Laporan Uji Coba Praktikum Probabilistic Learning - Bayesian Sampling Kecerdasan Komputasional (KK) - RKA Kelas N

Nama : Abdan Hafidz

NRP : 5054231021

In [4]:
```
from probability import *
from utils import print_table
from notebook import psource, pseudocode, heatmap
```

Inferensi Aproksimasi pada Bayesian Networks

Pada Bayesian Networks, metode inferensi yang *exact* (tepat) sering kali tidak dapat diskalakan dengan baik terutama pada jaringan yang sangat besar dan kompleks. Hal ini disebabkan oleh kompleksitas komputasi yang meningkat secara eksponensial seiring bertambahnya jumlah variabel dan dependensi antar variabel.

Mengapa Inferensi *Exact* Tidak Efisien?

Metode inferensi *exact* seperti **Variable Elimination** atau **Belief Propagation** membutuhkan banyak sumber daya komputasi karena perlu menghitung distribusi probabilitas yang lengkap untuk setiap variabel. Ketika jaringan menjadi lebih besar, jumlah kombinasi yang harus dievaluasi juga bertambah, sehingga waktu dan memori yang dibutuhkan menjadi sangat tinggi.

Inferensi Aproksimasi dengan Monte Carlo

Sebagai alternatif, kita dapat menggunakan metode **Inferensi Aproksimasi**. Salah satu pendekatan populer adalah **algoritma sampling acak**, yang juga dikenal sebagai **Monte Carlo algorithms**. Metode ini tidak berusaha menghitung nilai yang *exact*, melainkan melakukan pendekatan melalui sejumlah sampel acak untuk memperkirakan distribusi probabilitas.

Cara Kerja Algoritma Monte Carlo

Algoritma Monte Carlo bekerja dengan cara:

1. **Membangkitkan Sampel Acak**: Membuat sampel dari distribusi probabilitas sesuai dengan struktur Bayesian Network.
2. **Menghitung Frekuensi**: Menggunakan sampel yang dihasilkan untuk menghitung frekuensi kemunculan berbagai nilai atau kejadian.

3. **Memperkirakan Distribusi**: Frekuensi tersebut kemudian digunakan untuk mendekati nilai probabilitas yang diinginkan.

Keuntungan Inferensi Aproksimasi

- **Lebih Cepat untuk Jaringan Besar**: Pendekatan ini lebih efisien secara komputasi pada jaringan besar karena tidak perlu menghitung semua kemungkinan kombinasi.
- **Fleksibilitas Tinggi**: Dapat diterapkan pada berbagai jenis jaringan, bahkan yang memiliki dependensi kompleks.
- **Skalabilitas**: Dapat ditingkatkan skalanya dengan menambah jumlah sampel untuk mendapatkan hasil yang lebih akurat.

Contoh Algoritma Monte Carlo

Beberapa contoh algoritma Monte Carlo yang sering digunakan dalam inferensi aproksimasi adalah:

- **Gibbs Sampling**: Metode yang menggunakan sampel dari distribusi kondisional setiap variabel.
- **Likelihood Weighting**: Menggunakan bobot berdasarkan nilai variabel yang diketahui untuk memperkirakan probabilitas.
- **Metropolis-Hastings**: Algoritma yang menghasilkan sampel berdasarkan rasio probabilitas.

In [ ]:
```
psource(BayesNode.sample)
```

Metode Sampling pada Bayesian Networks

Sebelum membahas berbagai algoritma sampling, kita akan melihat metode **BayesNode.sample**. Metode ini berfungsi untuk mengambil sampel dari distribusi suatu variabel, dengan kondisi nilai-nilai pada **parent_variables** (variabel-variabel induk). Artinya, metode ini akan mengembalikan nilai **True** atau **False** secara acak, berdasarkan probabilitas kondisional yang diberikan oleh variabel-variabel induknya. Untuk kemudahan, kita menggunakan fungsi **probability** dari modul **utils**, yang akan mengembalikan **True** sesuai probabilitas yang diberikan sebagai parameter.

Prior Sampling

Prior Sampling adalah salah satu metode sampling sederhana yang dilakukan dengan mengikuti urutan topologis pada Bayesian Network.

Cara Kerja Prior Sampling

1. **Mulai dari Node Teratas**: Kita mulai dari node (variabel) yang berada di puncak jaringan, yaitu variabel tanpa induk.

2. **Sampel Berdasarkan Distribusi Kondisional**: Untuk setiap variabel $\mathbf{X_i}$, kita mengambil sampel berdasarkan $\mathbf{P(X_i \mid parents(X_i))}$. Artinya, distribusi probabilitas dari variabel tersebut dikondisikan oleh nilai-nilai variabel induknya yang sudah ditetapkan.
3. **Simulasi Jaringan**: Metode ini dapat dianggap sebagai simulasi yang menghasilkan skenario berdasarkan struktur dan probabilitas jaringan.

Contoh

Misalkan kita memiliki Bayesian Network sederhana dengan dua variabel:

- **Rain** (Hujan) tanpa variabel induk.
- **Sprinkler** yang bergantung pada **Rain**.

Jika kita menggunakan Prior Sampling:

- Kita mulai dengan mengambil sampel untuk **Rain** berdasarkan distribusinya.
- Setelah itu, kita mengambil sampel untuk **Sprinkler** yang dikondisikan oleh nilai **Rain** yang sudah diperoleh.

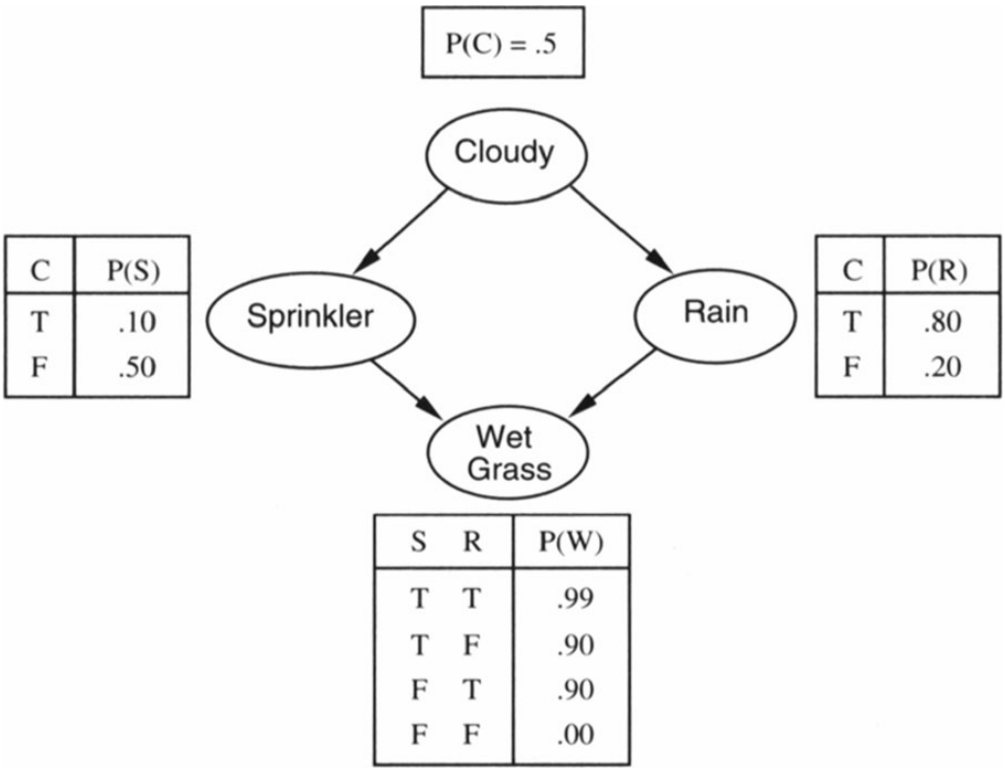
Keunggulan

- **Sederhana** dan mudah diimplementasikan.
- Dapat digunakan untuk menghasilkan data simulasi dari Bayesian Network.

Namun, perlu diingat bahwa metode ini mungkin tidak efisien pada jaringan yang sangat besar, terutama jika terdapat variabel yang jarang terjadi (*low probability events*), karena akan membutuhkan banyak sampel untuk mendapatkan estimasi yang akurat.

In [ ]:
```
psource(prior_sample)
```

The function **prior_sample** implements the algorithm described in **Figure 14.13** of the book. Nodes are sampled in the topological order. The old value of the event is passed as evidence for parent values. We will use the Bayesian Network in **Figure 14.12** to try out the **prior_sample**



Traversing the graph in topological order is important. There are two possible topological orderings for this particular directed acyclic graph.

1. `Cloudy -> Sprinkler -> Rain -> Wet Grass`
2. `Cloudy -> Rain -> Sprinkler -> Wet Grass`

We can follow any of the two orderings to sample from the network. Any ordering other than these two, however, cannot be used.
One way to think about this is that `Cloudy` can be seen as a precondition of both `Rain` and `Sprinkler` and just like we have seen in planning, preconditions need to be satisfied before a certain action can be executed.
We store the samples on the observations. Let us find **P(Rain=True)** by taking 1000 random samples from the network.

In [7]:
```
N = 1000
all_observations = [prior_sample(sprinkler) for x in range(N)]
```

Kode di atas melakukan **Prior Sampling** sebanyak `N = 1000` kali pada sebuah **Bayesian Network** .

- `N = 1000` : Menentukan jumlah sampel yang akan diambil, yaitu 1000 sampel.
- `prior_sample(sprinkler)` : Fungsi `prior_sample` digunakan untuk mengambil satu sampel dari jaringan dengan node**sprinkler** , mengikuti urutan topologis jaringan.
- `all_observations` : Variabel ini menyimpan semua hasil sampel dalam bentuk list, di mana setiap elemen adalah hasil dari satu panggilan `prior_sample(sprinkler)` .
- **List Comprehension** `[... for x in range(N)]` : Menjalankan fungsi `prior_sample(sprinkler)` sebanyak `N` kali dan menyimpan hasilnya ke dalam list `all_observations` .

Hasilnya adalah `all_observations`, list yang berisi 1000 sampel hasil dari prior sampling pada jaringan **sprinkler** .

Sekarang kita filter untuk mendapatkan observasi dimana Rain = True

In [8]:
```
rain_true = [observation for observation in all_observations if
observation['Rain'] == True]
```

P(Rain=True)

In [9]:
```
answer = len(rain_true) / N
print(answer)
```

```
0.513
```

Ketidakpastian dalam Sampling

Ketika melakukan sampling ulang, hasil yang diperoleh mungkin berbeda karena kita tidak memiliki kontrol penuh atas distribusi sampel acak. Hal ini disebabkan oleh sifat **acak** dari metode sampling, di mana setiap kali dijalankan, sampel diambil berdasarkan distribusi probabilitas yang bisa menghasilkan variasi berbeda. Oleh karena itu, hasil sampling dari Bayesian Network dapat bervariasi antar percobaan, meskipun dilakukan dengan parameter yang sama.

In [10]:
```
N = 1000
all_observations = [prior_sample(sprinkler) for x in range(N)]
rain_true = [observation for observation in all_observations if
observation['Rain'] == True]
answer = len(rain_true) / N
print(answer)
```

```
0.503
```

Evaluasi Distribusi Kondisional

Untuk mengevaluasi distribusi kondisional, kita dapat menggunakan proses **filtering dua langkah**:

1. **Langkah Pertama**: Memisahkan variabel-variabel yang konsisten dengan bukti yang ada. Misalnya, jika kita ingin mencari **P(Cloudy=True | Rain=True)**, kita terlebih dahulu memfilter data yang sesuai dengan **Rain=True** dan menyimpannya dalam variabel **rain_true**.

2. **Langkah Kedua**: Setelah itu, kita melakukan filter kedua pada **rain_true** untuk menghitung **P(Rain=True dan Cloudy=True)**. Dengan membandingkan hasil ini, kita dapat menghitung probabilitas kondisional yang diinginkan.

Metode ini memudahkan kita untuk menghitung probabilitas variabel query dengan menggunakan bukti yang telah diberikan.

In [11]:
```
rain_and_cloudy = [observation for observation in rain_true if observation['Cloudy'] == True]
answer = len(rain_and_cloudy) / len(rain_true)
print(answer)
```

```
0.7892644135188867
```

Rejection Sampling

Rejection Sampling adalah metode sampling yang mirip dengan **Prior Sampling**. Metode ini bekerja dengan terlebih dahulu menghasilkan sampel dari distribusi awal (prior) yang ditentukan oleh Bayesian Network, kemudian **menolak** semua sampel yang tidak sesuai dengan bukti yang diberikan.

Cara Kerja Rejection Sampling

1. **Generate Sampel**: Mengambil sampel dari distribusi prior yang ada di jaringan.
2. **Filter Berdasarkan Bukti**: Menolak semua sampel yang tidak konsisten dengan bukti (evidence) yang kita miliki.

Kelebihan Rejection Sampling

- Berguna jika kita sudah mengetahui query dan buktinya sebelum proses sampling.
- Berbeda dengan **Prior Sampling**, yang bisa bekerja untuk query apapun, metode ini lebih efisien dalam skenario di mana kita memiliki bukti dengan probabilitas sangat kecil.

Masalah pada Prior Sampling

Misalkan kita ingin mencari **P(A | e)** pada jaringan Bayesian dengan evidence **e** yang memiliki probabilitas sangat kecil. Dalam **Prior Sampling**, jika **e** sangat jarang terjadi, kita mungkin tidak akan mendapatkan sampel di mana **e** benar. Hal ini menyebabkan **P(e) = 0** sehingga **P(A | e) / P(e) = 0/0**, yang tidak terdefinisi. Meskipun kita bisa meningkatkan jumlah sampel, tidak ada jaminan kita akan menemukan sampel di mana **e** benar.

Solusi oleh Rejection Sampling

Dengan **Rejection Sampling**, kita hanya mempertimbangkan sampel yang konsisten dengan evidence **e**. Jadi, setiap sampel yang tidak sesuai dengan evidence akan langsung ditolak. Ini memastikan bahwa kita memiliki cukup data yang sesuai dengan evidence untuk menjawab query terkait.

Implementasi

Algoritma **Rejection Sampling** diimplementasikan dalam fungsi `rejection_sampling` seperti yang dijelaskan pada **Figure 14.14**.

In [ ]:
```
psource(rejection_sampling)
```

Penjelasan Fungsi Rejection Sampling

Fungsi **Rejection Sampling** menghitung jumlah setiap nilai yang mungkin dari variabel query dan meningkatkan hitungan ketika sampel yang diamati konsisten dengan bukti. Fungsi ini menerima input parameter sebagai berikut:

- **X**: Variabel Query yang ingin dicari probabilitasnya.
- **e**: Bukti yang diberikan yang harus konsisten dengan sampel.
- **bn**: Jaringan Bayesian (Bayes Net) yang digunakan.
- **N**: Jumlah sampel prior yang akan dihasilkan.

Fungsi ini menggunakan **consistent_with** untuk memeriksa apakah sampel yang diambil konsisten dengan bukti yang diberikan. Sampel yang tidak konsisten dengan bukti akan ditolak, sementara yang konsisten akan digunakan untuk memperbarui hitungan nilai query.

In [ ]:
```
psource(consistent_with)
```

P(Cloudy=True | Rain=True)

In [14]:
```
p = rejection_sampling('Cloudy', dict(Rain=True), sprinkler, 1000)
p[True]
```

Out[14]:
```
0.8180076628352491
```

Likelihood Weighting

Rejection Sampling dapat memakan waktu lama jika probabilitas menemukan bukti yang konsisten rendah, dan menjadi lebih lambat pada jaringan besar atau dengan banyak variabel bukti. Jika bukti terdiri dari banyak variabel, banyak sampel yang akan ditolak. **Likelihood Weighting** mengatasi masalah ini dengan cara memperbaiki bukti (tidak melakukan sampling pada bukti) dan menggunakan bobot untuk memastikan bahwa sampling tetap konsisten dengan bukti.

Pseudocode yang dijelaskan dalam **Figure 14.15** diimplementasikan dalam fungsi **likelihood_weighting** dan **weighted_sample**.

In [ ]:
```
psource(weighted_sample)
```

Penjelasan Fungsi **weighted_sample**

Fungsi **weighted_sample** mengambil sampel dari **Bayesian Network** yang konsisten dengan bukti **e** dan mengembalikan sampel tersebut beserta bobotnya, yaitu kemungkinan bahwa sampel tersebut sesuai dengan bukti yang diberikan. Fungsi ini menerima dua parameter:

- **bn**: Jaringan Bayesian yang digunakan.
- **e**: Bukti yang diberikan.

Bobot dihitung dengan mengalikan **$P(x_i \mid parents(x_i))$** untuk setiap node dalam bukti. Pada awal fungsi, nilai **event** diset sesuai dengan **e** (bukti) yang diberikan.

In [ ]:
```
psource(likelihood_weighting)
```

Penjelasan Fungsi **likelihood_weighting**

Fungsi **likelihood_weighting** mengimplementasikan algoritma untuk menyelesaikan masalah inferensi. Kode ini mirip dengan **rejection_sampling**, namun alih-alih menambah satu untuk setiap sampel, fungsi ini menambahkan bobot yang diperoleh dari **weighted_sample**.

Dengan menggunakan bobot, **likelihood_weighting** memastikan bahwa sampel yang diambil sesuai dengan bukti dan memberikan kontribusi yang proporsional terhadap hasil inferensi.

In [17]:
```
likelihood_weighting('Cloudy', dict(Rain=True), sprinkler,
200).show_approx()
```

Out[17]:
```
'False: 0.184, True: 0.816'
```

Gibbs Sampling

Pada **Likelihood Sampling**, bobot dapat menjadi sangat kecil jika variabel bukti berada di bagian bawah **Bayesian Network**, karena pengaruh hanya menyebar ke bawah dalam proses sampling ini.

Gibbs Sampling mengatasi masalah ini. Fungsi **gibbs_ask** yang diimplementasikan sesuai dengan algoritma yang dijelaskan pada **Figure 14.16** memungkinkan untuk melakukan sampling dengan cara yang lebih efektif, di mana pengaruh variabel dapat menyebar lebih merata di seluruh jaringan.

In [ ]:
```
psource(gibbs_ask)
```

Penjelasan Fungsi **gibbs_ask**

Pada fungsi **gibbs_ask**, kita menginisialisasi variabel non-bukti dengan nilai acak. Kemudian, kita memilih variabel non-bukti dan melakukan sampling dari **P(Variable | nilai dalam keadaan saat ini dari semua variabel lainnya)** secara berulang.

Untuk mempercepat proses, kita menggunakan **markov_blanket_sample** alih-alih sampling langsung, karena istilah yang tidak melibatkan variabel akan saling menghilangkan dalam perhitungan.

Argumen untuk **gibbs_ask** mirip dengan **likelihood_weighting**, yang mencakup jaringan Bayesian dan bukti yang diberikan.

In [19]:
```
gibbs_ask('Cloudy', dict(Rain=True), sprinkler, 200).show_approx()
```

Out[19]:
```
'False: 0.24, True: 0.76'
```

Analisis Waktu Eksekusi

Mari kita lihat berapa banyak waktu yang dibutuhkan oleh setiap algoritma untuk dijalankan.

In [20]:
```
%%timeit
all_observations = [prior_sample(sprinkler) for x in range(1000)]
rain_true = [observation for observation in all_observations if
observation['Rain'] == True]
len([observation for observation in rain_true if observation['Cloudy']
== True]) / len(rain_true)
```

```
5.4 ms ± 330 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [21]:
```
%%timeit
rejection_sampling('Cloudy', dict(Rain=True), sprinkler, 1000)
```

```
7.21 ms ± 520 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

In [22]:
```
%%timeit
likelihood_weighting('Cloudy', dict(Rain=True), sprinkler, 200)
```

```
1.16 ms ± 67.5 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

In [23]:
```
%%timeit
gibbs_ask('Cloudy', dict(Rain=True), sprinkler, 200)
```

```
27.3 ms ± 2.19 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

Analisis Waktu Eksekusi Algoritma

Seperti yang diharapkan, semua algoritma memiliki waktu eksekusi yang sangat mirip. Namun, **Rejection Sampling** akan lebih cepat dan lebih akurat ketika probabilitas menemukan sampel yang konsisten dengan bukti yang dibutuhkan sangat kecil.

Sementara itu, **Likelihood Weighting** adalah yang tercepat di antara semuanya karena tidak melibatkan penolakan sampel, meskipun memiliki variansi yang cukup tinggi.