Istituto
Dalle Molle
di Studi
sull'Intelligenza
Artificiale

IDSIA

Advanced Probabilistic Modelling

Probabilistic Inference

Marco Scutari

Dalle Molle Institute for
Artificial Intelligence (IDSIA)

Acedemic Year 2025–2026

A BN represents a working model of the world that a computer can understand; but how does a computer system use it to help and perform its assigned task?

We ask questions, and the computer system performs probabilistic inference to answer them and decide what to do in the process.

Questions that can be asked are called queries and are typically about an event of interest given some evidence. The evidence is the input to the computer system ("Someone with a high-school degree.") and the event is the output ("A man driving a car."). This is often called belief update: we observe some evidence and we update our beliefs before taking action.

The two most common queries are

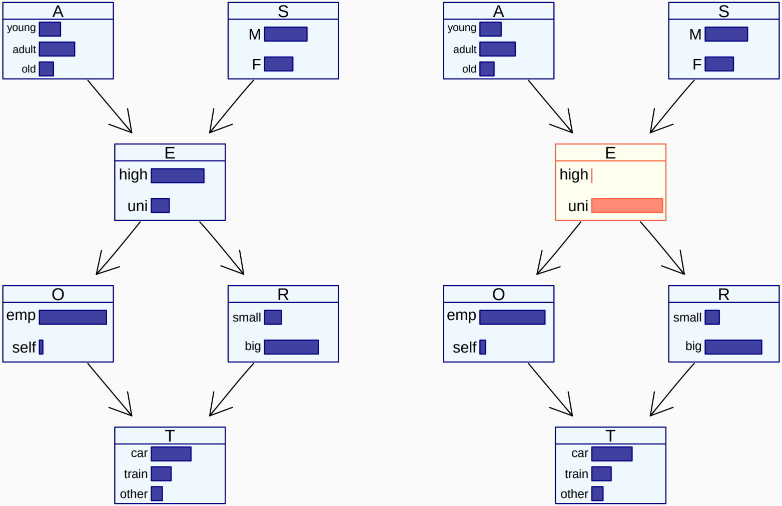
- conditional probability queries ("What is the probability that someone with high-school degree is a man driving a car?"); and
- most probable explanation queries ("What is the most probable sex and mode of transportation for someone with a high-school degree?")

In both cases the evidence is hard evidence: we set some variables to particular values. Then the computer system checks how the probabilities of other variables change and provides an answer to the query.

No more manual probability calculations..

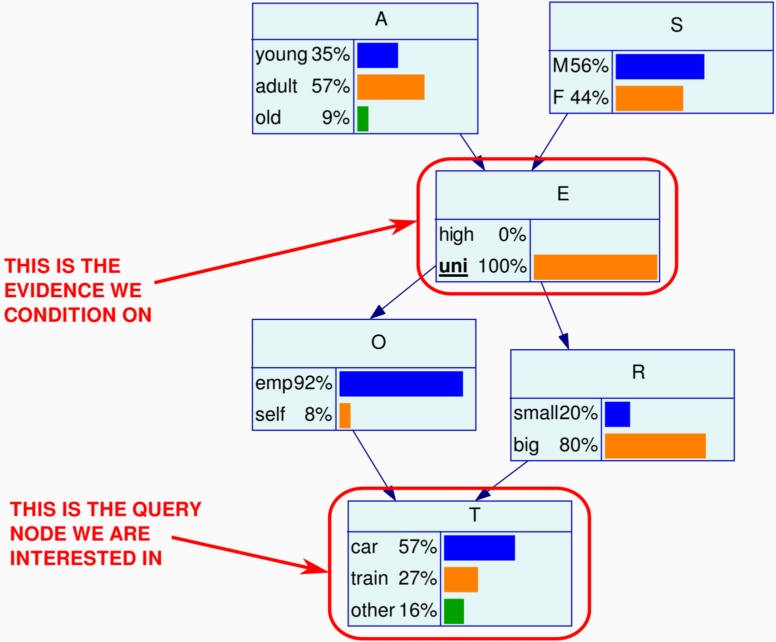
NOTE: we will initially consider only DBNs for ease of exposition, and get back to other types of BNs later.

The Effects of Conditioning on Hard Evidence

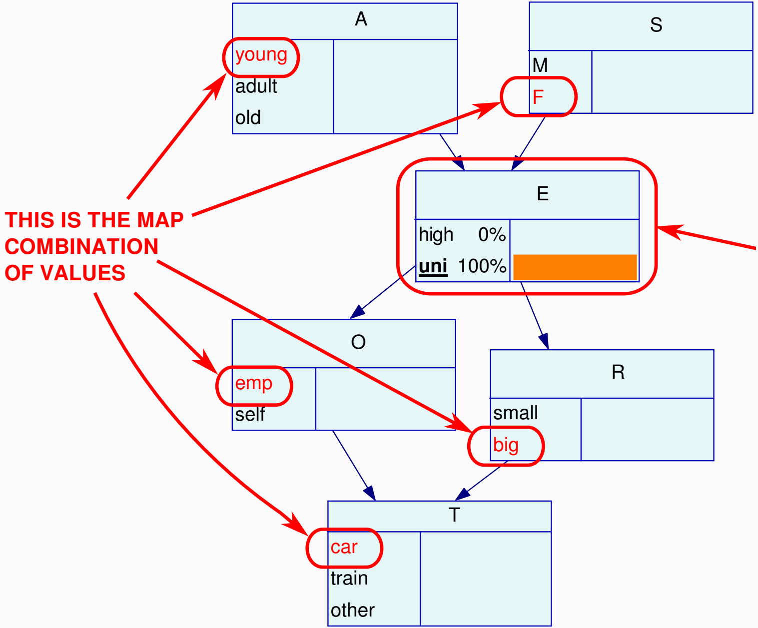


The original survey BN (left), and the posterior BN with hard evidence on Education (right).

Conditional Probability Queries in Pictures



Maximum a Posteriori Queries in Pictures



There are two approaches to answer queries using BNs.

Exact algorithms use the DAG to schedule and perform repeated applications of Bayes theorem on the local probability distributions in the BN. In other words, the computer system uses the DAG to perform all the math we did by hand in earlier lectures.

The two best known are

- variable elimination; and
- belief updates based on junction trees.

PROS: they return exact values for the probabilities of interest.
CONS: they do not scale well when BNs have many nodes and many arcs.

Approximate algorithms use the BN as a model of the world in a very literal sense. In the real world to answer some question in a scientific, rigorous way we would perform an experiment and observe the outcome. Approximate algorithms imitate this process by generating random observations from the BN, thus running a simulated experiment that approximates reality.

The two best known are

- logic sampling; and
- likelihood weighting.

PROS: they scale really well when BNs have many nodes and many arcs.
CONS: they return approximate, estimated values for the probabilities of interest.

INPUT: a BN, evidence E and query event Q.

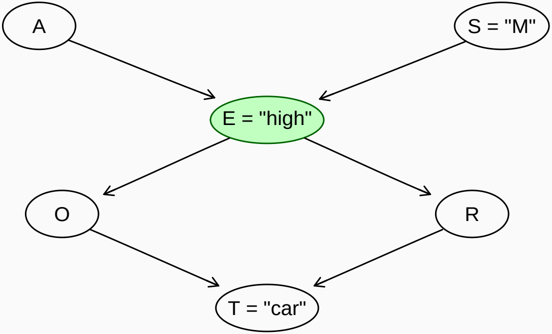
1. Order the variables in $\mathbf{X}$ according to the topological ordering in the DAG (from top to bottom), so that parents come before children.
2. Set $n_E = 0$ and $n_{E,Q} = 0$.
3. For a suitably large number of samples $\mathbf{x}$:
   3.1 generate a random value from each $X_i \mid \Pi_{X_i}$ taking advantage of the fact that, thanks to the topological ordering, by the time we are considering X_i we have already generated the values of all its parents Π_{X_i};
   3.2 if $\mathbf{x}$ includes E, set $n_E = n_E + 1$;
   3.3 if $\mathbf{x}$ includes both Q and E, set $n_{E,Q} = n_{E,Q} + 1$.
4. The answer to the query is the estimated probability $n_{E,Q}/n_E$.

A Survey Example

Consider:

- the evidence: someone whose Education (`E`) level is a high school diploma (`high`)...
- the event: ... is a man (`S` is equal to `M`) uses a `car` as a means of Transportation (`T`).

We will answer this query using the different inference algorithms.



First, we sample from the BN with `rbn()`, which takes a `bn.fit` object and the number of random samples to generate as arguments.

```
particles = rbn(bn, 10^6)
head(particles, n = 5)
```

```
      A    E   O   R S     T
1   old high emp big M train
2   old high emp big M   car
3 adult high emp big F   car
4   old high emp big M other
5 young high emp big M   car
```

The samples have the correct types and format as derived from the BN, and they are stored in a data frame that has the same structure as that of the data that were used to learn the BN (if any).

Stepping Through Logic Sampling

Then we count how many of those samples that match the evidence E to estimate $\mathrm{P}(E)$.

```
partE = particles[(particles[, "E"] == "high"), ]
nE = nrow(partE)
```

We also count how many of those samples that match the evidence E and the query event Q to estimate $\mathrm{P}(Q, E)$.

```
partEQ =
  partE[(partE[, "S"] == "M") & (partE[, "T"] == "car"), ]
nEQ = nrow(partEQ)
```

Finally, we estimate

$$\mathrm{P}(Q \mid E) = \frac{\mathrm{P}(Q, E)}{\mathrm{P}(E)}.$$

```
nEQ/nE
```

```
[1] 0.343
```

The `cpquery()` Function

These steps are implemented in `cpquery()`, with the obvious arguments:

- `event` is Q;
- `evidence` is E;
- `method` is `"ls"` for logic sampling (the default);
- `n` is the number of random samples.

```
cpquery(bn, event = (S == "M") & (T == "car"),
          evidence = (E == "high"), method = "ls", n = 10^6)
```

```
[1] 0.343
```

Both `event` end `evidence` are expressions that are evaluated on the random samples much like `subset()` would, so they must evaluate to a vector of `TRUE` and `FALSE` values (hence `&` and not `&&`).

More Advanced Queries with cpquery()

Specifying the arguments requires some care, but the result is an extremely flexible framework to compute the probability of arbitrary combinations of events.

As an example of a more complex query, we can compute

$$\mathrm{P}(\texttt{S} = \texttt{"M"}, \texttt{T} = \texttt{"car"} \mid \{\texttt{A} = \texttt{"young"}, \texttt{E} = \texttt{"uni"}\} \cup \{\texttt{A} = \texttt{"adult"}\}),$$

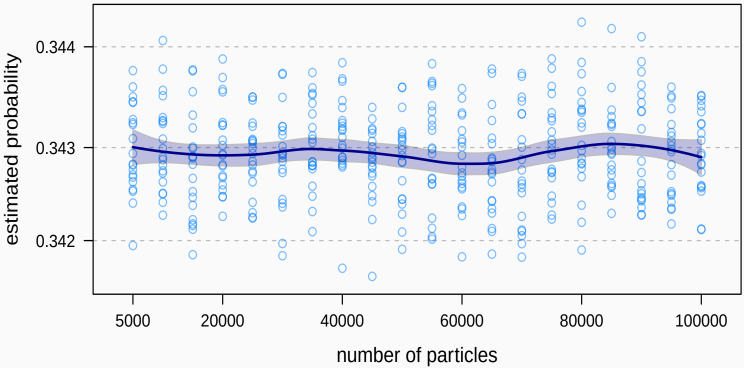
the probability of a man travelling by car given that his Age is "young" and his Education is "uni" or that he is an "adult", regardless of his Education. That would be:

```
cpquery(bn, event = (S == "M") & (T == "car"),
  evidence = ((A == "young") & (E == "uni")) | (A == "adult"))
```

```
[1] 0.349
```

Stepping Through Logic Sampling

```
nparticles = seq(from = 5 * 10^3, to = 10^5, by = 5 * 10^3)
prob = matrix(0, nrow = length(nparticles), ncol = 20)
for (i in seq_along(nparticles))
  for (j in 1:20)
    prob[i, j] = cpquery(bn, event = (S == "M") & (T == "car"),
                   evidence = (E == "high"), method = "ls", n = 10^6)
```



Notice anything in the figure in the previous slide?

- Logic sampling is obviously affected by sampling variability: every time we run it we get a different estimate of the probability that is the answer to our query because the random samples we generate will be different.
- Sampling variability decreases with the number of samples we generate, but it never goes to zero: there is always some uncertainty around the exact value we estimate (here 0.343 ± 0.001).
- Remember that we essentially discard all random samples that do not match the evidence we condition on, so if the evidence has low probability we are throwing out almost all samples we generate.

An improvement over logic sampling, designed to solve this problem, is the likelihood weighting algorithm. Unlike logic sampling, all the random samples generated by likelihood weighting include the evidence E by design.

1. Order the variables in $\mathbf{X}$ according to the topological ordering in the DAG (from top to bottom), so that parents come before children.
2. Set $w_E = 0$ and $w_{E,Q} = 0$.
3. For a suitably large number of samples $\mathbf{x}$:
    - 3.1 generate a random value from each $X_i \mid \Pi_{X_i}$ and fix the relevant variables to the values specified by the evidence E.
    - 3.2 compute the weight $w_{\mathbf{x}} = \mathrm{P}(E)$.
    - 3.3 set $w_E = w_E + w_{\mathbf{x}}$;
    - 3.4 if $\mathbf{x}$ includes Q , set $w_{E,Q} = w_{E,Q} + w_{\mathbf{x}}$.
4. The answer to the query is the estimated probability $w_{E,Q}/w_E$.

Stepping Through Likelihood Weighting

We do not want to sample from the original BN, but from the BN in which all the nodes covered by E are fixed. This network is called the mutilated network.

Compare:

```
coef(bn$E)
```

```
, , S = M

      A
E      young adult  old
  high  0.75  0.72 0.88
  uni   0.25  0.28 0.12

, , S = F

      A
E      young adult  old
  high  0.64  0.70 0.90
  uni   0.36  0.30 0.10
```

```
parents(bn, "E")
```

```
[1] "A" "S"
```

```
mutbn = mutilated(bn, list(E = "high"))
coef(mutbn$E)
```

```
high  uni
   1    0
```

```
parents(mutbn, "E")
```

```
character(0)
```

No parents, and the value is that in the evidence with probability equal to 1.

Simply sampling from `mutbn` is not a correct way of answering our query! A simple empirical check tells us that the naive estimate we would draw from `mutbn` is wrong, since it does not match the exact value we got earlier.

```
particles = rbn(mutbn, 10^6)
partE = particles[(particles[, "E"] == "high"), ]
partEQ = partE[(particles[, "S"] == "M") &
               (particles[, "T"] == "car"), ]
nrow(partEQ) / nrow(partE)
```

```
[1] 0.336
```

That is because `nrow(partE)` is identical to `nrow(particles)` by construction, so the conditional probability is not computed correctly. What we get is:

$$\mathrm{P}(Q, E) = \frac{n_{E,Q}}{n} \neq \mathrm{P}(Q \mid E).$$

The weights adjust for the fact that we are sampling from the mutilated BN instead of the original BN. The weights are just the likelihood components associated with the nodes we are conditioning on (E in this case):

```
w = logLik(bn, particles, nodes = "E", by.sample = TRUE)
wEQ = sum(exp(w[(particles[, "S"] == "M") &
                (particles[, "T"] == "car")]))
wE = sum(exp(w))
wEQ/wE
```

```
[1] 0.343
```

NOTE: the likelihood of an observation has the same mathematical expression as its probability, so for practical purposes here it is just $\mathrm{P}(E)$. `logLik()` returns $\log \mathrm{P}(E)$ in the code above.

More conveniently, we can perform likelihood weighting with `cpquery()` by setting `method = "lw"` and specifying the evidence as a named list with one element for each node we are conditioning on.

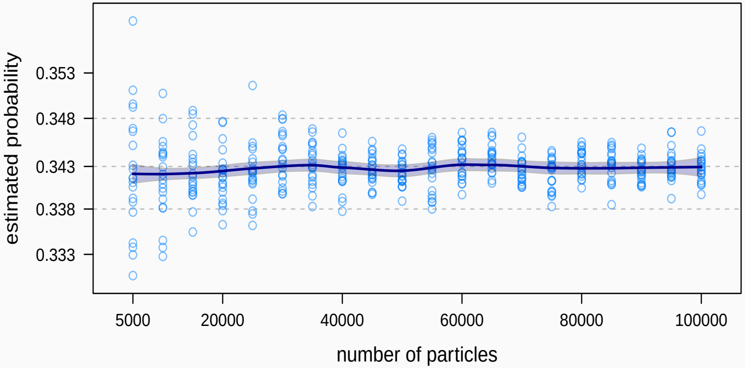
```
cpquery(bn, event = (S == "M") & (T == "car"),
          evidence = list(E = "high"), method = "lw", n = 5 * 10^4)
```

```
[1] 0.344
```

The estimate we obtain is still very precise with small numbers of random samples, as was the case for logic sampling, but the variability of the estimated probabilities is actually larger. There is no guarantee that likelihood weighting will always have lower variance than logic sampling.

Stepping Through Likelihood Weighting

```
nparticles = seq(from = 5 * 10^3, to = 10^5, by = 5 * 10^3)
prob = matrix(0, nrow = length(nparticles), ncol = 20)
for (i in seq_along(nparticles))
  for (j in 1:20)
    prob[i, j] = cpquery(bn, event = (S == "M") & (T == "car"),
                   evidence = list(E = "high"), method = "lw",
                   n = nparticles[i])
```



Then Why Use Likelihood Weighting?

Logic sampling will be computationally inefficient and very inaccurate if $\mathrm{P}(E)$ is small because most random samples will be discarded without contributing to the estimation of $\mathrm{P}(Q \mid E)$.

```r
extreme.dag = model2network("[A][B|A]")
A.prob = array(c(0.999999, 0.000001), dim = 2,
           dimnames = list(A = c("a1", "a2")))
B.prob = array(c(0.5, 0.5, 0.75, 0.25), dim = c(2, 2),
           dimnames = list(B = c("b1", "b2"), A = c("a1", "a2")))
extreme.bn = custom.fit(extreme.dag, list(A = A.prob, B = B.prob))
cpquery(extreme.bn, event = (B == "b2"), evidence = (A == "a2"),
  method = "ls", n = 10^6)
```

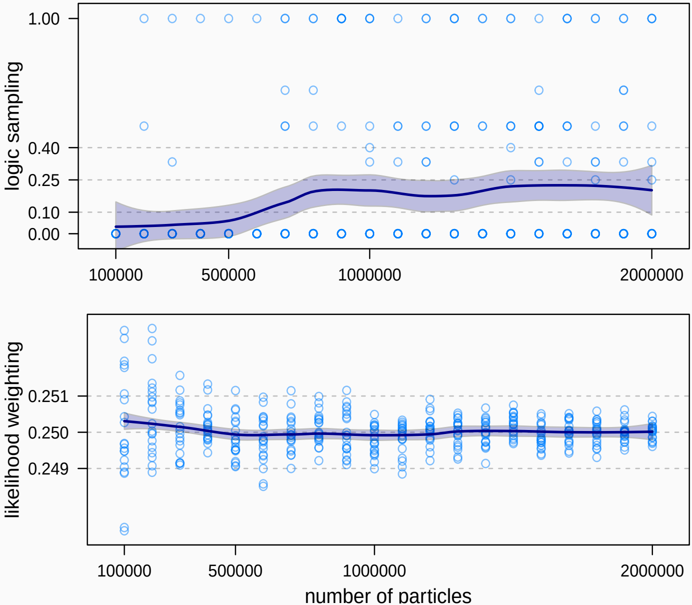
```
[1] 0.333
```

This simply does not happen with likelihood weighting.

```r
cpquery(extreme.bn, event = (B == "b2"), evidence = list(A = "a2"),
  method = "lw", n = 5 * 10^3)
```

```
[1] 0.249
```

A Comparison for Different Numbers of Random Samples



The `event` is still a general expression, which means it is possible to describe complex events. However, likelihood weighting relies on the fact that the `evidence` is fixed to a single value to compute the weights. In **bnlearn** this assumption is relaxed: the `event` can take more than one value for each variable. All combinations of values are given the same probability so as not to alter the weights.

```
cpquery(bn, event = (S == "M") & (T == "car"),
  evidence = list(A = c("young", "adult")), method = "lw", n = 10^6)
```

```
[1] 0.337
```

```
cpquery(bn, event = (S == "M") & (T == "car"),
  evidence = list(A = "young"), method = "lw", n = 10^6) * 0.5 +
cpquery(bn, event = (S == "M") & (T == "car"),
  evidence = list(A = "adult"), method = "lw", n = 10^6) * 0.5
```

```
[1] 0.337
```

Sampling and Conditioning

Last but not least, we can also use `cpdist()` to generate random samples conditional on some evidence E. Likelihood weighting works best, and attaches the weights to the samples (for use in later analyses).

```
cpdist(bn, nodes = c("S", "T"), evidence = list(A = "adult"),
  method = "lw", n = 5)
```

```
    S   T
  1 M car
  2 F car
  3 F car
  4 M car
  5 F car
```

Logic sampling works less well because it often returns far fewer observations than requested.

```
cpdist(bn, nodes = c("S", "T"), evidence = (A == "young"),
  method = "ls", n = 5)
```

```
 [1] S T
 <0 rows> (or 0-length row.names)
```

1. **Moralise:** create the moral graph of the BN $\mathcal{B}$.
2. **Triangulate:** break every cycle spanning 4 or more nodes into sub-cycles of exactly 3 nodes by adding arcs to the moral graph, thus obtaining a triangulated graph.
3. **Cliques:** identify the cliques $C_1, \dots, C_k$ of the triangulated graph, i.e., maximal subsets of nodes in which each element is adjacent to all the others.
4. **Junction Tree:** create a tree in which each clique is a node, and adjacent cliques are linked by arcs. The tree must satisfy the running intersection property: if a node belongs to two cliques C_i and C_j, it must be also included in all the cliques in the (unique) path that connects C_i and C_j.
5. **Parameters:** use the parameters of the local distributions of $\mathcal{B}$ to compute the parameter sets of the compound nodes of the junction tree.

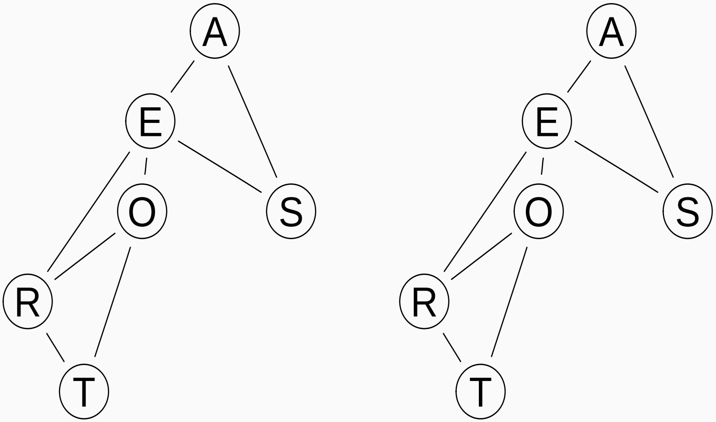
We saw how to create a moral graph earlier when introducing d-separation:

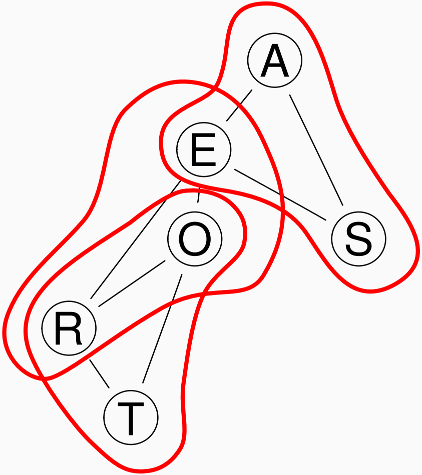
```
survey.dag = model2network("[A][S][E|A:S][O|E][R|E][T|O:R]")
survey.moral = moral(survey.dag)
```

NOTE: different DAGs can express the same set of dependencies and therefore will have the same moral graph. This in turn means that exact inference with junction trees will return the same results for conditional probability and maximum a posteriori queries. They are probabilistically indistinguishable.

Different DAGs, Same Moral Graph

```
survey.dag1 = model2network("[A][S][E|A:S][O|E][R|E][T|O:R]")
survey.dag2 = model2network("[A|E][S|A:E][E|O:R][O|R:T][R|T][T]")
graph.par(list(nodes = list(fontsize = 11)))
par(mfrow = c(1, 2))
graphviz.plot(moral(survey.dag1))
graphviz.plot(moral(survey.dag2))
```





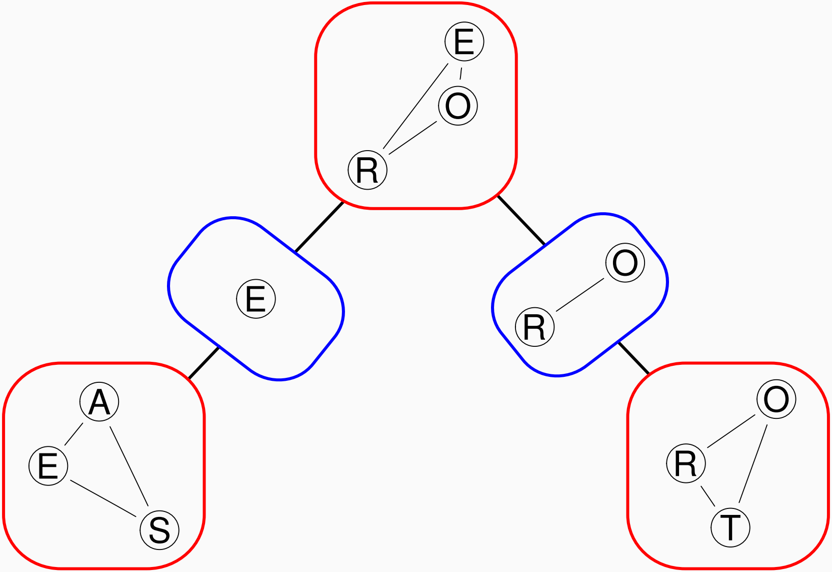
The moral graph is already triangulated, and we can see three cliques:

$$C_1 = \{A, E, S\}$$
$$C_2 = \{E, O, R\}$$
$$C_3 = \{O, R, T\}$$

with separators:

$$S_{12} = \{E\}$$
$$S_{23} = \{O, R\}$$

which we can use to build the junction tree.


E
O
R
E
O
R
A
E
S
O
R
T

In this example on the `survey` BN, the parameters for the cliques are:

$$\begin{aligned}
\Theta_{C_1} &= \mathrm{P}(A, E, S) = \mathrm{P}(A)\,\mathrm{P}(S)\,\mathrm{P}(E \mid A, S) \\
\Theta_{C_2} &= \mathrm{P}(E, O, R) = \mathrm{P}(O \mid E)\,\mathrm{P}(R \mid E)\,\mathrm{P}(E) \\
\Theta_{C_3} &= \mathrm{P}(O, R, T) = \mathrm{P}(T \mid O, R)\,\mathrm{P}(O), \mathrm{P}(R)
\end{aligned}$$

and those for the separators are:

$$\begin{aligned}
\Theta_{S_{12}} &= \mathrm{P}(E) \\
\Theta_{S_{23}} &= \mathrm{P}(O, R)
\end{aligned}$$

All can be readily computed from the local distributions in the BN.

Estimating the Parameters

```
C1 = coef(bn$E)
for (a in A.lv)
  for (s in S.lv)
    C1[, a, s] = C1[, A = a, S = s] * coef(bn$A)[a] * coef(bn$S)[s]
C1
```

```
, , S = M

      A
E       young  adult    old
  high 0.1350 0.2160 0.1056
  uni  0.0450 0.0840 0.0144

, , S = F

      A
E       young  adult    old
  high 0.0768 0.1400 0.0720
  uni  0.0432 0.0600 0.0080
```

```
S12 = margin.table(C1, 1)
S12
```

```
E
 high   uni
0.745 0.255
```

Estimating the Parameters

```
C2 = array(0, dim = c(2, 2, 2), dimnames = list(O = O.lv, R = R.lv, E = E.lv))
for (o in O.lv)
  for (r in R.lv)
    for (e in E.lv)
      C2[o, r, e] = coef(bn$O)[o, e] * coef(bn$R)[r, e] * S12[e]
C2
```

```
, , E = high

      R
O        small    big
  emp  0.17890 0.5367
  self 0.00745 0.0224

, , E = uni

      R
O        small    big
  emp  0.04685 0.1874
  self 0.00407 0.0163
```

Estimating the Parameters

```r
S23 = margin.table(C2, 1:2)
S23
```

```
       R
 O        small    big
   emp   0.2257 0.7241
   self  0.0115 0.0387
```

```r
C3 = coef(bn$T)
for (t in T.lv)
  for (o in O.lv)
    for (r in R.lv)
      C3[t, o, r] = C3[t, o, r] *
                      S23[o, r]
```

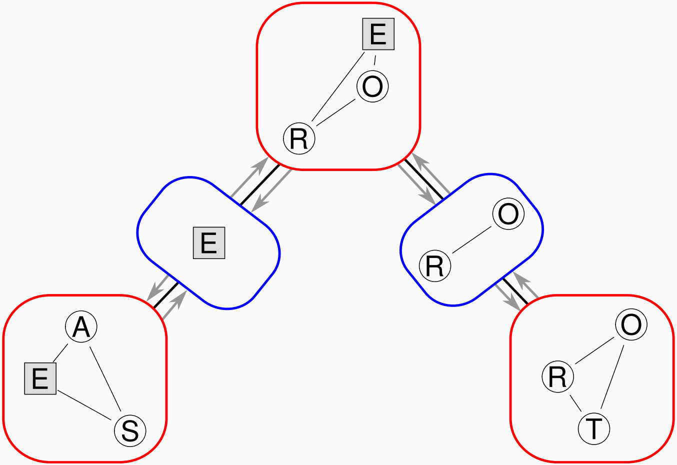
```r
C3
```

```
, , R = small

       O
T             emp     self
  car   0.108356 0.006455
  train 0.094812 0.004150
  other 0.022574 0.000922

, , R = big

       O
T             emp     self
  car   0.419963 0.027059
  train 0.173778 0.008118
  other 0.130333 0.003479
```



Say we set Education to “high school”: we can change it directly in S_{12}, but then we need to propagate the changes to C_1 and C_2; and from C_2 to S_{23} and to C_3. This is called belief propagation by message passing.

```r
new.S12 = S12
new.S12["high"] = 1
new.S12["uni"] = 0
new.S12
```

```
 high  uni
    1    0
```

```r
new.C1 = C1
for (e in E.lv)
  for (a in A.lv)
    for (s in S.lv)
      new.C1[e, a, s] =
        C1[e, a, s] / S12[e] *
                     new.S12[e]
```

```r
new.C1
```

```
, , S = M

      A
E       young  adult    old
  high 0.1811 0.2898 0.1417
  uni  0.0000 0.0000 0.0000

, , S = F

      A
E       young  adult    old
  high 0.1030 0.1878 0.0966
  uni  0.0000 0.0000 0.0000
```

```r
margin.table(new.C1, 1)
```

```
E
high  uni
   1    0
```

`margin.table(new.C1)` and `new.S12` match as expected.

Belief Propagation and Message Passing

```
new.C2 = C2
for (o in O.lv)
  for (r in R.lv)
    for (e in E.lv)
      new.C2[o, r, e] =
        C2[o, r, e] / S12[e] *
                     new.S12[e]
new.C2
```

```
, , E = high

      R
O       small  big
  emp    0.24 0.72
  self   0.01 0.03

, , E = uni

      R
O       small big
  emp       0    0
  self      0    0
```

```
new.S23 = margin.table(new.C2, 1:2)
new.S23
```

```
      R
O       small  big
  emp    0.24 0.72
  self   0.01 0.03
```

```
new.C3 = C3
for (t in T.lv)
  for (o in O.lv)
    for (r in R.lv)
      new.C3[t, o, r] =
        C3[t, o, r] / S23[o, r] *
                     new.S23[o, r]
```

Which completes the first iteration of belief propagation.

In more complex graphs and more complex queries we may need more than one iteration, but for this relatively simple network the belief propagation is complete.

Computing $\mathrm{P}(\texttt{S} = \texttt{"M"}, \texttt{T} = \texttt{"car"})$ at this point can be done easily by:

```
T = margin.table(new.C3, 1)
S = margin.table(new.C1, 3)
as.numeric(S["M"] * T["car"])
```

```
[1] 0.343
```

because Sex and Transportation are in different cliques and are separated by Education, and therefore independent.

Junction trees and belief propagation are implemented in the **gRain** package. In order to answer our query, we convert the BN from **bnlearn** to its equivalent in **gRain** with `as.grain()` and we construct the junction tree with `compile()`.

```
library(gRain)
junction = compile(as.grain(bn))
```

Then we set the evidence on the node, fixing it to “high school” with probability 1 with `setEvidence()`.

```
jedu = setEvidence(junction, nodes = "E", states = "high")
```

And after that, we can perform our conditional probability query with `querygrain()`, which also takes care of the belief propagation.

```
SxT.cpt = querygrain(jedu, nodes = c("S", "T"), type = "joint")
```

Joint and Marginal Conditional Probabilities

The result of our query is the joint distribution of Sex and Travel given that Education is "high school".

```
SxT.cpt
```

```
    T
S     car train  other
  M 0.343 0.174 0.0962
  F 0.217 0.110 0.0609
```

Similarly, we can use `querygrain()` compute the marginal distributions of Sex and Travel conditional on Education.

```
querygrain(jedu, nodes = c("S", "T"), type = "marginal")
```

```
$S
S
    M     F
0.613 0.387

$T
T
  car train other
0.559 0.283 0.157
```

Interestingly, we can also compute the conditional distribution of Sex given Travel (still conditioning on Education being "high school"), which turns out to be:

```
querygrain(jedu, nodes = c("S", "T"), type = "conditional")
```

```
   T
S     car train other
  M 0.613 0.613 0.613
  F 0.387 0.387 0.387
```

This makes sense in the light of d-separation, which implies conditional independence.

```
dsep(bn, x = "S", y = "T", z = "E")
```

```
[1] TRUE
```

Approximate inference works exactly in the same way as for DBNs. Sampling from a linear regression model is easy:

1. plug in the values of the parents;
2. generate the residuals from a normal distribution with mean zero and the specified variance.

The only major difference is that we cannot compute the probability of events that correspond to a point value, because probability is associated with intervals for continuous variables.

Exact inference is much easier than for DBNs. The distribution of some event nodes Q conditional on evidence nodes $E = \mathbf{e}$ has a (multivariate) normal distribution with

$$\widetilde{\boldsymbol{\mu}} = \boldsymbol{\mu}_Q + \Sigma_{QE}\Sigma_{EE}^{-1}(\mathbf{e} - \boldsymbol{\mu}_E) \quad \text{and} \quad \widetilde{\Sigma} = \Sigma_{QQ} - \Sigma_{QE}\Sigma_{EE}^{-1}\Sigma_{EQ}.$$

Use truncated normals in the case of interval evidence.

What is the probability that a student will get a distinction mark in algebra after getting a low mark at most in analysis?

```r
cpquery(marks.bn, (ALG >= 70), evidence = (ANL <= 50), method = "ls")
```

```
[1] 0.0015
```

```r
cpquery(marks.bn, (ALG >= 70), evidence = list(ANL = c(0, 50)), method = "lw")
```

```
[1] 0.00165
```

```r
mvn = gbn2mvnorm(marks.bn)
mu.tilde = mvn$mu["ALG"] + mvn$sigma["ANL", "ALG"] / mvn$sigma["ANL", "ANL"] *
  (50 - mvn$mu["ANL"])
sigma.tilde = mvn$sigma["ALG", "ALG"] -
  1/mvn$sigma["ANL", "ANL"] * mvn$sigma["ANL", "ALG"]^2
pnorm(70, mean = mu.tilde, sd = sqrt(sigma.tilde), lower.tail = FALSE)
```

```
[1] 0.00936
```

```r
mu.tilde = mvn$mu["ALG"] + mvn$sigma["ANL", "ALG"] / mvn$sigma["ANL", "ANL"] *
  (20 - mvn$mu["ANL"])
pnorm(70, mean = mu.tilde, sd = sqrt(sigma.tilde), lower.tail = FALSE)
```

```
[1] 0.00000614
```

Approximate inference for CGBNs is a combination of that for DBNs and GBNs: all we said earlier applies.

Exact inference, on the other hand, combines the worst of both worlds. We cannot work with the global distribution: like DBNs, it is too large. And we cannot use the junction tree algorithm from above either: there is an adaptation that works with CGBNs in package **BayesNetBP**, but it is slower and more memory intensive.

Rats: Weight Loss Among Females

What are the distributions of weight loss among female rats after one and two weeks?

```
particles =
  cpdist(rats.bn, nodes = c("WL1", "WL2"), evidence = list(SEX = "F"),
    method = "lw", n = 10^6)
summaries = sapply(particles, function(x) c(mean = mean(x), sd = sd(x)))
t(summaries)
```

```
    mean   sd
WL1 9.75 4.14
WL2 8.65 2.79
```

```
library(BayesNetBP)
node.class = sapply(rats.bn, function(ldist) is(ldist, "bn.fit.dnode"))
jtree = Initializer(dag = as.graphNEL(rats.dag), data = rbn(rats.bn, 10^5),
          node.class = node.class, propagate = TRUE)
jtree = AbsorbEvidence(jtree, "SEX", "F")
SummaryMarginals(Marginals(jtree, c("WL1", "WL2")))
```

```
    Mean   SD n
WL1 9.77 4.14 3
WL2 8.66 2.80 3
```

Exact inference is impossible: it relies on having closed-form representations of the joint distribution of arbitrary subsets of nodes.

Approximate inference, on the other hand, can take advantage of all the advanced Monte Carlo samples. **rstan** works very well for this.

If I have a somewhat reliable system, what is the probability that my reliability is greater than 99% if I observe 5 failures in the first phase and 20 failure is the second phase?

```r
params = list(
  Fp = c(2, 50),
  A1p = 20,
  A2p = 400
)
data = sampling(reliability.bn, algorithm = "Fixed_param", data = params,
        iter = 10^5, seed = 42)
nodes = c("FAILPROB", "ACCESS1", "ACCESS2", "FAILNUM1", "FAILNUM2")
particles = as.data.frame(extract(data)[nodes])
```

```r
partE = particles[(particles[, "FAILNUM1"] < 5) && (particles[, "FAILNUM2"] < 20), ]
nE = nrow(partE)
partEQ = partE[partE[, "FAILPROB"] < 0.01, ]
nEQ = nrow(partEQ)
nEQ/nE
```

```
[1] 0.0927
```

Relevant Functions in bnlearn

- `as.grain()` to export a fitted BN from **bnlearn** to **gRain**.
- `rbn()` to generate random samples from a BN.
- `cpdist()` to generate random samples from a BN conditional on some evidence.
- `cpquery()` to perform approximate inference with logic sampling and likelihood weighting.

Borrowed from **BayesNetBP**: `Initializer()`, `AbsorbEvidence()`, `Marginals()`.
Borrowed from **gRain**: `compile()`, `setEvidence()`, `querygrain()`.

1. Models in machine learning must be able to decide whether to perform particular actions given evidence on the surrounding environment.
2. The basis of these decisions are the predictions and the conditional probabilities computed after incorporating evidence into the model.
3. In the context of BNs computing these probability is called inference.
4. There are two classes of algorithms to perform inference: approximate and exact algorithms.
5. Approximate algorithms generate random samples to simulate real-world experiments.
6. Exact algorithms automate the mathematical steps we would perform to manipulate the probabilities in the model.

Thanks!

Any questions?