CS170 Phase 2 Write-Up: Algorithm

Jonathan Sun, Kevin Vo, Aleem Zaki

In our first algorithm we recognize that this problem reduces to the SAT problem. This means the wizard problem can be solved with an algorithm that solves a SAT problem.

Namely, each constraint can be modeled as a set of booleans. For example, given a constraint [Mario, Luigi, Peach], we know that this constraint can be satisfied several ways. Let each wizard denote a value corresponding to their index in the final ordering. Let M, L, P denote Mario, Luigi, and Peach. To satisfy the constraint, we need P to not be in the range of M and L. Another way to say this is: (Peach is after Mario and Luigi) OR (Peach is before Mario and Luigi).

$((P > M) \wedge (P > L)) \vee ((P < M) \wedge (P < L))$

Note something of the form $(X > Y)$ is not of standard SAT notation. Thus, we convert each of those into a standard SAT literal. We get:

$(x_1 \wedge x_2 \wedge \bar{x}_3 \wedge \bar{x}_4) \vee (x_3 \wedge x_4 \wedge \bar{x}_1 \wedge \bar{x}_2)$

Note the introduction of negation because $(X > Y)$ is the opposite of $(X < Y)$. Using this boolean predicate, the SAT solver determines which wizards are greater than what. For example, if a boolean assignment for variable $x_n =$"Luigi > Mario" is true, then Luigi is greater than Mario. In this case, Luigi comes after Mario in the ordering (although order doesn't matter as long as it's consistent for all boolean assignments).

Since these boolean assignments create dependencies between wizards, we then use a graph to model the dependencies and run a topological sort and yield the final ordering.

We provided two algorithms using the aforementioned concepts. `solver_sat.py` finds an optimal ordering. When cycles are introduced in the topological sort (impacting the runtime), then we needed an approximator. `solver_sat_exp.py` naively trims cycles and finds the best ordering within a number of samples.

Some external libraries were needed for this algorithm. Namely, we need a SAT solver (provided by MiniSat) and a python interface for setting up SAT problems (provided by SATisPy). We used the second install method for satispy from here: `https://github.com/netom/satispy`. This is the homebrew installer for minisat: `http://macappstore.org/minisat/`. We chose these libraries because they provided a versatile SAT solver (no need to convert expressions to CNF) and also worked on python which was our language of choice prior.

In [ ]:
```
import argparse
from satispy import Variable, Cnf
from satispy.solver import Minisat
from collections import deque
from collections import defaultdict

"""
======================================================================
  Complete the following function.
======================================================================
"""

def solve(num_wizards, num_constraints, wizards, constraints):
    """
    Write your algorithm here.
    Input:
        num_wizards: Number of wizards
        num_constraints: Number of constraints
        wizards: An array of wizard names, in no particular order
        constraints: A 2D-array of constraints,
                     where constraints[0] may take the form ['A', 'B', 'C']i

    Output:
        An array of wizard names in the ordering your algorithm returns
    """
    sameDictionary = []

    constraintToVariable = dict()

    exp = None
    for constraint in constraints:
        wiz1 = constraint[0]
        wiz2 = constraint[1]
        wiz3 = constraint[2]

        clause1 = wiz3 + " " + wiz1
        clause2 = wiz3 + " " + wiz2
        clause3 = wiz1 + " " + wiz3
        clause4 = wiz2 + " " + wiz3

        if clause1 in constraintToVariable:
            v1 = constraintToVariable[clause1]
        else:
            constraintToVariable[clause1] = Variable(clause1)
            v1 = constraintToVariable[clause1]

        if clause2 in constraintToVariable:
            v2 = constraintToVariable[clause2]
        else:
            constraintToVariable[clause2] = Variable(clause2)
            v2 = constraintToVariable[clause2]

        if clause3 in constraintToVariable:
            v3 = constraintToVariable[clause3]
        else:
            constraintToVariable[clause3] = Variable(clause3)
```

```
            v3 = constraintToVariable[clause3]

        if clause4 in constraintToVariable:
            v4 = constraintToVariable[clause4]
        else:
            constraintToVariable[clause4] = Variable(clause4)
            v4 = constraintToVariable[clause4]

        literal = ((v1 & v2 & -v3 & -v4) ^ (v3 & v4 & -v1 & -v2))
        if exp is None:
            exp = literal
        else:
            exp = exp & literal
    solver = Minisat()
    solution = solver.solve(exp)
    if solution.success:
        graph = dict()
        for wizard in wizards:
            graph[wizard] = set()
        for constraint in constraintToVariable:
            v = constraintToVariable[constraint]
            if solution[v] == True:
                w = str(v).split()
                vertexU = w[0]
                vertexV = w[1]
                graph[vertexU].add(vertexV)

        cycle = False
        try: topSortGraph = topological(graph)
        except ValueError: cycle = True
    iteration = 0
    graph = {}
    if not cycle:
        return list(topSortGraph)
    else:
        while True:
            dictCompare = {}

            g = Graph(num_wizards)
            for wiz, neighbors in graph.items():
                for n in neighbors:
                    g.addEdge(wiz, n)

            badOnes = []
            sccs = g.getSCCs()
            for scc in sccs:
                badGroup = set()
                if len(scc) > 1:
                    for u in scc:
                        for v in scc:
                            if g.containsEdge(u, v):
                                badGroup.add(u + " " + v)
                                g.removeEdge(u, v)

                    badOnes.append(badGroup)
            print("number of cycles:", len(badOnes))
            for count, i in enumerate(badOnes):
```

```
            print("length of cycle", count + 1, ":", len(i))

        for group in badOnes:
            lit = None
            for b in group:
                v = constraintToVariable[b]
                if lit is None:
                    lit = v
                else:
                    lit = lit & v
            exp = -(lit) & exp


        solver = Minisat()
        solution = solver.solve(exp)
        if solution.success:
            graph = dict()
            for wizard in wizards:
                graph[wizard] = set()
            for constraint in constraintToVariable:
                v = constraintToVariable[constraint]
                dictCompare[v] = solution[v]
                if solution[v] == True:
                    w = str(v).split()
                    vertexU = w[0]
                    vertexV = w[1]
                    graph[vertexU].add(vertexV)
            cycle = False
            try: topSortGraph = topological(graph)
            except ValueError: cycle = True
        else:
            print("SOMETHING WENT HORRIBLY WRONG")
        if not cycle:
            return list(topSortGraph)
        print("iteration: ", iteration)
        iteration += 1
        if dictCompare in sameDictionary:
            print("YOU'VE BEEN IN THIS STATE BEFORE")
        else:
            print("new state :)")
            sameDictionary.append(dictCompare)




GRAY, BLACK = 0, 1
def topological(graph):
    order, enter, state = deque(), set(graph), {}

    def dfs(node):
        state[node] = GRAY
        for k in graph.get(node, ()):
            sk = state.get(k, None)
            if sk == GRAY: raise ValueError("cycle")
            if sk == BLACK: continue
            enter.discard(k)
```

```
            dfs(k)
        order.appendleft(node)
        state[node] = BLACK

    while enter: dfs(enter.pop())
    return order


#This class represents a directed graph using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary to store graph

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    def removeEdge(self, u, v):
        self.graph[u].remove(v)

    def containsEdge(self, u, v):
        if u in self.graph:
            if v in self.graph[u]:
                return True
        return False

    # A function used by DFS
    def DFSUtil(self,v,visited):
        # Mark the current node as visited and print it
        visited[v]= True
        # print(v)
        #Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            if visited[i]==False:
                self.DFSUtil(i,visited)

    def DFSSet(self, v, visited):
        s = set()
        return self.DFSSetUtil(v, visited, s)


    def DFSSetUtil(self, v, visited, s):
        # Mark the current node as visited
        visited[v]= True
        s.add(v)
        #Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            if visited[i]==False:
                self.DFSSetUtil(i, visited, s)
        return s


    def fillOrder(self,v,visited, stack):
        # Mark the current node as visited
        visited[v]= True
```

```
        #Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            if visited[i]==False:
                self.fillOrder(i, visited, stack)
        stack = stack.append(v)


    # Function that returns reverse (or transpose) of this graph
    def getTranspose(self):
        g = Graph(self.V)

        # Recur for all the vertices adjacent to this vertex
        for i in self.graph:
            for j in self.graph[i]:
                g.addEdge(j,i)
        return g



    # The main function that finds and prints all strongly
    # connected components
    def getSCCs(self):

        stack = []
        # Mark all the vertices as not visited (For first DFS)
        # visited =[False]*(self.V)
        visited = dict()
        for u, neighbors in self.graph.items():
            visited[u] = False
            for n in neighbors:
                visited[n] = False
        # Fill vertices in stack according to their finishing
        # times
        # for i in range(self.V):
        #     if visited[i]==False:
        #         self.fillOrder(i, visited, stack)
        for u in visited:
            if visited[u] == False:
                self.fillOrder(u, visited, stack)

        # Create a reversed graph
        gr = self.getTranspose()

        # Mark all the vertices as not visited (For second DFS)
        # visited =[False]*(self.V)
        visited = dict()
        for u, neighbors in self.graph.items():
            visited[u] = False
            for n in neighbors:
                visited[n] = False

        # Now process all vertices in order defined by Stack
        sccs = []
        while stack:
            i = stack.pop()
            if visited[i]==False:
                # gr.DFSUtil(i, visited)
```

```
                s = gr.DFSSet(i, visited)
                sccs.append(s)
        return sccs

def checkConstraintsWithList(constraints, list):
    wizard_map = {}
    for i in range(len(list)):
        wizard_map[list[i]] = i
    return checkConstraintsCount(constraints, wizard_map)

#Check every constraint and count
def checkConstraintsCount(constraints, wizard_map):
    result = 0
    for c in constraints:
        if not checkConstraint(c, wizard_map):
            result += 1
    return result

#Check every constraint using the wizard dictionary.
def checkConstraints(constraints, wizard_map):
    for c in constraints:
        if not checkConstraint(c, wizard_map):
            return False
    return True

#Given the constraints and the wizard dictionary to get ages, check constra
def checkConstraint(constraint, wizard_map):
    wiz1 = constraint[0]
    wiz2 = constraint[1]
    wiz3 = constraint[2]
    age1 = wizard_map[wiz1]
    age2 = wizard_map[wiz2]
    age3 = wizard_map[wiz3]
    if age1 == -1 or age2 == -1 or age3 == -1:
        if age1 != -1 and age3 != -1 and age1 == age3:
            return False
        if age2 != -1 and age3 != -1 and age2 == age3:
            return False
        return True
    if age1 < age2:
        return age3 not in range(age1, age2 + 1)
    else:
        return age3 not in range(age2, age1 + 1)
"""
======================================================================
   No need to change any code below this line
======================================================================
"""

def read_input(filename):
    with open(filename) as f:
        num_wizards = int(f.readline())
        num_constraints = int(f.readline())
        constraints = []
        wizards = set()
        for _ in range(num_constraints):
            c = f.readline().split()
```

```
            constraints.append(c)
            for w in c:
                wizards.add(w)

    wizards = list(wizards)
    return num_wizards, num_constraints, wizards, constraints

def write_output(filename, solution):
    with open(filename, "w") as f:
        for wizard in solution:
            f.write("{0} ".format(wizard))

if __name__=="__main__":
    parser = argparse.ArgumentParser(description = "Constraint Solver.")
    parser.add_argument("input_file", type=str, help = "___.in")
    parser.add_argument("output_file", type=str, help = "___.out")
    args = parser.parse_args()

    num_wizards, num_constraints, wizards, constraints = read_input(args.inp

    solution = solve(num_wizards, num_constraints, wizards, constraints)
    write_output(args.output_file, solution)
```

In [ ]:
```
import argparse
from satispy import Variable, Cnf
from satispy.solver import Minisat
from collections import deque
from collections import defaultdict
from random import shuffle

"""
======================================================================
  Complete the following function.
======================================================================
"""

def solve(num_wizards, num_constraints, wizards, constraints):
    """
    Write your algorithm here.
    Input:
        num_wizards: Number of wizards
        num_constraints: Number of constraints
        wizards: An array of wizard names, in no particular order
        constraints: A 2D-array of constraints,
                     where constraints[0] may take the form ['A', 'B', 'C']i

    Output:
        An array of wizard names in the ordering your algorithm returns
    """
    maxVal = 0
    maxRet = wizards

    iteration = 0
    while True:
        constraintToVariable = dict()
        exp = None

        g1 = Graph(num_wizards)
        g2 = Graph(num_wizards)
        for constraint in constraints:
            wiz1 = constraint[0]
            wiz2 = constraint[1]
            wiz3 = constraint[2]

            clause1 = wiz3 + " " + wiz1
            clause2 = wiz3 + " " + wiz2
            clause3 = wiz1 + " " + wiz3
            clause4 = wiz2 + " " + wiz3

            g1.addEdge(wiz3, wiz1)
            g1.addEdge(wiz3, wiz2)
            g2.addEdge(wiz1, wiz3)
            g2.addEdge(wiz2, wiz3)

            if clause1 in constraintToVariable:
                v1 = constraintToVariable[clause1]
            else:
                constraintToVariable[clause1] = Variable(clause1)
                v1 = constraintToVariable[clause1]
```

```
        if clause2 in constraintToVariable:
            v2 = constraintToVariable[clause2]
        else:
            constraintToVariable[clause2] = Variable(clause2)
            v2 = constraintToVariable[clause2]

        if clause3 in constraintToVariable:
            v3 = constraintToVariable[clause3]
        else:
            constraintToVariable[clause3] = Variable(clause3)
            v3 = constraintToVariable[clause3]

        if clause4 in constraintToVariable:
            v4 = constraintToVariable[clause4]
        else:
            constraintToVariable[clause4] = Variable(clause4)
            v4 = constraintToVariable[clause4]

        literal = ((v1 & v2 & -v3 & -v4) ^ (v3 & v4 & -v1 & -v2))
        if exp is None:
            exp = literal
        else:
            exp = exp & literal

    solver = Minisat()
    solution = solver.solve(exp)
    if solution.success:
        graph = dict()
        g = Graph(num_wizards)
        for wizard in wizards:
            graph[wizard] = set()
        for constraint in constraintToVariable:
            v = constraintToVariable[constraint]
            if solution[v] == True:
                # print(v)
                w = str(v).split()
                vertexU = w[0]
                vertexV = w[1]
                graph[vertexU].add(vertexV)
                g.addEdge(vertexU, vertexV)

                cycle = False
                try: topological(graph)
                except ValueError: cycle = True
                if cycle:
                    graph[vertexU].remove(vertexV)
                    graph[vertexV].add(vertexU)
        cycle = False
        try: topSortGraph = topological(graph)
        except ValueError: cycle = True
    graph = {}
    if not cycle:
        ans = list(topSortGraph)
        comp = checkConstraintsWithList(constraints, ans)
        if comp > maxVal:
            maxVal = comp
```

```
                maxRet = ans
            if iteration > 100:
                return maxRet
            # print(comp)
        iteration += 1
        shuffle(wizards)
        shuffle(constraints)


GRAY, BLACK = 0, 1
def topological(graph):
    order, enter, state = deque(), set(graph), {}

    def dfs(node):
        state[node] = GRAY
        for k in graph.get(node, ()):
            sk = state.get(k, None)
            if sk == GRAY: raise ValueError("cycle")
            if sk == BLACK: continue
            enter.discard(k)
            dfs(k)
        order.appendleft(node)
        state[node] = BLACK

    while enter: dfs(enter.pop())
    return order


#This class represents a directed graph using adjacency list representation
class Graph:

    def __init__(self,vertices):
        self.V= vertices #No. of vertices
        self.graph = defaultdict(list) # default dictionary to store graph

    # function to add an edge to graph
    def addEdge(self,u,v):
        self.graph[u].append(v)

    def removeEdge(self, u, v):
        self.graph[u].remove(v)

    def containsEdge(self, u, v):
        if u in self.graph:
            if v in self.graph[u]:
                return True
        return False

    # A function used by DFS
    def DFSUtil(self,v,visited):
        # Mark the current node as visited and print it
        visited[v]= True
        # print(v)
        #Recur for all the vertices adjacent to this vertex
        for i in self.graph[v]:
            if visited[i]==False:
                self.DFSUtil(i,visited)
```

```python
def DFSSet(self, v, visited):
    s = set()
    return self.DFSSetUtil(v, visited, s)


def DFSSetUtil(self, v, visited, s):
    # Mark the current node as visited
    visited[v]= True
    s.add(v)
    #Recur for all the vertices adjacent to this vertex
    for i in self.graph[v]:
        if visited[i]==False:
            self.DFSSetUtil(i, visited, s)
    return s


def fillOrder(self,v,visited, stack):
    # Mark the current node as visited
    visited[v]= True
    #Recur for all the vertices adjacent to this vertex
    for i in self.graph[v]:
        if visited[i]==False:
            self.fillOrder(i, visited, stack)
    stack = stack.append(v)


# Function that returns reverse (or transpose) of this graph
def getTranspose(self):
    g = Graph(self.V)

    # Recur for all the vertices adjacent to this vertex
    for i in self.graph:
        for j in self.graph[i]:
            g.addEdge(j,i)
    return g



# The main function that finds and prints all strongly
# connected components
def getSCCs(self):

    stack = []
    # Mark all the vertices as not visited (For first DFS)
    # visited =[False]*(self.V)
    visited = dict()
    for u, neighbors in self.graph.items():
        visited[u] = False
        for n in neighbors:
            visited[n] = False
    # Fill vertices in stack according to their finishing
    # times
    # for i in range(self.V):
    #     if visited[i]==False:
    #         self.fillOrder(i, visited, stack)
    for u in visited:
```

```
            if visited[u] == False:
                self.fillOrder(u, visited, stack)

        # Create a reversed graph
        gr = self.getTranspose()

        # Mark all the vertices as not visited (For second DFS)
        # visited =[False]*(self.V)
        visited = dict()
        for u, neighbors in self.graph.items():
            visited[u] = False
            for n in neighbors:
                visited[n] = False

        # Now process all vertices in order defined by Stack
        sccs = []
        while stack:
            i = stack.pop()
            if visited[i]==False:
                # gr.DFSUtil(i, visited)
                s = gr.DFSSet(i, visited)
                sccs.append(s)
        return sccs

def checkConstraintsWithList(constraints, list):
    wizard_map = {}
    for i in range(len(list)):
        wizard_map[list[i]] = i
    return checkConstraintsCount(constraints, wizard_map)

def checkConstraintsCount(constraints, wizard_map):
    result = 0
    for c in constraints:
        if checkConstraint(c, wizard_map):
            result += 1
    return result

#Check every constraint using the wizard dictionary.
def checkConstraints(constraints, wizard_map):
    for c in constraints:
        if not checkConstraint(c, wizard_map):
            return False
    return True

#Given the constraints and the wizard dictionary to get ages, check constra
def checkConstraint(constraint, wizard_map):
    wiz1 = constraint[0]
    wiz2 = constraint[1]
    wiz3 = constraint[2]
    age1 = wizard_map[wiz1]
    age2 = wizard_map[wiz2]
    age3 = wizard_map[wiz3]
    if age1 == -1 or age2 == -1 or age3 == -1:
        if age1 != -1 and age3 != -1 and age1 == age3:
            return False
        if age2 != -1 and age3 != -1 and age2 == age3:
            return False
```

```
        return True
    if age1 < age2:
        return age3 not in range(age1, age2 + 1)
    else:
        return age3 not in range(age2, age1 + 1)
"""
======================================================================
   No need to change any code below this line
======================================================================
"""

def read_input(filename):
    with open(filename) as f:
        num_wizards = int(f.readline())
        num_constraints = int(f.readline())
        constraints = []
        wizards = set()
        for _ in range(num_constraints):
            c = f.readline().split()
            constraints.append(c)
            for w in c:
                wizards.add(w)

    wizards = list(wizards)
    return num_wizards, num_constraints, wizards, constraints

def write_output(filename, solution):
    with open(filename, "w") as f:
        for wizard in solution:
            f.write("{0} ".format(wizard))

if __name__=="__main__":
    parser = argparse.ArgumentParser(description = "Constraint Solver.")
    parser.add_argument("input_file", type=str, help = "___.in")
    parser.add_argument("output_file", type=str, help = "___.out")
    args = parser.parse_args()

    num_wizards, num_constraints, wizards, constraints = read_input(args.inp
###TEST
    # cycle = True
    # solution = None
    # while cycle == True:
    #     try: solution = solve(num_wizards, num_constraints, wizards, const
    #     except ValueError:
    #         cycle = True
    #         print("cycle")
    #         continue
    #     cycle = False
###ENDTEST
    solution = solve(num_wizards, num_constraints, wizards, constraints)
    write_output(args.output_file, solution)
```