Constraint Satisfaction Problem

10S3001 – Artificial Intelligence

Samuel I. G. Situmeang

Faculty of Informatics and Electrical Engineering

Objectives

Students are able:

- to formulate problems with the constraint satisfaction problem (CSP) type,
- to perform inference in CSP, which is also called constraint propagation,
- to apply backtracking search for CSP, and
- to structure CSP in the form of a constraint graph.

Siswa mampu:

- untuk merumuskan masalah dengan tipe *constraint satisfaction problem* (CSP),
- untuk melakukan inferensi dalam CSP, yang disebut juga *constraint propagation*,
- untuk menerapkan pencarian *backtracking* untuk CSP, dan
- untuk menstrukturkan CSP dalam bentuk *constraint graph*.

Pada pembelajaran di sesi ini, kita membahas:

- Yang pertama adalah pendefinisian *constraint satisfaction problem* untuk memahami cara memformulasi sebuah masalah dalam tipe CSP.
- Yang kedua adalah *constraint propagation*. Ini adalah suatu jenis inferensi, di mana kita menggunakan batasan untuk mengurangi banyaknya nilai yang legal pada suatu variabel, yang selanjutnya dapat mengurangi banyaknya nilai yang legal pada variabel lain, dan seterusnya. Bayangin suatu variabel, katakanlah W yang memiliki beberapa nilai yang legal yaitu merah, biru, dan hijau. Nah, *constraint propagation* kita gunakan untuk mengurangi nilai yang legal pada variabel W tersebut. Kenapa harus dikurangin? Sabar, kita akan bahas pada waktunya.

- Terkadang kita dapat menyelesaikan proses *constraint propagation* dan masih memiliki variabel dengan beberapa kemungkinan nilai. Dalam kasus ini, kita membahas algoritma penelusuran mundur atau *backtracking search* yang dapat digunakan pada *partial assignments*. Apa maksudnya *partial assignment*? Bayangin masalah pewarnaan wilayah dalam sebuah peta. Diketahui dalam peta tersebut, terdapat 4 wilayah, katakanlah A, B, C, dan D. Pada kondisi awal (*initial state*), A dan B diketahui masing-masing telah diwarnai merah dan biru, sementara C dan D belum diwarnai sama sekali. Inilah contoh *partial assignment* tersebut, di mana kita mendapati bahwa tidak semua variabel dalam *state* yang kita sedang periksa memiliki nilai.
- Kita juga akan memeriksa cara-cara di mana struktur masalah, misalnya dalam bentuk *constraint graph*, dapat digunakan untuk menemukan solusi dengan cepat.

Recall

- Search problems:
  - Find the **sequence of actions** that leads to the goal.
  - Sequence of actions means a **path** in the search space.
  - Paths come with different costs and depths.
  - We use “rules of thumb” aka **heuristics** to guide the search efficiently.
- Constraint satisfaction problems:
  - A search problem too!
  - We care about the **goal itself**.

Pada bahasan *uninformed search*, *informed search*, dan *local search* dipaparkan gagasan bahwa suatu masalah dapat diselesaikan dengan melakukan pencarian dalam ruang keadaan (***state space***). *State space* pada bahasan tersebut kita pahami sebagai suatu graf di mana setiap simpul (***node***) pada graf tersebut merepresentasi sebuah *state* dan garis (***edge****)* penghubung simpul-simpul tersebut adalah aksi yang misalnya dilakukan oleh agen pencarian.

Dalam hal ini, kita mencoba membuat agen pencarian yang dapat menemukan urutan tindakan (***sequence of actions***) yang mengarah ke tujuan. Urutan tindakan berarti suatu jalur (***path***) di ruang pencarian. Setiap jalur memiliki ***cost*** dan ***depth*** yang berbeda-beda. Kita menggunakan "aturan praktis“ (**rule of thumb**) alias **heuristik** untuk memandu pencarian secara efisien.

Constraint satisfaction problems juga suatu masalah pencarian. Namanya juga pencarian, tentulah kita peduli dengan **tujuan** pencarian tersebut.

CSPs definition

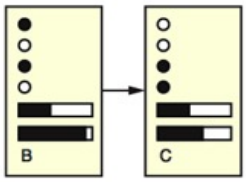
- **Search problems**:
  - A state is a **black box**, implemented as some data structure. Recall atomic representation.
  - A goal test is a function over the states.

Kita melihat bahwa heuristik khusus domain (misalnya heuristik "jumlah ubin yang salah tempat" dalam domain permainan 8-Puzzle) dapat memperkirakan *cost* untuk mencapai tujuan dari *state* tertentu, tetapi dari sudut pandang algoritma penelusuran atau pencarian, setiap *state* bersifat ***atomic***, atau tidak dapat dibagi – bayangin sebuah kotak hitam (***black box***) tanpa struktur internal. Untuk setiap masalah, kita memerlukan kode khusus domain untuk menjelaskan transisi antar-*state*.

Perhatikan *state* B dan *state* C tidak memiliki struktur internal.

CSPs definition

- **CSPs**:
  - A state: defined by variables X_i with values from domain D_i. Recall factored representation.
  - A goal test is a **set of constraints** specifying **allowable combinations** of values for subsets of variables.



Dalam bahasan kali ini, kita akan membuka kotak hitam tersebut dengan menggunakan representasi berfaktor (***factored representation***) untuk setiap *state*, di mana setiap *state* memiliki satu set variabel (***variable***), yang masing-masing variabelnya memiliki nilai (***value***). Suatu masalah diselesaikan ketika setiap variabel memiliki nilai yang memenuhi semua batasan pada variabel tersebut. Masalah yang dijelaskan dengan cara ini disebut ***constraint satisfaction problem***, atau **CSP**.

Algoritma pencarian pada CSP memanfaatkan struktur *state* dan menggunakan **heuristik umum** daripada **heuristik khusus domain** untuk memungkinkan solusi didapatkan dari masalah yang kompleks. Ide utamanya adalah menghilangkan sebagian besar bagian ruang pencarian (*search space*) sekaligus dengan mengidentifikasi kombinasi *variable* / *value* yang melanggar batasan. CSP memiliki kelebihan lagi yaitu model aksi dan transisi dapat disimpulkan dari deskripsi masalah.

Perhatikan *state* B dan *state* C tidak lagi hanya kotak hitam tanpa struktur internal. Setiap *state* tersebut memiliki representasi yang lebih kompleks yang memungkinkan pemahaman yang lebih cerdas dan lebih dalam pada masalah yang dihadapi untuk menyelesaikannya.

CSPs definition

- A constraint satisfaction problem consists of **three elements**:
  - A set of **variables**, $X = \{X_1, X_2, \dots, X_n\}$
  - A set of **domains** for each variable: $D = \{D_1, D_2, \dots, D_n\}$
    - D_i consists of a set of allowable values $\{v_1, \dots, v_k\}$, for variable X_i,.
  - A set of **constraints** C that specify allowable combinations of values.
    - C_j consists of a pair $\langle scope, rel \rangle$. e.g. if X_1 and X_2 both have domain $\{1,2,3\}$, then the constraint saying that X_1 must be greater than X_2 can be written as $\langle (X_1, X_2), \{(3,1), (3,2), (2,1)\} \rangle$ or $\langle (X_1, X_2), X_1 > X_2 \rangle$.
- Solving the CSP: **finding the assignment(s)** that **satisfy all constraints**.
- Concepts: problem formalization, backtracking search, arc consistency, etc.
- We call a solution, a **consistent assignment**.

Suatu *constraint satisfaction problem* terdiri dari tiga elemen:

- Satu set ***variables***, $X = \{X_1, X_2, \dots, X_n\}$
- Satu set ***domains*** untuk setiap variabel: $D = \{D_1, D_2, \dots, D_n\}$
- Satu set ***constraints*** C yang menspesifikasikan kombinasi nilai yang diperbolehkan.

Sebuah domain, D_i, terdiri dari atu set nilai yang diperbolehkan, $\{v_1, \dots, v_k\}$, untuk variabel X_i. Sebagai contoh, variabel Boolean akan memiliki domain $\{\text{true}, \text{false}\}$. Variabel yang berbeda dapat memiliki domain berbeda dengan ukuran berbeda pula.

C_j terdiri dari pasangan $\langle scope, rel \rangle$. e.g. jika X_1 dan X_2 masing-masing memiliki domain $\{1,2,3\}$, maka batasan atau *constraint* yang menyatakan bahwa X_1 harus lebih besar daripada X_2 dapat ditulis dalam bentuk $\langle (X_1, X_2), \{(3,1), (3,2), (2,1)\} \rangle$ atau $\langle (X_1, X_2), X_1 > X_2 \rangle$.

CSP menangani ***assignment*** nilai ke variabel, $\{X_i = v_i, X_j = v_j, \dots\}$. *Assignment* yang tidak melanggar *constraint* apa pun disebut ***consistent assignment*** atau ***legal assignment***. ***Complete assignment*** adalah *assignment* di mana setiap variabel diberi nilai, dan ***solution*** untuk CSP adalah sebuah *assignment* yang *consistent* dan

complete. ***Partial assignment*** adalah *assignment* yang membiarkan beberapa variabel tidak diberi nilai, dan ***partial solution*** adalah sebuah *partial assignment* yang konsisten. Secara umum, menyelesaikan CSP dapat dikategorikan sebagai masalah ***NP-complete***, meskipun ada *subclass* penting dari CSP yang dapat diselesaikan dengan sangat efisien.

Example problem: Map Coloring

- **Variables**: $X = \{WA, NT, Q, NSW, V, SA, T\}$
- **Domains**: $D_i = \{red, green, blue\}$
- **Constraints**: adjacent regions must have different colors; $C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$
  - Here we are using abbreviations; $SA \neq WA$ is a shortcut for $\langle (SA, WA), \{(red, green), (red, blue), (green, red), (blue, red), (blue, green)\}\rangle$ or $\langle (SA, WA), SA \neq WA\rangle$

Pewarnaan peta terinspirasi oleh teori graf. Ini masalah terkenal dalam teori graf di mana kita diberi tugas mewarnai peta dengan warna *red*, *green*, dan *blue*. Kita tidak boleh menggunakan warna yang sama pada dua wilayah yang berdekatan.

Pada contoh ini, variabel adalah setiap wilayah dalam peta yaitu WA, NT, Q, NSW, V, SA, T. Domain untuk setiap variabel adalah adalah *red*, *green*, dan *blue. Contraint* pada masalah ini mengharuskan daerah yang berdekatan harus memiliki warna yang berbeda. Karena ada 9 wilayah yang berbatasan, maka ada 9 *constraint*, yaitu $SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V$. Penulisan singkatan seperti ini adalah bentuk singkatan dari bentuk formal penulisan *constraint*.

Example problem: Map Coloring

- **Example of Solution (comp ıments):**
  - $\{WA = red, NT = green, Q =$ $A = blue, T = green\}$

Ada banyak solusi yang memungkinkan pada masalah ini, salah satunya adalah $WA = red, NT = green, Q = red, NSW = green, V = red, SA = blue, T = green$. Solusi ini disebut **complete and consistent assignment**.

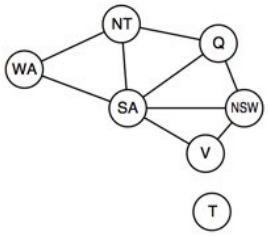
Real-world CSPs

- Assignment problems, e.g., who teaches what class?
- Timetabling problems, e.g., which class is offered when and where?
- Hardware configuration
- Spreadsheets
- Transportation scheduling
- Factory scheduling
- Floor planning
- Notice that many real-world problems involve real-valued variables

CSP dapat kita gunakan pada berbagai masalah di dunia nyata, seperti:
Masalah penugasan perkuliahan, misalnya, siapa yang mengajar kelas apa?
Masalah pengaturan waktu perkuliahan, misalnya, kelas mana yang ditawarkan kapan dan di mana?
Konfigurasi perangkat keras
Spreadsheet
Penjadwalan transportasi
Penjadwalan pabrik
Perencanaan lantai

Perhatikan bahwa banyak masalah dunia nyata melibatkan variabel dengan bilangan riil.

Constraint Graph



- **Binary CSP**: each constraint relates at most two variables constraint graph: nodes are variables, arcs show constraints
- **CSP algorithms**: use the graph structure to speed up search.
  E.g., Tasmania is an independent subproblem!

Terkadang, kita perlu merepresentasikan CSP sebagai *constraint graph*, di mana kita memiliki simpul (*node*) yang mewakili variabel dan garis (*edge*) yang mewakili batasan atau *constraint* antar-simpul.

Varieties of Variables

- **Discrete variables**:
  - Finite domains:
    - assume n variables, d values, then the number of complete assignments is $O(d^n)$.
    - e.g., map coloring, 8-queens problem
  - Infinite domains (integers, strings, etc.):
    - need to use a constraint language,
    - e.g., job scheduling. $T_1 + d \leq T_2$.
- **Continuous variables**:
  - Common in operations research
  - Linear programming problems with linear or non linear equalities.

Variabel diskrit adalah variabel yang nilainya diperoleh dengan menghitung.
Contoh: jumlah siswa yang hadir, jumlah buku merah dalam rak buku, jumlah kepala saat melempar tiga koin, tingkat kelas siswa.

Variabel kontinu adalah variabel yang nilainya diperoleh dengan mengukur.
Contoh: tinggi badan siswa di kelas, berat siswa di kelas, waktu yang dibutuhkan untuk sampai ke sekolah, jarak tempuh antar kelas.

Ada dua tipe domain variabel diskrit, yakni:
1. *finite domain* contohnya variabel yang digunakan pada *map coloring* dan 8-queens problem
2. *inifinite domain* contohnya variabel yang digunakan pada *job scheduling*

Varieties of Constraints

- **Unary constraints**: involve a single variable e.g., $SA \neq green$
- **Binary constraints**: involve pairs of variables e.g., $SA \neq WA$
- **Global constraints**: involve 3 or more variables e.g., *Alldiff* that specifies that all variables must have different values (e.g., cryptarithmetic puzzles, Sudoku)
- **Preference constraints** (**soft constraints**) indicating which solutions are preferred.
  - Example: red is better than green
  - Often represented by a cost for each variable assignment
  - constrained optimization problems

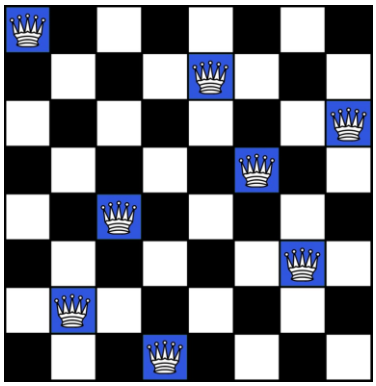
Selain memeriksa jenis variabel yang dapat muncul di CSP, ada baiknya untuk melihat jenis *constraint* juga.

- ***Unary constraints***: melibatkan satu variabel misalnya, $SA \neq green$
- ***Binary constraints***: melibatkan pasangan variabel misalnya, $SA \neq WA$
- ***Global constraints***: melibatkan 3 atau lebih variabel misalnya, *Alldiff* yang menetapkan bahwa semua variabel harus memiliki nilai yang berbeda (ini biasanya digunakan pada masalah teka-teki kriptaritmatika dan Sudoku)
- ***Preference constraints***: mengindikasi solusi mana yang lebih disukai.
  - Contoh: merah lebih baik dari hijau
  - Contoh lain misalnya, dalam masalah penjadwalan perkuliahan di universitas, Prof X mungkin lebih suka mengajar di pagi hari sedangkan Prof Y lebih suka mengajar di sore hari. Pada jadwal pengajaran, tertulis bahwa Prof. X pada pukul 2 siang. Nah, ini akan tetap menjadi solusi, tetapi tidak akan menjadi solusi yang optimal.
  - *Preference constraint* sering kali dapat dikodekan sebagai *cost* pada *individual variable assignment*-misalnya, dengan menetapkan slot sore untuk Prof. X maka sistem akan kena penalti atau *cost* 2 poin terhadap fungsi tujuan keseluruhan, sedangkan slot pagi memiliki *cost* 1. Dengan formulasi ini, CSP dengan preferensi dapat diselesaikan menggunakan

metode pencarian pengoptimalan, baik berbasis *path* atau *local*. Masalah ini disebut ***constrained optimization problem*** (**COP**).

Example problem: 8-queen

- **8-Queen:** Place 8 queens on an 8×8 chess board so no queen can attack another one.
- **Problem formalization?**

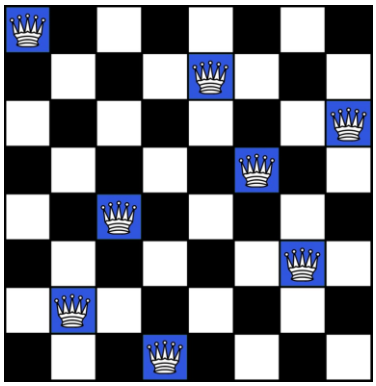


Contoh lain adalah **8-Queen** problem. Pada puzzle ini, kita diminta untuk menempatkan 8 ratu di papan catur 8 × 8 sehingga tidak ada ratu yang dapat menyerang yang lain.

Bagaimana formulasi masalahnya?

Example problem: 8-queen

- **8-Queen:** Place 8 queens on an 8×8 chess board so no queen can attack another one.
- **Problem formalization 1:**
  - One variable per queen, $Q_1, Q_2, \dots, Q_8$.
  - Each variable could have a value between 1 and 64.
  - Solution: $Q_1 = 1, Q_2 = 13, Q_3 = 24, \dots, Q_8 = 60$.

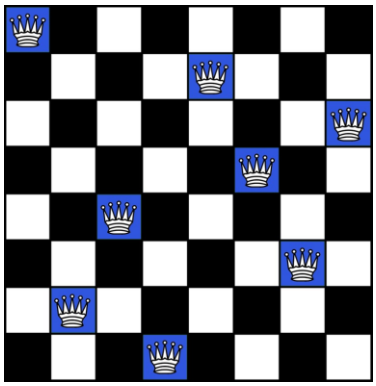


Bentuk formulasi masalah yang pertama yaitu:

- Menggunakan satu variabel per ratu, $Q_1, Q_2, \dots, Q_8$.
- Setiap variabel dapat memiliki nilai antara 1 dan 64.
- Solusi: $Q_1 = 1, Q_2 = 13, Q_3 = 24, \dots, Q_8 = 60$.

Example problem: 8-queen

- **8-Queen:** Place 8 queens on an 8×8 chess board so no queen can attack another one.
- **Problem formalization 2:**
  - One variable per queen, $Q_1, Q_2, \dots, Q_8$.
  - Each variable could have a value between 1 and 8 (columns).
  - Solution: $Q_1 = 1, Q_2 = 7, Q_3 = 5, \dots, Q_8 = 3$.

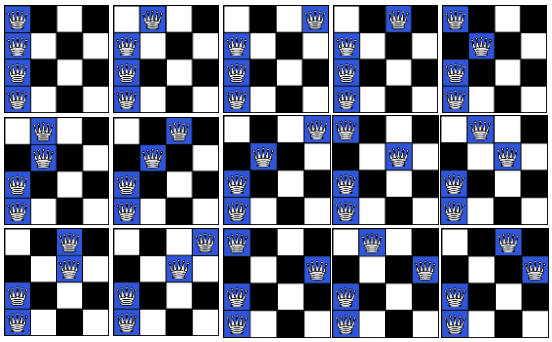


Bentuk formulasi masalah yang kedua yaitu:

- Menggunakan satu variabel per ratu, $Q_1, Q_2, \dots, Q_8$.
- Setiap variabel dapat memiliki nilai antara 1 dan 8 yang merepresentasi kolom.
- Solusi: $Q_1 = 1, Q_2 = 7, Q_3 = 5, \dots, Q_8 = 3$.

Brute Force?

- Should we simply generate and test all configurations?

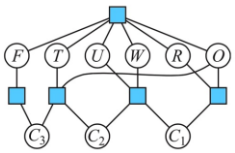


...

Jadi jelas, *brute forcing*, yang menghasilkan semua kemungkinan konfigurasi, dan memeriksa semua *constraint* tersebut, bukanlah pendekatan yang baik untuk CSP.

Example problem: Cryptarithmetic

$$\begin{array}{r} T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array}$$



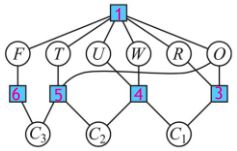
Kriptaritmatika adalah ilmu dan seni membuat dan memecahkan kriptaritma. Kriptaritmatika adalah genre teka-teki matematika di mana angka-angka diganti dengan huruf alfabet atau simbol lainnya.

Example problem: Cryptarithmetic

- **Variables**: $X = \{F, T, U, W, R, O, C_1, C_2, C_3\}$
- **Domain**: $D = \{0, 1, 2, \ldots, 9\}$
- **Constraints**:
  1. $Alldiff(F, T, U, W, R, O)$
  2. $T \neq 0,\ F \neq 0$
  3. $O + O = R + 10 \times C_1$
  4. $C_1 + W + W = U + 10 \times C_2$
  5. $C_2 + T + T = O + 10 \times C_3$
  6. $C_3 = F$

C3 C2 C1

$$\begin{array}{r} T\ W\ O \\ +\ T\ W\ O \\ \hline F\ O\ U\ R \end{array}$$



n-ary constraint (6-ary)

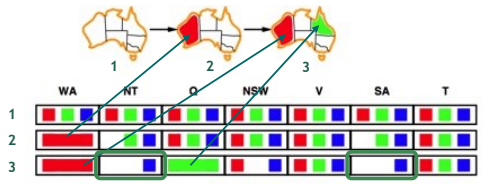
$$\begin{array}{r} 7\ 3\ 4 \\ 7\ 3\ 4 \\ \hline 1\ 4\ 6\ 8 \end{array}$$

solusi

Pada contoh ini, kita tidak ingin ada awalan 0. Jadi kita ingin T tidak merepresentasi 0, dan Anda ingin F berbeda dari 0.

Constraint Propagation

- Forward checking propagates information from assigned to unassigned variables.
- Observe:



  - Forward checking does not check interaction between unassigned variables! Here SA and NT! (They both must be blue but can't be blue!).
- Forward checking improves backtracking search but does not look very far in the future, hence does not detect all failures.
- We use constraint propagation, reasoning from constraint to constraint. e.g., arc consistency test.

Forward checking mempropagasi informasi dari *assigned variable* ke *unassigned variable*.

Perhatikan gambar pada slide. *Forward checking* tidak memeriksa interaksi antara *unassigned variable*! Di sini, SA dan NT! (berdasarkan *forward checking*, keduanya harus berwarna biru, padahal ini tidak diperbolehkan karena keduanya bertetangga!).

Forward checking meningkatkan performa *backtracking search* tetapi tidak melihat jauh di masa depan, karenanya tidak dapat mendeteksi semua kegagalan.

Kita menggunakan constraint propagation, penalaran dari *constraint* ke *constraint*. misalnya, uji konsistensi busur (*arc consistency test*).

Types of Consistency

- **Node-consistency** (unary constraints): A variable X_i is **node-consistent** if all the values of *Domain*(X_i) satisfy all unary constraints.
- **Arc-consistency** (binary constraints): $X \rightarrow Y$ is arc-consistent if and only if every value x of X is consistent with some value y of Y.
- **Path-consistency** (n-ary constraints): generalizes arc-consistency from binary to multiple constraints.
- **Note**: It is always possible to transform all n-ary constraints into binary constraints. Often, CSPs solvers are designed to work with binary constraints.

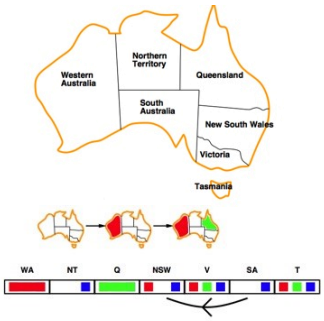
Terdapat tiga tipe konsistensi yang dapat kita gunakan pada *arc consistency test*, yaitu:

- Konsistensi node (berhubungan dengan *unary constraints*): Variabel X_i dikatakan *node-consistent* jika semua nilai Domain (X_i) memenuhi semua batasan *unary*.
- Konsistensi busur (berhubungan dengan *binary constraints*): $X \rightarrow Y$ dikatakan *arc-consistent* jika dan hanya jika setiap nilai x dari X konsisten dengan beberapa nilai y dari Y.
- Konsistensi jalur (berhubungan dengan n-*ary constraints*): menggeneralisasi *arc-consistency* dari biner ke beberapa *constraint*.

Catatan: kita dapat mengubah semua batasan n-ary menjadi *binary constraints*. Seringkali, CSP *solver* dirancang untuk bekerja dengan *binary constraints*.

Arc Consistency

- **AC**: Simplest form of propagation makes each arc consistent.
- $X \rightarrow Y$ is consistent IFF for every value x of X, there is some allowed y.



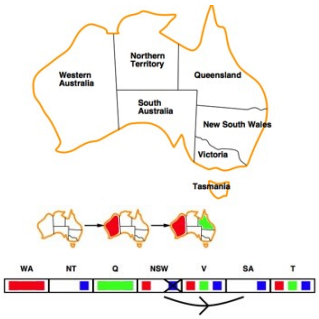
Perhatikan gambar pada *slide*. Ini merupakan kelanjutan halaman 19. Gambar ini mendemonstrasikan *arc consistency*.

Ingat daftar *constraint* pada masalah pewarnaan peta ini adalah $C = \{SA \neq WA, SA \neq NT, SA \neq Q, \boldsymbol{SA \neq NSW}, SA \neq V, WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$.

$SA \rightarrow NSW$ dikatakan *arc-consistent* karena setiap nilai pada SA (yaitu biru) konsisten dengan beberapa nilai pada NSW (yaitu merah – meski hanya satu tetap kita perhitungkan, asalkan jangan sama semua nilai di SA dengan di NSW). Perhatikan posisi "kita" ketika melakukan pemeriksaan adalah SA.

Arc Consistency

- **AC**: Simplest form of propagation makes each arc consistent.
- $X \rightarrow Y$ is consistent IFF for every value x of X, there is some allowed y.



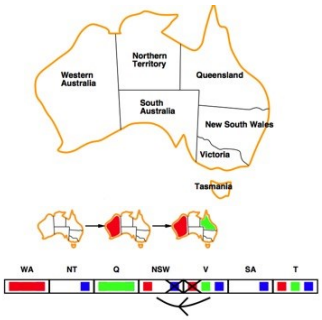
Perhatikan gambar pada *slide*. Ini merupakan kelanjutan halaman 21. Gambar ini juga masih mendemonstrasikan *arc consistency*.

Ingat daftar *constraint* pada masalah pewarnaan peta ini adalah $C = \{SA \neq WA, SA \neq NT, SA \neq Q, \boldsymbol{SA \neq NSW}, SA \neq V, WA \neq NT, NT \neq Q, Q \neq NSW, NSW \neq V\}$.

Sekarang lanjutkan pemeriksaan dengan arah panah dibalik dari $NSW \rightarrow SA$. Nah, supaya $NSW \rightarrow SA$ dikatakan *arc-consistent*, maka nilai biru pada NSW dihapus. Perhatikan bahwa dengan mengapus nilai biru pada NSW, *constraint* $\boldsymbol{SA \neq NSW}$ terpenuhi, dalam kata lain $NSW \rightarrow SA$ sudah *arc-consistent*. Perhatikan posisi "kita" ketika melakukan pemeriksaan adalah NSW.

Arc Consistency

- **AC**: Simplest form of propagation makes each arc consistent.
- $X \rightarrow Y$ is consistent IFF for every value x of X, there is some allowed y.



- If X loses a value, neighbors of X need to be rechecked

Jika X kehilangan nilainya, tetangga dari X perlu diperiksa ulang.

Perhatikan gambar pada *slide.* Ini merupakan kelanjutan halaman 22. Gambar ini juga masih mendemonstrasikan *arc consistency*.

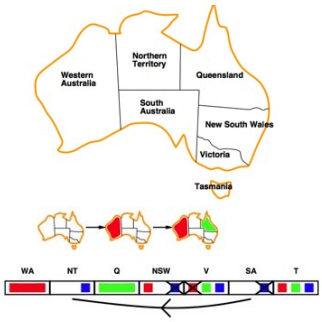
Tetangga NSW adalah Q, SA, dan V. Mari kita pilih V.

Ingat lagi daftar *constraint* pada masalah pewarnaan peta ini adalah $C = \{SA \neq WA, SA \neq NT, SA \neq Q, SA \neq NSW, SA \neq V, WA \neq NT, NT \neq Q, Q \neq NSW, \boldsymbol{NSW \neq V}\}$.

Sekarang lanjutkan pemeriksaan dengan arah panah dari $V \rightarrow NSW$. Nah, supaya $V \rightarrow NSW$ dikatakan *arc-consistent*, maka nilai merah pada NSW harus dihapus.

Arc Consistency

- **AC**: Simplest form of propagation makes each arc consistent.
- $X \rightarrow Y$ is consistent IFF for every value x of X, there is some allowed y.



- If X loses a value, neighbors of X need to be rechecked
- Arc consistency detects failure earlier than forward checking
- Can be run as a preprocessor or after each assignment

Arc consistency mendeteksi kegagalan lebih awal daripada *forward checking*. *Arc consistency* dapat dijalankan sebagai *preprocessor* atau setelah setiap *assignment*.

Perhatikan gambar pada *slide*. Gambar ini juga masih mendemonstrasikan *arc consistency*.

Supaya $SA \rightarrow NT$ dikatakan *arc-consistent*, maka nilai biru pada SA harus dihapus.

Arc Consistency Algorithm AC-3

Algorithm that makes a CSP arc-consistent!

```
function AC-3( csp)
returns False if an inconsistency is found, True otherwise
inputs: csp, a binary CSP with components (X, D, C)
local variables: queue, a queue of arcs, initially all the arcs in csp
while queue is not empty do
    (X_i, X_j) = REMOVE-FIRST(queue)
    if REVISE(csp, X_i, X_j)then
        if size of D_i = 0 then return False
        for each X_k in X_i.NEIGHBORS − {X_j} do
            add (X_k, X_i) to queue
return true

function REVISE( csp, X_i, X_j)
returns True iff we revise the domain of X_i
revised = False
for each x in D_i do
    if no value y in D_j allows (x, y) to satisfy the constraint between X_i and X_j then
        delete x from D_i
        revised = True
return revised
```


Pada slide adalah pseudocode Arc Consistency Algorithm AC-3.

Complexity of AC-3

- Let n be the number of variables, and d be the domain size.
- If every node (variable) is connected to the rest of the variables, then we have $n * (n-1)$ arcs (constraints) $\rightarrow O(n^2)$
- Each arc can be inserted in the queue d times $O(d)$
- Checking the consistency of an arc costs $\rightarrow O(d^2)$.
- Overall complexity is $O(n^2 d^2)$

Misalkan n adalah banyaknya variabel, dan d adalah ukuran domain.
Jika setiap simpul (variabel) terhubung ke variabel lainnya, maka kita memiliki $n * (n-1)$ busur (*constraint*) $\rightarrow O(n^2)$
Setiap busur dapat dimasukkan dalam antrian d kali $O(d)$
Memeriksa konsistensi *cost* suatu busur $\rightarrow O(d^2)$.
Kompleksitas keseluruhan adalah $O(n^2 d^2)$

Solving CSPs

IMPORTANT

- State-space search algorithms: search!
- CSP Algorithms: Algorithm can do two things:
  - Search: choose a new variable assignment from many possibilities
  - Inference: constraint propagation, use the constraints to spread the word: reduce the number of values for a variable which will reduce the legal values of other variables etc.
- As a preprocessing step, constraint propagation can sometimes solve the problem entirely without search.
- Constraint propagation can be intertwined with search.

Algoritma pencarian **state-space** adalah pencarian.
Algoritma pada CSP dapat melakukan dua hal:

- **Pencarian**: pilih *variable assignment* baru dari banyak kemungkinan
- **Inferensi**: *constraint propagation*, gunakan *constraint* untuk menyebarkan nilai: kurangi jumlah nilai untuk variabel yang akan mengurangi nilai hukum variabel lain, dll.

Sebagai langkah *preprocessing*, *constraint propagation* terkadang dapat menyelesaikan masalah sepenuhnya tanpa pencarian.
Constraint propagation dapat terjalin dengan pencarian.

Solving CSPs

- **BFS:** Develop the complete tree
- **DFS:** Fine but time consuming
- **BTS: Backtracking search** is the basic uninformed search for CSPs. It's a DFS s.t.
  - Assign one variable at a time: assignments are commutative. e.g., (WA=red, NT=green) is same as (NT=green, WA=red)
  - Check constraints on the go: consider values that do not conflict with previous assignments.

Untuk menyelesaikan CSP, kita dapat menggunakan

- BFS: Kembangkan pohon lengkap
- DFS: Bagus tapi memakan waktu
- BTS: Pencarian backtracking adalah *basic uninformed search* untuk CSP. Ini termasuk DFS s.t.
  - Tetapkan satu variabel pada satu waktu: *assignment* bersifat komutatif. misalnya, (WA = merah, NT = hijau) sama dengan (NT = hijau, WA = merah)
  - Periksa *constraint* saat dalam pengeksekusian algoritma: pertimbangkan nilai yang tidak bertentangan dengan *assignment* nilai sebelumnya.

Solving CSPs

- **Initial state**: empty assignment {}
- **States**: are partial assignments
- **Successor function (operator)**: assign a value to an unassigned variable
- **Goal test**: the current assignment is complete and satisfies all constraints

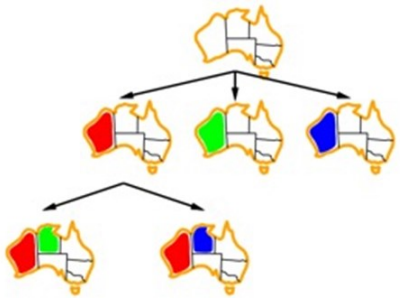
Keadaan awal: *empty assignment* {}
State: adalah *partial assignments*
Fungsi penerus (operator): menetapkan nilai ke *unassigned variable*
Tes tujuan: *assignment* saat ini selesai dan memenuhi semua *constraint*

Backtracking Search Example

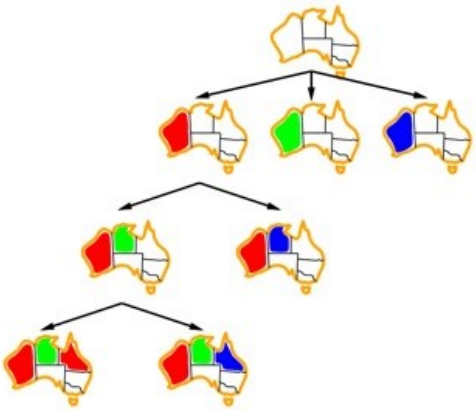
Berikut ilustrasi *backtracking search*.

Backtracking Search Example

Backtracking Search Example



Backtracking Search Example



Backtracking Search

```
function BACKTRACKING_SEARCH(csp) returns a solution, or failure
    return BACKTRACK({}, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
    var = SELECT_UNASSIGNED-VARIABLE(csp)
    for each value in ORDER_DOMAIN_VALUES (var, assignment, csp)
        if value is consistent with assignment then
            add {var = value} to assignment
            result = BACKTRACK(assignment, csp)
            if result ≠ failure then return result
        remove {var = value} from assignment
    return failure
```


Berikut pseudocode *backtracking search*.

Improving Backtracking Search Efficiency

Heuristics are back!

- **General-purpose** methods can give huge speed gains:
  1. Which variable should be assigned next?
  2. In what order should its values be tried?
  3. Can we detect inevitable failure early?

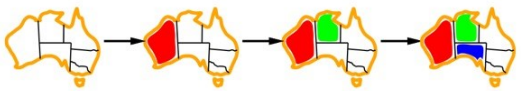
Kita dapat meningkatkan efisiensi *backtracking search* dengan menggunakan heuristik seperti:

1. Variabel mana yang nilainya harus di-*assign* selanjutnya?
2. Dalam urutan apa nilai-nilainya harus dicoba?
3. Bisakah kita mendeteksi kegagalan yang tak terelakkan sejak dini?

Minimum Remaining Values

1. Which variable should be assigned next?

- A 

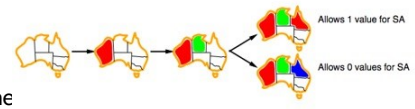
able with th 

Pick the hardest!

MRV: Pilih variabel dengan nilai legal paling sedikit di domainnya

Least Constraining Value

2. In what order should its values be tried?



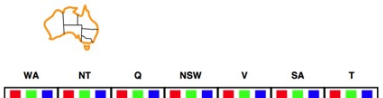
- L , choose the ut the
  f emaining variables.

Pick the ones that are likely to work!

LCV: Diberikan variabel, pilih nilai yang paling sedikit membatasi: nilai yang mengesampingkan nilai paling sedikit di variabel yang tersisa.

Forward Checking

3. Can we detect inevitable failure early?

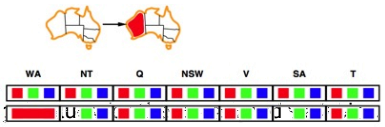


- F… …aining legal values for the unassigned variables. Terminate when any v… …values.

FC: Lacak nilai legal yang tersisa untuk variabel yang belum ditetapkan. Hentikan jika variabel apa pun tidak memiliki nilai legal.

Forward Checking

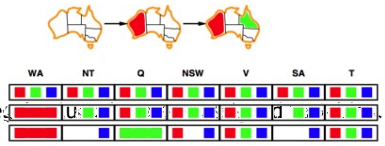
3. Can we detect inevitable failure early?



- F … haining le… Terminate when any v… values.

Forward Checking

3. Can we detect inevitable failure early?

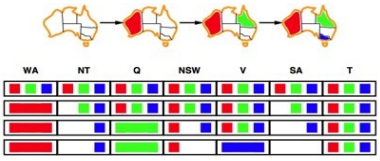


- F… …aining le… …. Terminate when any v… …values.

Forward Checking

3. Can we detect inevitable failure early?

- F … haining le … values.



Terminate when any

Solving CSPs: Sudoku

- All 3x3 boxes, rows, columns, must contain all digits 1…9.

8		9	5		1	7	3	6
2		7		6	3			
1	6							
				9		4		7
	9		3		7		2	
7		6		8				
							6	3
			9	3		5		2
5	3	2	6		4	8		9

Solving CSPs: Sudoku

- All 3x3 boxes, rows, columns, must contain all digits 1…9.
  - **Variables**: $V = \{A_1, \dots, A_9, B_1, \dots, B_9, \dots, I_1, \dots, I_9\}$, $|V| = 81$.
  - **Domain**: $D = \{1, 2, \dots, 9\}$, the filled squares have a single value.
  - **Constraints**: 27 constraints
    - $\text{Alldiff}(A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9)$
    - . . .
    - $\text{Alldiff}(A_1, B_1, C_1, D_1, E_1, F_1, G_1, H_1, I_1)$
    - . . .
    - $\text{Alldiff}(A_1, A_2, A_3, B_1, B_2, B_3, C_1, C_2, C_3)$

8		9	5		1	7	3	6
2		7		6	3			
1	6							
				9		4		7
	9		3		7		2	
7		6		8				
							6	3
			9	3		5		2
5	3	2	6		4	8		9

Berikut adalah contoh CSP pada Sudoku.

Solving CSPs: Sudoku

- All 3x3 boxes, rows, columns, must contain all digits 1…9.
- More Complex Inference Strategies
  - Naked doubles (triples): find two (three) cells in a 3×3 grid that have only the same candidates left, eliminate these two (three) values from all possible assignments in that box.
  - Locked pair, Locked triples, etc.

8		9	5		1	7	3	6
2		7		6	3			
1	6							
				9		4		7
	9		3		7		2	
7		6		8				
							6	3
			9	3		5		2
5	3	2	6		4	8		9

Strategi Inferensi yang Lebih Kompleks juga dapat digunakan untuk menyelesaikannya.

Solving CSPs: Sudoku

- All 3x3 boxes, rows, columns, must contain all digits 1…9.

8		9	5		1	7	3	6
2		7		6	3			
1	6							
				9		4		7
	9		3		7		2	
7		6		8				
							6	3
			9	3		5		2
5	3	2	6		4	8		9

8	4	9	5	2	1	7	3	6
2	5	7	8	6	3	9	1	4
1	6	3	7	4	9	2	5	8
3	2	5	1	9	6	4	8	7
4	9	8	3	5	7	6	2	1
7	1	6	4	8	2	3	9	5
9	8	4	2	7	5	1	6	3
6	7	1	9	3	8	5	4	2
5	3	2	6	1	4	8	7	9

Berikut adalah contoh solusinya.

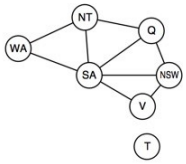
Backtracking w/ Inference

```
function BACKTRACKING_SEARCH(csp) returns a solution, or failure
    return BACKTRACK({}, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
    if assignment is complete then return assignment
    var = SELECT_UNASSIGNED-VARIABLE(csp)
    for each value in ORDER_DOMAIN_VALUES (var, assignment, csp)
        if value is consistent with assignment then
            add {var = value} to assignment
            inferences = INFERENCE(csp, var, value)
            if inferences ≠ failure then
                add inferences to assignment
                result = BACKTRACK(assignment, csp)
                if result ≠ failure then return result
        remove {var = value} and inferences from assignment
    return failure
```


Berikut adalah pseudocode Backtracking w/ Inference

Problem Structure



- Idea: Leverage the problem structure to make the search more efficient.
- Example: Tasmania is an independent problem.
- Identify the connected component of a graph constraint.
- Work on independent subproblems.

Ide: Manfaatkan struktur masalah untuk membuat pencarian lebih efisien.
Contoh: Tasmania adalah masalah independen.
Identifikasi komponen yang terhubung dari batasan grafik.
Kerjakan subproblem independen.

Problem Structure

- **Complexity**:
  - Let d be the size of the domain and n be the number of variables.
  - Time complexity for BTS is $O(d^n)$.
  - Suppose we decompose into subproblems, with c variables per subproblem.
  - Then we have $\frac{n}{c}$ subproblems.
  - c variables per subproblem takes $O(d^c)$.
  - The total for all subproblems takes $O\left(\frac{n}{c} d^c\right)$ in the worst case.

Perhatikan bahwa kompleksitas berkutang dengan menstrukturkan masalah sebelum menerapkan strategi penyelesaian CSP.

Problem Structure

- **Example**:
  - Assume $n = 80$, $d = 2$.
  - Assume we can decompose into 4 subproblems with $c = 20$.
  - Assume processing at 10 million nodes per second.
  - Without decomposition of the problem we need:
    $$2^{80} = 1.2 \times 10^{24}$$
    3.83 million years!
  - With decomposition of the problem we need:
    $$4 \times 2^{20} = 4.2 \times 10^{6}$$
    0.4 seconds!

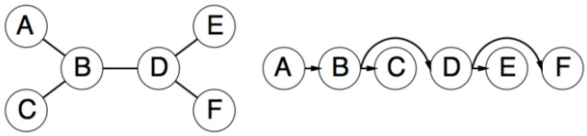
Berikut ini perbandingan waktu penyelesaian CSP tanpa dekomposisi masalah dan dengan dekomposisi masalah.

Problem Structure

- Turning a problem into independent subproblems is not always possible.
- Can we leverage other graph structures?
- Yes, if the graph is tree-structured or nearly tree-structured.
- A graph is a tree if any two variables are connected by only one path.
- Idea: use DAC, Directed Arc Consistency
- A CSP is said to be directed arc-consistent under an ordering $X_1, X_2, \ldots, X_n$ IFF every X_i is arc-consistent with each X_j for $j > i$.

Mengubah masalah menjadi subproblem independen tidak selalu memungkinkan.
Bisakah kita memanfaatkan struktur grafik lainnya?
Ya, jika grafiknya berstruktur pohon atau hampir berstruktur pohon.
Grafik adalah pohon jika ada dua variabel yang dihubungkan hanya oleh satu jalur.
Ide: gunakan DAC, Directed Arc Consistency

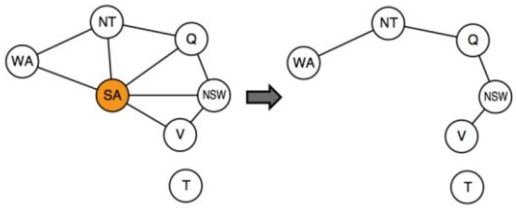
Problem Structure



- First pick a variable to the be the root.
- Do a topological sorting: choose an ordering of the variables s.t. each variable appears after its parent in the tree.
- For n nodes, we have $n - 1$ edges.
- Make the tree directed arc-consistent takes $O(n)$
- Each consistency check takes up to $O(d^2)$ (compare d possible values for 2 variables).
- The CSP can be solved in $O(nd^2)$

Berikut adalah DAC.

Nearly Tree-structured CSPs



- Assign a variable or a set of variables and prune all the neighbors domains.
- This will turn the constraint graph into a tree :)
- There are other tricks to explore, have fun.

Berikut adalah contoh Nearly Tree-structured CSPs. Ada trik lain untuk dijelajahi, silahkan eksplorasi sendiri.

Summary

- CSPs are a special kind of search problems:
  - states defined by values of a fixed set of variables
  - goal test defined by constraints on variable values
- Backtracking = depth-first search with one variable assigned per node
- Variable ordering and value selection heuristics help
- Forward checking prevents assignments that guarantee later failure

Summary

- Constraint propagation (e.g., arc consistency) is an important mechanism in CSPs.
- It does additional work to constrain values and detect inconsistencies.
- Tree-structured CSPs can be solved in linear time
- Further exploration: How can local search be used for CSPs?
- The power of CSPs: domain-independent, that is you only need to define the problem and then use a solver that implements CSPs mechanisms.
- Play with CSP solver? Try http://aispace.org/constraint/.

References

- S. J. Russell and P. Borvig. (2020). Artificial Intelligence: A Modern Approach (4th Edition), Prentice Hall International.
  - Chapter 6
- Constraint Satisfaction Problems, 6.034 - Artificial Intelligence, Massachusetts Institute of Technology, 2012 (https://courses.csail.mit.edu/6.034s/handouts/spring12/csp.pdf)
- Constraint Satisfaction Problems by Francisco Iacobelli (https://www.youtube.com/watch?v=lCrHYT_EhDs)

eof