Causal Discovery under Causal Sufficiency

In certain case, it is reasonable to assume that there are no latent confounders and no selection bias. In this case, we assume that given a dataset, the true underlying causal diagram, over the measured variables, is a directed acyclic graph (DAG).

A common algorithm used in these cases is the PC algorithm (named after its inventors Peter Spirtes and Clark Glymour). An efficient algorithm is the recursive autonomy identification (RAI) algorithm. In the large sample limit, both algorithms a proved to recover the equivalence class of the true underlying DAG. This equivalence class is called completed partially directed graph (CPDAG), or essential graph.

For demonstrating PC and RAI, we follow these steps.

Initially, the required classes and methods are imported.

In [ ]:
```
import sys
sys.path.append('..')

import numpy as np
from causal_discovery_utils.cond_indep_tests import CondIndepCMI
from causal_discovery_algs import LearnStructRAI, LearnStructPC, LearnStruct
from causal_discovery_utils.data_utils import get_var_size
from graphical_models import DAG, PDAG
from causal_discovery_utils.performance_measures import structural_hamming_d
from experiment_utils.threshold_select_ci_test import search_threshold_bdeu
from matplotlib import pyplot as plt
from plot_utils import draw_graph

from tqdm import tqdm
```

Experiment Setup

Firstly, we will use the ALARM monitoring system (Beinlich et al., 1989) as the true underlying graph. It is a Bayesian network consisting of 37 nodes (8 diagnoses, 16 findings and 13 intermediate variables), and 46 edges. As an example, we use one dataset sampled from this Bayesian network by Tsamardinos et al. (2006). The full database they created, covering a range of Bayesian network and a range of datasets size can be found in the supplementary material for the max-min hill-climbing (MMHC) algorithm.

Initialization

Initially, we define the location and file name of the datasets and graph structure.

In [ ]:
```
data_path = '../example_data/sachs_data/discrete_sachs.txt'
true_dag_adj_matrix_path = '../example_data/sachs_data/sachs_true_dag_adj_ma
```

Load Training Data

The training data will be used to learn the underlying graph structure.

In [ ]:
```
data_train = np.loadtxt(data_path,
                        dtype=int)
n_samples, n_vars = data_train.shape  # data is assumed a numpy 2d-array
graph_nodes = set(range(n_vars))  # create a set containing the nodes indice

print('Data loaded')
print('Data size:', n_samples)
print('Number of variables:', n_vars)
```

```
Data loaded
Data size: 853
Number of variables: 11
```

The loaded data, `data_train`, is a 2D numpy array, where its first axis is the sample index and its second axis is the variable index.

Learn the Equivalence Class (CPDAG) of the Underlying DAGs

Initially, we set the conditional independence test. We select the conditional mutual information (CMI) as it is suitable for estimating the level of correlation between discrete variables (all the variables in the ALARM network are discrete).

In [ ]:
```
CITest = CondIndepCMI  # class of the ci test to be used
```

In addition, we search the CMI threshold that maximizes the likelihood of the graph. This strategy was suggested by Yehezkel and Lerner (2009) for the RAI algorithm. We use it here for the PC algorithm as well. The method that searches for the threshold is `search_threshold_bdeu`. It utilizes the caching mechanism of the CI test class to search the threshold efficiently.

Set the list of candidate thresholds.

In [ ]:
```
th_range = [i / 10000 + 0.01 for i in range(100)]  # list of candidate thres
```

Find the threshold for RAI

In [ ]:
```
th_rai, all_scores_rai = search_threshold_bdeu(LearnStructRAI, data_train, C
print('Selected RAI threshold = {:.4f}'.format(th_rai))
```

```
Selected RAI threshold = 0.0199
```

Find the threshold for PC

In [ ]:
```
th_pc, all_scores_pc = search_threshold_bdeu(LearnStructPC, data_train, CITe
print('Selected PC threshold = {:.4f}'.format(th_pc))
```

```
Selected PC threshold = 0.0111
```

Learn using the RAI algorithm

First, instantiate a CI test with the selected threshold and training dataset

In [ ]:
```
ci_test_rai_auto = CITest(dataset=data_train, threshold=th_rai, count_tests=
rai = LearnStructRAI(nodes_set=graph_nodes, ci_test=ci_test_rai_auto)
rai.learn_structure()  # learn structure
rai_graph_with_auto_threshold = rai.graph

# # Learn structure with varying values of threshold for RAI algorihtm range
min = 0.01
max = 0.2
n = 100
th_range = np.linspace(min, max, n)

learned_rai_graphs = []

for th in tqdm(th_range):
    ci_test_rai = CITest(dataset=data_train, threshold=th, count_tests=True)
    rai = LearnStructRAI(nodes_set=graph_nodes, ci_test=ci_test_rai)
    rai.learn_structure()  # learn structure
    learned_rai_graphs.append(rai.graph)
```

```
100%|██████████| 100/100 [00:01<00:00, 79.52it/s]
```

Learn using the PC algorithm

Instantiate a CI test and a PC learner, and learn the structure.

In [ ]:
```
ci_test_pc_auto = CITest(dataset=data_train, threshold=th_pc, count_tests=Tr
pc = LearnStructPC(nodes_set=graph_nodes, ci_test=ci_test_pc_auto)
pc.learn_structure()  # learn structure
pc_graph_with_auto_threshold = pc.graph

# Learn structure with varying values of threshold for PC algorihtm range fr
min = 0.01
max = 0.2
n = 100
```

```
th_range = np.linspace(min, max, n)

learned_pc_graphs = []

for th in tqdm(th_range):
    ci_test_pc = CITest(dataset=data_train, threshold=th, count_tests=True)
    pc = LearnStructPC(nodes_set=graph_nodes, ci_test=ci_test_pc)
    pc.learn_structure()  # learn structure
    learned_pc_graphs.append(pc.graph)
```

```
100%|██████████| 100/100 [00:01<00:00, 55.67it/s]
```

Examine Results

We compare the PC and RAI algorithms using three measures:

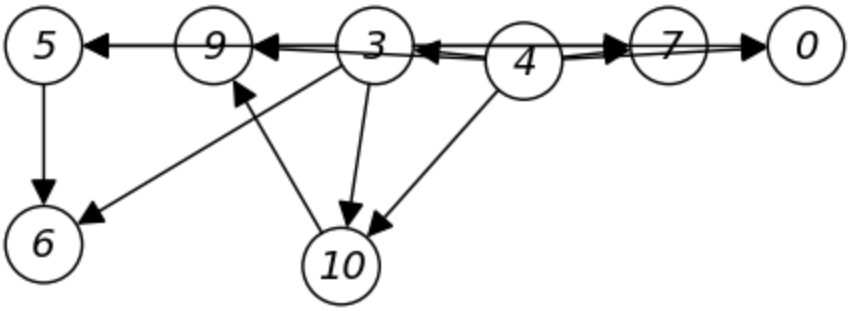
1. *Complexity* in terms of the number of CI tests required for learning the graph
2. *Structural Hamming distance* of the learned graph from the true underlying CPDAG
3. *BDeu* score of the learned graph calculated using a novel, large, test dataset

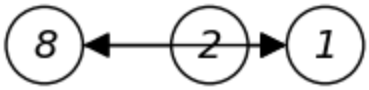
Initially, load the true underlying DAG and calculate its equivalence class CPDAG.

In [ ]:
```
# Load True Graph
true_dag_np = np.loadtxt(true_dag_adj_matrix_path, dtype=int)
true_dag = DAG(graph_nodes)
true_dag.init_from_adj_mat(true_dag_np)
true_cpdag = PDAG(true_dag.nodes_set)
true_dag.convert_to_cpdag(true_cpdag)  # create equivalence class of th true
```

Draw True DAG

In [ ]:
```
fig = draw_graph(true_dag)
```



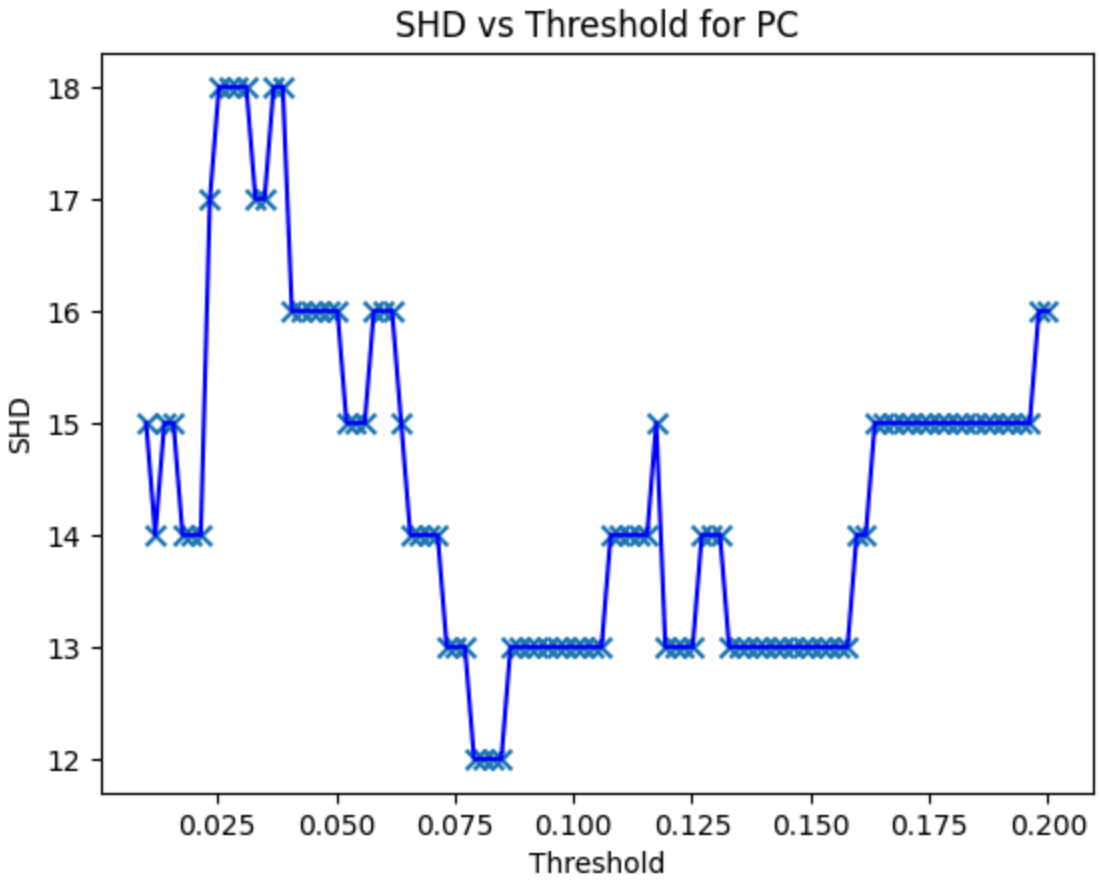


Examine the Quality of the Learned Graphs

Calculate the structural Hamming distance of the graphs learned by each of the algorithms

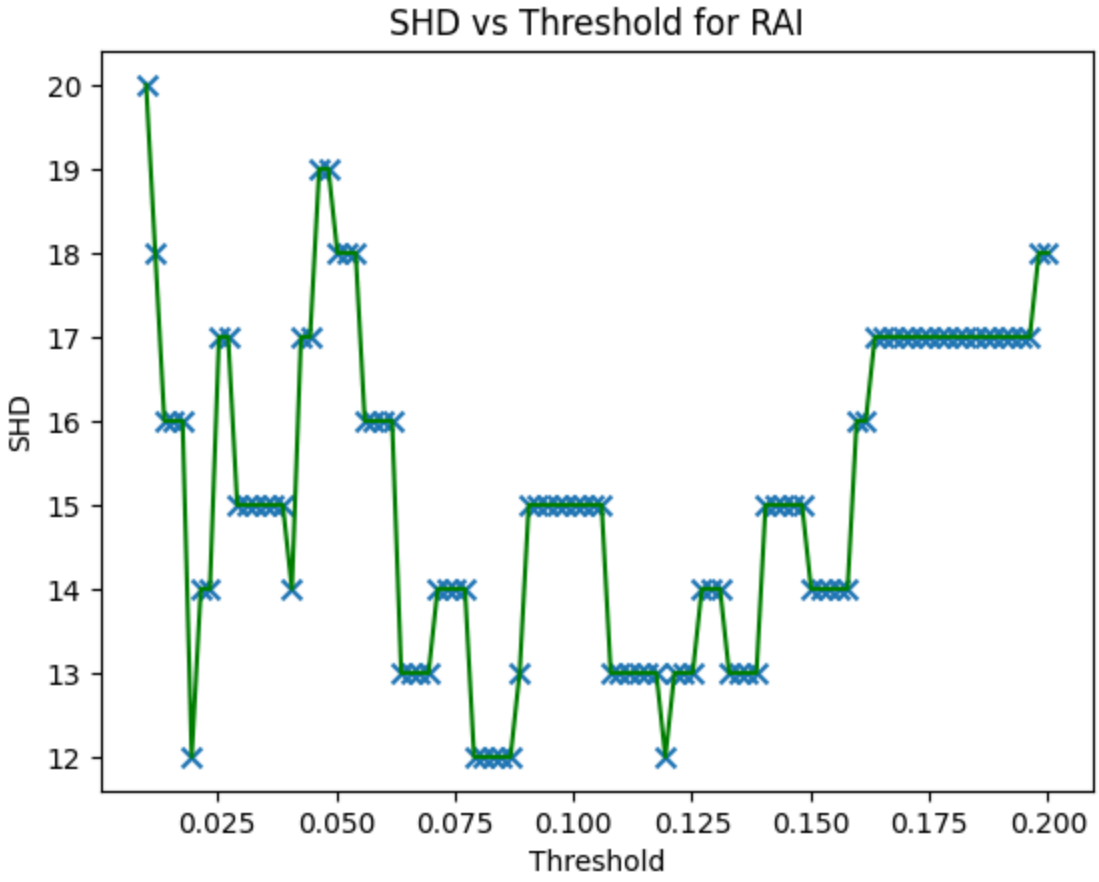
In [ ]:
```
# compute shd for all learned graphs using pc
shds_pc = [structural_hamming_distance_cpdag(g, true_cpdag)['total'] for g i

plt.scatter(th_range, shds_pc, s=50, marker='x')
plt.plot(th_range, shds_pc, 'b')
plt.xlabel('Threshold')
plt.ylabel('SHD')
plt.title('SHD vs Threshold for PC')
plt.show()
```



In [ ]:
```
# compute shd for all learned graphs using rai
shds_rai = [structural_hamming_distance_cpdag(g, true_cpdag)['total'] for g

plt.scatter(th_range, shds_rai, s=50, marker='x')
plt.plot(th_range, shds_rai, 'g')
plt.xlabel('Threshold')
plt.ylabel('SHD')
plt.title('SHD vs Threshold for RAI')
plt.show()
```



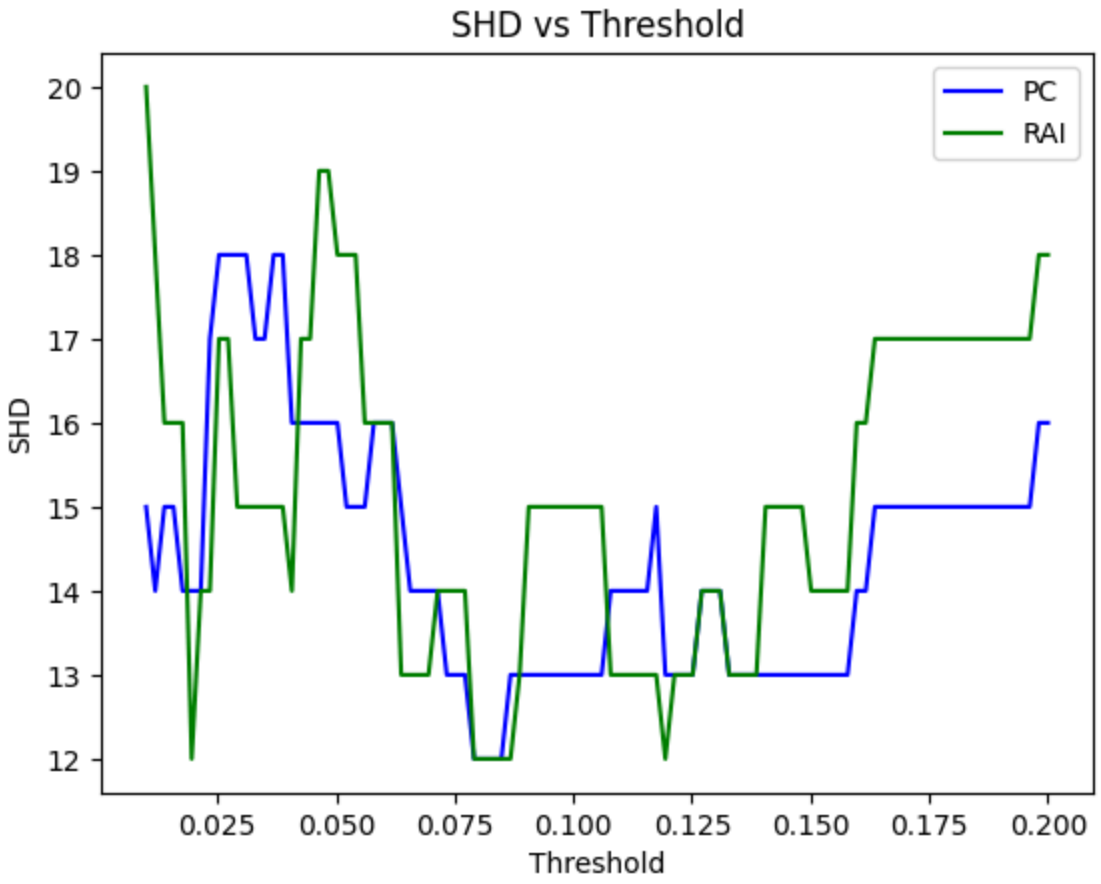
In [ ]:
```
# compute shd for all learned graphs using pc
shds_pc = [structural_hamming_distance_cpdag(g, true_cpdag)['total'] for g i

# compute shd for all learned graphs using rai
shds_rai = [structural_hamming_distance_cpdag(g, true_cpdag)['total'] for g

# plt.scatter(th_range, shds_pc, s=50, marker='x', color='b', label='PC')
plt.plot(th_range, shds_pc, 'b', label='PC')

# plt.scatter(th_range, shds_rai, s=50, marker='x', color='g', label='RAI')
plt.plot(th_range, shds_rai, 'g', label='RAI')

plt.xlabel('Threshold')
plt.ylabel('SHD')
plt.title('SHD vs Threshold')
plt.legend()
plt.show()
```



Print Quality Measures of the Learned Structures

Print structural hamming distance

In [ ]:
```
shd_rai = structural_hamming_distance_cpdag(rai_graph_with_auto_threshold, t
shd_pc = structural_hamming_distance_cpdag(pc_graph_with_auto_threshold, tru

def print_shd(shd, alg_name):
    print(alg_name, '\tEdges: extra|missing', shd['edge']['extra'], shd['edg
          '\tOrientation: extra|missing|revered',
          shd['arrowhead']['extra'], shd['arrowhead']['missing'], shd['arrow
          '\tTotal SHD:', shd['total'])

print_shd(shd_rai, 'RAI.')
print_shd(shd_pc, 'PC.')
```

```
RAI.    Edges: extra|missing 1 1        Orientation: extra|missing|revered 1
0 0 0   Total SHD: 12
PC.     Edges: extra|missing 1 1        Orientation: extra|missing|revered 9
0 0     Total SHD: 11
```

Plot the number of CI tests required by each algorithm

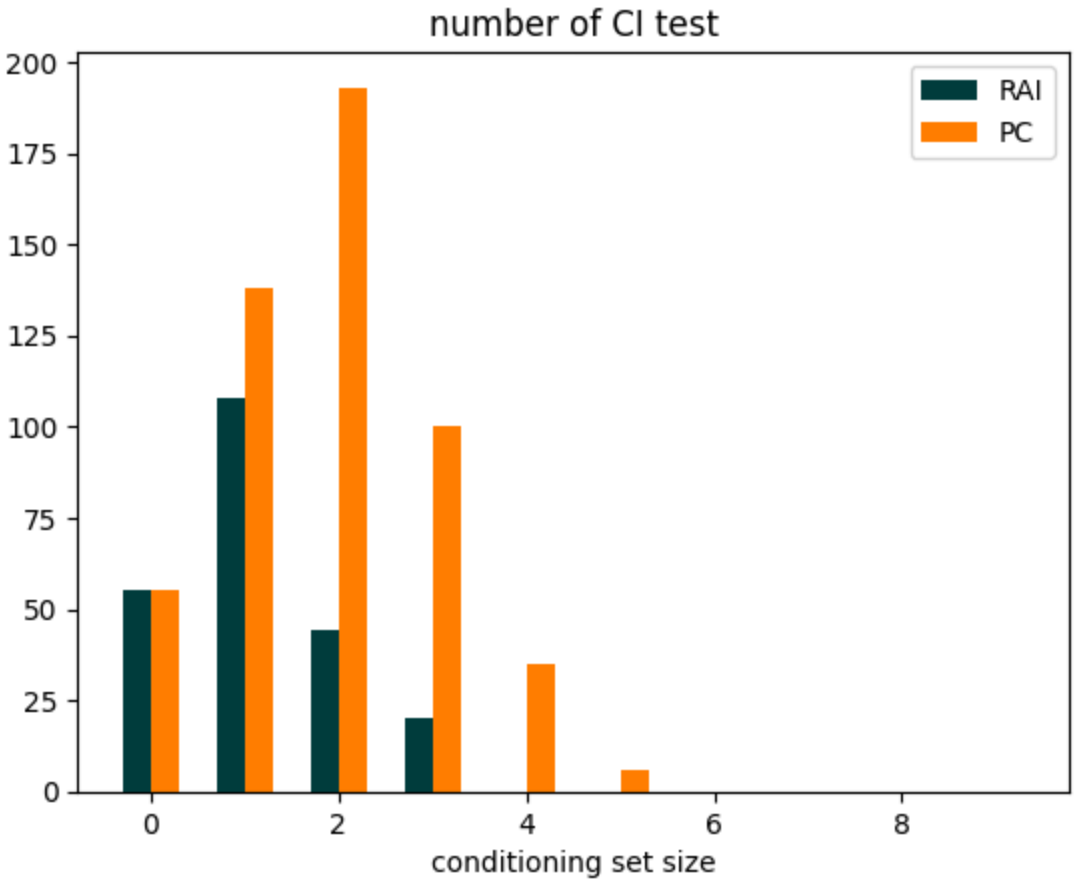
In [ ]:
```
num_ci_order_to_plot = 10
ci_orders = np.array(range(num_ci_order_to_plot), dtype=float)
plt.figure()
plt.bar(ci_orders-0.15, ci_test_rai_auto.test_counter[0:num_ci_order_to_plot
```

```
        color=[0, 0.25, 0.25])
plt.bar(ci_orders+0.15, ci_test_pc_auto.test_counter[0:num_ci_order_to_plot]
        color=[1, 0.5, 0])
plt.title('number of CI test')
plt.xlabel('conditioning set size')
plt.legend(['RAI', 'PC'])

# Print total number of CI tests for each algorithm (using sum(ci_test_pc.te
print('Total number of CI tests for RAI:', "{:,}".format(sum(ci_test_rai_aut
print('Total number of CI tests for PC:', "{:,}".format(sum(ci_test_pc_auto.

plt.show()
```

```
Total number of CI tests for RAI: 227
Total number of CI tests for PC: 527
```



References

- Beinlich, Ingo A., Henri Jacques Suermondt, R. Martin Chavez, and Gregory F. Cooper. "The ALARM monitoring system: A case study with two probabilistic inference techniques for belief networks." In AIME 89, pp. 247-256. Springer, Berlin, Heidelberg, 1989.
- Tsamardinos, Ioannis, Laura E. Brown, and Constantin F. Aliferis. "The max-min hill-climbing Bayesian network structure learning algorithm." Machine learning 65, no. 1

(2006): 31-78.

- Yehezkel, Raanan, and Boaz Lerner. "Bayesian Network Structure Learning by Recursive Autonomy Identification." Journal of Machine Learning Research 10, no. 7 (2009).