# Usage of the solver

This jupyter notebook details the code we have written for our project. First, our generators and solver will be shown, and at the end there is example code that can be used to run the solver.

```
# General setup
from numpy.random import randint, choice
from typing import Dict
from copy import deepcopy
from networkx import DiGraph, draw
from matplotlib.pyplot import show
from pickle import load, dump
```

First, we need to define what a model can be. We have chosen to group a set of worlds, agents, relations and valuations all into one class seen below. This class, unless initiated with specific values for these 4 things, will randomly generate an S5 Kripke model for our solver to use.

```
class Model():
    def __init__(self, worlds: list = None, agents: list = None,
                 relations=None, valuations=None):
        self.worlds = worlds
        self.agents = agents
        self.relations = relations
        self.valuations = valuations
        # Generate all values if they were not given.
        if self.valuations is None:
            self.create_valuations()
        if self.relations is None:
            self.create_relations()
        if self.agents is None:
            self.create_agents()
        if self.worlds is None:
            self.create_worlds()

    def __eq__(self, value) -> bool:
        # Overwrite the equality operator (==)
        if isinstance(value, self.__class__):
            return self.relations == value.relations and
self.valuations == value.valuations
        else:
            return False

    def __ne__(self, value) -> bool:
        # Overwrite the non equality operator (!=)
        return not self.__eq__(value)

    def create_worlds(self, max: int = 20) -> list[str]:
        """
        generate a random number of worlds named w1-w{max}
        :param max: maximum amount of worlds in the model. Default is
20.
        """
        self.worlds = []
        for i in range(randint(1, max + 1)):
            self.worlds.append(f"w{i}")
        return self.worlds

    def create_agents(self, max: int = 5) -> list[str]:
        """
        generate a random number of worlds named A-Z
        :param max: maximum amount of worlds in the model. Default is
5.
        """

        if max > 26:
            raise ValueError("Having more agents than letters in the
alphabet is not possible")

        self.agents = []
        for i in range(randint(1, max + 1)):
            self.agents.append(chr(ord('A')+i))
        return self.agents

    def create_relations(self) -> Dict[str, Dict[str, set[str]]]:
        """
        generate a random number of relations for each agent and each
world.
        These relations follow S5 rules.
        """
        # First, make sure we have worlds and agents in our model
        if self.worlds is None:
            self.create_worlds()
        if self.agents is None:
            self.create_agents()
        # Then, generate a set of relations for each agent from each
world
        self.relations = {}
        for a in self.agents:
            self.relations[a] = {}
            for world in self.worlds:
                # model is reflexive, so each world at least has
itself in the set
                # each connection between different worlds has a 30%
chance of existing
                self.relations[a][world] = {w for w in self.worlds
                                            if w == world
                                            or randint(0, 10) < 3}
        # Make sure the model is transitive
        # Deepcopy to make sure we're not changing while iterating
        copy = deepcopy(self.relations)
        for a in copy:
            for world in copy[a]:
                for connection in copy[a][world]:
                    # add any missing relations
                    self.relations[a][world].update(copy[a]
[connection])


        # Make sure the model is euclidean
        copy = deepcopy(self.relations)
        for a in copy:
            for world in copy[a]:
                lst = list(copy[a][world])
                for connection in lst:
                    for second in lst[lst.index(connection):]:
                        # add any missing relations
                        self.relations[a][connection].add(second)
                        self.relations[a][second].add(connection)

        return self.relations

    def create_valuations(self, max: int = 10):
        """
        generate a random number of atoms and give each valuations for
each world.
        Each atom has a 50% chance of being true in each world.
        """
        # Create worlds if there are none yet
        if self.worlds is None:
            self.create_worlds()

        if max > 26:
            raise ValueError("Having more atoms than letters in the
alphabet is not possible")

        # Create a random number of atoms named a-z
        atoms = [chr(ord('a')+i) for i in range(randint(1, max + 1))]
        self.valuations = {}
        for atom in atoms:
            # Decide whether this atom is true in a world with 50%
chance
            self.valuations[atom] = {w for w in self.worlds
                                     if randint(0, 10) < 5}

        return self.valuations
```

The following function creates a formula that can be used as announcement in the riddle.

```
def create_formula(model: Model, low:int = 5, high:int = 12) -> str:
    """
    Creates a formula matching a model. Adds a random amount of K-
operators to the start of the formula.
    There is a 20% chance any K-operator, atom or conjunction has a
negation in front of it
    :param low: Lowest amount of K-operators that can be added to the
start of a formula
    :param high: Highest amount of K-operators that can be added to
the start of a formula
    """
    # set some default values
    formula = ""
    neg = False

    # add a random number of K-operators between 'low' and 'high'
    for _ in range(randint(low, high)):
        # 1 in 5 chance of adding a negation
        if randint(5) == 1:
            formula += '!'
        formula += "K" + choice(model.agents)
    # 90% chance of adding an atom at the end
    if randint(10) != 1:
        # 1 in 5 chance of adding a negation
        if randint(5) == 1:
            formula += '!'
        formula += choice(list(model.valuations.keys()))
    # 10% chance of adding a conjunction instead
    else:
        # 1 in 5 chance of adding a negation
        if randint(5) == 1:
            formula += '!('
            neg = True
        formula += '(' + create_formula(model, 0, 2) + ")&(" +
create_formula(model, 0, 2) + ')'
        if neg:
            formula += ')'

    return formula
```

The following is a Solver class. This class will find the product of the announcement formula's action model and a model. If no formula or model was given, it will create one, although this is not used.

```
class Solver:
    def __init__(self, formula: str = None, model: Model = None,
verbose: bool = False):
        """
        Initialize the solver with a starting formula and world

        :param formula: A formula that serves as the announcement to the riddle
        :param model: The Kripke model used in the riddle
        :param verbose: Whether or not the solver should print what it's working on.
        """
        self.start = formula if formula else create_formula()
        self.level = 0
        self.f = formula
        self.model = model if model else Model()
        self.truths = {}
        self.talks = verbose

    def _eval_formula(self, formula: str, world: str) -> bool:
        """
        Evaluates the truth value of a formula in a specific world in the model.
        It does this recursively, storing valuations in a dictionary to look up later.
        :param formula: The formula to evaluate
        :param world: The world in which to evaluate this formula
        """
        current = world + formula
        # Check if we already know the valuation of this formula in this world
        if (truth := self.truths.get(current, None)) is not None:
            return truth

        # What is the thing we need to evaluate?
        match atom := formula[0]:
            case atom if atom in self.model.valuations:
                # Atoms are true in this world if the world is in the set of valuations related to this atom
                self.truths[current] = world in self.model.valuations.get(atom, set())
            case 'K':
                # To evaluate a K-operator, we need to know which agent supposedly knows what
                agent, sub_formula = formula[1], formula[2:]
                # Then, the subformula must hold in all worlds this agent can reach from the current world.
                self.truths[current] = all(self._eval_formula(sub_formula, connected_world) for connected_world in self.model.relations[agent][world])
            case '!':
                # The not-operator simply negates the valuation of the
```

```
rest of the line
                self.truths[current] =
not(self._eval_formula(formula[1:], world))
            case '(':
                # Brackets can only appear on their own or in
combination with an "&" (and)
                # So all subformulas within the brackets must hold
                this_bracket = [sub_formula for i, sub_formula in
self._parse_brackets(formula) if i == 0]
                self.truths[current] =
all(self._eval_formula(sub_formula, world) for sub_formula in
this_bracket)
        # We stored our findings in the hashmap, so we can return the
same value.
        return self.truths[current]

    def _parse_brackets(self, string: str):
        """
        Helper function
        :param string: String to parse the brackets of
        """
        stack = []
        for i, c in enumerate(string):
            if c == '(':
                stack.append(i)             # Keep track of where the
brackets are
            elif c == ')' and stack:
                start = stack.pop()         # Remember where the last
bracket started
                yield (len(stack), string[start + 1: i])    # Return
the depth and content of each bracket

    def _satisfying_worlds(self):
        # Find all worlds remaining after the product
        return [w for w in self.model.worlds if
self._eval_formula(self.f, w)]

    def _strip_outer_K(self):
        if self.f[0] == 'K':
            self.f = self.f[2:]
        elif self.f[0:2] == '!K':
            self.f = self.f[3:]
        else:
            return False
        return True


    def find_riddle(self):
        """
        Main function
        """
        answers = []

        while True:
            # Find all worlds that satisfy the current state of the
riddle
            ws = self._satisfying_worlds()

            if self.talks:
                print("\n")
                print("Formula:", self.f)
                print("Possible worlds:", ws)

            # What is the answer to this riddle?
            if not ws:
                # No satisfying worlds means the announcement was not
consistent
                if self.talks:
                    print("→ inconsistent announcement")
                answers.append(None)
            elif len(ws) == 1:
                # There is a single satisfying world
                if self.talks:
                    print("→ unique world:", ws[0])
                answers.append(ws[0])
            else:
                # There are multiple satisfying worlds
                answer = []
                # Find what we know about all the atoms
                for atom in self.model.valuations:
                    truths = [w in self.model.valuations[atom] for w
in ws]

                    if all(truths):
                        # We know this atom is true
                        answer.append(atom)
                    elif not(any(truths)):
                        # We know this atom is false
                        answer.append(f"!{atom}")
                    else:
                        # If neither, we can't say anything about the
atom
                        answer.append(f"{atom}?")
                if self.talks:
                    print(f"My answer is → {', '.join(answer)}")
                # Keep track of the answers given for this order of
ToM
                answers.insert(0, ", ".join(answer))

            # If there's no more K-operators to remove, stop the loop
            if not self._strip_outer_K():
                break

        # If a satisfactory riddle was found, print this and return
True
        if len(answers) != 0 and len(answers) == len(set(answers)):
            print("\nUnique different answer found for each K operator
removed.")
            print("Formula:", self.start)
            print("worlds:", self.model.worlds)
            print("relations:", self.model.relations)
            print("valuations:", self.model.valuations)
            print("Answers:", [a for a in answers])
            for agent in self.model.agents:
                # Draw graphs of the found model for each agent
                G = DiGraph(self.model.relations[agent])
                draw(G, with_labels=True, arrows=True)
                show()
            return True
        else:
            if self.talks: print("\t\tNo unique different answer found
for each K operator removed.")
            return False
```

The following code shows two examples of how our code can be used. The first tries a limited number of times, while the second continues trying until a riddle has been found or the user interrupts the code.

```
# Iterate over, in this case, 2 tries
found = False
for i in range(tries := 2):
    # Keep track of tried combinations
    tried_combinations = {}
    # create a new model and announcement
    model = Model()
    formula = create_formula(model)
    # See if this combination of model and announcement gets us a
satisfying riddle
    if (formula, model) not in tried_combinations.items():
        tried_combinations.setdefault(formula, []).append(model)
        found = Solver(formula, model, True).find_riddle()
if not found:
    print("Have not found a satisfying riddle within", tries,
"tries.")



Formula: !KAKAKAKAKBd
Possible worlds: ['w0', 'w1', 'w2', 'w3', 'w4', 'w5', 'w6', 'w7',
'w8', 'w9', 'w10', 'w11', 'w12', 'w13', 'w14', 'w15', 'w16', 'w17',
'w18', 'w19']
My answer is → a?, b?, c?, d?, e?, f?, g?, h?, i?


Formula: KAKAKAKBd
Possible worlds: []
→ inconsistent announcement


Formula: KAKAKBd
Possible worlds: []
→ inconsistent announcement


Formula: KAKBd
Possible worlds: []
→ inconsistent announcement


Formula: KBd
Possible worlds: []
→ inconsistent announcement


Formula: d
Possible worlds: ['w0', 'w2', 'w3', 'w4', 'w5', 'w7', 'w8', 'w9',
'w12', 'w13', 'w18']
My answer is → a?, b?, c?, d, e?, f?, g?, h?, i?
           No unique different answer found for each K operator
removed.


Formula: KCKDKAKBKBKBKCKDKCKA(a)&(KD!c)
Possible worlds: []
→ inconsistent announcement


Formula: KDKAKBKBKBKCKDKCKA(a)&(KD!c)
Possible worlds: []
→ inconsistent announcement


Formula: KAKBKBKBKCKDKCKA(a)&(KD!c)
Possible worlds: []
→ inconsistent announcement


Formula: KBKBKBKCKDKCKA(a)&(KD!c)
Possible worlds: []
→ inconsistent announcement
```

```
Formula: KBKBKCKDKCKA(a)&(KD!c)
Possible worlds: []
→ inconsistent announcement


Formula: KBKCKDKCKA(a)&(KD!c)
Possible worlds: []
→ inconsistent announcement


Formula: KCKDKCKA(a)&(KD!c)
Possible worlds: []
→ inconsistent announcement


Formula: KDKCKA(a)&(KD!c)
Possible worlds: []
→ inconsistent announcement


Formula: KCKA(a)&(KD!c)
Possible worlds: []
→ inconsistent announcement


Formula: KA(a)&(KD!c)
Possible worlds: []
→ inconsistent announcement


Formula: (a)&(KD!c)
Possible worlds: []
→ inconsistent announcement
           No unique different answer found for each K operator
removed.
Have not found a satisfying riddle within 2 tries.
```

```python
try:
    # Define some variable defaults
    found = False
    tries = 0
    unique = 0

    # Keep track of tried combinations, load it from a file if such
exists
    try:
        with open('tried_combinations.pickle', 'rb') as file_tried:
            tried_combinations = load(file_tried)
    except:
        tried_combinations = {}
    print("Finished loading tried_combinations.pickle")

    # Infinite loop, unless it finds a riddle
    while not found:
        # create a new model and announcement
        model = Model()
        formula = create_formula(model)
        # See if this combination of model and announcement gets us a
satisfying riddle
        if formula in tried_combinations and model not in
tried_combinations[formula]:
            tried_combinations.setdefault(formula, []).append(model)
            found = Solver(formula, model).find_riddle()
            unique += 1
        tries += 1
except KeyboardInterrupt:
    print(f"Solver stopped by user after {tries} tries.\nOf those,
{unique}/{tries} were unique.")
    try:
        with open('tried_combinations.pickle', 'wb') as file_tried:
            dump(tried_combinations, file_tried, protocol=-1)
        print("Tried combinations were written to the file
tried_combinations.pickle.")
    except:
        print("Something went wrong when writing the tried
combinations to a file.")
```

```
Finished loading tried_combinations.pickle
Solver stopped by user after 19989 tries.
Of those, 0/19989 were unique.
Tried combinations were written to the file tried_combinations.pickle.
```