a)^b)+a)
s.push()
s.add(ForAll(x, lhs == rhs))
. . .
s.pop()
s.add(Exists(x, lhs != rhs))
result = s.check()
print(result)
print(s.sexpr())
if result == sat:
    print(s.model())

```

Custom Sorts – Datatypes

- Beyond the built-in Sorts
- Datatypes allow us to define more complex data structures, e.g. lists, trees, etc.

Custom Sorts – Datatypes

- Beyond the built-in Sorts
- Datatypes allow us to define more complex data structures , e.g. lists, trees, etc.

- Simple Case: Enum

```
ColoursDatatype = Datatype("Colour")
ColoursDatatype.declare("RED")
ColoursDatatype.declare("GREEN")
ColoursDatatype.declare("BLUE")
ColoursDatatype.declare("MAGENTA")
ColoursSort = ColoursDatatype.create()

x = Const("x", ColoursSort)
```

Uninterpreted Functions

- Generally, we have:
 - $f: A_0 \times \dots \times A_n \rightarrow B$
 - f maps values from $A_0 \times \dots \times A_n$ to B

Uninterpreted Functions

- Generally, we have:
 - $f: A_0 \times \dots \times A_n \rightarrow B$
 - f maps values from $A_0 \times \dots \times A_n$ to B
- Uninterpreted Functions have no know “structure”
 - Z3 decides the output based on the constraint
 - f can be seen as a lookup-table

Uninterpreted Functions

- Generally, we have:
 - $f: A_0 \times \dots \times A_n \rightarrow B$
 - f maps values from $A_0 \times \dots \times A_n$ to B
- Uninterpreted Functions have no know “structure”
 - Z3 decides the output based on the constraint
 - f can be seen as a lookup-table
- `f = Function('f', IntSort(), IntSort())`
- `f = Function('f', IntSort(), ColoursSort)`

1D – Colouring Problem

```
from itertools import combinations
from z3 import *

solver = Solver()

Colours = Datatype("Colours")
...

Colour = Colours.create()
f      = Function('f', IntSort(), Colour)

variables = list()
for i in range(0,5):
    variables.append(Int(i))
    solver.add(0 <= variables[-1])
    solver.add(variables[-1] <= 5)
solver.add(Distinct(variables))

for combi in combinations(variables,2):
    solver.add(Implies(Abs(combi[0] - combi[1]) == 1, f(combi[0]) != f(combi[1])))

result = solver.check()
if result == sat:
    print(solver.model())
```

Z3 Check Results

- So far we have only used `solver.model()`
 - Only available if assumptions are satisfiable

Z3 Check Results

- So far we have only used `solver.model()`
 - Only available if assumptions are satisfiable
- Can we use insights from `unsat` results?

Z3 Check Results

- So far we have only used `solver.model()`
 - Only available if assumptions are satisfiable
- Can we use insights from `unsat` results?
 - Yes! Z3 produces a so-called *unsatisfiable core*

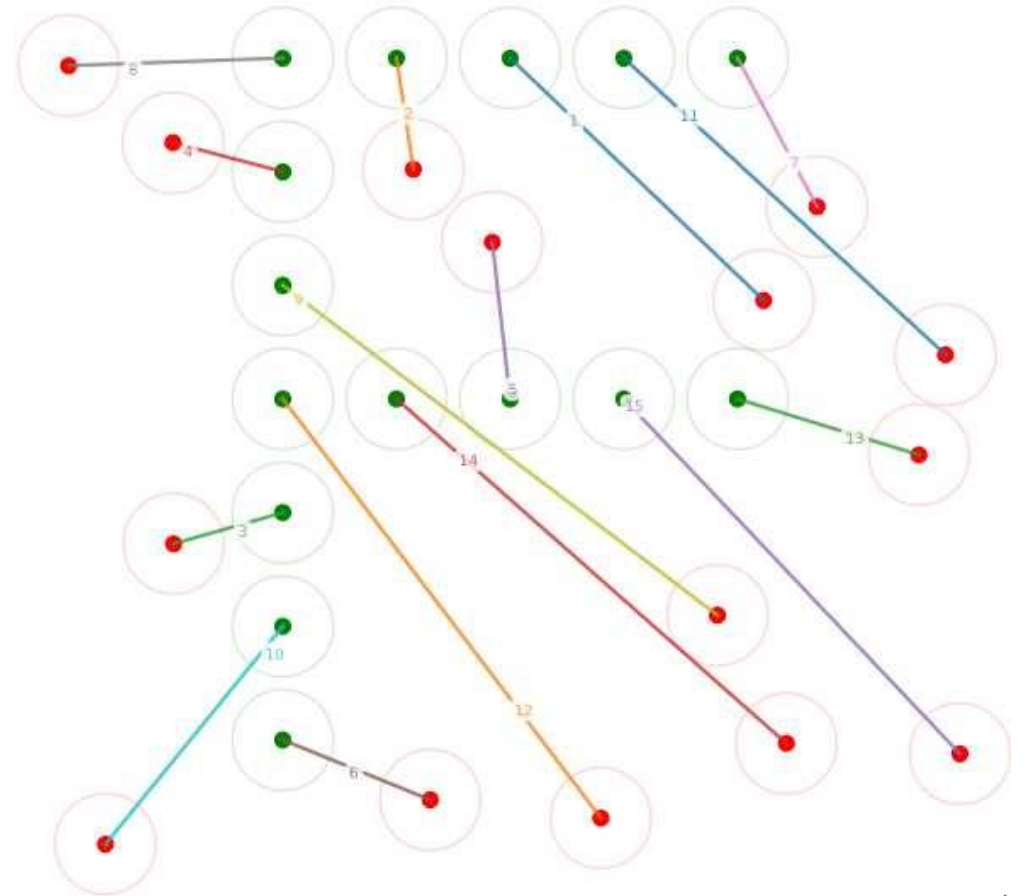
Unsatisfiable Cores

- Can we use insights from `unsat` results?
 - Yes! Z3 produces a so-called *unsatisfiable core*
 - Tell Z3 to “track” certain assumptions:
 - `solver.assert_and_track(<phi>, <Bool|String>)`

```
p, q = Bools('p q')
s = Solver()
s.add(Not(q))
s.assert_and_track(q, 'p1')
s.assert_and_track(p, 'p2')
print(s.check())
print(s.unsat_core())
```

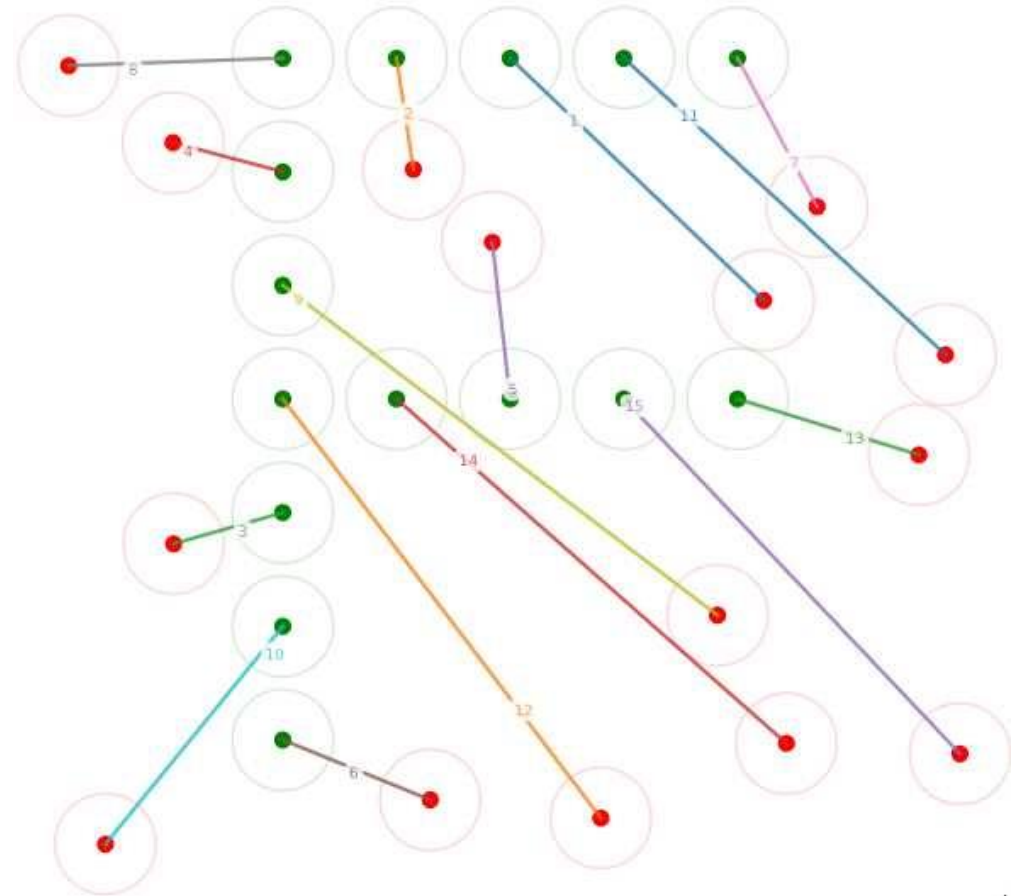
Excursion – Scheduling Molecule Movement

- We are given a list of **start positions** s_i and **goal positions** g_i



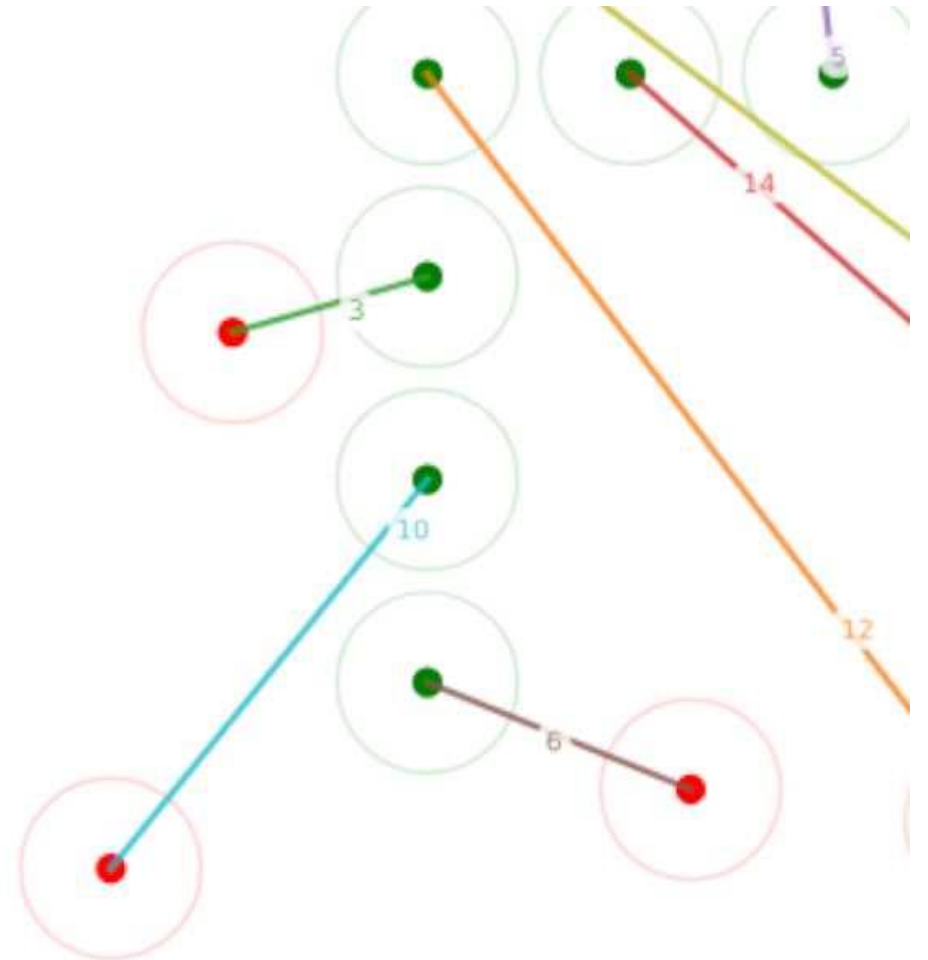
Excursion – Scheduling Molecule Movement

- We are given a list of **start positions** s_i and **goal positions** g_i
- In each timestep j we want to move one molecule,
- without them crossing any other molecule while moving!



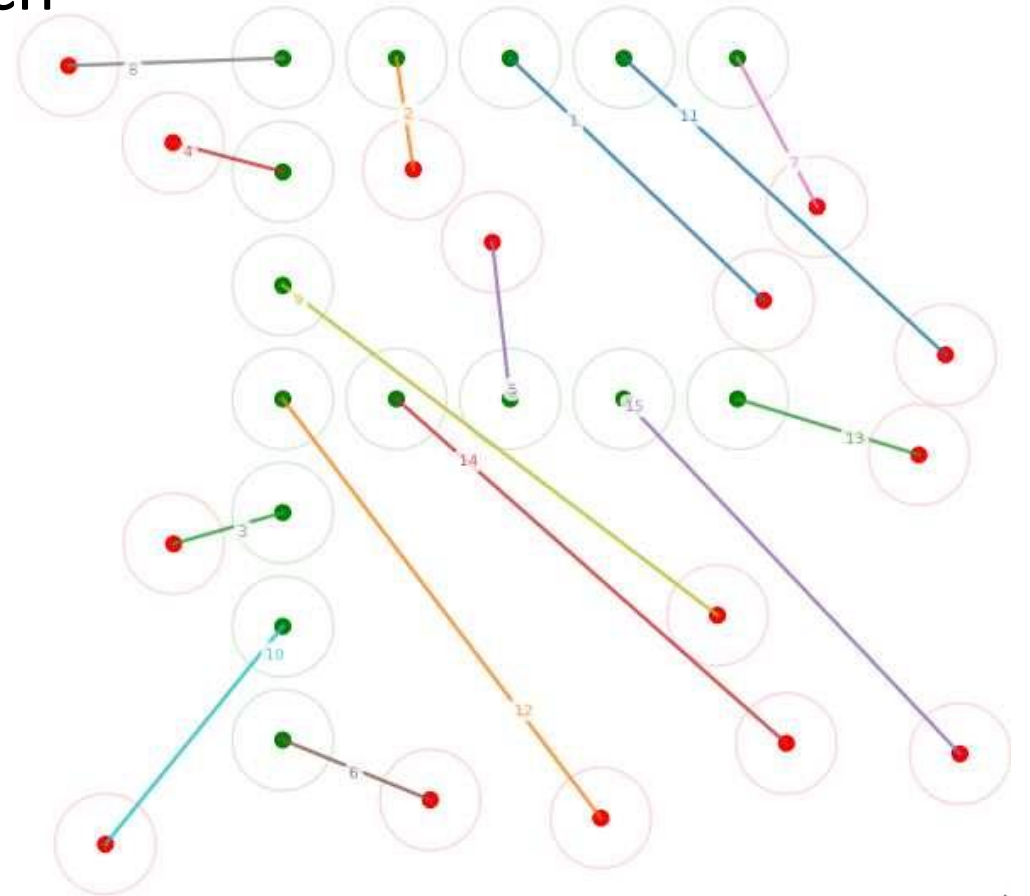
Excursion – Scheduling Molecule Movement

- We are given a list of **start positions** s_i and **goal positions** g_i
- In each timestep j we want to move one molecule,
- without them crossing any other molecule while moving!
- E.g.: $i=10$ needs to move before 6



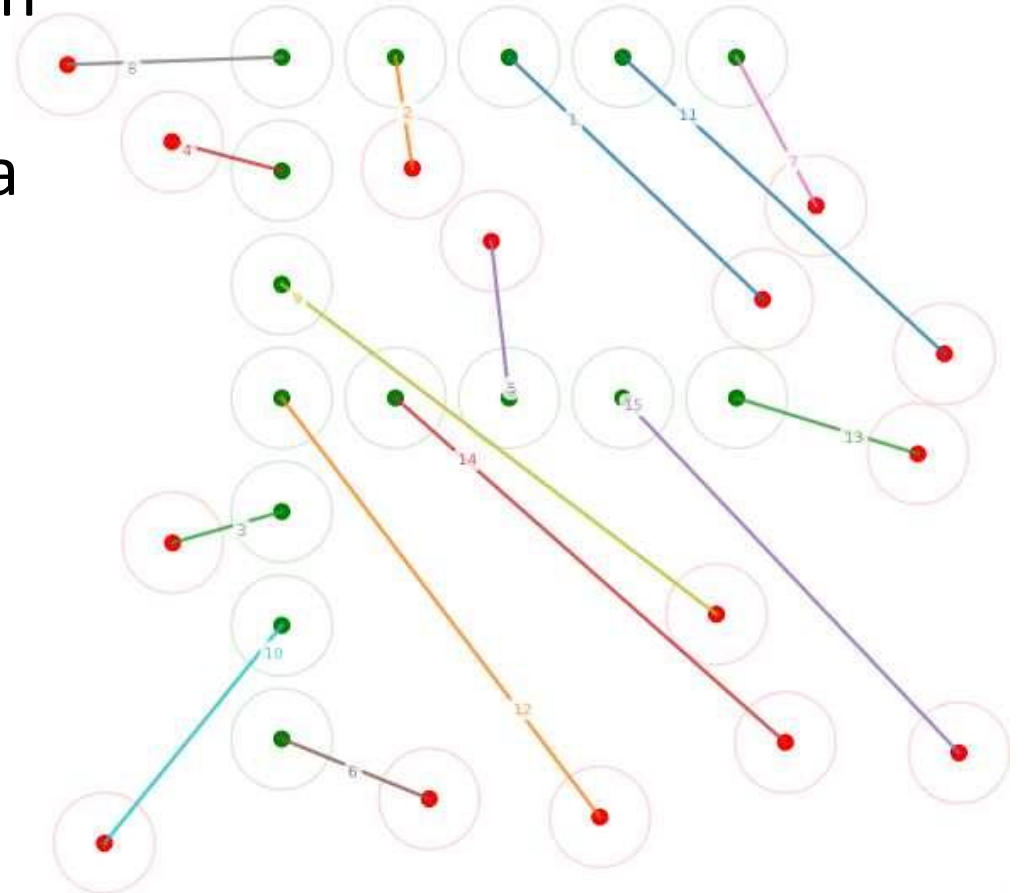
Excursion – Scheduling Molecule Movement

- We are given a list of **start positions** s_i and **goal positions** g_i
- Which molecule i should move in which timestep j ?



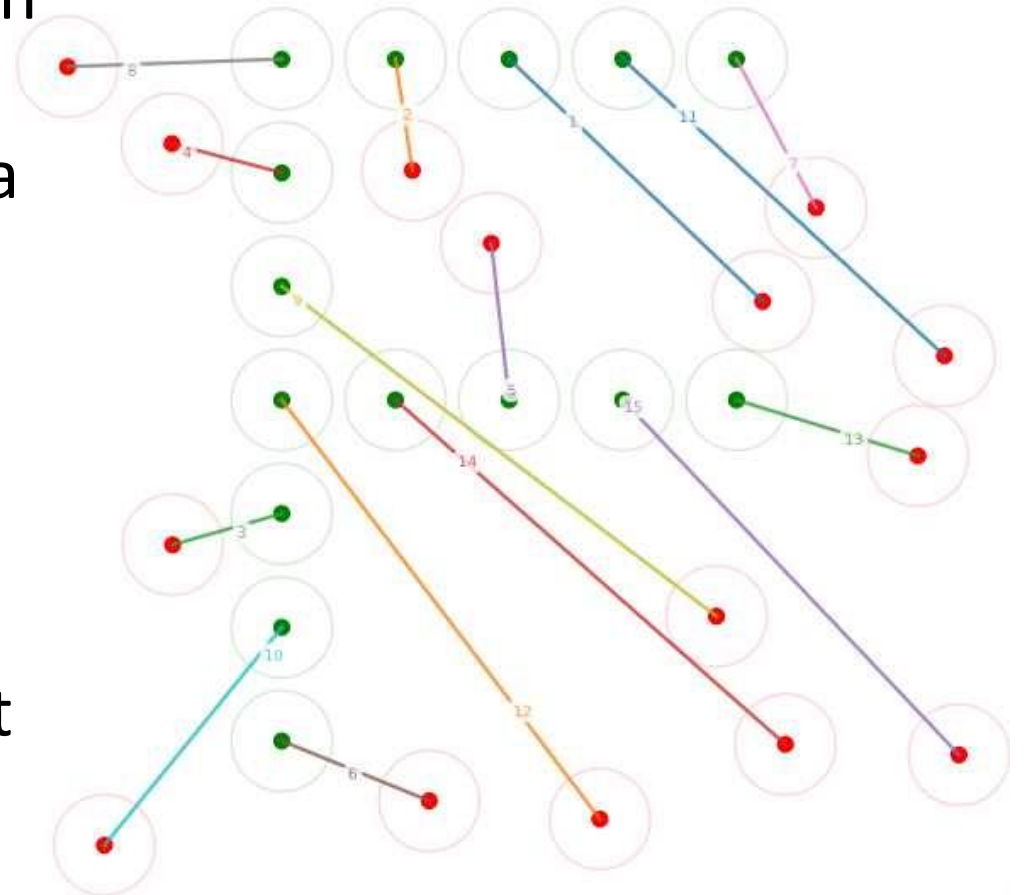
Excursion – Scheduling Molecule Movement

- We are given a list of **start positions** s_i and **goal positions** g_i
- Which molecule i should move in which timestep j ?
- What do we need to tell Z3 to give us a solution?



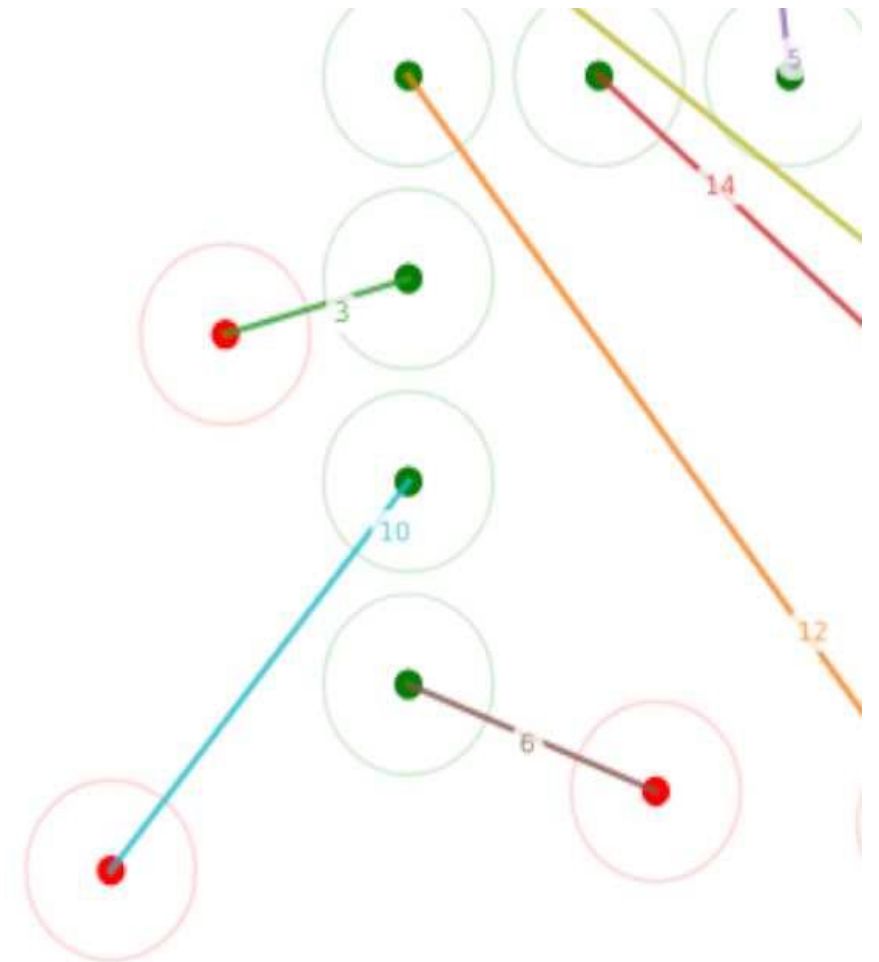
Excursion – Scheduling Molecule Movement

- We are given a list of **start positions** s_i and **goal positions** g_i
- Which molecule i should move in which timestep j ?
- What do we need to tell Z3 to give us a solution?
- Booleans:
 - „Is molecule i at the start“,
 - „is molecule i at the goal“
 - „Is molecule i scheduled to move at timestep j “



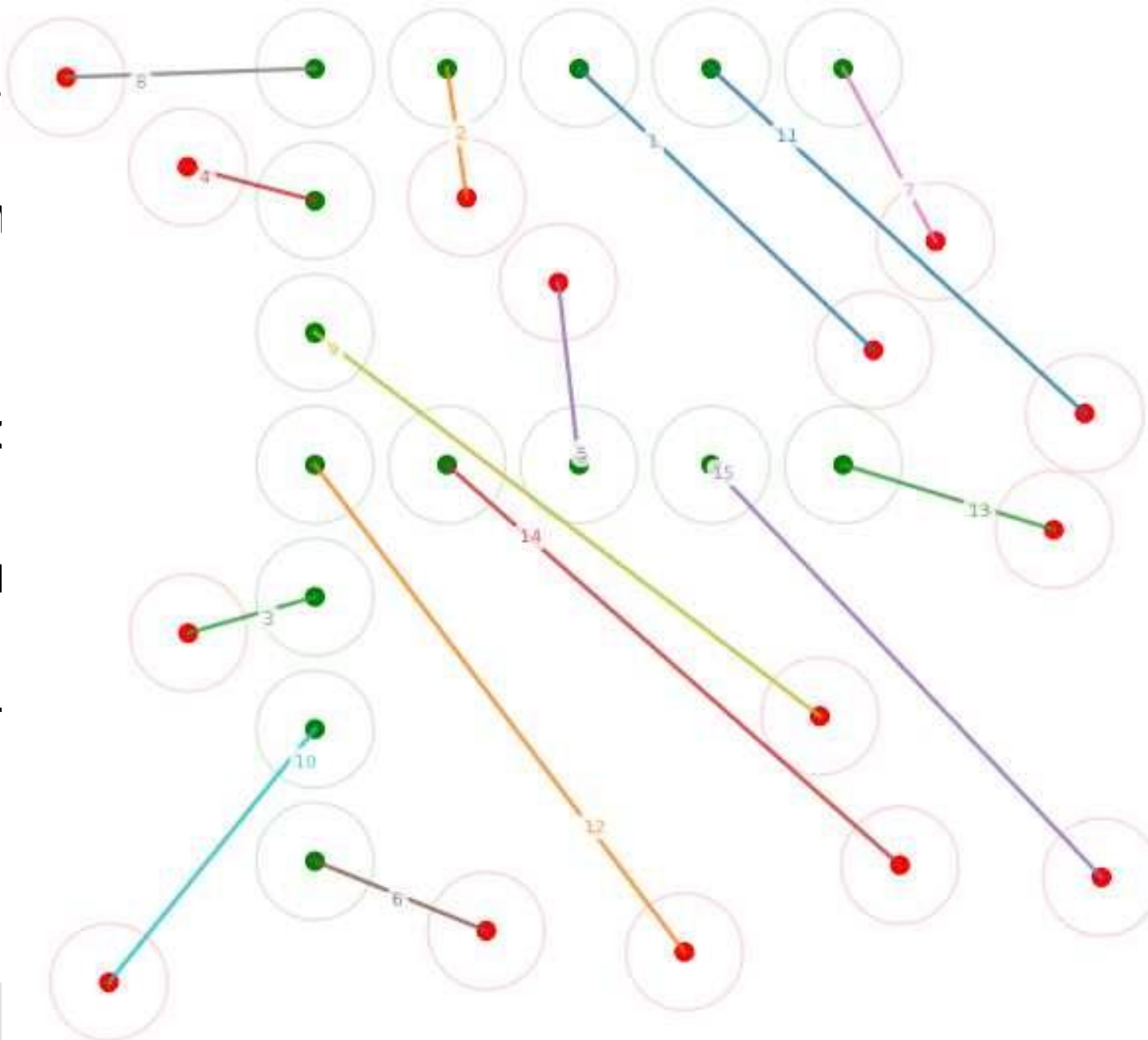
Excursion – Scheduling Molecule Movement

- We are given a list of **start positions** s_i and **goal positions** g_i
- Which molecule i should move in which timestep j ?
- What do we need to tell Z3 to give us a solution?
- Constraints:
 - „If 10 moves at timestep j , then 6 has to move in a timestep $> j$ “
 - „If a moves at timestep j , then b has to have moved in a timestep $< j$ “



Excursion

- We are given a set of points in the plane
- Which points are connected by edges, and which are not?
- What is the structure of the graph? Can we use a search algorithm to find a path from a source to a target?
- Constraints
 - „If time is limited, we can only visit a limited number of points"

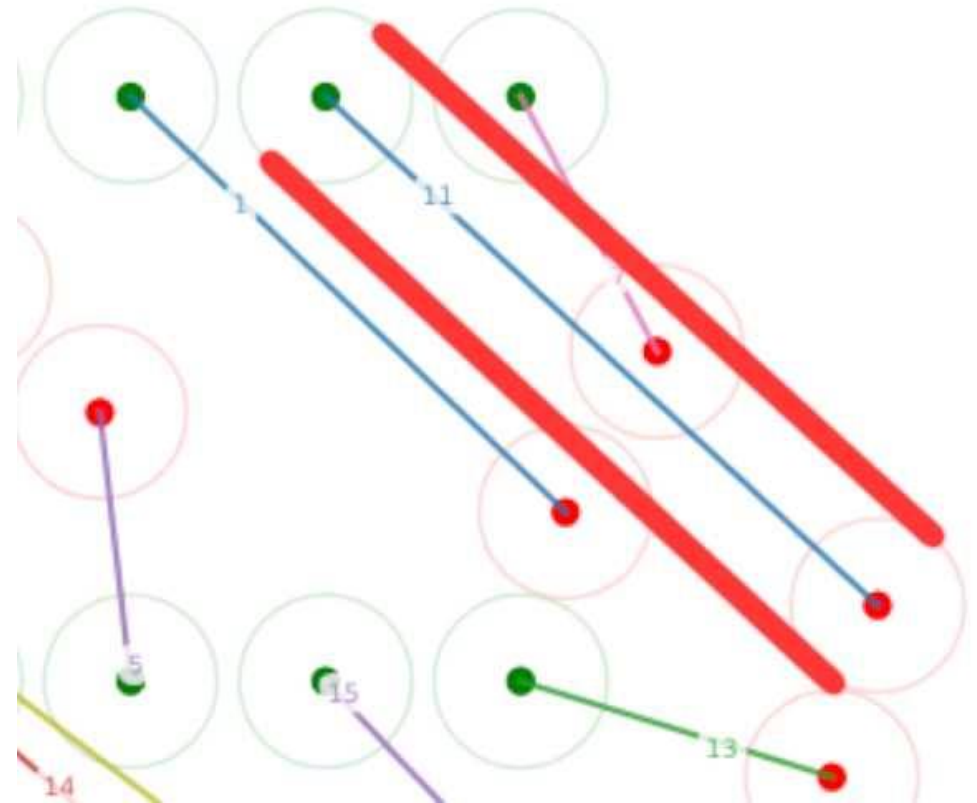


Movement

positions g_i

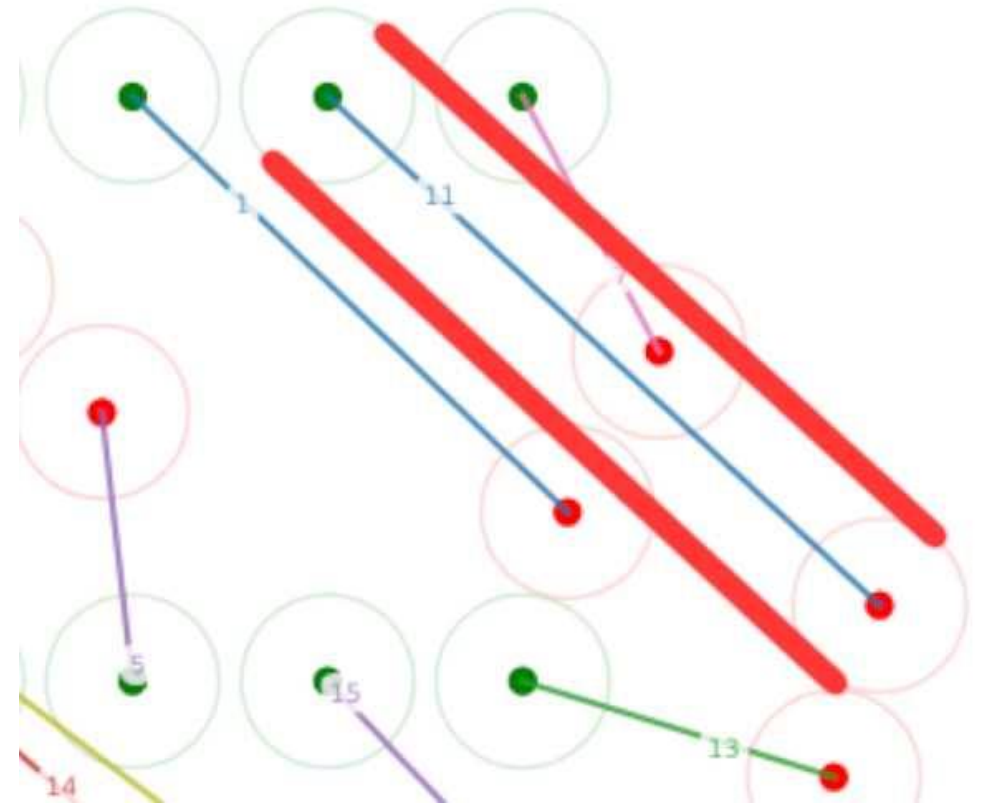
Excursion – Scheduling Molecule Movement

- We are given a list of **start positions** s_i and **goal positions** g_i
- Which molecule i should move in which timestep j ?
- What do we need to tell Z3 to give us a solution?
- Unsatisfiable:
 - 7 will never move out of the way for 11
 - $[g_{i1:7}; i2:11, s_{i1:7}; i2:11]$

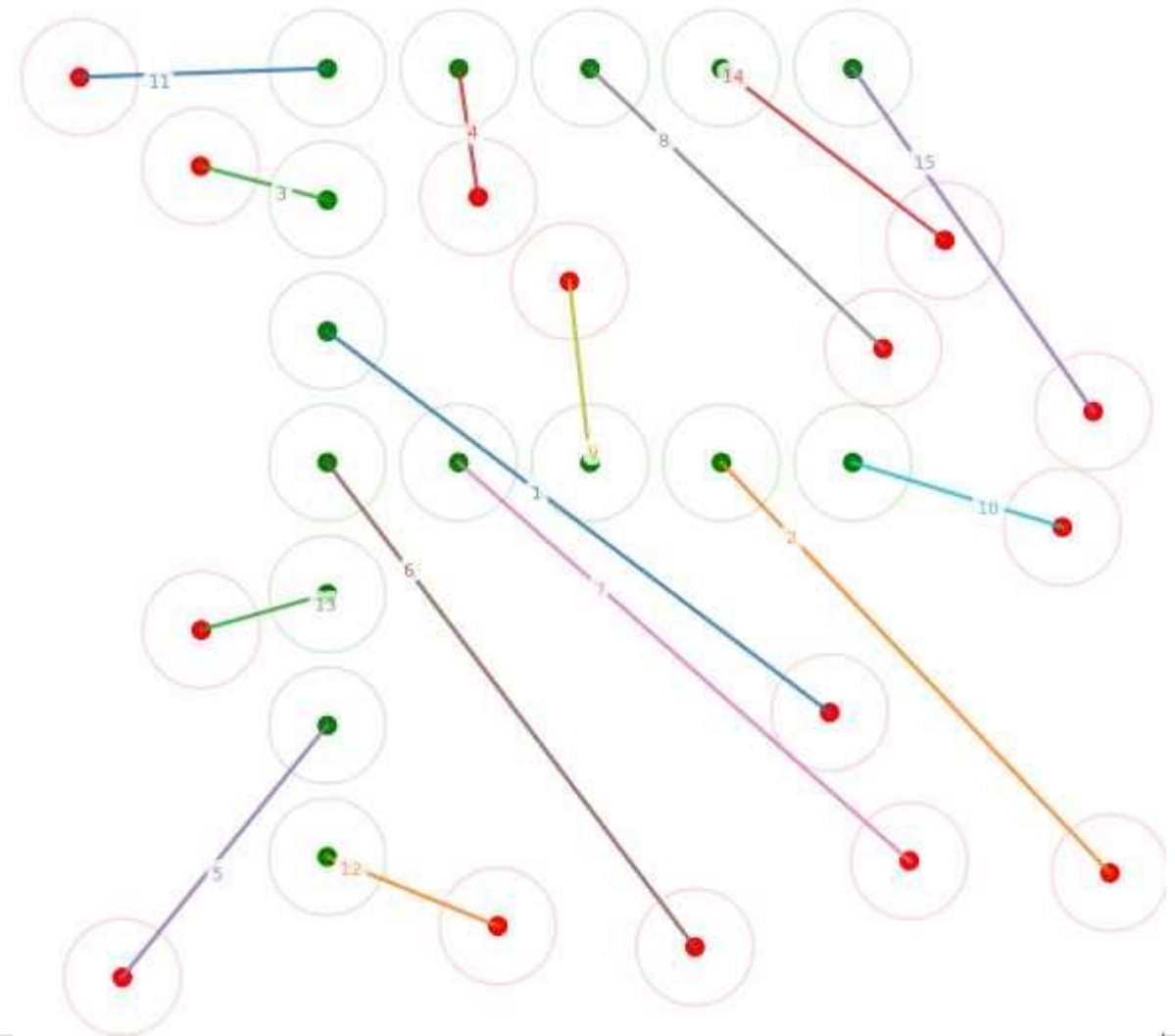
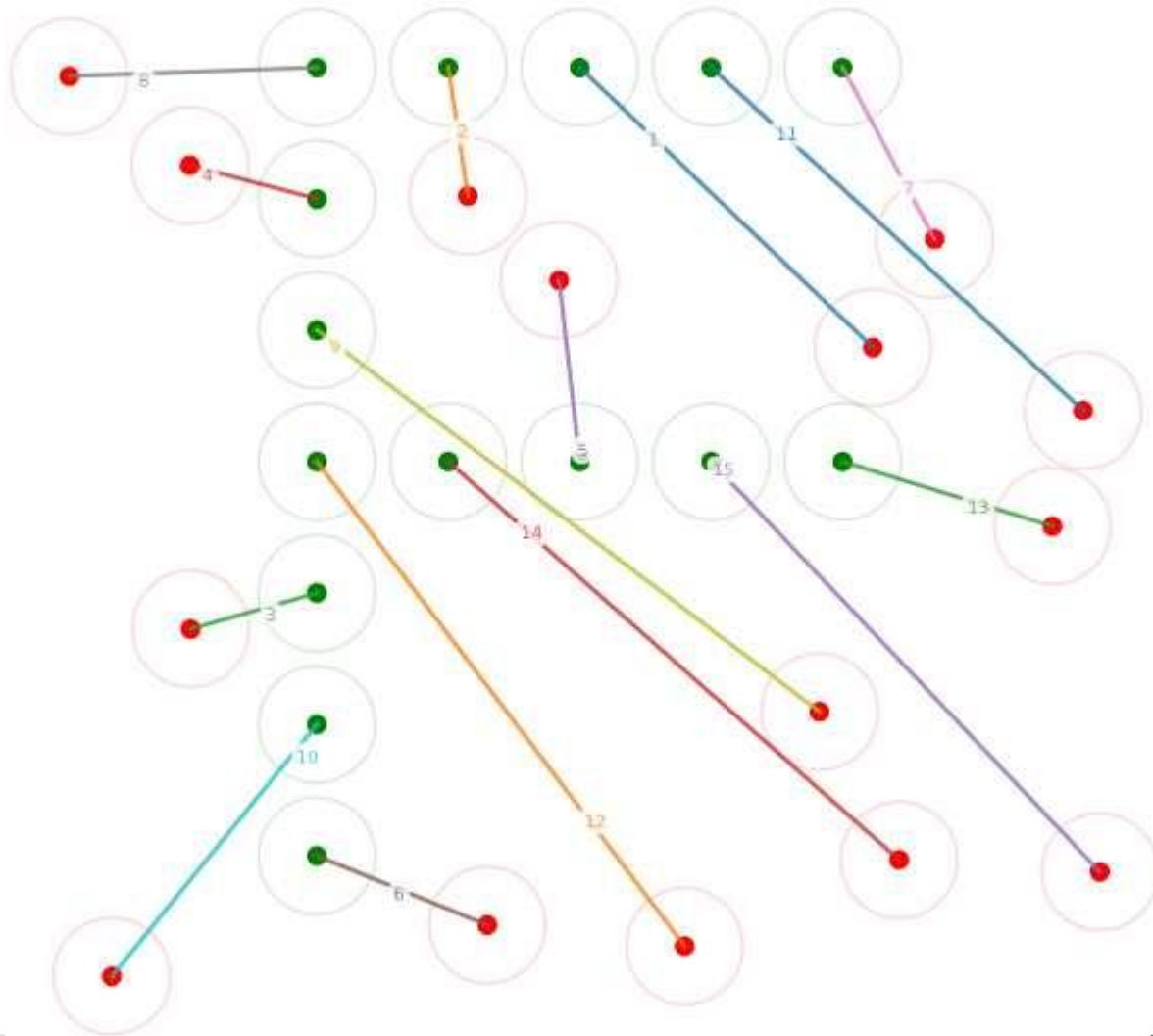


Excursion – Scheduling Molecule Movement

- We are given a list of **start positions** s_i and **goal positions** g_i
- Which molecule i should move in which timestep j ?
- What do we need to tell Z3 to give us a solution?
- Unsatisfiable:
 - 7 will never move out of the way for 11
 - `[g-i1:7;i2:11, s-i1:7;i2:11]`
 - Therefore: 7 and 11 need to switch goal positions!

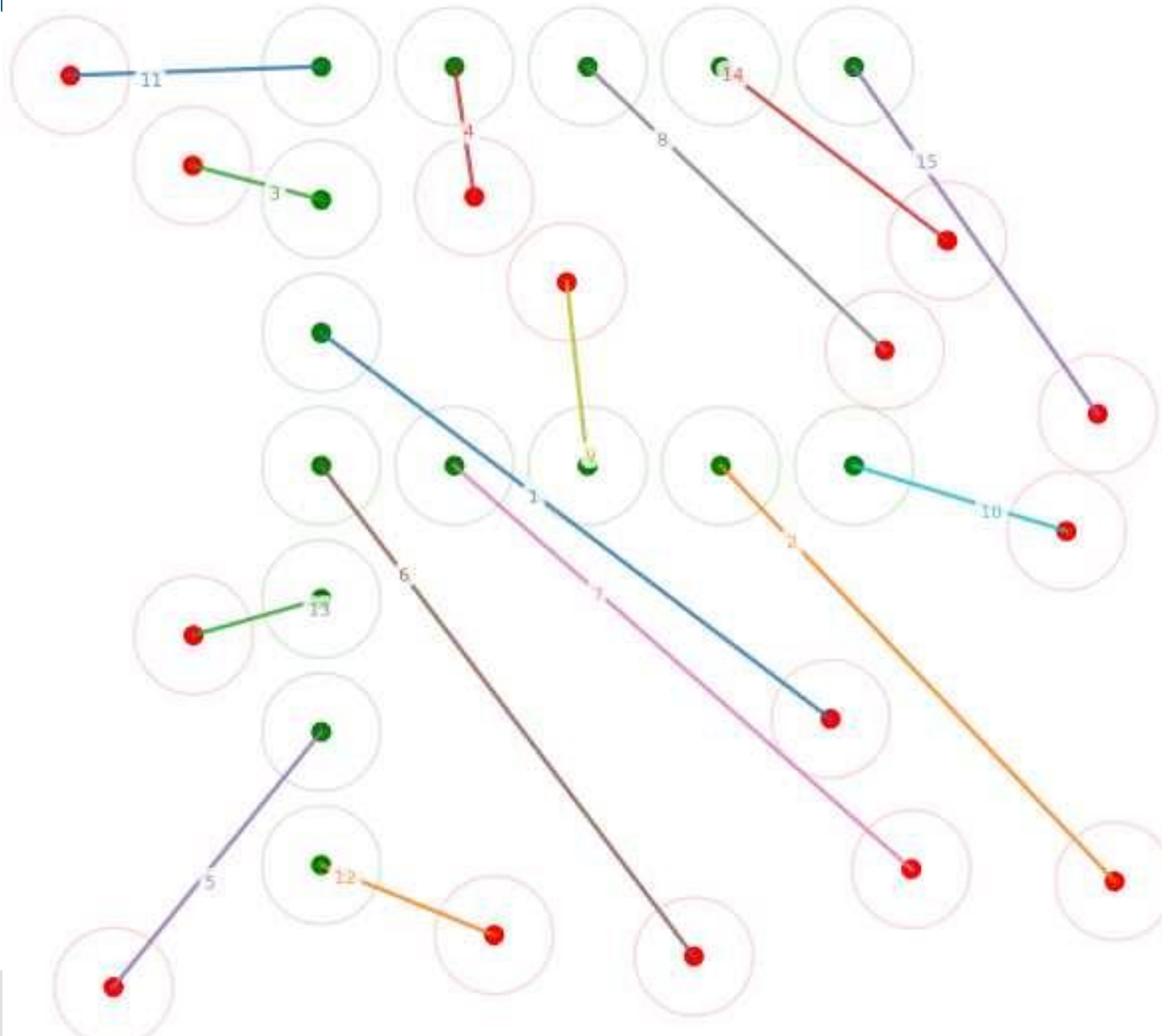


Excursion – Scheduling Molecule Movement



Excu

ovement



More Z3...

- `solver = Solver()`
 - A general, multi-purpose solver
- Z3 contains multiple proof methods that can be fine-tuned for a problem setting

More Z3...

- `solver = Solver()`
 - A general, multi-purpose solver
- Z3 contains multiple proof methods that can be fine-tuned for a problem setting
- Tactics and Tacticals:
 - A `Tactic(<String>)` describes a heuristic or algorithm to be applied
 - E.g.: `ackermannize-bv`, `euf-completion` (congruence-closure), `simplify`, `solve-eqs`, `smt`, `sat`, ...
 - A `Tactical` describes the application order of the different tactics:
 - `Then([<String>, ...])` apply tactics consecutively,
 - `OrElse(t, s)` apply `t`, if it fails apply `s`,
 - `Repeat(t)` repeat `t` until no changes happen
 - `.solver()` in order to get the according Solver

More Z3...

- Tactics and Tacticals:
 - A `Tactic (<String>)` describes a heuristic or algorithm to be applied
 - E.g.: `ackermannize-bv`, `euf-completion` (congruence-closure), `simplify`, `solve-eqs`, `smt`, `sat`, ...
 - A `Tactical` describes the application order of the different tactics:
 - `Then([<String>, ...])` apply tactics consecutively,
 - `.solver()` in order to get the according `Solver`

```

bv_solver = Then('simplify,
                 'solve-eqs',
                 'bit-blast',
                 'sat').solver()

```

```

x, y = BitVecs('x y', 16)
bv_solver.add(x*32 + y == 13, x & y < 10, y > -100)
print(bv_solver.check())

```