**Project 1 – Josselin SOMERVILLE**

**https://github.com/JosselinSomervilleRoberts/AA228-Project-1**

**Overview of the algorithms tried**

I have tried many algorithms:

* Iteration through random orderings to perform K2
* Local Search with random initialization
* Local Search with restarts, simulated annealing and tabu
* Local Search with restarts, simulated annealing and tabu (using K2 as initialization)
* Genetic algorithm on graphs using genes between 0 and 2 to represent:
  + 0: no edge between I and j
  + 1: edge from i to j
  + 2: edge from j to i
* Genetic algorithm on orderings that are then used for K2
* Genetic algorithm on orderings that are then used for K2 followed by a local search with restarts, simulated annealing and tabu (this one gave the best results).

**Bayesian score Optimization**

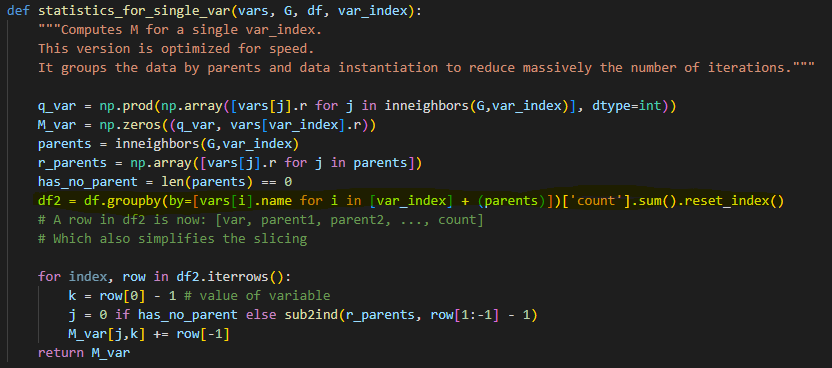
One key aspect is that I very much optimized the computation of the Bayesian score.

* I simplified the Bayesian\_score\_component function, using the fact that alpha is simply a matrix full of ones. So we can get rid of the construction of alpha and only computing its shape and then adapt the function using the fact that alpha is full of ones:

Une image contenant texte

Description générée automatiquement

* I only recomputed the component that changed when adding a parent rather than the entire score.
* I did two group by. First, when I loaded the data, I grouped it by identical realizations. Then, what really made a difference is when I compute M, I group by the data by the realization of the node and its parent, which makes it so that instead of looping through 10k lines, we only get a few hundreds at most. Also I only recomputed the M[i] needed and all the Ms, just as explained in the previous point.



* I precomputed a lot of terms that were computed at each iteration in the algorithm such as parents, r\_parents, …

**In the end, K2 ran in about 1.2s for the medium dataset with all the optimization compared to the 2 minutes people were talking about on Ed.**

**Local Search**

For local search, I added restarts if after no more than X iterations, no improvement was found. I also added Simulated annealing, which simply consists in:



Then I added a tabu. To understand the tabu, I will first explain how I generate the neighbors:

* First we chose randomly i, j such that I < j.
* Then, we chose an action 0, 1 or 2 corresponding to:
  + 0: Delete edge between I and j or j and i
  + 1: edge from i to j and delete edge from j to i if there is one
  + 2: edge from j to I and delete edge from i to j if there is one

Of course for a given graph, only two actions are possible for each edge. So there are n\*(n-1) / 2 \* 2 = n(n-1) actions possible per state. W

* We keep track of this tuple (i,j,a) and if it is already in the tabu, we don’t do the action and repeat the previous steps until finding a new action.

**Genetic algorithm**

The genetic algorithm supported all the vanilla concepts of genetic algorithms: selection, crossover, mutation and elitism. For the population initialization, I chose to either have empty graphs, random graphs or graphs generated by K2 from random orderings.

I implemented a genetic algorithm on the graphs. There were n(n-1)/2 genes corresponding for each to the state between i and j:

* + 0: no edge between I and j
  + 1: edge from i to j
  + 2: edge from j to i

The crossover was implemented as simply taking a subpart of the gene of parent 1 and filling the rest with parent 2. The mutation simply consisted in changing a gene.

**Genetic algorithm on orderings**

I tried to do a genetic algorithm on orderings to produce the best K2 solution. The genes were therefore permutations of [1, …, n]. The evaluation simply consisted on running K2 with the genes as the ordering and then computing the Bayesian score (this was expensive).

The mutation consisted of simply swapping two random nodes in the ordering.

The crossover was a bit more interesting as a node can only appear once. The idea was therefore to take a subpart of the ordering from parent 1. Then fill the rest with the ordering of parent 2 that had the subpart of parent 1 chosen removed. Here is the algorithm:

Une image contenant texte

Description générée automatiquement

**Final algorithm**

I used the Genetic algorithm on orderings and the after a certain number of generations selected the best individual and ran a local search with tabu, simulated annealing (with a quite low temperature) and tabu to optimize it even more.

**Running time**

One run of K2:

* Small: about 0.2 seconds
* Medium: about 1.2 seconds
* Large: about 120 seconds

Final algorithm full pipeline:

* Small (20 generations of 200 followed by 1000 iterations of local search): about 20 minutes
* Medium (20 generations of 100 followed by 5000 iterations of local search): about 45 minutes
* Large (5 generations of 30 followed by 20000 iterations of local search): about 6 hours.

**Graphs**

**Small**

Une image contenant bateau

Description générée automatiquement

**Medium**

**Une image contenant ciel, bateau

Description générée automatiquement**

**Large**

**Une image contenant ciel, différent

Description générée automatiquement**

**Code**

Utils.py

# This file provides useful function to work with graphs.

import pandas as pd

from datetime import datetime

import networkx as nx

import os

def seconds\_since\_beginning\_of\_project():

    # Returns the amount of seconds since the 02/01/2023 00:00:00

    return int(round((datetime.now() - datetime(2023, 2, 1)).total\_seconds()))

def seconds\_since\_beginning\_of\_project\_at\_first\_execution():

    # Returns the amount of seconds since the 02/01/2023 00:00:00

    # when the function is first called and then always returns the same value.

    if not hasattr(seconds\_since\_beginning\_of\_project\_at\_first\_execution, "first\_execution"):

        seconds\_since\_beginning\_of\_project\_at\_first\_execution.first\_execution = seconds\_since\_beginning\_of\_project()

    return seconds\_since\_beginning\_of\_project\_at\_first\_execution.first\_execution

def load\_data(infile):

    df = pd.read\_csv(infile, delimiter=',')

    df\_max = df.max()

    var\_names = list(df.columns)

    df = df.groupby(var\_names).size().reset\_index(name='count')

    vars = [Variable(var\_names[i], df\_max[i]) for i in range(len(var\_names))]

    return df, vars

def is\_cyclic(G):

    return nx.is\_directed\_acyclic\_graph(G) == False

def load\_gph(gph\_file, vars):

    G = nx.DiGraph()

    G.add\_nodes\_from(list(range(len(vars))))

    names2idx = {vars[i].name: i for i in range(len(vars))}

    with open(gph\_file, 'r') as f:

        for line in f:

            edge = line.replace('\n', '').replace(' ', '').split(',')

            G.add\_edge(names2idx[edge[0]], names2idx[edge[1]])

    return G

def write\_gph(dag, vars, data\_name, gph\_name, score=None):

    # create directory if not exists

    sec = seconds\_since\_beginning\_of\_project\_at\_first\_execution()

    score\_filename = "results/{}/{}/scores.txt".format(data\_name, sec)

    if not os.path.exists('results/{}/{}'.format(data\_name, sec)):

        os.makedirs('results/{}/{}'.format(data\_name, sec))

        f = open(score\_filename, "a") # Create a log score file

        f.close()

    idx2names = {i: vars[i].name for i in range(len(vars))}

    filename = "results/{}/{}/{}.gph".format(data\_name, sec, gph\_name)

    with open(filename, 'w') as f:

        for edge in dag.edges():

            f.write("{}, {}\n".format(idx2names[edge[0]], idx2names[edge[1]]))

    with open(score\_filename, 'a') as file:

        file.write('{} : {}\n'.format(gph\_name, score))

def inneighbors(G, i):

    """Helper function for finding the parents of a variable."""

    return list(G.predecessors(i))

class Variable:

    def \_\_init\_\_(self, name, r):

        self.name = name

        self.r = r

if \_\_name\_\_ == "\_\_main\_\_":

    print("Seconds since beginning of project: {}".format(seconds\_since\_beginning\_of\_project()))

Bayesian\_scoring,py

import scipy.special

import numpy as np

import networkx as nx

from utils import inneighbors, is\_cyclic

def prior\_shape(vars, G):

    """Algorithm 4.2 - Page 81 of the book.

    that this function returns takes the

    same form as the statistics generated by algorithm 4.1. To determine

    the appropriate dimensions, the

    function takes as input the list of

    variables vars and structure G."""

    n = len(vars)

    r = [vars[i].r for i in range(n)]

    q = np.array([np.prod(np.array([r[j] for j in inneighbors(G,i)])) for i in range(n)], dtype=int)

    return [(q[i], r[i]) for i in range(n)]

def prior\_shape\_for\_single\_var(vars, G, var\_index):

    q\_var = np.prod(np.array([vars[j].r for j in inneighbors(G,var\_index)]))

    return (q\_var, vars[var\_index].r)

def sub2ind(siz, x):

    """Algorithm 4.1. - Page 75 of the book - Helper function."""

    return np.ravel\_multi\_index(x, siz)

def statistics(vars, G, df):

    """Algorithm 4.1. - Page 75 of the book.

    A function for extracting the statistics, or counts,

    from a discrete data set D, assuming a Bayesian network with variables vars and structure G. The

    data set is an n x m matrix, where

    n is the number of variables and

    m is the number of data points.

    This function returns an array M of

    length n. The ith component consists of a qi x ri matrix of counts.

    The sub2ind(siz, x) function returns a linear index into an array

    with dimensions specified by siz

    given coordinates x. It is used to

    identify which parental instantiation is relevant to a particular data

    point and variable."""

    n = len(vars)

    r = np.array([vars[i].r for i in range(n)])

    q = np.array([np.prod(np.array([r[j] for j in inneighbors(G,i)])) for i in range(n)], dtype=int)

    M = [np.zeros((q[i], r[i])) for i in range(n)]

    for var\_index in range(n):

        parents = inneighbors(G,var\_index)

        r\_parents = np.array([vars[j].r for j in parents])

        has\_no\_parent = len(parents) == 0

        df2 = df.groupby(by=[vars[i].name for i in [var\_index] + (parents)])['count'].sum().reset\_index()

        for index, row in df2.iterrows():

            k = row[0] - 1 # value of variable

            j = 0 if has\_no\_parent else sub2ind(r\_parents, row[1:-1] - 1)

            M[var\_index][j,k] += row[-1]

    return M

def statistics\_for\_single\_var(vars, G, df, var\_index):

    """Computes M for a single var\_index.

    This version is optimized for speed.

    It groups the data by parents and data instantiation to reduce massively the number of iterations."""

    q\_var = np.prod(np.array([vars[j].r for j in inneighbors(G,var\_index)], dtype=int))

    M\_var = np.zeros((q\_var, vars[var\_index].r))

    parents = inneighbors(G,var\_index)

    r\_parents = np.array([vars[j].r for j in parents])

    has\_no\_parent = len(parents) == 0

    df2 = df.groupby(by=[vars[i].name for i in [var\_index] + (parents)])['count'].sum().reset\_index()

    # A row in df2 is now: [var, parent1, parent2, ..., count]

    # Which also simplifies the slicing

    for index, row in df2.iterrows():

        k = row[0] - 1 # value of variable

        j = 0 if has\_no\_parent else sub2ind(r\_parents, row[1:-1] - 1)

        M\_var[j,k] += row[-1]

    return M\_var

def bayesian\_score\_component(M, alpha\_shape):

    """Algorithm 5.1 - Page 98 of the book - Helper function."""

    # I've optimized the next line by using the fact that alpha is a vector of 1s

    # p = np.sum(scipy.special.loggamma(alpha + M))

    p = np.sum(scipy.special.loggamma(1 + M))

    # I've removed the next line because with a prior of 1, the loggamma of alpha is 0

    # p -= np.sum(scipy.special.loggamma(alpha))

    # The next line has been removed to be optimized by what follows (using the fact that alpha is a vector of 1s)

    # p += np.sum(scipy.special.loggamma(np.sum(alpha, axis=1)))

    p += alpha\_shape[0] \* np.log(scipy.special.factorial(alpha\_shape[1] - 1))

    # I've optimized the next line by using the fact that alpha is a vector of 1s

    # p -= np.sum(scipy.special.loggamma(np.sum(alpha, axis=1) + np.sum(M, axis=1)))

    p -= np.sum(scipy.special.loggamma(alpha\_shape[1] + np.sum(M, axis=1)))

    return p

def bayesian\_score(vars, G, D):

    """Algorithm 5.1 - Page 98 of the book.

    for computing the Bayesian score

    for a list of variables vars and

    a graph G given data D. This

    method uses a uniform prior

    αijk = 1 for all i, j, and k

    as generated by algorithm 4.2.

    The loggamma function is provided

    by SpecialFunctions.jl. Chapter 4 introduced the statistics

    and prior functions. Note that

    log(Γ(α)/Γ(α + m)) = log Γ(α) −

    log Γ(α + m), and that log Γ(1) =

    0."""

    if is\_cyclic(G):

        return - np.inf, None

    n = len(vars)

    M = statistics(vars, G, D)

    alpha\_shape = prior\_shape(vars, G)

    score\_components = np.array([bayesian\_score\_component(M[i], alpha\_shape[i]) for i in range(n)])

    score = np.sum(score\_components)

    return score, score\_components

def bayesian\_score\_recompute\_single\_var(previous\_score, previous\_score\_components, vars, G, D, var\_index):

    """Recomputes the bayesian score form a previous score after adding a node from i to j."""

    M\_j = statistics\_for\_single\_var(vars, G, D, var\_index)

    alpha\_shape\_j = prior\_shape\_for\_single\_var(vars, G, var\_index)

    new\_score\_component = bayesian\_score\_component(M\_j, alpha\_shape\_j)

    score = previous\_score + new\_score\_component - previous\_score\_components[var\_index]

    return score, new\_score\_component

if \_\_name\_\_ == "\_\_main\_\_":

    from tqdm import tqdm

    import pandas as pd

    import networkx as nx

    from utils import Variable

    df = pd.read\_csv('example/example.csv', delimiter=',')

    df\_max = df.max()

    var\_names = list(df.columns)

    df = df.groupby(var\_names).size().reset\_index(name='count')

    vars = [Variable(var\_names[i], df\_max[i]) for i in range(len(var\_names))]

    # JUST FOR TESTING

    G = nx.DiGraph()

    for i in range(len(vars)): G.add\_node(i)

    for i in range(len(vars)//2): G.add\_edge(2\*i, 2\*i+1)

    NUM\_ITER\_TIMEIT = 100

    from time import time

    initial\_score, init\_comp = bayesian\_score(vars, G, df)

    print("Initial Bayesian score: {}".format(initial\_score))

    print("Initial Bayesian score components: {}".format(init\_comp))

    G.add\_edge(0, 2)

    t\_start = time()

    for \_ in tqdm(range(NUM\_ITER\_TIMEIT)):

        new\_score, new\_comp = bayesian\_score(vars, G, df)

    t\_end = time()

    print("\nNew Bayesian score: {}".format(new\_score))

    print("New Bayesian score components: {}".format(new\_comp))

    print("Time taken for {} iterations: {} s".format(NUM\_ITER\_TIMEIT, round(t\_end - t\_start, 2)))

    t\_start = time()

    for \_ in tqdm(range(NUM\_ITER\_TIMEIT)):

        clever\_score, clever\_comp = bayesian\_score\_recompute\_single\_var(initial\_score, init\_comp, vars, G, df, 2)

    t\_end = time()

    clever\_compoments = init\_comp.copy()

    clever\_compoments[2] = clever\_comp

    print("\nClever Bayesian score: {}".format(clever\_score))

    print("Clever Bayesian score components: {}".format(clever\_compoments))

    print("Time taken for {} iterations: {} s".format(NUM\_ITER\_TIMEIT, round(t\_end - t\_start, 2)))

Evaluate\_graph.py

from bayesian\_scoring import bayesian\_score

from utils import load\_gph, load\_data

import sys

if \_\_name\_\_ == "\_\_main\_\_":

    # Check arguments

    if len(sys.argv) != 3:

        raise Exception("usage: python evaluate\_graph.py <infile>.csv, <gphfile>.gph")

    inputfilename = sys.argv[1]

    gphfilename = sys.argv[2]

    # Load data

    df, vars = load\_data(inputfilename)

    # Load graph

    G = load\_gph(gphfilename, vars)

    # Score graph

    score = bayesian\_score(vars, G, df)

    print("Score: {}".format(score))

K2\_search.py

import networkx as nx

from tqdm import tqdm

import numpy as np

from bayesian\_scoring import bayesian\_score, bayesian\_score\_recompute\_single\_var

from utils import write\_gph, inneighbors

def k2\_iter(vars, df, num\_iter, max\_parents=2, data\_name="small"):

    #past\_orderings = set()

    best\_score = -np.inf

    best\_G = None

    # Compute empty score

    G = nx.DiGraph()

    G.add\_nodes\_from(list(range(len(vars))))

    empty\_score, empty\_score\_comp = bayesian\_score(vars, G, df)

    for idx in tqdm(range(num\_iter)):

        # generate a random ordering

        ordering = np.random.permutation(len(vars))

        #while ordering in past\_orderings:

        #    ordering = np.random.permutation(len(vars))

        #past\_orderings.add(ordering)

        # run k2 on the ordering

        G, score = k2(ordering, vars, df, max\_parents=max\_parents, empty\_score=empty\_score, empty\_score\_comp=empty\_score\_comp.copy())

        if score > best\_score:

            best\_score = score

            best\_G = G

            write\_gph(best\_G, vars, data\_name=data\_name, gph\_name="k2\_" + str(idx), score=best\_score)

            print("New best score: {}".format(best\_score))

    return best\_G, best\_score

# K2 algorithm

def k2(ordering, vars, df, max\_parents=2, empty\_score=None, empty\_score\_comp=None):

    G = nx.DiGraph()

    G.add\_nodes\_from(list(range(len(ordering))))

    score, score\_comp = empty\_score, empty\_score\_comp

    if score is None or score\_comp is None: score, score\_comp = bayesian\_score(vars, G, df)

    for (k, i) in enumerate(ordering[1:]):

        if len(inneighbors(G, i)) >= max\_parents:

            continue

        while True:

            score\_best, j\_best, score\_comp\_best = -np.inf, 0, None

            for j in ordering[:k]:

                if not G.has\_edge(j, i):

                    G.add\_edge(j, i)

                    new\_score, new\_score\_comp = bayesian\_score\_recompute\_single\_var(score, score\_comp, vars, G, df, i)

                    if new\_score > score\_best:

                        score\_best, j\_best, score\_comp\_best = new\_score, j, new\_score\_comp

                    G.remove\_edge(j, i)

            if score\_best > score:

                score = score\_best

                score\_comp[i] = score\_comp\_best

                G.add\_edge(j\_best, i)

            else:

                break

    return G, score\_best

if \_\_name\_\_ == "\_\_main\_\_":

    import sys

    from utils import load\_data, write\_gph

    # Check arguments

    if len(sys.argv) != 2:

        raise Exception("usage: python k2\_search.py <infile>.csv")

    inputfilename = sys.argv[1]

    data\_name = inputfilename.split("/")[-1].split(".")[0]

    # Load data

    df, vars = load\_data(inputfilename)

    # Run k2

    G, score = k2\_iter(vars, df, 1000, max\_parents=4, data\_name=data\_name)

    print("Best score: {}".format(score))

    write\_gph(G, vars, data\_name=data\_name, gph\_name="k2\_best", score=score)

Local\_search.py

from utils import write\_gph, is\_cyclic

import networkx as nx

from tqdm import tqdm

import numpy as np

from bayesian\_scoring import bayesian\_score, bayesian\_score\_recompute\_single\_var

def rand\_graph\_neighbor\_with\_score(G, score, score\_comp, df, vars, tabu=None):

    # There is a total of n(n-1) posssible actions

    n = G.number\_of\_nodes()

    # Tabu search

    first\_iter = True

    while tabu is not None and (first\_iter or (i, j, action) in tabu):

        first\_iter = False

        i = np.random.randint(1, n)

        j = i

        while j == i:

            j = np.random.randint(1, n)

        if i > j: i, j = j, i

        actions = [0, 1, 2]

        if G.has\_edge(i, j):

            actions.remove(1)

        elif G.has\_edge(j, i):

            actions.remove(2)

        else:

            actions.remove(0)

        action = actions[0]

        possible\_actions = [action for action in actions if (i, j, action) not in tabu]

        action = np.random.choice(possible\_actions)

    tabu.add((i, j))

    # Generate neighbor

    G\_prime = G.copy()

    recompute\_i = False

    recompute\_j = False

    if action == 0:

        if G.has\_edge(i, j):

            G\_prime.remove\_edge(i, j)

            recompute\_j = True

        if G.has\_edge(j, i):

            G\_prime.remove\_edge(j, i)

            recompute\_i = True

    elif action == 1:

        if G.has\_edge(j, i):

            G\_prime.remove\_edge(j, i)

            recompute\_i = True

        G\_prime.add\_edge(i, j)

        recompute\_j = True

    elif action == 2:

        if G.has\_edge(i, j):

            G\_prime.remove\_edge(i, j)

            recompute\_j = True

        G\_prime.add\_edge(j, i)

        recompute\_i = True

    # Check if neighbor is cyclic

    if is\_cyclic(G\_prime):

        return G\_prime, None, (None, None), (i, j)

    # Compute new score

    score\_prime, score\_comp\_prime\_i, score\_comp\_prime\_j = score, score\_comp[i], score\_comp[j]

    if recompute\_i:

        score\_prime, score\_comp\_prime\_i = bayesian\_score\_recompute\_single\_var(score\_prime, score\_comp, vars, G\_prime, df, i)

    if recompute\_j:

        score\_prime, score\_comp\_prime\_j = bayesian\_score\_recompute\_single\_var(score\_prime, score\_comp, vars, G\_prime, df, j)

    return G\_prime, score\_prime, (score\_comp\_prime\_i, score\_comp\_prime\_j), (i, j)

# Local Search algorithm

def local\_search(vars, df, k\_max, data\_name, G=None):

    # Generate initial graph

    if G is None:

        G = nx.DiGraph()

        G.add\_nodes\_from(list(range(len(vars))))

    score, score\_comp = bayesian\_score(vars, G, df)

    for k in tqdm(range(k\_max)):

        G\_prime, score\_prime, score\_comp\_prime, j = rand\_graph\_neighbor\_with\_score(G, score, score\_comp, df, vars)

        if is\_cyclic(G\_prime):

            continue

        if score\_prime > score:

            score = score\_prime

            score\_comp[j] = score\_comp\_prime

            G = G\_prime

            print("New best score: {}".format(score))

            if score > -425000: write\_gph(G, vars, data\_name=data\_name, gph\_name="local\_best\_" + str(k), score=score)

    return G, score

# Local Search algorithm with Simulated annealing, random restarts and random initializations

def random\_graph\_init(vars, df):

    G = nx.DiGraph()

    G.add\_nodes\_from(list(range(len(vars))))

    score, score\_comp = bayesian\_score(vars, G, df)

    return G, score, score\_comp

def local\_search\_with\_optis(vars, df, k\_max, data\_name, G=None,

                            t\_max=5,

                            k\_max\_without\_improvements=2000,

                            score\_improvement\_to\_save=5,

                            score\_min\_to\_save=-4200,

                            log\_score\_every=1000,

                            return\_on\_restart=False):

    # Generate initial graph

    score, score\_comp = None, None

    init\_G = G.copy()

    if G is None:

        G, score, score\_comp = random\_graph\_init(vars, df)

    else:

        score, score\_comp = bayesian\_score(vars, G, df)

    # To keep track of the best graph

    last\_saved\_score = -np.inf

    k\_last\_improvement = -1

    k\_last\_restart = 0

    score\_of\_last\_improvement = - np.inf

    best\_G = G.copy()

    # Tabu

    tabu = set()

    tabu\_full\_threshold = 0.9 \* len(vars) \* (len(vars) - 1)

    for k in tqdm(range(k\_max)):

        temp = t\_max \* (1 - (k) / (k\_max) )

        G\_prime, score\_prime, (score\_comp\_prime\_i, score\_comp\_prime\_j), (i, j) = rand\_graph\_neighbor\_with\_score(G, score, score\_comp, df, vars, tabu=tabu)

        if score\_prime is None: continue # This means that the graph is cyclic

        # Simulated annealing

        diff = score\_prime - score

        if diff > 0 or np.random.rand() < np.exp(diff/temp):

            score = score\_prime

            score\_comp[i] = score\_comp\_prime\_i

            score\_comp[j] = score\_comp\_prime\_j

            G = G\_prime

            tabu.clear()

        # Random restarts

        if score > score\_of\_last\_improvement:

            k\_last\_improvement = k

            score\_of\_last\_improvement = score

            best\_G = G.copy()

        if k - k\_last\_improvement > k\_max\_without\_improvements or len(tabu) > tabu\_full\_threshold: # No improvement for k\_max\_without\_improvements steps or Tabu is 90% full

            if return\_on\_restart:

                print("Stopping at step {} ({} steps without improvement)".format(k, k - k\_last\_improvement))

                return G, score

            print("Restarting at step {} ({} steps without improvement)".format(k, k - k\_last\_improvement))

            k\_last\_restart = k

            k\_last\_improvement = k

            G = best\_G.copy()

            score, score\_comp = bayesian\_score(vars, G, df)

            tabu.clear()

        # Saving best graph

        if diff > 0 and score >= score\_min\_to\_save and score >= last\_saved\_score + score\_improvement\_to\_save:

            last\_saved\_score = score

            print("New best score: {}".format(score))

            write\_gph(G, vars, data\_name=data\_name, gph\_name="tabu\_score\_" + str(int(round(-score))), score=score)

        # Logging

        if k % log\_score\_every == 0:

            print("Current score: {}".format(score))

    return G, score

if \_\_name\_\_ == "\_\_main\_\_":

    import sys

    from utils import load\_data, write\_gph

    # Check arguments

    if len(sys.argv) != 2:

        raise Exception("usage: python k2\_search.py <infile>.csv")

    inputfilename = sys.argv[1]

    data\_name = inputfilename.split("/")[-1].split(".")[0]

    # Optims

    use\_optims = True if len(sys.argv) == 3 and sys.argv[2] == "optims" else False

    if use\_optims:

        print("Using optims")

    else:

        print("Not using optims")

    # Load data

    df, vars = load\_data(inputfilename)

    # Run k2

    G, score = None, None

    if use\_optims:

        G, score = local\_search\_with\_optis(vars, df, 1000, data\_name=data\_name)

    else:

        G, score = local\_search(vars, df, 1000, data\_name=data\_name)

    print("Best score: {}".format(score))

    write\_gph(G, vars, data\_name=data\_name, gph\_name="local\_best", score=score)

Genetic\_utils.py

# This file provides efficient functions to run genetic algorithm

# It handles population initialization, crossover, mutation, and selection

import random

import numpy as np

import matplotlib.pyplot as plt

import time

import copy

import matplotlib.pyplot as plt

from scipy.stats import norm

from scipy import stats

import networkx as nx

from tqdm import tqdm

from utils import write\_gph

# This function initializes the population

# Input: population size, number of genes, gene range

# Output: population

def init\_population(pop\_size, num\_genes, gene\_range):

    population = np.random.randint(gene\_range[0], gene\_range[1], (pop\_size, num\_genes))

    # population = np.zeros((pop\_size, num\_genes))

    return population

# This function performs crossover

# Input: population, crossover rate

# Output: population after crossover

def crossover(population, cross\_rate):

    pop\_size = population.shape[0]

    num\_genes = population.shape[1]

    for i in range(0, pop\_size, 2):

        if random.random() < cross\_rate:

            cross\_point = random.randint(0, num\_genes - 1)

            temp1 = copy.deepcopy(population[i, cross\_point:])

            temp2 = copy.deepcopy(population[i + 1, cross\_point:])

            population[i, cross\_point:] = temp2

            population[i + 1, cross\_point:] = temp1

    return population

# This function performs mutation

# Input: population, mutation rate

# Output: population after mutation

def mutation(population, mut\_rate):

    pop\_size = population.shape[0]

    num\_genes = population.shape[1]

    for i in range(pop\_size):

        if random.random() < mut\_rate:

            mut\_point = random.randint(0, num\_genes - 1)

            population[i, mut\_point] = random.randint(0, 2)

    return population

# This function performs selection

# Input: population, fitness, selection rate

# Output: population after selection

def selection(population, fitness, select\_rate):

    pop\_size = population.shape[0]

    num\_genes = population.shape[1]

    fitness = fitness / np.sum(fitness)

    fitness = np.cumsum(fitness)

    new\_population = np.zeros((pop\_size, num\_genes))

    for i in range(pop\_size):

        rand = random.random()

        for j in range(pop\_size):

            if rand < fitness[j]:

                new\_population[i, :] = population[j, :]

                break

    return new\_population

# This function performs genetic algorithm

# Input: population size, number of genes, gene range, crossover rate, mutation rate, selection rate, number of generations

# Output: best individual, best fitness, average fitness, and time

def genetic(pop\_size, num\_genes, gene\_range, cross\_rate, mut\_rate, select\_rate, num\_generations, fitness\_function, saving\_function=None):

    print('Start Genetic Algorithm')

    print('Population Size: ', pop\_size, ' Number of Genes: ', num\_genes, ' Gene Range: ', gene\_range, ' Crossover Rate: ', cross\_rate, ' Mutation Rate: ', mut\_rate, ' Selection Rate: ', select\_rate, ' Number of Generations: ', num\_generations)

    population = init\_population(pop\_size, num\_genes, gene\_range)

    best\_fitness = np.zeros(num\_generations)

    average\_fitness = np.zeros(num\_generations)

    start\_time = time.time()

    for i in range(num\_generations):

        fitness = np.zeros(pop\_size)

        for j in tqdm(range(pop\_size)):

            fitness[j] = fitness\_function(population[j, :])

        best\_fitness[i] = np.max(fitness)

        average\_fitness[i] = np.mean(fitness)

        population = selection(population, fitness, select\_rate)

        population = crossover(population, cross\_rate)

        population = mutation(population, mut\_rate)

        if saving\_function is not None: print('Generation: ', i, ' Best Fitness: ', best\_fitness[i], ' Average Fitness: ', average\_fitness[i])

        else: print('Generation: ', i, ' Best Fitness: ', best\_fitness[i], ' Average Fitness: ', average\_fitness[i], ' Best sentence: ', ''.join(chr(int(i)) for i in population[np.argmax(fitness), :]))

        if saving\_function is not None:

            saving\_function(population[np.argmax(fitness), :], best\_fitness[i], i)

    end\_time = time.time()

    best\_individual = population[np.argmax(fitness), :]

    return best\_individual, best\_fitness, average\_fitness, end\_time - start\_time

# This function plots the fitness

# Input: best fitness, average fitness, number of generations

# Output: plot

def plot\_fitness(best\_fitness, average\_fitness, num\_generations):

    x = np.arange(num\_generations)

    plt.plot(x, best\_fitness, 'r', label = 'Best Fitness')

    plt.plot(x, average\_fitness, 'b', label = 'Average Fitness')

    plt.xlabel('Generation')

    plt.ylabel('Fitness')

    plt.legend()

    plt.show()

# This function shows that the genetic algorithm works

# The example is to find a string equal to "Hello World"

# The genes are each character in the string.

# The fitness function is the number of characters that are correct.

# The gene range is 0 to 127, which is the ASCII code for each character.

# The crossover rate is 0.8, the mutation rate is 0.01, the selection rate is 0.2, and the number of generations is 100.

# The population size is 1000, and the number of genes is 11.

# The best individual is "Hello World", and the best fitness is 11.

# The average fitness is 5.5.

# The time is 0.5 seconds.

def test():

    pop\_size = 1000

    num\_genes = 11

    gene\_range = [0, 127]

    cross\_rate = 0.8

    mut\_rate = 0.01

    select\_rate = 0.2

    num\_generations = 100

    best\_individual, best\_fitness, average\_fitness, run\_time = genetic(pop\_size, num\_genes, gene\_range, cross\_rate, mut\_rate, select\_rate, num\_generations, fitness\_function\_string)

    print('Best Individual: ', best\_individual)

    print('Best Fitness: ', best\_fitness[-1])

    print('String of best individual', ''.join(chr(int(i)) for i in best\_individual))

    print('Average Fitness: ', average\_fitness[-1])

    print('Time: ', run\_time)

    plot\_fitness(best\_fitness, average\_fitness, num\_generations)

# This function is the fitness function

# Input: individual

# Output: fitness

def fitness\_function\_string(individual):

    fitness = 0

    for i in range(len(individual)):

        if individual[i] == ord('Hello World'[i]):

            fitness += 1

    return fitness

if \_\_name\_\_ == '\_\_main\_\_':

    test()

# This function creates a graph given an individual

# Input: individual, list of column names

# Output: graph

def create\_graph\_from\_gene(individual, vars):

    graph = nx.DiGraph()

    graph.add\_nodes\_from(list(range(len(vars))))

    index = 0

    for i in range(len(vars)):

        for j in range(i + 1, len(vars)):

            if individual[index] == 1:

                graph.add\_edge(i, j)

            elif individual[index] == 2:

                graph.add\_edge(j, i)

            index += 1

    return graph

Genetic.py

from genetic\_utils import genetic, create\_graph\_from\_gene, plot\_fitness

from bayesian\_scoring import bayesian\_score

import numpy as np

# This function uses the genetic algorithm to find the best bayesian network

# given an input pandas dataframe called df (provided as a parameter)

# and a list of column names called cols (provided as a parameter)

# The fitness function is the Bayesian Score of the network.

# Only part of the Bayesian score is recomputed to save time.

# The gene range is 0 to 2, which corresponds to:

# 0 = no edge

# 1 = directed edge from i to j

# 2 = directed edge from j to i

# The number of genes is the number of possible edges in the network, which is n(n-1)/2.

# The crossover rate is 0.8, the mutation rate is 0.01, the selection rate is 0.2, and the number of generations is 100.

# The population size is 1000.

# The best fitness is the Bayesian Score of the best network.

# The time is 0.5 seconds.

def fitness\_function\_bayesian(individual, df, vars):

    G = create\_graph\_from\_gene(individual, vars)

    score, score\_comp = bayesian\_score(vars, G, df)

    if score < -10000000000: return 0

    return score + 4350

def run(df, vars, data\_name):

    pop\_size = 500

    num\_genes = int(len(vars) \* (len(vars) - 1) / 2)

    gene\_range = [0, 2]

    cross\_rate = 0.6

    mut\_rate = 0.05

    select\_rate = 0.3

    num\_generations = 200

    fitness\_function = lambda individual: fitness\_function\_bayesian(individual, df, vars)

    saving\_function = lambda individual, score, iter: write\_gph(create\_graph\_from\_gene(individual, vars), vars, data\_name=data\_name, gph\_name="gen\_" + str(iter), score=score)

    best\_individual, best\_fitness, average\_fitness, run\_time = genetic(pop\_size, num\_genes, gene\_range, cross\_rate, mut\_rate, select\_rate, num\_generations, fitness\_function, saving\_function=saving\_function)

    print('Best Individual: ', best\_individual)

    print('Best Fitness: ', best\_fitness[-1])

    print('Time: ', run\_time)

    plot\_fitness(best\_fitness, average\_fitness, num\_generations)

    return create\_graph\_from\_gene(best\_individual, vars), best\_fitness[-1]

if \_\_name\_\_ == '\_\_main\_\_':

    import sys

    from utils import load\_data, write\_gph

    # Check arguments

    if len(sys.argv) != 2:

        raise Exception("usage: python genetic.py <infile>.csv")

    inputfilename = sys.argv[1]

    data\_name = inputfilename.split("/")[-1].split(".")[0]

    # Load data

    df, vars = load\_data(inputfilename)

    # run the genetic algorithm

    G, score = run(df, vars, data\_name)

    print("Best score: {}".format(score))

    write\_gph(G, vars, data\_name=data\_name, gph\_name="best", score=score)

Genetic\_permutations.py

# This file is mostly a duplicate of genetic\_utils.py with some variations

# of the algorithm to handle permutations

import random

import numpy as np

import matplotlib.pyplot as plt

import time

import copy

import matplotlib.pyplot as plt

from scipy.stats import norm

from scipy import stats

import networkx as nx

from tqdm import tqdm

from utils import write\_gph

import random

from k2\_search import k2

from bayesian\_scoring import bayesian\_score

from heapq import heappush, heappushpop, heappop

from local\_search import local\_search\_with\_optis

# This function initializes the population

# Input: population size, number of genes, gene range

# Output: population

def init\_population(pop\_size, num\_genes):

    population = np.array([np.random.permutation(num\_genes) for i in range(pop\_size)])

    return population

def fitness\_fn\_permutation(individual, vars, df, max\_parents=2, empty\_score=None, empty\_score\_comp=None):

    # This function computes the fitness of an individual by computing

    # the bayesian score of the best graph found by the K2 algorithm

    # with the individual as the ordering of the variables

    # Compute the best graph found by the K2 algorithm

    G, score = k2(individual, vars, df, max\_parents=max\_parents, empty\_score=empty\_score, empty\_score\_comp=empty\_score\_comp)

    return score, G

class Record:

    def \_\_init\_\_(self, fitness, individual, G=None):

        self.fitness = fitness

        self.individual = individual

        self.G = G

    def \_\_lt\_\_(self, other):

        return self.fitness < other.fitness

    def \_\_eq\_\_(self, other):

        return self.fitness == other.fitness

def genetic\_algorithm(fitness\_fn, size\_population, nb\_genes,

                     num\_generations, fraction\_elites, probability\_crossover,

                     probability\_mutation,

                     data\_name, vars,

                     keep\_best\_nb=10,

                     nb\_gens\_max\_without\_improvement=10):

    # Print parameters of the algorithm

    num\_elites = int(size\_population - 2 \* np.ceil(((1 -fraction\_elites) \* size\_population) / 2))

    print(f"Population size: {size\_population} | Number of generations: {num\_generations} | Number of elites: {num\_elites} | Probability of crossover: {probability\_crossover} | Probability of mutation: {probability\_mutation}")

    print(f"Number of genes: {nb\_genes} | Data name: {data\_name} | Keep best nb: {keep\_best\_nb} | Max gens without improvement: {nb\_gens\_max\_without\_improvement}")

    # Initialize the population

    population = init\_population(size\_population, nb\_genes)

    fitness\_scores = np.zeros(size\_population)

    new\_fitness = [None] \* size\_population

    best\_individuals = []

    G = [None] \* size\_population

    idx\_best\_saved = 0

    # best score (for ealy stopping)

    best\_score = -np.inf

    last\_gen\_improvement = -1

    for generation in range(num\_generations):

        # Evaluate the fitness of each individual in the population

        for i, individual in enumerate(tqdm(population)):

            if new\_fitness[i] is None: # If the individual was modified, it needs to be recomputed

                fitness\_scores[i], G[i] = fitness\_fn(individual)

            else: # it is an individual from the previous generation, so we already know it's score

                fitness\_scores[i] = new\_fitness[i]

        new\_fitness = [None] \* size\_population

        # Sort the population by fitness in descending order

        indices = np.argsort(-fitness\_scores)

        for i in range(min(keep\_best\_nb, size\_population)):

            rec = Record(fitness\_scores[indices[i]], population[indices[i]], G[indices[i]])

            if len(best\_individuals) < keep\_best\_nb:

                heappush(best\_individuals, rec)

                idx\_best\_saved += 1

                write\_gph(rec.G, vars, data\_name=data\_name, gph\_name="genetic\_" + str(idx\_best\_saved), score=fitness\_scores[indices[i]])

            elif not rec in best\_individuals:

                rec\_popped = heappushpop(best\_individuals, rec)

                if rec\_popped.fitness == fitness\_scores[indices[i]]: # if the individual was not added to the heap

                    # no individual will be added since they are sorted, so it will only get worse

                    break

                else: # new best individual, so we save its graph

                    idx\_best\_saved += 1

                    write\_gph(rec.G, vars, data\_name=data\_name, gph\_name="genetic\_" + str(idx\_best\_saved), score=fitness\_scores[indices[i]])

        # Displays useful information about the current generation

        print(f"Generation {generation + 1} | Best fitness: {fitness\_scores[indices[0]]} | Average fitness: {np.mean(fitness\_scores)}")

        # Early stoping if no improvement for X generations

        if fitness\_scores[indices[0]] > best\_score:

            best\_score = fitness\_scores[indices[0]]

            last\_gen\_improvement = generation

        elif generation - last\_gen\_improvement >= nb\_gens\_max\_without\_improvement:

            break

        # Select the elites individuals

        elites = [population[idx] for idx in indices[:num\_elites]]

        new\_fitness[:num\_elites] = fitness\_scores[indices[:num\_elites]]

        # Generate the offspring for the next generation

        offspring = elites[:]

        while len(offspring) < size\_population:

            # Select two parents using the tournament selection method

            parent1, fitness\_parent\_1, parent2, fitness\_parent\_2 = tournament\_selection(population, fitness\_scores, 2)

            # Crossover with a probability of `probability\_crossover`

            if random.random() < probability\_crossover:

                child1, child2 = crossover(parent1, parent2)

                offspring.append(child1)

                offspring.append(child2)

            else:

                new\_fitness[len(offspring)] = fitness\_parent\_1

                offspring.append(parent1)

                new\_fitness[len(offspring)] = fitness\_parent\_2

                offspring.append(parent2)

        # Mutate the offspring with a probability of `probability\_mutation`

        for i in range(len(offspring)):

            if random.random() < probability\_mutation:

                mutate(offspring[i])

                new\_fitness[i] = None

        # Replace the population with the offspring

        population = np.array(offspring)

    # Return the best individuals found

    return best\_individuals

def tournament\_selection(population, fitness\_scores, size\_tournament):

    indices = np.random.choice(len(population), size\_tournament, replace=False)

    tournament = population[indices]

    tournament\_fitness = fitness\_scores[indices]

    indices = np.argsort(-tournament\_fitness)

    return tournament[indices[0]], tournament\_fitness[indices[0]], tournament[indices[1]], tournament\_fitness[indices[1]]

def mix\_genes(genes1, genes2, i, j):

    new\_genes = np.zeros(len(genes1), dtype=int)

    new\_genes[i:j] = genes1[i:j]

    remaining = [gene for gene in genes2 if gene not in genes1[i:j]]

    new\_genes[:i] = remaining[:i]

    new\_genes[j:] = remaining[i:]

    return new\_genes

def crossover(parent1, parent2):

    # randomly keeps a section of parent1 and fills the rest by the order specified by the gene of parent2

    # It is not possible to simply fill the missing genes byt the genes of parent2 as we are modeling permutations

    # and therefore each gene can only appear once

    i = random.randint(0, len(parent1) - 1)

    j = random.randint(0, len(parent1) - 1)

    if i > j:

        i, j = j, i

    child1 = mix\_genes(parent1, parent2, i, j)

    child2 = mix\_genes(parent2, parent1, i, j)

    return child1, child2

def mutate(individual):

    # randomly swaps two genes of the individual

    i = random.randint(0, len(individual) - 1)

    j = i

    while j == i: j = random.randint(0, len(individual) - 1)

    individual[i], individual[j] = individual[j], individual[i]

if \_\_name\_\_ == "\_\_main\_\_":

    import sys

    from utils import load\_data, write\_gph, load\_gph

    # Check arguments

    if len(sys.argv) < 2:

        raise Exception("usage: python k2\_search.py <infile>.csv")

    inputfilename = sys.argv[1]

    data\_name = inputfilename.split("/")[-1].split(".")[0]

    # Load data

    df, vars = load\_data(inputfilename)

    # If it's only reoptimization

    best\_G = None

    only\_reoptimization = False

    if len(sys.argv) == 3:

        gph\_name = sys.argv[2]

        best\_G = load\_gph(gph\_name, vars)

        only\_reoptimization = True

        print("Reoptimization of the graph " + gph\_name)

    if not only\_reoptimization:

        # Compute empty score

        G = nx.DiGraph()

        G.add\_nodes\_from(list(range(len(vars))))

        empty\_score, empty\_score\_comp = bayesian\_score(vars, G, df)

        # Constants

        max\_parents = 4

        # Define the fitness function

        fitness\_fn = lambda individual: fitness\_fn\_permutation(individual, vars, df, max\_parents=max\_parents, empty\_score=empty\_score, empty\_score\_comp=empty\_score\_comp.copy())

        # Run the genetic algorithm

        best\_individuals = genetic\_algorithm(fitness\_fn,

                                            nb\_genes=len(vars),

                                            size\_population=30,

                                            num\_generations=10,

                                            fraction\_elites=0.18,

                                            probability\_crossover=0.8,

                                            probability\_mutation=0.1,

                                            data\_name=data\_name,

                                            vars=vars,

                                            nb\_gens\_max\_without\_improvement=3)

        best\_ordering, best\_score, best\_G = None, None, None

        while len(best\_individuals) > 0:

            rec = heappop(best\_individuals)

            best\_score = rec.fitness

            best\_ordering = rec.individual

            best\_G = rec.G

        if best\_G is None: best\_G, best\_score = k2(best\_ordering, vars, df, max\_parents=max\_parents, empty\_score=empty\_score, empty\_score\_comp=empty\_score\_comp)

        print("Best score: {}".format(best\_score))

        write\_gph(best\_G, vars, data\_name=data\_name, gph\_name="best", score=best\_score)

    # Improve with local search

    best\_G, best\_score = local\_search\_with\_optis(vars, df, k\_max=100000, data\_name=data\_name, G=best\_G, t\_max=40.0,

                            k\_max\_without\_improvements=1000,

                            score\_improvement\_to\_save=1.0,

                            score\_min\_to\_save=-np.inf,

                            log\_score\_every=100,

                            return\_on\_restart=False)

    print("Best score after local search: {}".format(best\_score))

    write\_gph(best\_G, vars, data\_name=data\_name, gph\_name="best\_after\_optims", score=best\_score)