ATE estimations from generated observational data

This notebook examines the use of Bayesian Networks for estimating Average Treatment Effects (ATE) in Observational Studies within the Neyman-Rubin potential outcome framework from generated data: Lunceford & Davidian (2004)

Context

In contrast to randomized controlled trials (RCTs) where the ignorability assumption is satisfied, estimating treatment effects from observational data introduces new complexities.

Under RCT conditions, an unbiased estimator of the Average Treatment Effect (ATE) can be effectively computed by comparing the means of the observed treated subjects and the observed untreated subjects. However, in observational data, the ignorability assumption often does not hold, as subjects with certain treatment outcomes may be more or less likely to receive the treatment (i.e. $\mathbb{E}[Y(t)|T=t] \neq \mathbb{E}[Y(t)],\ t \in \{0,1\}$). Consequently, previous estmation methods are not guaranteed to be unbiased.

Nevertheless, if we can identify the confounders that *d-separates* the potential outcomes from the treatement assignment, we can achieve conditional independence between the treatment and the outcomes by conditionning on the confounders. Suppose that the covariate vector contains all such confounders, then:

$$T \perp\!\!\!\perp \{Y(0), Y(1)\} \mid X \quad \text{(Unconfoundedness)}$$

This conditional independence allows for estimations of the ATE from observational data:

$$\begin{aligned}\tau &= \mathbb{E}[Y(1) - Y(0)] \\ &= \mathbb{E}_X[\mathbb{E}[Y(1) \mid X] - \mathbb{E}[Y(0) \mid X]] \\ &= \mathbb{E}_X[\mathbb{E}[Y \mid T=1, X] - \mathbb{E}[Y \mid T=0, X]] \qquad \text{(Unconfoundedness)} \\ &= \mathbb{E}_X[\tau(X)]\end{aligned}$$

Where $\tau(x) = \mathbb{E}[Y \mid T=1, X=x] - \mathbb{E}[Y \mid T=0, X=x]$ is the conditional treatment effect given the covariates x.

In [1]:
```python
import pyAgrum as gum
import pyAgrum.lib.notebook as gnb
import pyAgrum.skbn as skbn

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from scipy.stats import norm, logistic
```

Generated Data

- The *outcome* variable Y is generated according the following equation:

$$\begin{aligned}Y &= -X_1 + X_2 - X_3 + 2T - V_1 + V_2 + V_3 \\ &= \langle \nu, \boldsymbol{Z} \rangle + \langle \xi, \boldsymbol{V} \rangle\end{aligned}$$

Where $\nu = (0, -1, 1, -1, 2)^\intercal$, $\boldsymbol{Z} = (1, X_1, X_2, X_3, T)^\intercal$, $\xi = (-1, 1, 1)^\intercal$ and $\boldsymbol{V} = (V_1, V_2, V_3)^\intercal$.

- The *covariates* are distributed as $X_3 \sim \text{Bernoulli}\,(0.2)$. Conditionally X_3, the distribution of the other variables is defined as:

If $X_3 = 0$, $V_3 \sim \text{Bernoulli}\,(0.25)$ and $(X_1, V_1, X_2, V_2)^\intercal \sim \mathcal{N}_4(\tau_0, \Sigma)$

If $X_3 = 1$, $V_3 \sim \text{Bernoulli}\,(0.75)$ and $(X_1, V_1, X_2, V_2)^\intercal \sim \mathcal{N}_4(\tau_1, \Sigma)$ with

$$\tau_1 = \begin{pmatrix} 1 \\ 1 \\ -1 \\ -1 \end{pmatrix}, \tau_0 = \begin{pmatrix} -1 \\ -1 \\ 1 \\ 1 \end{pmatrix} \text{ and } \Sigma = \begin{pmatrix} 1 & 0.5 & -0.5 & -0.5 \\ 0.5 & 1 & -0.5 & -0.5 \\ -0.5 & -0.5 & 1 & 0.5 \\ -0.5 & -0.5 & 0.5 & 1 \end{pmatrix}$$

- The *treatment* T is generated as a Bernoulli of the *propensity score*:

$$\begin{aligned}\mathbb{P}[T=1|X] &= e\,(X, \beta) \\ &= (1 + \exp(-0.6X_1 + 0.6X_2 - 0.6X_3))^{-1} \\ &= \frac{1}{1 + e^{-\langle \beta, \boldsymbol{X} \rangle}} \\ \mathbb{P}[T=0|X] &= 1 - \mathbb{P}[T=1|X]\end{aligned}$$

With $\beta = (0, 0.6, -0.6, 0.6)^\intercal$ and $\boldsymbol{X} = (1, X_1, X_2, X_3)^\intercal$.

In [2]:
```python
# Model parameters
XI = np.array([-1, 1, 1])
NU = np.array([0, -1, 1, -1, 2])
BETA = np.array([0, 0.6, -0.6, 0.6])
TAU_0 = np.array([-1, -1, 1, 1])
TAU_1 = TAU_0 * -1
SIGMA = np.array([[1, 0.5, -0.5, -0.5],
                  [0.5, 1, -0.5, -0.5],
                  [-0.5, -0.5, 1, 0.5],
                  [-0.5, -0.5, 0.5, 1]], dtype=float)

def generate_lunceford(n=1000):
    # Generate data
    x3 = np.random.binomial(1, 0.2, n)
    v3 = np.random.binomial(1, (0.75 * x3 + (0.25 * (1 - x3))), n)

    # If x3=0 you have a model, if x3=1 you have another one
    x1v1x2v2_x3_0_matrix = np.random.multivariate_normal(TAU_0, SIGMA, size=n, check_valid='warn', tol=1e-8)
    x1v1x2v2_x3_1_matrix = np.random.multivariate_normal(TAU_1, SIGMA, size=n, check_valid='warn', tol=1e-8)
    x1v1x2v2_x3 = np.where(np.repeat(x3[:, np.newaxis], 4, axis=1) == 0, x1v1x2v2_x3_0_matrix, x1v1x2v2_x3_1_matrix)

    # Concatenate values
    xv = np.concatenate([x1v1x2v2_x3, np.expand_dims(x3, axis=1), np.expand_dims(v3, axis=1)], axis=1)

    # Compute e, a, and y
    x = xv[:, [0,2,4]]
    v = xv[:, [1,3,5]]
    e = np.power(1 + np.exp(- BETA[0] - x.dot(BETA[1:])), -1)
    a = np.random.binomial(1, e, n)
    y = x.dot(NU[1:-1]) + v.dot(XI) + a*NU[-1] + np.random.binomial(1, e, n) + np.random.normal(0, 1, n)

    # Create the final df
    synthetic_data_df = pd.DataFrame(np.concatenate([x, np.expand_dims(a, axis=1), v, np.expand_dims(y, axis=1)], axis=1), columns=["X1", "X2", "X3", "T", "V1", "V2", "V3", "Y"])
    synthetic_data_df["X3"] = synthetic_data_df["X3"].astype(int)
    synthetic_data_df["V3"] = synthetic_data_df["V3"].astype(int)
    synthetic_data_df["T"] = synthetic_data_df["T"].astype(int)

    return synthetic_data_df
```

Here, the exact ATE can be explicitly calculated using the previously defined assumptions.

$$
\begin{aligned}
\mathbb{E}[Y(1)-Y(0)] &= \mathbb{E}_{X,V}[\mathbb{E}[Y \mid T=1,X,V]-\mathbb{E}[Y \mid T=0,X,V]] \\
&= \mathbb{E}[(-X_1+X_2-X_3+2\times 1-V_1+V_2+V_3)]-\mathbb{E}[(-X_1+X_2-X_3+2\times 0-V_1+V_2+V_3)] \\
&= 2
\end{aligned}
$$

In [3]:
```
df = generate_lunceford(int(1e6))
df.head()
```

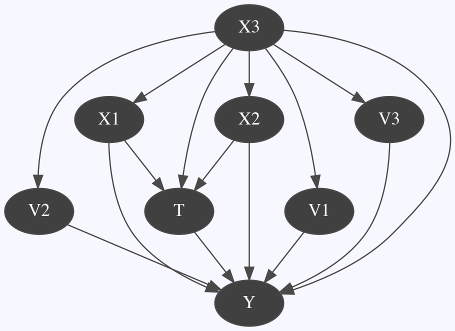
Out[3]:

	X1	X2	X3	T	V1	V2	V3	Y
0	1.991811	-2.024519	1	1	1.667370	-1.348843	1	-3.878638
1	-2.505670	1.946466	0	0	-1.947466	1.214491	0	7.037154
2	-0.973763	1.637130	0	0	-0.780443	0.608886	1	4.541941
3	-0.725768	1.440723	0	0	-1.683306	1.149354	1	3.995614
4	-0.468293	0.675582	0	0	-0.016903	-0.657962	0	0.931900

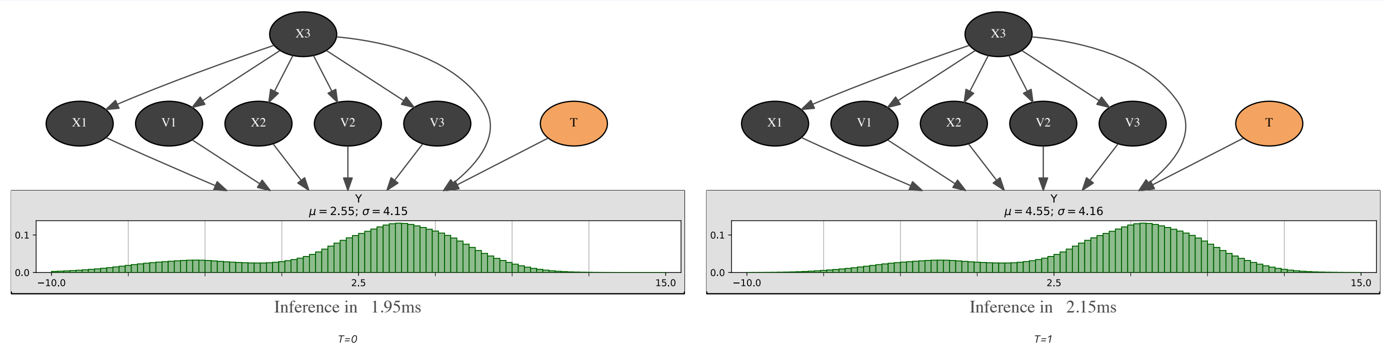
1 - "Exact" Computation

In [4]:
```
# Declarations of functions used in this section

def getBN(# Covariate parameters
          covariate_start : int = -5.0,
          covariate_end : int = 5.0 ,
          covariate_num_split : int = 10,
          # Outcome parameters
          outcome_start = -10.0 ,
          outcome_end = 15.0 ,
          outcome_num_split = 60,
          # Other
          data : pd.DataFrame | None = None,
          add_arcs : bool = True,
          fill_distribution : bool = True) -> gum.BayesNet:
    """
    Returns Baysian Network corresponding to the model by discretising
    countinous variables with given parameters.
    """
    if data is None:
        #plus = "" if fill_distribution else "+"
        plus = "+"
        bn = gum.BayesNet()
        for i in range(1,3):
            bn.add(f"X{i}{plus}[{covariate_start}:{covariate_end}:{covariate_num_split}]")
            bn.add(f"V{i}{plus}[{covariate_start}:{covariate_end}:{covariate_num_split}]")
        bn.add(f"X3[2]")
        bn.add(f"V3[2]")
        bn.add("T[2]")
        bn.add(f"Y{plus}[{outcome_start}:{outcome_end}:{outcome_num_split}]")

    else :
        disc = skbn.BNDiscretizer(defaultDiscretizationMethod="uniform",
                                  defaultNumberOfBins=covariate_num_split)
        disc.setDiscretizationParameters("Y", 'uniform', outcome_num_split)
        bn = disc.discretizedBN(data)

    if add_arcs :
        bn.beginTopologyTransformation()
        for _, name in bn:
            if name != "Y":
                bn.addArc(name, "Y")
        for X in ["X1", "X2", "X3"]:
            bn.addArc(X, "T")
        for XV in ["X1", "V1", "X2", "V2"]:
            bn.addArc("X3", XV)
        bn.addArc("X3", "V3")
        bn.endTopologyTransformation()

    if add_arcs and fill_distribution:
        bn.cpt("X3").fillWith([0.8, 0.2])
        bn.cpt("V3")[:] = [[0.75, 0.25], [0.25, 0.75]]
        for XV in ["X", "V"]:
            bn.cpt(f"{XV}1").fillFromDistribution(norm, loc="2*X3-1", scale=1)
            bn.cpt(f"{XV}2").fillFromDistribution(norm, loc="1-2*X3", scale=1)
        bn.cpt("T").fillFromDistribution(logistic, loc="-0.6*X1+0.6*X2-0.6*X3", scale=1)
        bn.cpt("Y").fillFromDistribution(norm, loc="-X1+X2-X3+2*T-V1+V2+V3", scale=1)

    return bn

def mutilateBN(bn : gum.BayesNet) -> gum.BayesNet:
    """
    Returns a copy of the Bayesian Network with all incoming arcs to the variable T removed.
    """
    res = gum.BayesNet(bn)
    for p_id in bn.parents("T"):
        res.eraseArc(p_id, bn.idFromName("T"))
    return res

def ATE(bn : gum.BayesNet) -> float:
    """
    Returns estimation of the ATE directly from Baysian Network.
    """

    ie = gum.LazyPropagation(mutilateBN(bn))

    ie.setEvidence({"T": 0})
    ie.makeInference()
    p0 = ie.posterior("Y")

    ie.chgEvidence("T",1)
    ie.makeInference()
    p1 = ie.posterior("Y")

    dif = p1 - p0
    return dif.expectedValue(lambda d: dif.variable(0).numerical(d[dif.variable(0).name()]))
```

In [5]:
```
exbn = getBN(covariate_num_split=5, outcome_num_split=100)
gnb.showBN(exbn, size="50")
```



In [13]:
```
gnb.sideBySide(gnb.getInference(mutilateBN(exbn), evs={"T":0}, targets={"Y"}),
               gnb.getInference(mutilateBN(exbn), evs={"T":1}, targets={"Y"}),
               captions=["T=0", "T=1"])

print(exbn)
print(f"{ATE(exbn) = }")
```




```
BN{nodes: 8, arcs: 15, domainSize: 500000, dim: 495085, mem: 3Mo 835Ko 400o}
ATE(exbn) = 1.997431830290404
```

Varying the granularity of the discretization of covariates and the outcome variable yields different results in ATE estimation. For exact computations, the degree of discretization has minimal impact on the final result, as the expression of the outcome distribution is directly incorporated into the variable. However, in the case of learning estimators, a coarser discretization may hinder the model's ability to capture the intricacies of the outcome distribution, while an excessively fine discretization can introduce an excessive number of parameters to be learned from the data. Consequently, the quality of the estimation is sensitive to the discretization settings. Therefore, selecting the appropriate level of discretization is crucial for the accuracy of the estimators.

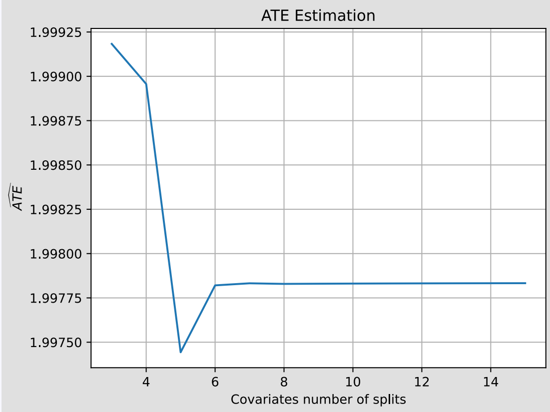
We observed that the number of outcome splits has minimal impact on the estimation. Therefore, we will choose 50 uniform splits on the outcome to study how the covariate splits affect the estimation.

In [33]:
```
covariate_split_list = range(3,16,1)
outcome_split_list = range(50,51,200)

e_grid = np.zeros((len(covariate_split_list), len(outcome_split_list)))

for i in range(len(covariate_split_list)):
    for j in range(len(outcome_split_list)):
        bn = getBN(covariate_num_split=covariate_split_list[i],
                   outcome_num_split=outcome_split_list[j])
        ate = ATE(bn)
        e_grid[i][j] = ate

        print(f"{covariate_split_list[i] = }, "\
              f"{outcome_split_list[j] = }, " \
              f"{ate = }")
```

```
covariate_split_list[i] = 3, outcome_split_list[j] = 50, ate = 1.999183008586101
covariate_split_list[i] = 4, outcome_split_list[j] = 50, ate = 1.99895668995137
covariate_split_list[i] = 5, outcome_split_list[j] = 50, ate = 1.9974432482262632
covariate_split_list[i] = 6, outcome_split_list[j] = 50, ate = 1.9978207054750634
covariate_split_list[i] = 7, outcome_split_list[j] = 50, ate = 1.9978323873990398
covariate_split_list[i] = 8, outcome_split_list[j] = 50, ate = 1.9978290354405723
covariate_split_list[i] = 9, outcome_split_list[j] = 50, ate = 1.997830061613167
covariate_split_list[i] = 10, outcome_split_list[j] = 50, ate = 1.997830840380035
covariate_split_list[i] = 11, outcome_split_list[j] = 50, ate = 1.9978314546477722
covariate_split_list[i] = 12, outcome_split_list[j] = 50, ate = 1.9978319620989407
covariate_split_list[i] = 13, outcome_split_list[j] = 50, ate = 1.9978323796985789
covariate_split_list[i] = 14, outcome_split_list[j] = 50, ate = 1.9978327253923807
covariate_split_list[i] = 15, outcome_split_list[j] = 50, ate = 1.997833013792365
```

We see that outcome discretization does not have a major impact on the accuracy of the prediction.

In [34]:
```
plt.plot(covariate_split_list, e_grid[:,0])
plt.ylabel("$\widehat{ATE}$")
plt.xlabel("Covariates number of splits")
plt.title("ATE Estimation")
plt.grid(True)

plt.show()
```



2 - Parameter Learning

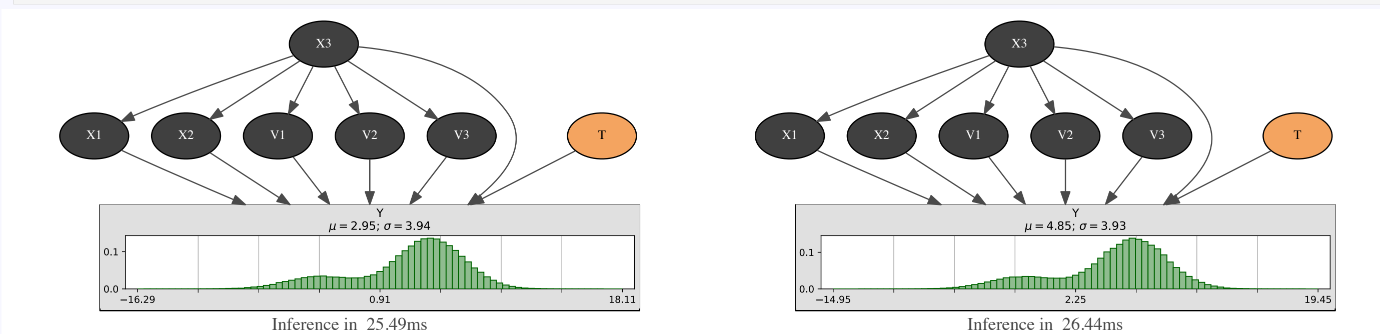
We will first evaluate the results of the parameter learning algorithm using a Bayesian network where the variable domains come from the generated data. Here, we used `skbn.BNDiscretizer` with uniform discretization.

In [14]:
```
discretized_p_template = getBN(data=df, covariate_num_split=9, outcome_num_split=80)
```

In [15]:
```
disc_p_learner = gum.BNLearner(df, discretized_p_template)
disc_p_learner.useNMLCorrection()

#disc_p_learner.useSmoothingPrior(1e-9)
disc_p_learner.useDirichletPrior(discretized_p_template)

disc_plbn = gum.BayesNet(discretized_p_template)
disc_p_learner.fitParameters(disc_plbn)

print(disc_p_learner)
```

```
Filename                          : /tmp/tmp5m80hr0b.csv
Size                              : (1000000,8)
Variables                         : X1[9], X2[9], X3[2], T[2], V1[9], V2[9], V3[2], Y[80]
Induced types                     : False
Missing values                    : False
Algorithm                         : MIIC
Score                             : BDeu  (Not used for constraint-based algorithms)
Correction                        : NML  (Not used for score-based algorithms)
Prior                             : Dirichlet
Dirichlet from Bayesian network : : BN{nodes: 8, arcs: 15, domainSize: 10^6.62315, dim: 4146781, mem: 32Mo 40Ko 144o}
Prior weight                      : 1.000000
```

In [16]:
```
gnb.sideBySide(gnb.getInference(mutilateBN(disc_plbn), evs={"T":0}, targets={"Y"}),
               gnb.getInference(mutilateBN(disc_plbn), evs={"T":1}, targets={"Y"}),
               captions=["T=0", "T=1"])

print(disc_plbn)
print(f"{ATE(disc_plbn) = }")
```



T=0

T=1

```
BN{nodes: 8, arcs: 15, domainSize: 10^6.62315, dim: 4146781, mem: 32Mo 40Ko 144o}
ATE(disc_plbn) = 1.8966741071588857
```

The estimated ATE is less than the expected ATE of 2.

Next, we will evaluate the performance of the parameter learning algorithm when provided with the same sample spaces as the variables from the exact Baysian Network.

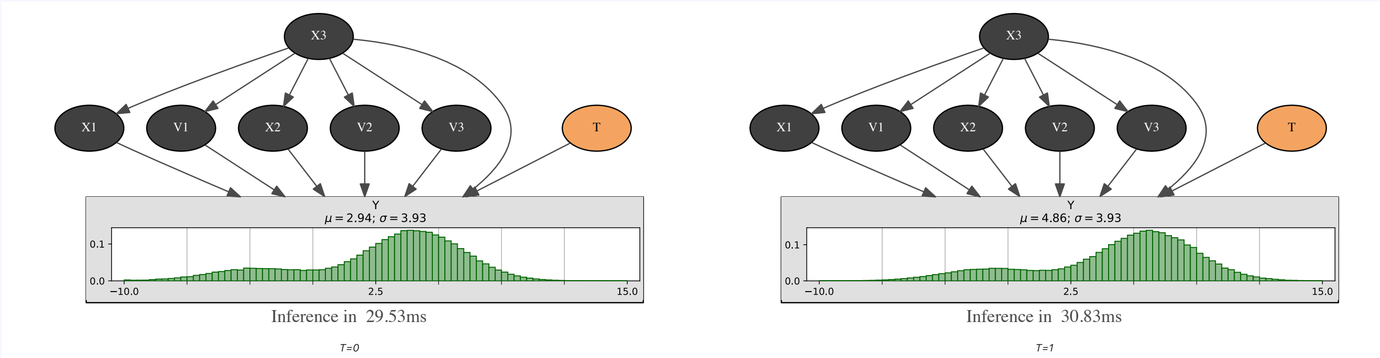
In [17]:
```
custom_p_template = getBN(fill_distribution=True, covariate_num_split=9, outcome_num_split=80)
```

In [18]:
```
cstm_p_learner = gum.BNLearner(df, custom_p_template)
cstm_p_learner.useNMLCorrection()

#cstm_p_learner.useSmoothingPrior(1e-9)
cstm_p_learner.useDirichletPrior(custom_p_template)

cstm_plbn = gum.BayesNet(custom_p_template)
cstm_p_learner.fitParameters(cstm_plbn)

print(cstm_p_learner)
```

```
Filename                          : /tmp/tmpu1twcecp.csv
Size                              : (1000000,8)
Variables                         : X1[9], V1[9], X2[9], V2[9], X3[2], V3[2], T[2], Y[80]
Induced types                     : False
Missing values                    : False
Algorithm                         : MIIC
Score                             : BDeu  (Not used for constraint-based algorithms)
Correction                        : NML  (Not used for score-based algorithms)
Prior                             : Dirichlet
Dirichlet from Bayesian network : : BN{nodes: 8, arcs: 15, domainSize: 10^6.62315, dim: 4146781, mem: 32Mo 40Ko 144o}
Prior weight                      : 1.000000
```

In [19]:
```
gnb.sideBySide(gnb.getInference(mutilateBN(cstm_plbn), evs={"T":0}, targets={"Y"}),
               gnb.getInference(mutilateBN(cstm_plbn), evs={"T":1}, targets={"Y"}),
               captions=["T=0", "T=1"])
```

```
print(cstm_plbn)
print(f"{ATE(cstm_plbn) = }")
```

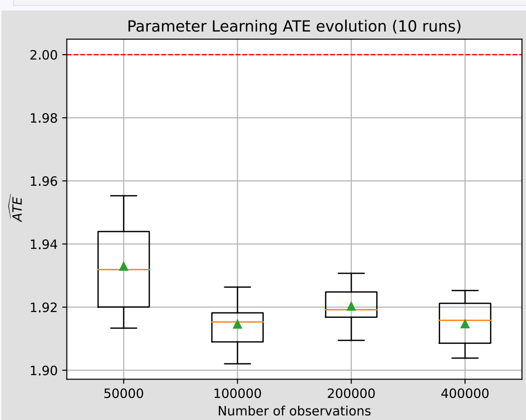



```
BN{nodes: 8, arcs: 15, domainSize: 10^6.62315, dim: 4146781, mem: 32Mo 40Ko 144o}
ATE(cstm_plbn) = 1.915481473886809
```

The estimated ATE is similar to the previously obtained one when using the discretized template.

In [20]:
```
p_template = custom_p_template

num_obs_list = np.array([5e4, 1e5, 2e5, 4e5]).astype(int)
num_shots = 10
pl_tau_hat_arr = list()

for i in num_obs_list:
    pl_tau_hat_arr.append(list())
    for j in range(num_shots):
        df = generate_lunceford(int(i))

        p_learner = gum.BNLearner(df, p_template)
        p_learner.useNMLCorrection()
        #p_learner.useSmoothingPrior(1e-9)
        p_learner.useDirichletPrior(p_template)

        plbn = gum.BayesNet(p_template)
        p_learner.fitParameters(plbn)

        pl_tau_hat_arr[-1].append(ATE(plbn))
```

In [21]:
```
plt.boxplot(pl_tau_hat_arr, labels=num_obs_list, meanline=False,
            showmeans=True, showcaps=True)
plt.axhline(y=2, color='r', linestyle='--', linewidth=1)
plt.title(f"Parameter Learning ATE evolution ({num_shots} runs)")
plt.xlabel("Number of observations")
plt.ylabel(r"$\widehat{ATE}$")
plt.grid(True)
plt.show()
```

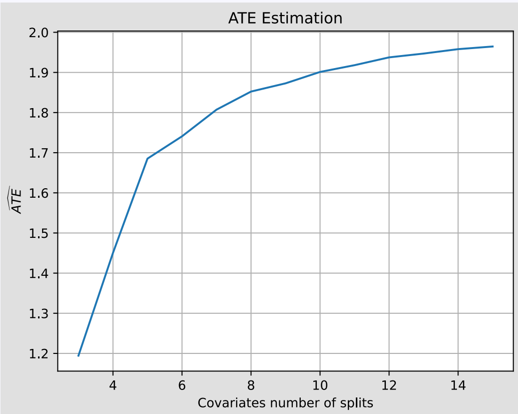


Again, let's see how the fineness of the discretization of the sample space affect the ATE estimation.

In [43]:
```
covariate_split_list = range(3,16,1)
outcome_split_list = range(50,51,1)

p_grid = np.zeros((len(covariate_split_list), len(outcome_split_list)))

for i in range(len(covariate_split_list)):
    for j in range(len(outcome_split_list)):
        template = getBN(data=df,
                         covariate_num_split=covariate_split_list[i],
                         outcome_num_split=outcome_split_list[j])

        p_learner = gum.BNLearner(df, template)
        p_learner.useNMLCorrection()

        #p_learner.useSmoothingPrior(1e-9)
        p_learner.useDirichletPrior(template)

        bn = gum.BayesNet(template)
        p_learner.fitParameters(bn)

        ate = ATE(bn)
        p_grid[i][j] = ate

        print(f"{covariate_split_list[i] = }, "\
              f"{outcome_split_list[j] = }, " \
              f"{ate = }")

p_grid[p_grid == 0.0] = None
```

```
covariate_split_list[i] = 3, outcome_split_list[j] = 50, ate = 1.1703006861540841
covariate_split_list[i] = 4, outcome_split_list[j] = 50, ate = 1.514577805919881
covariate_split_list[i] = 5, outcome_split_list[j] = 50, ate = 1.6760043111579213
covariate_split_list[i] = 6, outcome_split_list[j] = 50, ate = 1.761115821370386
covariate_split_list[i] = 7, outcome_split_list[j] = 50, ate = 1.824424408643453
covariate_split_list[i] = 8, outcome_split_list[j] = 50, ate = 1.8572228848914383
covariate_split_list[i] = 9, outcome_split_list[j] = 50, ate = 1.8875650335814362
covariate_split_list[i] = 10, outcome_split_list[j] = 50, ate = 1.9151964511120083
covariate_split_list[i] = 11, outcome_split_list[j] = 50, ate = 1.9310981241955458
covariate_split_list[i] = 12, outcome_split_list[j] = 50, ate = 1.940231702184745
covariate_split_list[i] = 13, outcome_split_list[j] = 50, ate = 1.9576029649606854
```

In [ ]:
```python
plt.plot(covariate_split_list, p_grid[:,0])
plt.ylabel("$\widehat{ATE}$")
plt.xlabel("Covariates number of splits")
plt.title("ATE Estimation")
plt.grid(True)

plt.show()
```



3 - Structure Learning

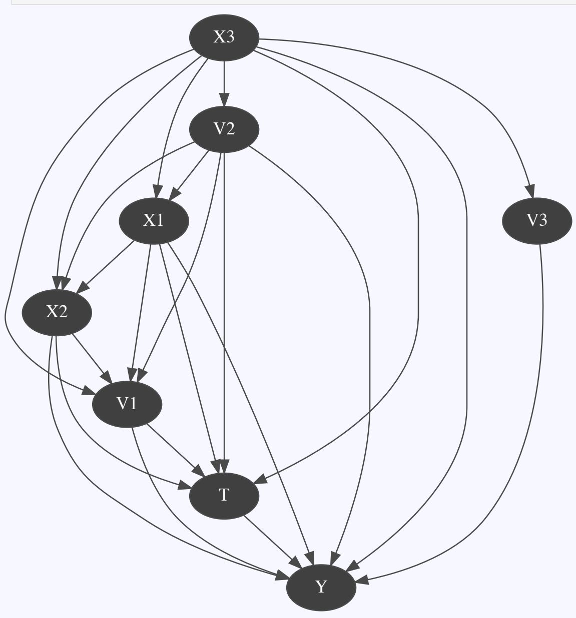
As before, structure learning can be performed on discretized variables derived from the data or by specifying the variables to be used in the process. Here, we observed that a 5-bins discretisation for the covariates and the outcome yielded the best results.

In [6]:
```python
discretized_s_template = getBN(data=df, add_arcs=False, covariate_num_split=5, outcome_num_split=5)
```

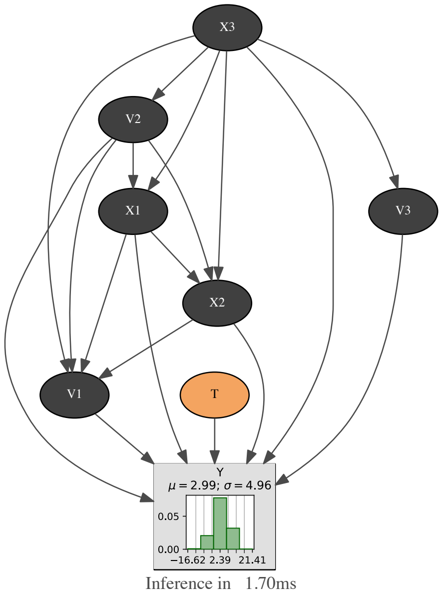
In [7]:
```python
disc_s_learner = gum.BNLearner(df, discretized_s_template)
disc_s_learner.useNMLCorrection()

disc_s_learner.useSmoothingPrior(1e-9)

disc_s_learner.setSliceOrder([["X3"], ["X1","X2","V1","V2","V3"], ["T"], ["Y"]])
disc_slbn = disc_s_learner.learnBN()

print(disc_s_learner)
```

```
Filename              : /tmp/tmpfaz2otvb.csv
Size                  : (1000000,8)
Variables             : X1[5], X2[5], X3[2], T[2], V1[5], V2[5], V3[2], Y[5]
Induced types         : False
Missing values        : False
Algorithm             : MIIC
Score                 : BDeu  (Not used for constraint-based algorithms)
Correction            : NML  (Not used for score-based algorithms)
Prior                 : Smoothing
Prior weight          : 0.000000
Constraint Slice Order : {T:2, X2:1, V3:1, V1:1, Y:3, X3:0, X1:1, V2:1}
```

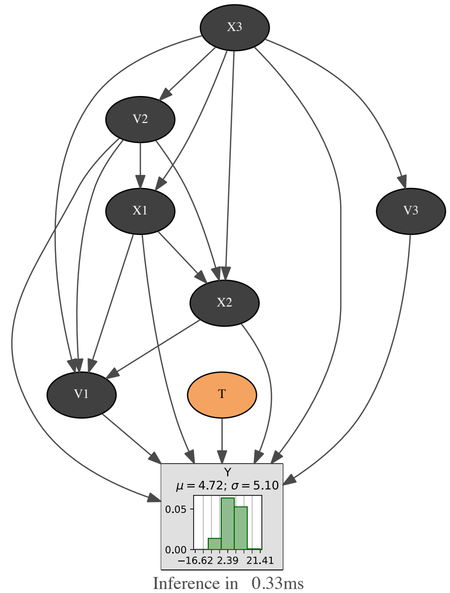
In [8]:
```python
gnb.showBN(disc_slbn, size="50")
```



In [9]:
```python
gnb.sideBySide(gnb.getInference(mutilateBN(disc_slbn), evs={"T":0}, targets={"Y"}),
               gnb.getInference(mutilateBN(disc_slbn), evs={"T":1}, targets={"Y"}),
               captions=["T=0", "T=1"])

print(disc_slbn)
print(f"{ATE(disc_slbn) = }")
```



T=0



T=1

```
BN{nodes: 8, arcs: 23, domainSize: 25000, dim: 22501, mem: 227Ko 80o}
ATE(disc_slbn) = 1.7302183743131738
```

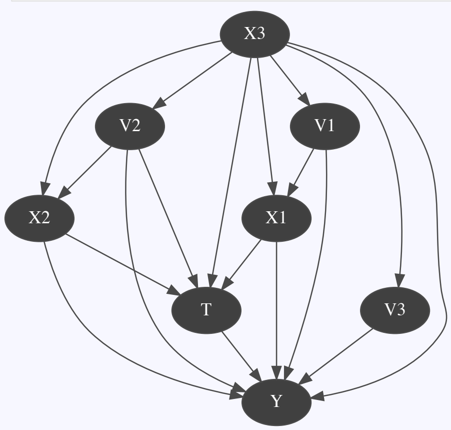
The structure learning algorithm using discretized variables performs worse than the parameter learning algorithm, as evidenced by the greater bias of the ATE.

In [10]:
```
custom_s_template = getBN(fill_distribution=False, add_arcs=False, covariate_num_split=5, outcome_num_split=5)
```

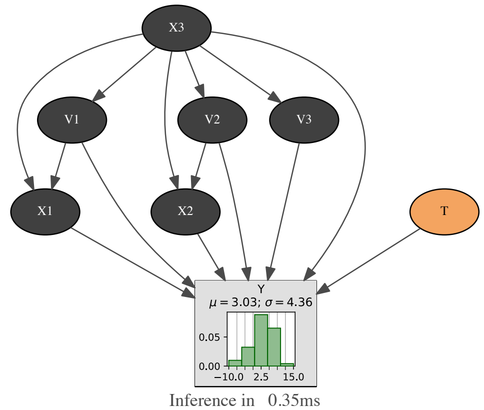
In [11]:
```
cstm_s_learner = gum.BNLearner(df, custom_s_template)
cstm_s_learner.useNMLCorrection()

cstm_s_learner.useSmoothingPrior(1e-9)

cstm_s_learner.setSliceOrder([["X3"], ["X1","X2","V1","V2","V3"], ["T"], ["Y"]])
cstm_slbn = cstm_s_learner.learnBN()

print(cstm_s_learner)
```

```
Filename               : /tmp/tmpbiuibuhp.csv
Size                   : (1000000,8)
Variables              : X1[5], V1[5], X2[5], V2[5], X3[2], V3[2], T[2], Y[5]
Induced types          : False
Missing values         : False
Algorithm              : MIIC
Score                  : BDeu  (Not used for constraint-based algorithms)
Correction             : NML  (Not used for score-based algorithms)
Prior                  : Smoothing
Prior weight           : 0.000000
Constraint Slice Order : {V2:1, V1:1, T:2, X3:0, Y:3, X1:1, X2:1, V3:1}
```

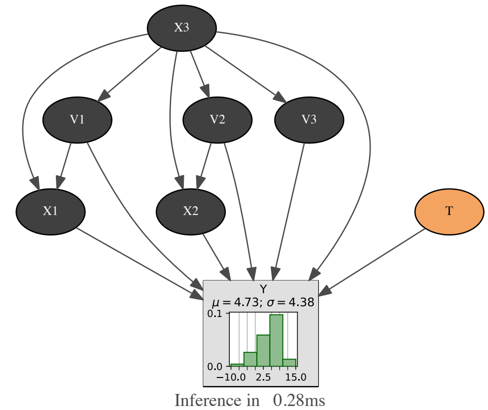
In [12]:
```
gnb.showBN(cstm_slbn, size="50")
```



In [13]:
```
gnb.sideBySide(gnb.getInference(mutilateBN(cstm_slbn), evs={"T":0}, targets={"Y"}),
               gnb.getInference(mutilateBN(cstm_slbn), evs={"T":1}, targets={"Y"}),
               captions=["T=0", "T=1"])

print(cstm_slbn)
print(f"{ATE(cstm_slbn) = }")
```



T=0



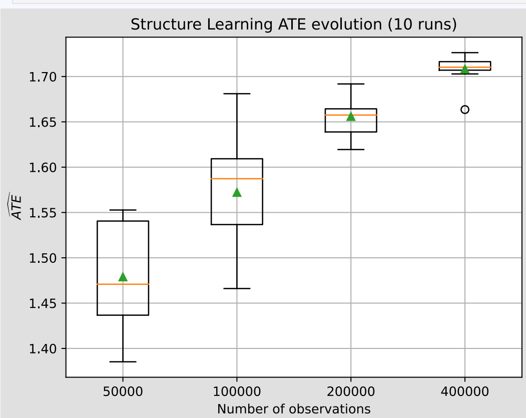
T=1

```
BN{nodes: 8, arcs: 18, domainSize: 25000, dim: 20349, mem: 200Ko 208o}
ATE(cstm_slbn) = 1.6981043499362127
```

Again, let's evaluate the evolution of the estimated ATE.

In [ ]:
```python
template = custom_s_template

num_obs_list = np.array([5e4, 1e5, 2e5, 4e5]).astype(int)
num_shots = 10
sl_tau_hat_arr = list()

for i in num_obs_list:
    sl_tau_hat_arr.append(list())
    for j in range(num_shots):
        df = generate_lunceford(int(i))

        s_learner = gum.BNLearner(df, template)
        s_learner.useNMLCorrection()
        s_learner.useSmoothingPrior(1e-6)
        s_learner.setSliceOrder([["X3"], ["X1","X2","V1","V2","V3"], ["T"], ["Y"]])
        slbn = s_learner.learnBN()

        sl_tau_hat_arr[-1].append(ATE(slbn))
```

In [15]:
```python
plt.boxplot(sl_tau_hat_arr, labels=num_obs_list, meanline=False,
            showmeans=True, showcaps=True)
#plt.axhline(y=2, color='r', linestyle='--', linewidth=1)
plt.title(f"Structure Learning ATE evolution ({num_shots} runs)")
plt.xlabel("Number of observations")
plt.ylabel(r"$\widehat{ATE}$")
plt.grid(True)
plt.show()
```



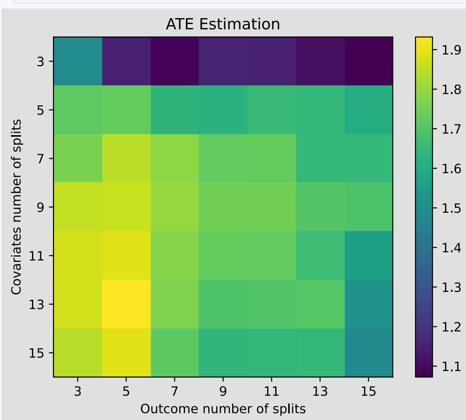

```
The Kernel crashed while executing code in the current cell or a previous cell.

Please review the code in the cell(s) to identify a possible cause of the failure.

Click <a href='https://aka.ms/vscodeJupyterKernelCrash'>here</a> for more info.

View Jupyter <a href='command:jupyter.viewOutput'>log</a> for further details.
```

In [14]:
```python
covariate_split_list = range(3,16,2)
outcome_split_list = range(3,16,2)

s_grid = np.zeros((len(covariate_split_list), len(outcome_split_list)))

for i in range(len(covariate_split_list)):
    for j in range(len(outcome_split_list)):
        template = getBN(data=df, add_arcs=False,
                         covariate_num_split=covariate_split_list[i],
                         outcome_num_split=outcome_split_list[j])

        s_learner = gum.BNLearner(df, template)
        s_learner.useNMLCorrection()
        s_learner.useSmoothingPrior(1e-6)
        s_learner.setSliceOrder([["X3"], ["X1","X2","V1","V2","V3"], ["T"], ["Y"]])
        bn = s_learner.learnBN()

        ate = ATE(bn)
        s_grid[i][j] = ate

        print(f"{covariate_split_list[i] = }, "\
              f"{outcome_split_list[j] = }, " \
              f"{ate = }")

s_grid[s_grid == 0.0] = None
```

```
covariate_split_list[i] = 3, outcome_split_list[j] = 3, ate = 1.4869220077545415
covariate_split_list[i] = 3, outcome_split_list[j] = 5, ate = 1.147325088093789
covariate_split_list[i] = 3, outcome_split_list[j] = 7, ate = 1.0812410867604025
covariate_split_list[i] = 3, outcome_split_list[j] = 9, ate = 1.1568108398106196
covariate_split_list[i] = 3, outcome_split_list[j] = 11, ate = 1.1471055761572806
covariate_split_list[i] = 3, outcome_split_list[j] = 13, ate = 1.108749530238853
covariate_split_list[i] = 3, outcome_split_list[j] = 15, ate = 1.0724837043703936
covariate_split_list[i] = 5, outcome_split_list[j] = 3, ate = 1.7192073301804336
covariate_split_list[i] = 5, outcome_split_list[j] = 5, ate = 1.7302182004419768
covariate_split_list[i] = 5, outcome_split_list[j] = 7, ate = 1.6301812169752428
covariate_split_list[i] = 5, outcome_split_list[j] = 9, ate = 1.6226166244543616
covariate_split_list[i] = 5, outcome_split_list[j] = 11, ate = 1.650704582222923
covariate_split_list[i] = 5, outcome_split_list[j] = 13, ate = 1.6453622958677532
covariate_split_list[i] = 5, outcome_split_list[j] = 15, ate = 1.604904711220915
covariate_split_list[i] = 7, outcome_split_list[j] = 3, ate = 1.7596921065174993
covariate_split_list[i] = 7, outcome_split_list[j] = 5, ate = 1.8425699837667469
covariate_split_list[i] = 7, outcome_split_list[j] = 7, ate = 1.7844564113612513
covariate_split_list[i] = 7, outcome_split_list[j] = 9, ate = 1.7261812557618934
covariate_split_list[i] = 7, outcome_split_list[j] = 11, ate = 1.7301321511455072
covariate_split_list[i] = 7, outcome_split_list[j] = 13, ate = 1.6464776312117801
covariate_split_list[i] = 7, outcome_split_list[j] = 15, ate = 1.644885880602469
covariate_split_list[i] = 9, outcome_split_list[j] = 3, ate = 1.8514545818629045
covariate_split_list[i] = 9, outcome_split_list[j] = 5, ate = 1.859911324763014
covariate_split_list[i] = 9, outcome_split_list[j] = 7, ate = 1.7957627303440402
covariate_split_list[i] = 9, outcome_split_list[j] = 9, ate = 1.7474029918957537
covariate_split_list[i] = 9, outcome_split_list[j] = 11, ate = 1.7505546892731922
covariate_split_list[i] = 9, outcome_split_list[j] = 13, ate = 1.7016296448966015
covariate_split_list[i] = 9, outcome_split_list[j] = 15, ate = 1.6924347692876223
covariate_split_list[i] = 11, outcome_split_list[j] = 3, ate = 1.8750651638327536
covariate_split_list[i] = 11, outcome_split_list[j] = 5, ate = 1.8901845803093433
covariate_split_list[i] = 11, outcome_split_list[j] = 7, ate = 1.7747371313135762
covariate_split_list[i] = 11, outcome_split_list[j] = 9, ate = 1.729162352447335
covariate_split_list[i] = 11, outcome_split_list[j] = 11, ate = 1.7263555403569049
covariate_split_list[i] = 11, outcome_split_list[j] = 13, ate = 1.6661563457248063
covariate_split_list[i] = 11, outcome_split_list[j] = 15, ate = 1.5603582892041863
covariate_split_list[i] = 13, outcome_split_list[j] = 3, ate = 1.873764179857285
covariate_split_list[i] = 13, outcome_split_list[j] = 5, ate = 1.9318644837676637
covariate_split_list[i] = 13, outcome_split_list[j] = 7, ate = 1.7649686234917352
covariate_split_list[i] = 13, outcome_split_list[j] = 9, ate = 1.6950907119297063
covariate_split_list[i] = 13, outcome_split_list[j] = 11, ate = 1.6995803635106643
covariate_split_list[i] = 13, outcome_split_list[j] = 13, ate = 1.7048582569721784
covariate_split_list[i] = 13, outcome_split_list[j] = 15, ate = 1.5107412836455203
covariate_split_list[i] = 15, outcome_split_list[j] = 3, ate = 1.8410091666691049
covariate_split_list[i] = 15, outcome_split_list[j] = 5, ate = 1.8923184912183442
covariate_split_list[i] = 15, outcome_split_list[j] = 7, ate = 1.713919228409103
covariate_split_list[i] = 15, outcome_split_list[j] = 9, ate = 1.6397949865949841
covariate_split_list[i] = 15, outcome_split_list[j] = 11, ate = 1.6406918989938948
covariate_split_list[i] = 15, outcome_split_list[j] = 13, ate = 1.6462838283059618
covariate_split_list[i] = 15, outcome_split_list[j] = 15, ate = 1.4835319731665504
```

In [16]:

```python
plt.imshow(s_grid)
plt.yticks(np.arange(len(covariate_split_list)), labels=covariate_split_list)
plt.xticks(np.arange(len(outcome_split_list)), labels=outcome_split_list)
plt.colorbar()
plt.xlabel("Outcome number of splits")
plt.ylabel("Covariates number of splits")
plt.title("ATE Estimation")

plt.show()
```



It appears that using an overly fine discretization of the outcome variable can lead the algorithm to incorrectly infer independence between the treatment and outcome variables. This misinterpretation can result in an estimated ATE of zero.

We observe that Bayesian Network-based estimators consistently underestimate the ATE. This underestimation arises due to the discretization and learning processes, which tend to homogenize the data by averaging it out. Since the ATE is computed as the difference between two expectations, this induced homogeneity reduces the variance between the groups, leading to a smaller difference and consequently an underestimation of the ATE.