Local Search:
Goal Versus Path Search

CS3243: Introduction to Artificial Intelligence – Lecture 5a

6 February 2023

Contents

Reference: AIMA 4th Edition, Section 4.1 & Section 5.1

Administrative Matters

Midterm Examination

- Schedule
  - Week 7 Lecture Slot
  - Monday (27 FEB), 1030-1130 hrs (Arrive by 1010 hrs)
- Venue
  - MPSH1a (Conducted in-person)
- Format
  - Duration = 1 hour
  - Total = 30 marks
  - Closed-book + Cheat Sheet (1 × Double-sided A4 Sheet)
  - Lectures 1-5 (i.e., everything up to and including this lecture)
- Practice Papers
  - Canvas > CS3244 > Files > Past Papers

Consultations

- Project 1
  - Consultation recording → Canvas
  - Important notes on grid representation → Canvas
  - For more support → Message TA
  - Last resort → Email me (dler@comp.nus.edu.sg)

- Midterm
  - Review past midterm papers
  - Message TAs for clarifications

Upcoming…

- Deadlines
  - TA3 (released last week)
    - *Due in your Week 5 tutorial session*
    - *Submit the a physical copy (more instructions on the Tutorial Worksheet)*
  - Prepare for the tutorial!
    - Participation marks = 5%

  - Project 1
    - *Due next Sunday (19 February), 2359 hrs*

Goal Versus Path Search

Slightly Different Problems

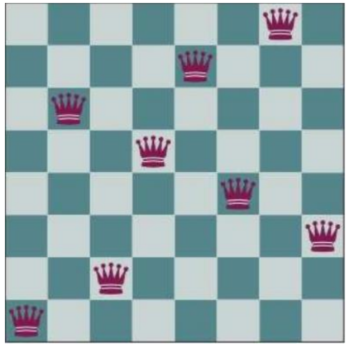
- Thus far: finding a path to a goal
  - Algorithms track paths
  - Systematically search paths
- What if only interested in goal state?
  - Have goal test, but not values to satisfy it
  - Only want goal state values

  - Optimisation problems
    - Vertex cover problems
    - Boolean satisfiability problems (SAT)
    - Travelling salesman problem
    - Timetabling / scheduling problems

- Sudoku

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

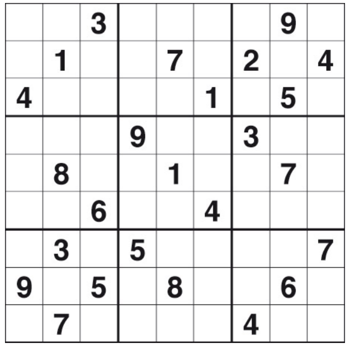
- n-queens



Path Versus Goal

- Search problems – path planning
  - Path to a goal necessary
  - Path cost is important
- Local search – goal determination
  - Abandon systematic search – ignore path (and path cost)
  - Maintain “best” successor state – greedy search
- Advantages
  - Only store current and immediate successor states
    - Space complexity: O(b)
      - Note that space complexity may be reduced to O(1) if successors may be processed one at a time
  - Applicable to very large or infinite search spaces

Path planning can satisfy the objective of goal search but does more than it needs to since we don’t need the path



Local Search is incomplete

Local Search via Hill-Climbing

Hill-Climbing Algorithm

```
current = initial_state
while true:
      neighbour = highest_valued_successor(current)
      if value(neighbour) ≤ value(current): return current
      current = neighbour
```

- How it works (steepest ascent – greedy strategy)
  - Starts with a *random* initial state (typically) – more on this later
  - Only store the current state
  - In each iteration, find a successor that ***improves*** on current state
    - Requires **`actions`** and **`transition`** to determine successors
    - Requires **`value`**; a way to value each state – e.g., f(n) = -h(n)
  - If none exists, return current state as the best option
    - This algorithm *can fail*; may return a non-goal state

Requires heuristic (similar to informed search heuristic)

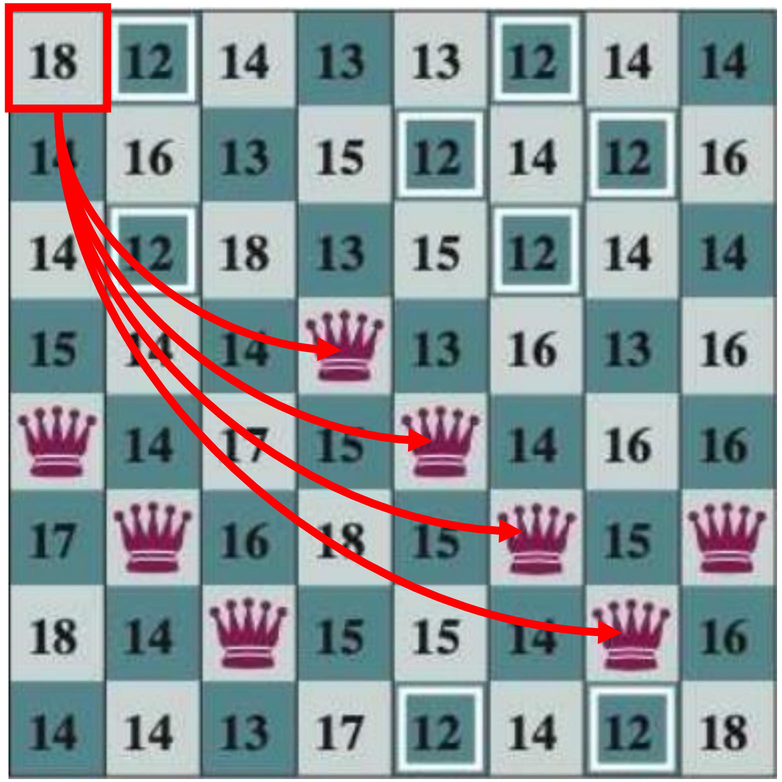
8-Queens Example

- Given an 8×8 chess board
  - Place 8 queens
  - No queen must threaten another
  - h: *# pairs of queens threatening each other*
- Search problem
  - State: 1 queen per column
  - Action: move 1 queen to different col. position
  - Goal: 0 pairs threatening
- Example h
  - Consider top-most left-most cell (h-value is 18)

8-Queens Example

- Given an 8×8 chess board
  - Place 8 queens
  - No queen must threaten another
  - h: *# pairs of queens threatening each other*
- Search problem
  - State: 1 queen per column
  - Action: move 1 queen to different col. position
  - Goal: 0 pairs threatening
- Example h
  - Consider top-most left-most cell (h-value is 18)

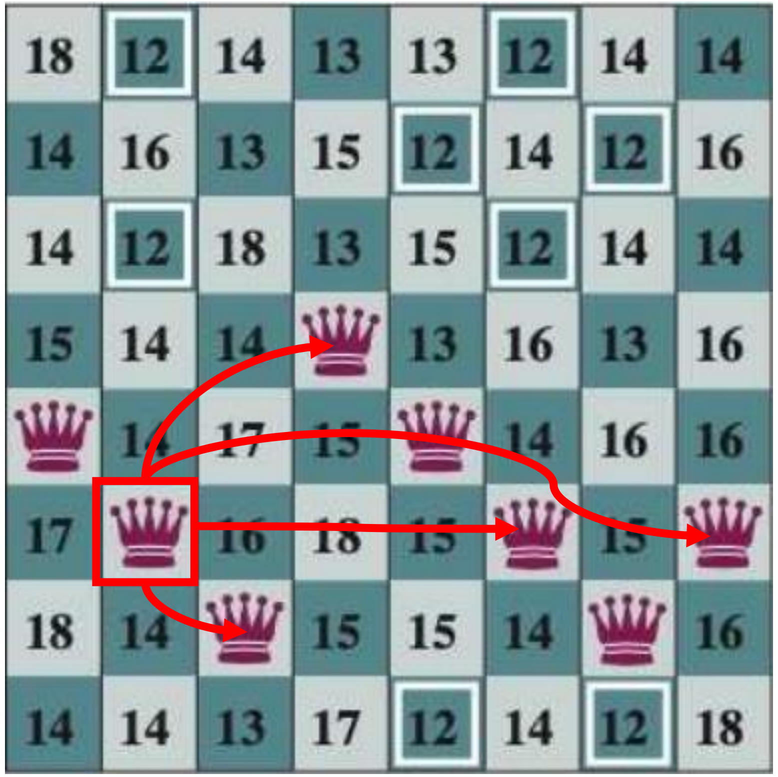
C1 (now in top-most left-most call) attacks C4, C5, C6, C7 [4]



8-Queens Example

- Given an 8×8 chess board
  - Place 8 queens
  - No queen must threaten another
  - h: # *pairs of queens threatening each other*
- Search problem
  - State: 1 queen per column
  - Action: move 1 queen to different col. position
  - Goal: 0 pairs threatening
- Example h
  - Consider top-most left-most cell (h-value is 18)

    C1 (now in top-most left-most call) attacks C4, C5, C6, C7 [4]



C2 attacks C3, C4, C6, C8 [4]

8-Queens Example

- Given an 8×8 chess board
  - Place 8 queens
  - No queen must threaten another
  - h: *# pairs of queens threatening each other*
- Search problem
  - State: 1 queen per column
  - Action: move 1 queen to different col. position
  - Goal: 0 pairs threatening
- Example h
  - Consider top-most left-most cell (h-value is 18)

C1 (now in top-most left-most call) attacks C4, C5, C6, C7 [4]

C2 attacks C3, C4, C6, C8 [4]

C3 attacks C5, C7 [2]

8-Queens Example

- Given an 8×8 chess board
  - Place 8 queens
  - No queen must threaten another
  - h: *# pairs of queens threatening each other*
- Search problem
  - State: 1 queen per column
  - Action: move 1 queen to different col. position
  - Goal: 0 pairs threatening
- Example h
  - Consider top-most left-most cell (h-value is 18)

C1 (now in top-most left-most call) attacks C4, C5, C6, C7 [4]

C2 attacks C3, C4, C6, C8 [4]

C3 attacks C5, C7 [2]

C4 attacks C5, C6, C7 [3]

8-Queens Example

- Given an 8×8 chess board
  - Place 8 queens
  - No queen must threaten another
  - h: *# pairs of queens threatening each other*
- Search problem
  - State: 1 queen per column
  - Action: move 1 queen to different col. position
  - Goal: 0 pairs threatening
- Example h
  - Consider top-most left-most cell (h-value is 18)

C1 (now in top-most left-most call) attacks C4, C5, C6, C7 [4]

C2 attacks C3, C4, C6, C8 [4]

C3 attacks C5, C7 [2]

C4 attacks C5, C6, C7 [3]

$\boxed{\text{C5 attacks C6, C7 [2]}}$

8-Queens Example

- Given an 8×8 chess board
  - Place 8 queens
  - No queen must threaten another
  - h: *# pairs of queens threatening each other*
- Search problem
  - State: 1 queen per column
  - Action: move 1 queen to different col. position
  - Goal: 0 pairs threatening
- Example h
  - Consider top-most left-most cell (h-value is 18)

C1 (now in top-most left-most call) attacks C4, C5, C6, C7 [4]

C2 attacks C3, C4, C6, C8 [4]

C3 attacks C5, C7 [2]

C4 attacks C5, C6, C7 [3]

C5 attacks C6, C7 [2]

C6 attacks C7, C8 [2]

8-Queens Example

- Given an 8×8 chess board
  - Place 8 queens
  - No queen must threaten another
  - h: *# pairs of queens threatening each other*
- Search problem
  - State: 1 queen per column
  - Action: move 1 queen to different col. position
  - Goal: 0 pairs threatening
- Example h
  - Consider top-most left-most cell (h-value is 18)

C1 (now in top-most left-most call) attacks C4, C5, C6, C7 [4]   C2 attacks C3, C4, C6, C8 [4]   C3 attacks C5, C7 [2]

C4 attacks C5, C6, C7 [3]   C5 attacks C6, C7 [2]   C6 attacks C7, C8 [2]   C7 attacks C8 [1]

Complete-State Formulations

- States in the 8-Queens search problem have all 8 queens present
- Every state has all components of a solution
  - No partially completed states
  - All actions perturb current state by 1 move
- Each state is a potential solution
  - Apt for problems where path is not important
    - Simply “guess” a solution
    - “Check” its value
    - Make a “systemic guess” by moving to states of higher value (e.g., via f(n) = -h(n))
      - Assumes that states with higher f values are closer to the goal (i.e., more likely to reach a goal)
- Most local search problems may be formulated in this manner

Practically, it is fine to use f(n) = h(n) and seek a local minima as well. In such cases, we simply replace the ≤ in the algorithm with ≥.

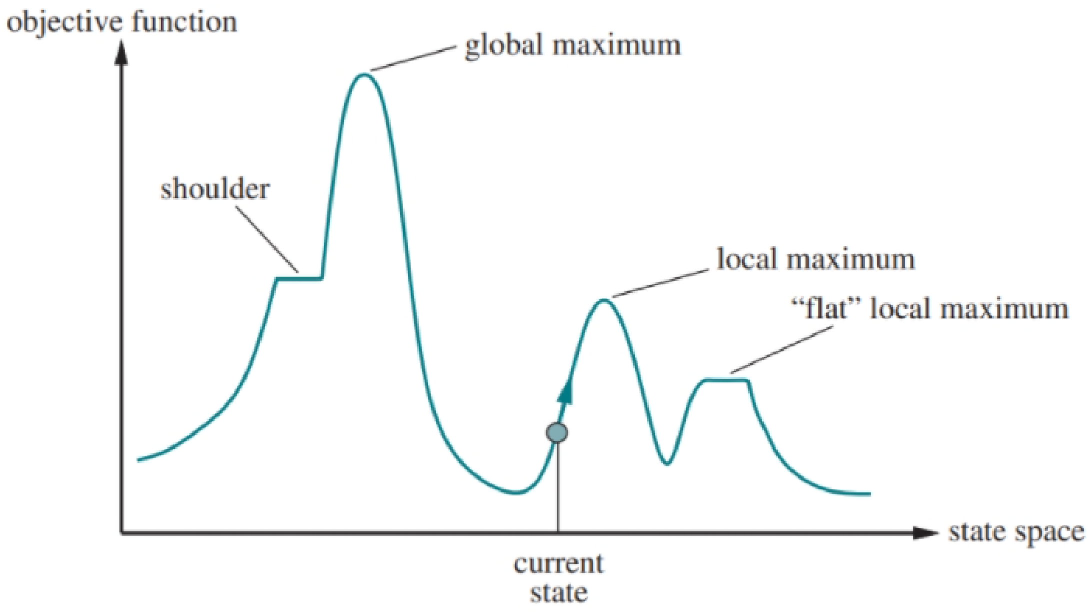
Hill-Climbing Algorithm (Revisited)

```
current = initial_state
while true:
      neighbour = highest_valued_successor(current)
      if value(neighbour) ≤ value(current): return current
      current = neighbour
```

- NOT guaranteed to find a goal!
  - **value** defined by informed search heuristic, h; e.g., f(n) = -h(n)
  - Goal → h(n) = 0
- What happens if the returned state is not a goal state?
- When does this happen?

Issues & the Potential for Failure

- Hill-climbing may not return a solution



- May get stuck at
  - Local Maxima
  - Shoulder or Plateau
  - Ridge (sequence of local maxima)
- Require strategies to counter these problems

Hill-Climbing Variants

- Stochastic hill climbing
  - Changes `highest_valued_successor(..)`
  - Chooses randomly among states with values better than current
  - May take longer to find a solution but sometimes leads to better solutions

- First-choice hill climbing
  - Changes `highest_valued_successor(..)`
  - Handles high b by randomly generating successors until one with better value than current is found (instead of generating all possible successors)

Hill-Climbing Variants

- Sideways move
  - Replaces ≤ with <; allows continuation when `value(neighbour) == value(current)`
  - Can traverse shoulders / plateaus

- Random-restart hill climbing
  - Different algorithm
  - Adds an outer loop which randomly picks a new starting state
  - Keeps attempting random restarts until a solution is found

Random Restarts Hill-Climbing Algorithm

```
current = NULL
while current is NULL or not isGoal(current):
      current = random_initial_state()
      while true:
            neighbour = highest_valued_successor(current)
            if value(neighbour) < value(current):
                  break
            current = neighbour
return current
```

- Changes from the Hill-Climbing Algorithm
  - Utilises **`isGoal`**; if goal not found then loops with a random restart
  - Requires function to generate random initial state: ***`random_initial_state`***`()`
  - Considers sideways moves since it utilises < instead of ≤

Back to 8-Queens: Analysis

- Hill climbing (via steepest-ascent) with random restarts
  - Solution: p_1 = 14% (expected solution in 4 steps; expected failure in 3 steps)
  - Expected computation = $1\times$(steps for success) + $((1 - p_1) / p_1)\times$(steps for failure)
    = $1\times(4)$ + $(0.86/0.14) \times(3)$
    = 22.428571428571427 steps

(1 - p_1) / p_1) determines the expected number of failed attempts

- Adding sideways moves
  - Solution: p_2 = 94% (expected solution in 21 steps; expected failure in 64 steps)
  - Expected computation = $1\times$(steps for success) + $((1 - p_1) / p_1)\times$(steps for failure)
    = $1\times(21)$ + $(0.06/0.94) \times(64)$
    = 25.085106382978722 steps

- 8-Queens possible states = 8^8 = 16777216

Extremely efficient for such a large space

Expected values taken from AIMA pp. 131

Local Beam Search

Local Beam Search

- Store k states instead of 1
  - Hill climbing just stores the current state
  - Beam (window) stores k
- Algorithm
  - Begins with k random starts
  - Each iteration generate successors for all k states
  - Repeat with best k among ALL generated successors unless goal found
- Better than k parallel random restarts
  - Since best k among ALL successors taken (not best from each set of successors, k times)
- Stochastic beam search
  - Original variant may still get stuck in a local cluster
  - Adopt stochastic strategy similar to stochastic hill climbing to increase state diversity

Questions about the Lecture?

- Was anything unclear?
- Do you need to clarify anything?

- Ask on Archipelago
  - Specify a question
  - Upvote someone else's question

Invitation Link (Use NUS Email --- starts with E)
https://archipelago.rocks/app/resend-invite/12384352999

Constraint Satisfaction Problems: Generalising Goal Search I

CS3243: Introduction to Artificial Intelligence – Lecture 5b

6 February 2023

Systematic Goal Search

- With local search we apply greedy search strategies
  - Are there more ***systematic*** search strategies applicable?
- Issues with systematic searching
  - Systematic approaches tend to be computationally expensive
    - Incorporating domain knowledge via heuristics helped direct the search such that less was searched
    - Need to reduce the search space to make a systematic search more viable
- A general solution
  - Use a factored representation for each state
    - State: set of variables $\boldsymbol{X} = \{x_1, ..., x_n\}$, where each variable x_i has a domain $D_i = \{d_1, ..., d_m\}$
  - Divide the goal test into a set of constraints
    - If a state satisfies all constraints, it is a goal state
  - Constraint satisfaction problem (CSP)
    - Any state that does not satisfy a constraint should not be further explored

CSPs systematically search for goal states by pruning invalid subtrees as early as possible

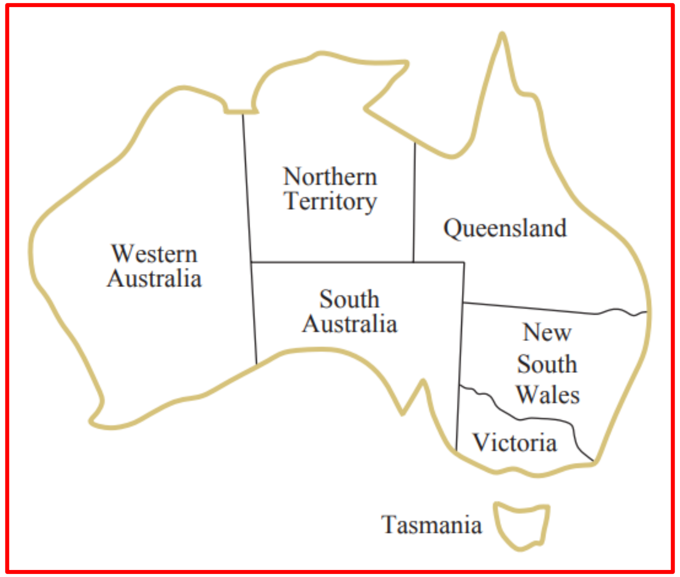
CSP Formulation

Formulating CSPs

- State representation
  - Variables: $X = \{x_1, ..., x_n\}$
  - Domains: $D = \{d_1, ..., d_k\}$
    - Such that x_i has a domain d_i
  - Initial state: all variables unassigned
  - Intermediate state: partial assignment
- Actions, costs and transition
  - Assignment of values (within domain) to variables
  - Costs are not utilised
- Goal test
  - Constraints: $C = \{c_1, ..., c_m\}$
    - Defined via a constraint language
      - Algebra, Logic, Sets
    - Each c_i corresponds to a requirement on some subset of X
  - Objective is a ***complete*** and ***consistent*** assignment
    - Find a legal assignment $(y_1, ..., y_n)$
      - $y_i \in d_i$ for all $i \in [n]$
    - Complete: all variables assigned values
    - Consistent: all constraints C satisfied

CSP Formulation Example 1: Graph Colouring

- Colour each state of Australia such that no two adjacent states share the same colour
- Variables
  - $X = \{ WA, NT, Q, NSW, V, SA, T \}$
- Domains
  - $d_i = \{ \text{Red, Green, Blue} \}$
- Constraints
  - $\forall (x_i, x_j) \in E$, $\mathbf{colour}(x_i) \neq \mathbf{colour}(x_j)$



CSP Formulation Example 2: Cryptarithmetic Puzzle

- Given that each letter represents a digit, determine the letter-digit mapping that solves the given sum

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

- Variables
  - $X = \{ T, W, O, F, U, R, B_1, B_2, B_3 \}$
  - Where B_1, B_2, B_3 are carry bits for ($2O$, $2W$, $2T$ respectively)
- Domains
  - $d_i = \{ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 \}$
  - Strictly, B_1, B_2, B_3 should have domain {0, 1}

- Constraints
  - **`alldiff`**(T, W, O, F, U, R)
  - $O + O = R + 10.B_1$
  - $B_1 + W + W = U + 10.B_2$
  - $B_2 + T + T = O + 10.B_3$
  - $B_3 = F$
  - $T, F \neq 0$

CSP Formulation Example 3: Sudoku

- Variables
  - $X = \{ A_1, \ldots, A_9, \ldots, I_1, \ldots, I_9 \}$
  - 81 variables
- Domains
  - $d_i = \{ 1, 2, 3, 4, 5, 6, 7, 8, 9 \}$
- Constraints
  - **`alldiff`**(...)
    - 27 cases
      - 9 columns
      - 9 rows
      - 9 boxes

	1	2	3	4	5	6	7	8	9
A			3		2		6		
B	9			3		5			1
C			1	8		6	4		
D			8	1		2	9		
E	7								8
F			6	7		8	2		
G			2	6		9	5		
H	8			2		3			9
I			5		1		3		

	1	2	3	4	5	6	7	8	9
A	4	8	3	9	2	1	6	5	7
B	9	6	7	3	4	5	8	2	1
C	2	5	1	8	7	6	4	9	3
D	5	4	8	1	3	2	9	7	6
E	7	2	9	5	6	4	1	3	8
F	1	3	6	7	9	8	2	4	5
G	3	7	2	6	8	9	5	1	4
H	8	1	4	2	5	3	7	6	9
I	6	9	5	4	1	7	3	8	2

Variable Domain Types & Constraint Types

- Variable domain types
  - Continuous
  - Discrete
  - Finite
  - Infinite

  - Continuous and Infinite
    - Real values
  - Discrete and Infinite
    - All integers
  - Discrete and finite
    - Sudoku

CS3243 focuses on discrete, finite domains

- Constraint types
  - Linear
  - Nonlinear

Continuous domain and linear constraints → linear programming

Not covered in CS3243

More on Constraints

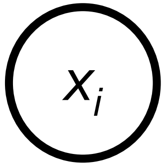
- A language is necessary to express the constraints
  - Arithmetic
  - Sets (of legal values)
  - Logic
  - For example, x_1 greater than x_2 given d = {1, 2, 3} may be written
    - $\langle (x_1, x_2), x_1 > x_2 \rangle$
    - $\langle (x_1, x_2), \{ (2, 1), (3, 1), (3, 2) \} \rangle$

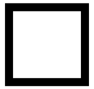
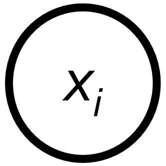
- Each constraint, c_i,
  - Describes the necessary relationship, ***rel***, between a set of variables, ***scope***
    - For the example above, ***scope*** = (x_1, x_2). ***rel*** = $x_1 > x_2$

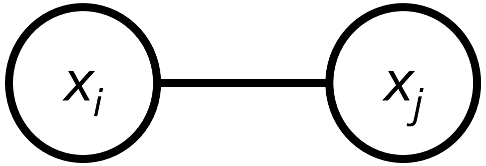
- Types of constraints
  - Unary: | ***scope*** | = 1
  - Binary: | ***scope*** | = 2
  - Global: | ***scope*** | > 2 (i.e., higher-order constraints)

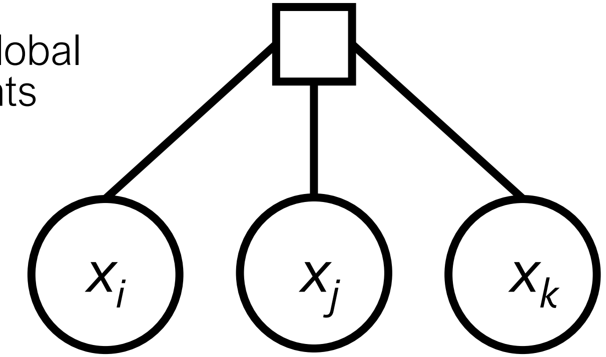
Constraint Graphs

Drawing Constraint Graphs and Hypergraphs

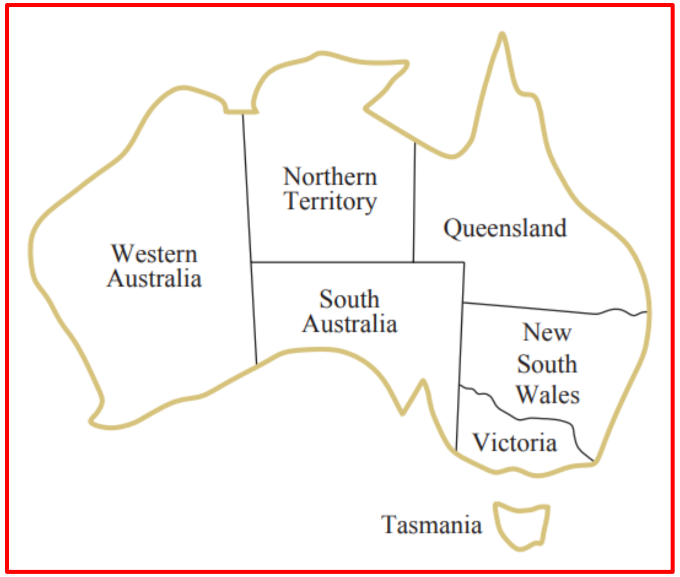
- Constraint graphs represent the constraints in a CSP
  - Simple Vertex: variable 

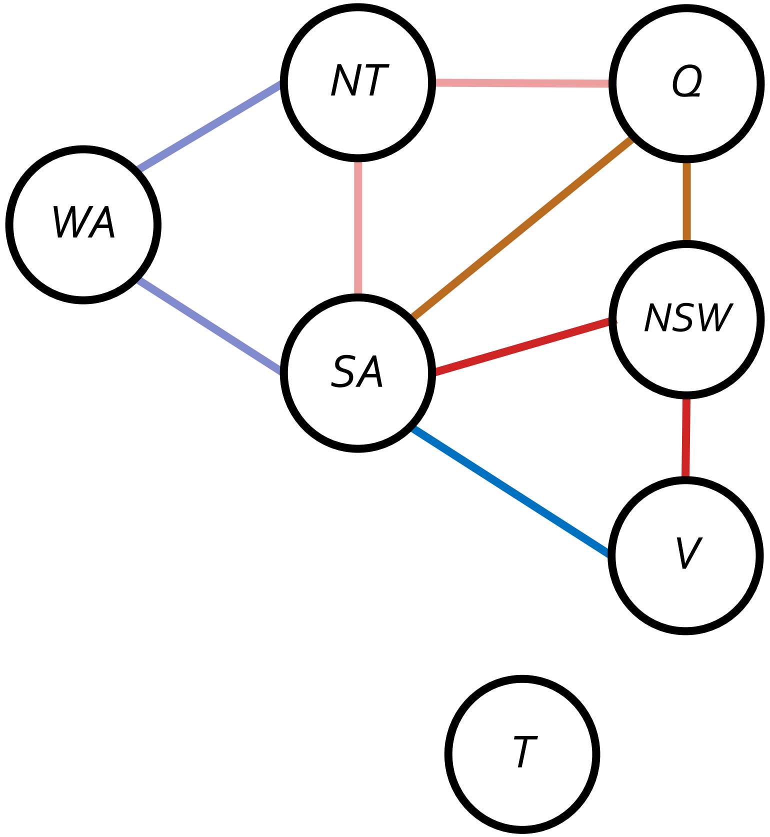
  - Linking Vertex: for global constraints 
  - Edge: links all variables in the scope of a constraint (*rel*)
    - Unary constraints 

    - Binary constraints 

    - Binary/Global constraints 

Constraint Graph for Example 1: Graph Colouring

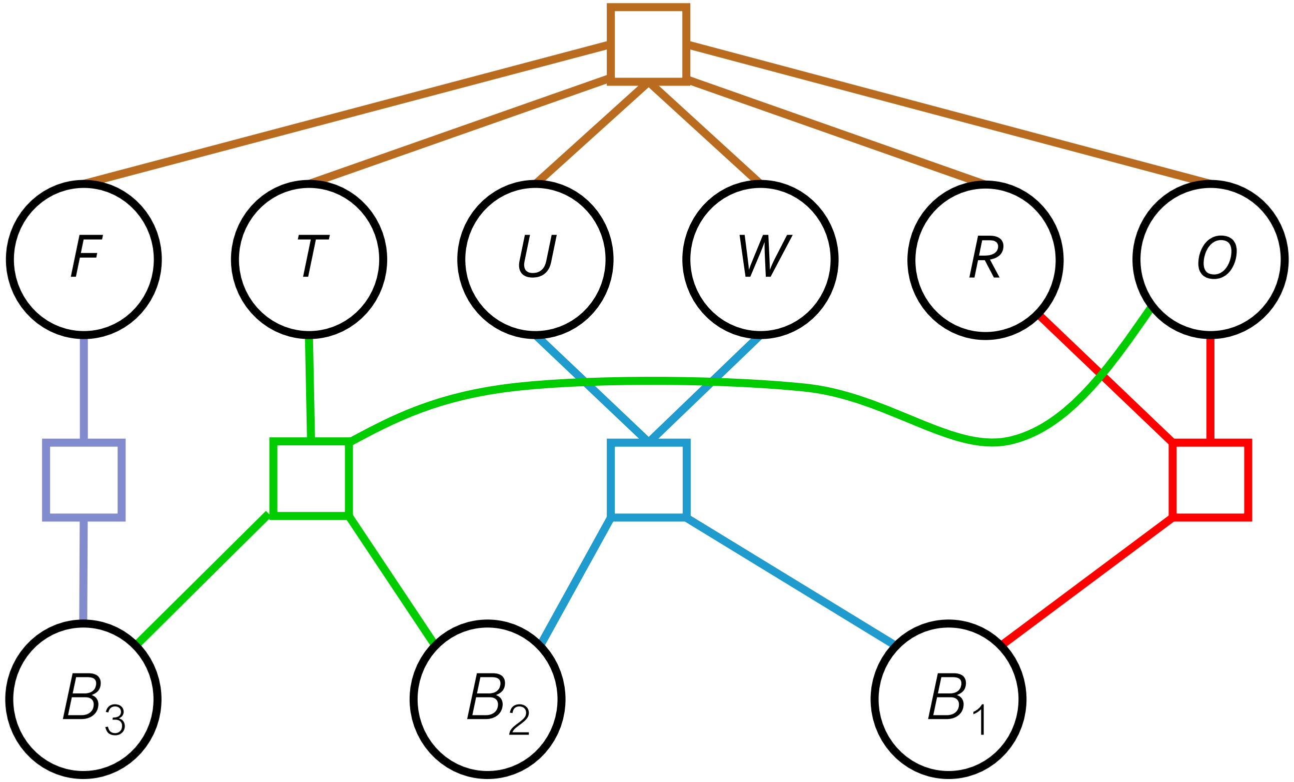




Constraint Graph for Example 2: Cryptarithmetic Puzzle

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

- Constraints
  - **`alldiff`**(T, W, O, F, U, R)
  - $O + O = R + 10.B_1$
  - $B_1 + W + W = U + 10.B_2$
  - $B_2 + T + T = O + 10.B_3$
  - $B_3 = F$
  - $T, F \neq 0$



A First Look at an Algorithm for CSPs

General Idea for the Algorithm

```
assignments = initial state (no assignments made)
while assignments incomplete:
      if no possible assignments left return failure
      current = assign a value to non-assigned variable
      if current consistent then assignments.store(current)
return assignments
```

- Applicable to all CSPs
- Search path irrelevant
  - May use complete-state formulation
- All solutions require |X| = n assignments

Which algorithm should be used?

DFS

Search Tree Size

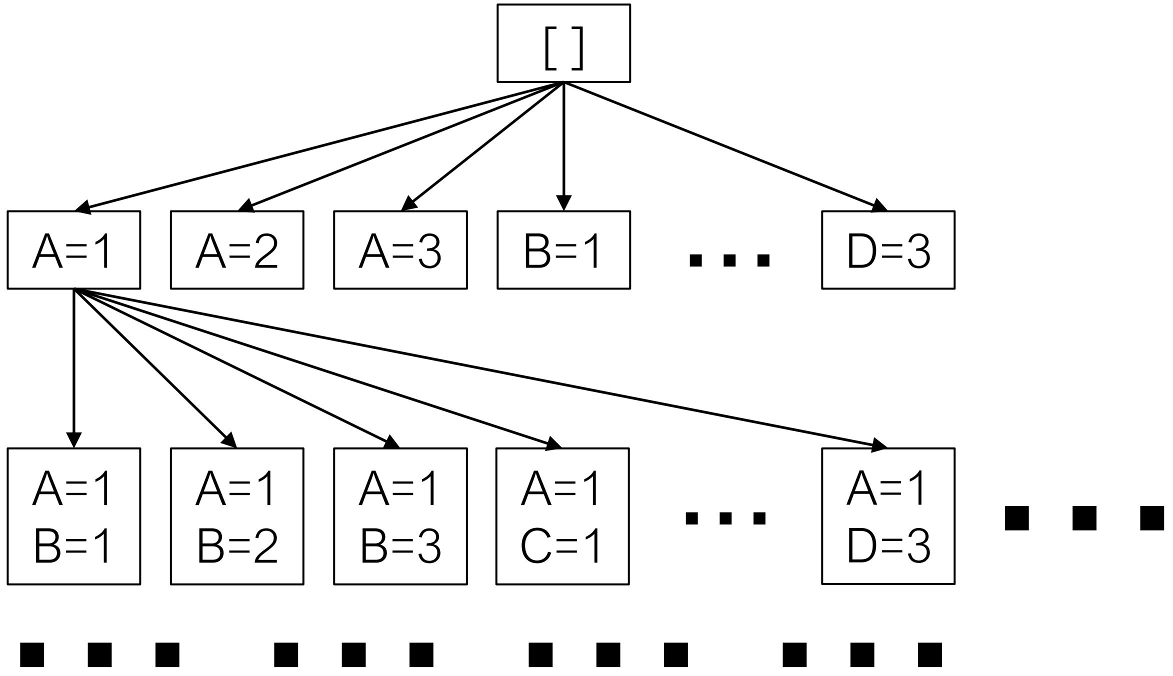
- Example CSP
  - X = {A, B, C, D}
  - All domains: d = {1, 2, 3}
  - No constraints
- Analysis

  b at depth 1: 4 variables × 3 values = 12 states
  b at depth 2: 3 variables × 3 values = 9 states
  b at depth 3: 2 variables × 3 values = 6 states
  b at depth 4: 1 variables × 3 values = 3 states

  At depth ℓ : $(|X| - \ell).|d|$ states

  Total number of leaf states:
  $nm \times (n-1)m \times (n-2)m \times \ldots \times 2m \times m = n!m^n$

  where $n = |X|$ and $m = |d|$



Order of variable assignments not important
Just consider assignments to ONE variable per level (m^n leaves)

Basic uninformed search for CSPs: Backtracking
Backtrack when no legal assignments

Backtracking Algorithm for CSPs

```
function BACKTRACKING-SEARCH(csp) returns a solution or failure
  return BACKTRACK(csp, { })

function BACKTRACK(csp, assignment) returns a solution or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp, assignment)
  for each value in ORDER-DOMAIN-VALUES(csp, var, assignment) do
     if value is consistent with assignment then
        add {var = value} to assignment
        inferences ← INFERENCE(csp, var, assignment)
        if inferences ≠ failure then
           add inferences to csp
           result ← BACKTRACK(csp, assignment)
           if result ≠ failure then return result
           remove inferences from csp
        remove {var = value} from assignment
  return failure
```

Determine the variable to assign to

Determine the value to assign

Trying to determine if the chosen assignment will lead to a terminal state

Continues recursively as long as the ***assignment*** is ***viable***

We will look into making these choices in the next lecture

Questions about the Lecture?

- Was anything unclear?
- Do you need to clarify anything?

- Ask on Archipelago
  - Specify a question
  - Upvote someone else's question

Invitation Link (Use NUS Email --- starts with E)
https://archipelago.rocks/app/resend-invite/12384352999