Local Search:
Goal Versus Path Search

CS3243: Introduction to Artificial Intelligence – Lecture 5a

NUS | Computing
National University of Singapore

1 Administrative Matters

Upcoming...

- Deadlines
  - Tutorial Assignment 3 (released last week)
    - Post-tutorial submission: Due this Friday!
  - Tutorial Assignment 4 (released today)
    - Pre-tutorial Submission: Due this Sunday!
    - Post-tutorial Submission: Due next Friday!
  - Project 1.1 (released 26 August)
    - Due THIS Sunday! 15 September 2024
  - Project 1.2 (released 26 August)
    - Due NEXT Sunday! 22 September 2024

Late Penalties
- < deadline +24 hours = 80% of score
- < deadline +48 hours = 50% of score
- ≥ deadline +48 hours = 0% of score

Project 1 Consultations:
Thursdays via Zoom
Canvas > Announcements

Contents

- Goal Versus Path Search
- Local Search via Hill-Climbing
- Local Beam Search
- Constraint Satisfaction Problems (CSPs)
- CSP Formulation
- A First Look at an Algorithm for CSPs

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

2 Goal Versus Path Search

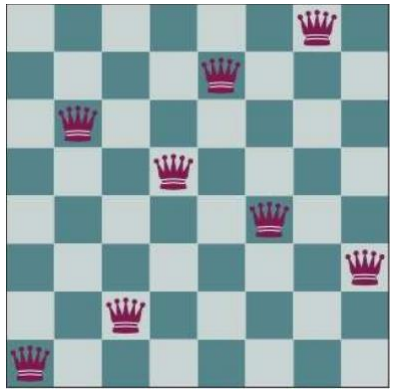
Slightly Different Problems

- Objective thus far: find a path to a goal
  - Algorithms systematically search paths
- What if we are only interested in goal state?
  - Have goal test, but not values to satisfy it
  - We only want legal goal state values
  - Some examples include optimisation problems
    - Vertex cover problems
    - Travelling salesman problem
    - Timetabling / scheduling problems
    - Boolean satisfiability problems (SAT)

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
  - Only maintain "best" successor state – greedy approach
- 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

		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		

Local Search is incomplete

Sacrifice systematic nature for superior space complexity

3 Local Search via Hill-Climbing

Hill-Climbing Algorithm

```
Function HillClimbing(initial_state, eval_fn, highest_valued_successor):
   current = initial_state
   while True:
         neighbour = highest_valued_successor(current)
         if eval_fn(neighbour) ≤ eval_fn(current): return current
         current = neighbour
```

- How it works (steepest ascent – greedy strategy)

Hill-Climbing Algorithm

```
Function HillClimbing(initial_state, eval_fn, highest_valued_successor):
   current = initial_state
   while True:
         neighbour = highest_valued_successor(current)
         if eval_fn(neighbour) ≤ eval_fn(current): return current
         current = neighbour
```

- How it works (steepest ascent – greedy strategy)
  - Starts with a random initial state (typically) – more on this later
  - Frontier only stores the current (best) state

Hill-Climbing Algorithm

```
Function HillClimbing(initial_state, eval_fn, highest_valued_successor):
   current = initial_state
   while True:
        neighbour = highest_valued_successor(current)
        if eval_fn(neighbour) ≤ eval_fn(current): return current
        current = neighbour
```

- How it works (steepest ascent – greedy strategy)
  - Starts with a random initial state (typically) – more on this later
  - Frontier only stores the current (best) state
  - In each iteration, `highest_valued_successor(..)` finds a successor that improves on the current state
    - Requires `actions` and `T` (i.e., transition model) to determine successors
    - Requires `eval_fn` (i.e., evaluation function); a way to score each state – e.g., `f(n) = -h(n)`

Requires a heuristic (similar to informed search heuristic)

Hill-Climbing Algorithm

```
Function HillClimbing(initial_state, eval_fn, highest_valued_successor):
   current = initial_state
   while True:
         neighbour = highest_valued_successor(current)
         if eval_fn(neighbour) ≤ eval_fn(current): return current
         current = neighbour
```

- How it works (steepest ascent – greedy strategy)
  - Starts with a random initial state (typically) – more on this later
  - Frontier only stores the current (best) state
  - In each iteration, `highest_valued_successor(..)` finds a successor that improves on the current state
    - Requires `actions` and `T` (i.e., transition model) to determine successors
    - Requires `eval_fn` (i.e., evaluation function); a way to score each state – e.g., `f(n) = -h(n)`

    Requires a heuristic (similar to informed search heuristic)
  - If none exists, return `current` as the best state
    - This algorithm can fail; may return a non-goal state

Hill-Climbing Algorithm

```
Function HillClimbing(initial_state, eval_fn, highest_valued_successor):
   current = initial_state
   while True:
         neighbour = highest_valued_successor(current)
         if eval_fn(neighbour) ≤ eval_fn(current): return current
         current = neighbour
```

- How it works (steepest ascent – greedy strategy)
  - Starts with a random initial state (typically) – more on this later
  - Frontier only stores the current (best) state
  - In each iteration, `highest_valued_successor(..)` finds a successor that improves on the current state
    - Requires `actions` and `T` (i.e., transition model) to determine successors
    - Requires `eval_fn` (i.e., evaluation function); a way to score each state – e.g., `f(n) = -h(n)`

    Requires a heuristic (similar to informed search heuristic)
  - If none exists, return `current` as the best state
    - This algorithm can fail; may return a non-goal state
  - Else continue on to the next iteration by updating `current` with the new best state

Hill-Climbing Algorithm

```
Function HillClimbing(initial_state, eval_fn, highest_valued_successor):
   current = initial_state
   while True:
         neighbour = highest_valued_successor(current)
         if eval_fn(neighbour) ≤ eval_fn(current): return current
         current = neighbour
```

- Frequently asked questions
  - The variable `current` only stores one state; so do we still require nodes?
    - No; since we no longer care about path information, and are only concerned with state information
    - Only the state itself is stored; from this we can still use the actions and transition to search further
  - Should we still consider tree or graph search implementations?
    - No; tree and graph search relate to path searching, whereas in local search, we only search for legal states
    - The idea is to reduce space complexity since a path is no longer necessary; a reached hash table would defeat the purpose of the space saving
  - Isn't this the same as greedy search?
    - Note that greedy search still searches paths – i.e., there are nodes on the frontier (with path information)

8-Queens Example

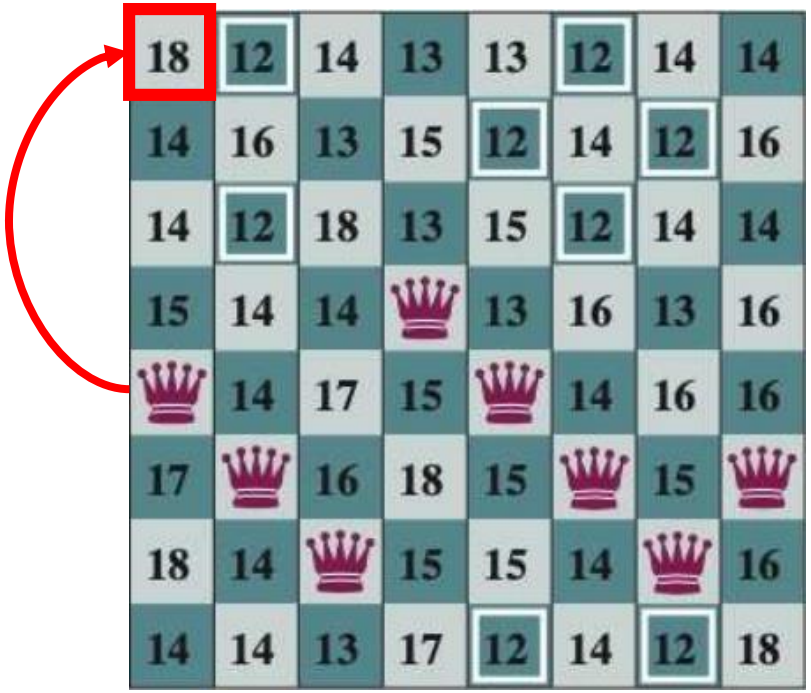
- Given an 8×8 chess board
  - Place 8 queens
  - No queen must threaten another
  - **h**: # pairs of queens threatening each other
- Search problem
  - State: 1 queen per column
  - Action*: move 1 queen to different row position
    (note that each queen stays in its column)
  - Goal: 0 pairs threatening

* Trivial transition model given this actions function

What is the initial state? What is the cost function?

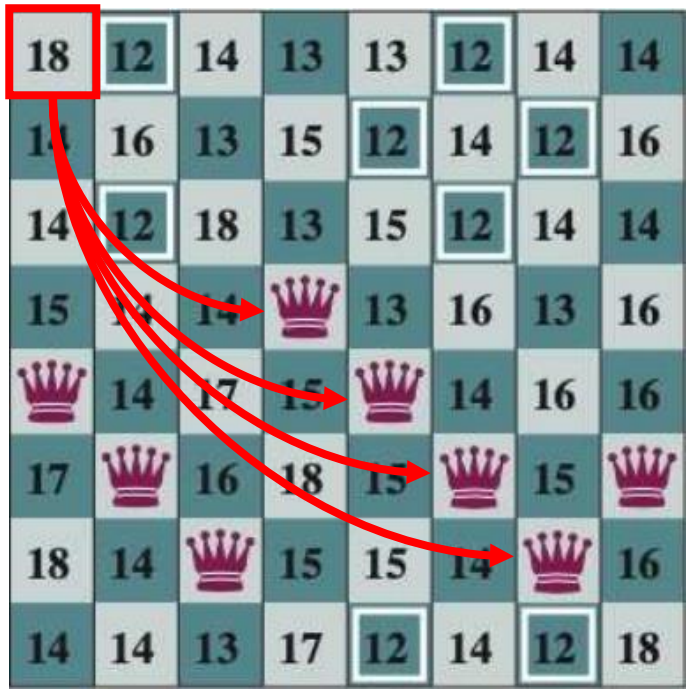
8-Queens Example

- Example `h` value calculation
  - Consider the state where the queen in the first column (C1) is moved to the top-most left-most cell?
  - Why is the `h` value of this state 18?



8-Queens Example

- Example `h` value calculation
  - Consider the state where the queen in the first column (C1) is moved to the top-most left-most cell?
  - Why is the `h` value of this state 18?
    - C1 (now in top-most left-most call) attacks C4, C5, C6, C7 → `h += 4`



8-Queens Example

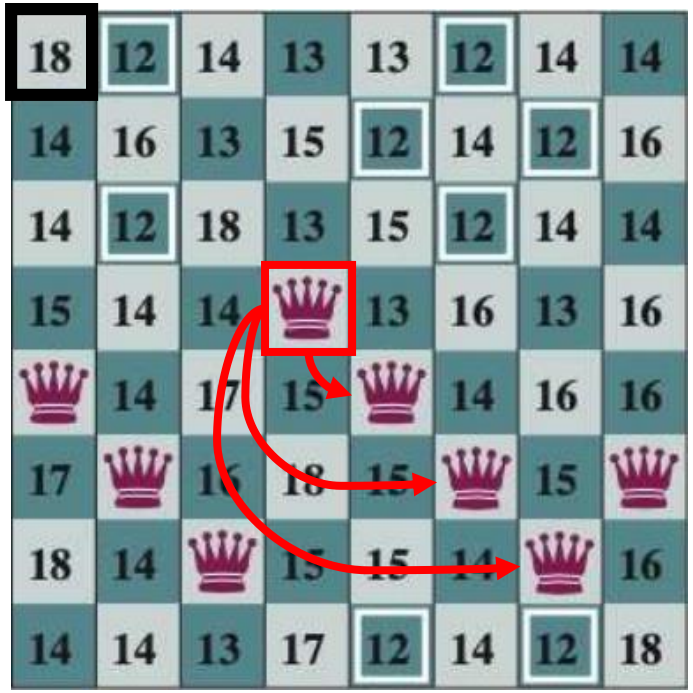
- Example `h` value calculation
  - Consider the state where the queen in the first column (C1) is moved to the top-most left-most cell?
  - Why is the `h` value of this state 18?
    - C1 (now in top-most left-most call) attacks C4, C5, C6, C7 → `h += 4`
    - C2 attacks C3, C4, C6, C8 → `h += 4`

8-Queens Example

- Example `h` value calculation
  - Consider the state where the queen in the first column (C1) is moved to the top-most left-most cell?
  - Why is the `h` value of this state 18?
    - C1 (now in top-most left-most call) attacks C4, C5, C6, C7 → `h += 4`
    - C2 attacks C3, C4, C6, C8 → `h += 4`
    - C3 attacks C5, C7 → `h += 2`

8-Queens Example

- Example `h` value calculation
  - Consider the state where the queen in the first column (C1) is moved to the top-most left-most cell?
  - Why is the `h` value of this state 18?
    - C1 (now in top-most left-most call) attacks C4, C5, C6, C7 → `h += 4`
    - C2 attacks C3, C4, C6, C8 → `h += 4`
    - C3 attacks C5, C7 → `h += 2`
    - C4 attacks C5, C6, C7 → `h += 3`



8-Queens Example

- Example `h` value calculation
  - Consider the state where the queen in the first column (C1) is moved to the top-most left-most cell?
  - Why is the `h` value of this state 18?
    - C1 (now in top-most left-most call) attacks C4, C5, C6, C7 → `h += 4`
    - C2 attacks C3, C4, C6, C8 → `h += 4`
    - C3 attacks C5, C7 → `h += 2`
    - C4 attacks C5, C6, C7 → `h += 3`
    - C5 attacks C6, C7 → `h += 2`

8-Queens Example

- Example `h` value calculation
  - Consider the state where the queen in the first column (C1) is moved to the top-most left-most cell?
  - Why is the `h` value of this state 18?
    - C1 (now in top-most left-most call) attacks C4, C5, C6, C7 → `h += 4`
    - C2 attacks C3, C4, C6, C8 → `h += 4`
    - C3 attacks C5, C7 → `h += 2`
    - C4 attacks C5, C6, C7 → `h += 3`
    - C5 attacks C6, C7 → `h += 2`
    - C6 attacks C7, C8 → `h += 2`

8-Queens Example

- Example `h` value calculation
  - Consider the state where the queen in the first column (C1) is moved to the top-most left-most cell?
  - Why is the `h` value of this state 18?
    - C1 (now in top-most left-most call) attacks C4, C5, C6, C7 → `h += 4`
    - C2 attacks C3, C4, C6, C8 → `h += 4`
    - C3 attacks C5, C7 → `h += 2`
    - C4 attacks C5, C6, C7 → `h += 3`
    - C5 attacks C6, C7 → `h += 2`
    - C6 attacks C7, C8 → `h += 2`
    - C7 attacks C8 → `h += 1`
    - C8 has no queens on the right (so attacks none)

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 validity
    - 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

Note that, 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 ≥

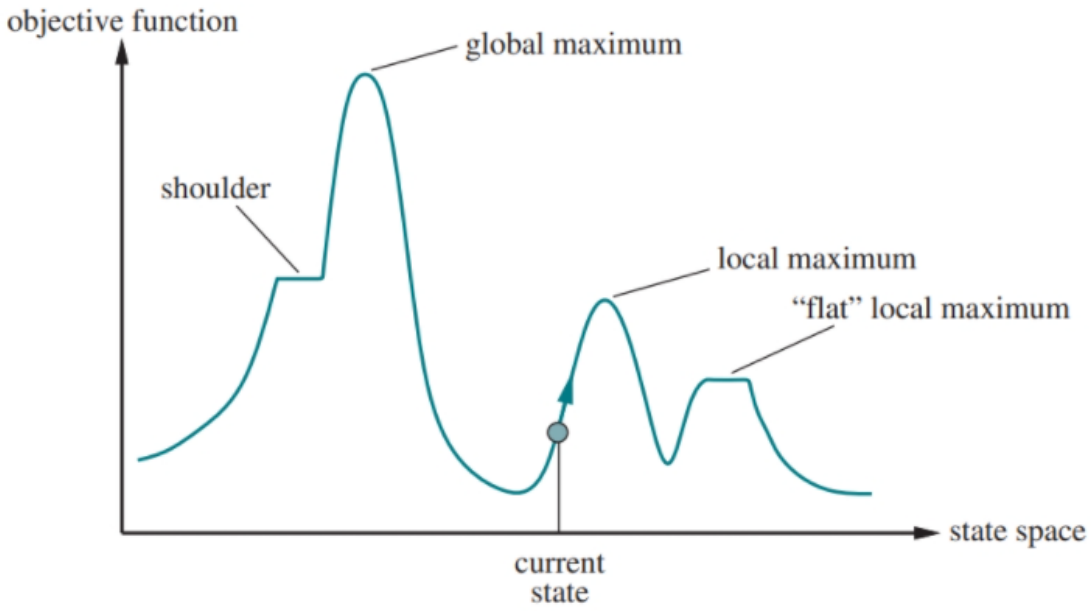
Hill-Climbing Algorithm (Revisited)

```
Function HillClimbing(initial_state, eval_fn, highest_valued_successor):
   current = initial_state
   while True:
        neighbour = highest_valued_successor(current)
        if eval_fn(neighbour) ≤ eval_fn(current): return current
        current = neighbour
```

- NOT guaranteed to find a goal!
  - `eval_fn` 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 the function `highest_valued_successor(..)`
  - Chooses randomly among states with `f`-values better than `current`
  - May take longer to find a solution but sometimes leads to better solutions

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

Hill-Climbing Variants

- Sideways move
  - Replaces ≤ with <
  - Allows continuation when `eval_fn(neighbour) == eval_fn(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

```
Function RandomRestarts(isGoal, random_initial_state, eval_fn, highest_valued_successor):
   current = NULL
   while current is NULL or not isGoal(current):
         current = random_initial_state()
         current = HillClimbing(current, eval_fn, highest_valued_successor)
   return current
```

- Differences as compared to the Hill-Climbing Algorithm
  - Wrapper to the `HillClimbing` algorithm
  - Repeatedly runs `HillClimbing` with different (random) initial states until a goal is found
    - Requires function to generate random initial state: `random_initial_state()`

Back to 8-Queens: Analysis

- Hill climbing (via steepest-ascent) with random restarts
  - Successful 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

Note that $((1 - p_1) / p_1)$ determines the expected number of failed attempts before success

- Adding sideways moves
  - Successful 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 $= 8^8 = 16,777,216$

Extremely efficient for such a large space

Expected values taken from: AIMA 4th Edition, pp 131

4 Local Beam Search

Local Beam Search

- Store `k` states instead of 1
  - Hill climbing just stores the current state
  - Beam (window) stores `k`
- General algorithm
  - Begins with `k` random starts
  - Each iteration generates successors for each of the `k` random start 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

Local Beam Search Algorithm

```
Function BeamSearch(random_initial_state, isGoal, actions, T, choose_best_k):
    frontier = {}
    for i in 1 to k:
        s = random_initial_state()
        if isGoal(s): return s
            frontier.push(s)
    while True:
        new_frontier = {}
        while frontier not empty:
            s = frontier.pop()
            for a in actions(s):
                new_s = T(s, a)
                if isGoal(new_s): return new_s
                new_frontier.push(new_s)
        frontier = choose_best_k(new_frontier)
```

As the search may loop infinitely, it may be wise to define and use an iteration threshold

We may also apply stochastic beam search where instead of choosing the best k, it chooses successors with probability proportional to the successor's value

- Notice that we may allow states with lower values than their parents to be added since `choose_best_k` will simply choose the best `k` successor states generated
  - We may vary this rule but then may not be able to always ensure we have `k` new candidates

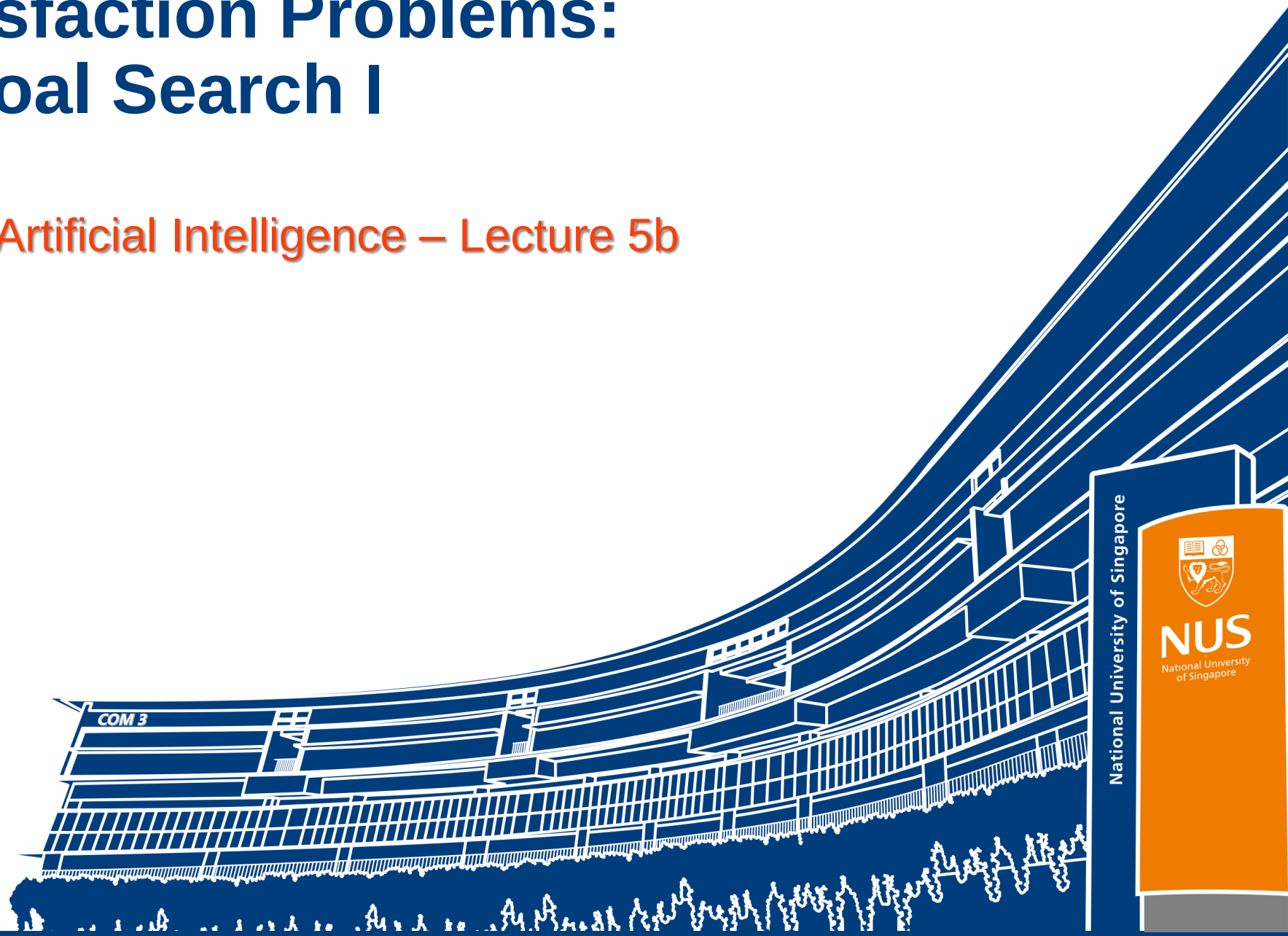
Questions about the Lecture?

- 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/99329793204

Constraint Satisfaction Problems: Generalising Goal Search I

CS3243: Introduction to Artificial Intelligence – Lecture 5b



NUS | Computing
National University of Singapore

Generalised Goal Search

- With local search we apply greedy search strategies that require h
  - Are there more general search strategies applicable?
- A general solution
  - Use a factored representation for each state
    - State: set of variables $X = \{x_1, ..., x_n\}$, where each variable x_i has a domain $D_i = \{d_1, ..., d_k\}$
  - 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 any constraint should not be further explored

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

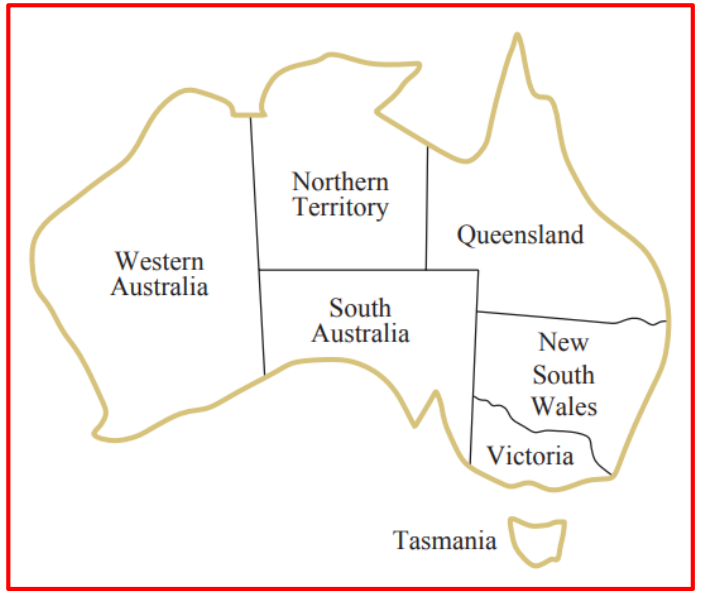
5 CSP Formulation

Formulating CSPs

- State representation
  - Variables: $X = \{x_1, ..., x_n\}$
  - Domains: $D = \{d_1, ..., d_k\}$
    - Such that xi has a domain di
  - Initial state: all variables unassigned
  - Intermediate state: partial assignment
- 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
- Actions, costs and transition
  - Assignment of values (within domain) to variables
  - Costs are unnecessary
  - Objective is a complete and consistent assignment
    - Find a legal assignment $(y_1, ..., y_n)$
      - $y_i \in D_i$ for all $i \in [1, n]$
  - Complete: all variables assigned values
  - Consistent: all constraints in 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
    - $d_i = \{ Red, Green, Blue \}$
  - Constraints
    - $\forall (x_i, x_j) \in E, colour(x_i) \neq colour(x_j)$



CSP Formulation Example 2: Cryptarithmetic Puzzle

- Given that each letter represents a digit, determine the letter-digit mapping that solves the given sum
  - 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
    - $\text{alldiff}(T, W, O, F, U, R)$
    - $O + O = R + 10 \cdot B_1$
    - $B_1 + W + W = U + 10 \cdot B_2$
    - $B_2 + T + T = O + 10 \cdot B_3$
    - $B_3 = F$
    - $T, F \neq 0$

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

CSP Formulation Example 3: Sudoku

- Colour each state of Australia such that no two adjacent states share the same colour
  - 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(T, W, O, F, U, R)
      - 27 cases
        - 9 columns
        - 9 rows
        - 9 (3×3) 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
    - E.g., real values
  - Discrete and Infinite
    - E.g., all integers
  - Discrete and finite
    - E.g., Sudoku

CS3243 focuses on discrete, finite domains

- Constraint types
  - Arithmetic*
    - e.g., inequalities
  - Logical
    - i.e., literals with semantic meaning
    - e.g., `AllDifferent`
  - Extensional
    - i.e., set of values that defines when the constraint is satisfied

* You may have also heard about constraint optimisation problems (COP) and their solutions, such as linear programming (CSPs are a subset of COPs; COPs also include objective functions (e.g., cost/utility functions) that must be optimised)

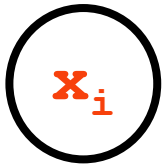
Note that such methods (outside CSPs) are not covered in CS3243

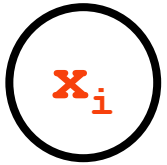
More on Constraints

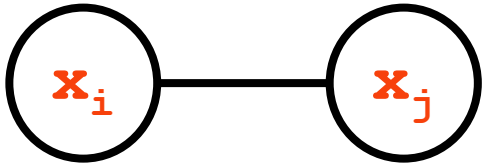
- A language is necessary to express the constraints (based on constraint type)
  - Algebraic for arithmetic
    - e.g., ⟨ (x_1, x_2), $x_1 > x_2$ ⟩ → x_1 greater than x_2 given $D = \{1, 2, 3\}$
  - Propositional logic for logical
  - Sets (of legal values) for extensional
    - e.g., ⟨ (x_1, x_2), { (2, 1), (3, 1), (3, 2) } ⟩ with the same example as above
- 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)

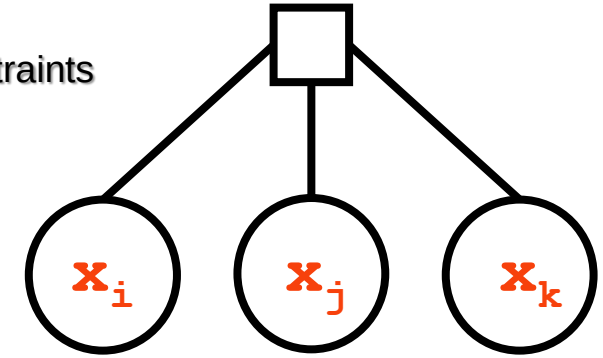
6 Constraint Graphs

Drawing Constraint Graphs and Hypergraphs

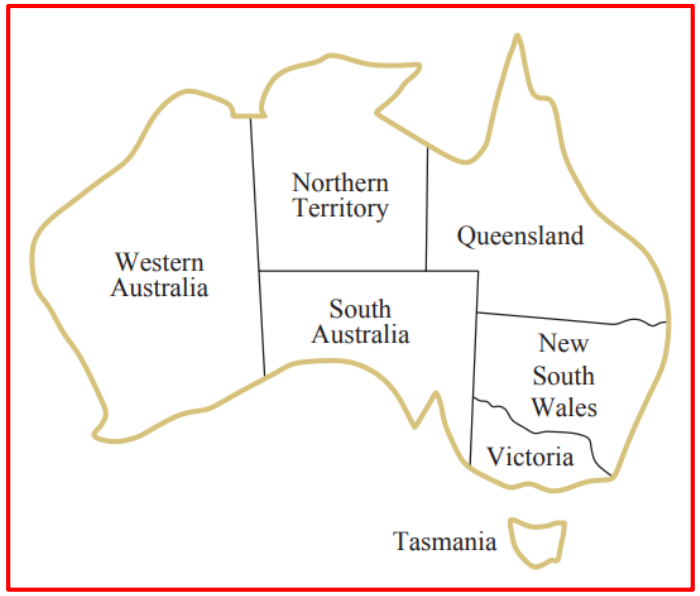
- Constraint graphs represent the constraints in a CSP
  - Simple Vertex: variable 

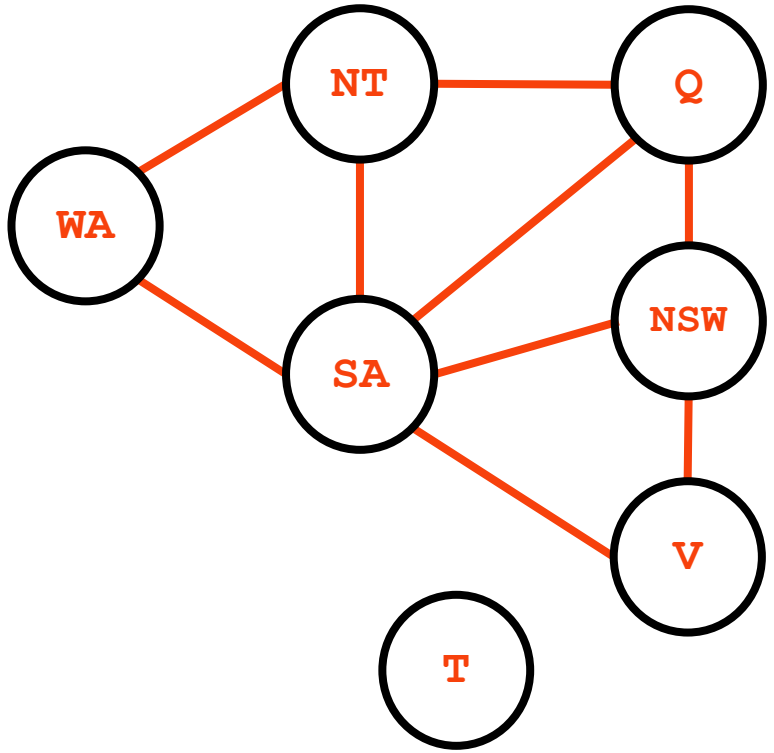
  - Linking Vertex: for global constraints 
  - Edge: links all variables in the `scope` of a constraint (`rel`)
    - Unary constraints 

    - Binary constraints 

    - Binary/Global constraints 

Constraint Graph for Example 1: Graph Colouring



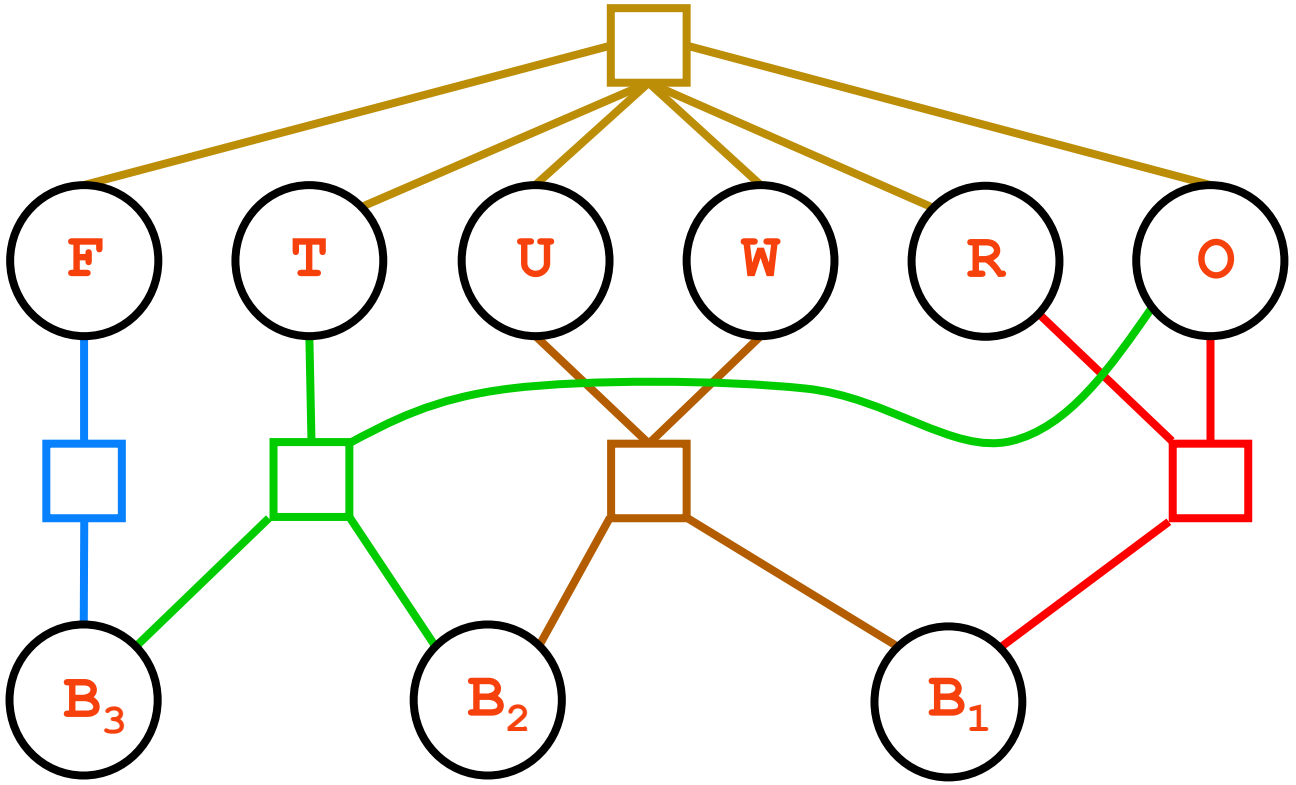


- Constraints
  - $\forall(x_i, x_j) \in E, \texttt{colour}(x_i) \neq \texttt{colour}(x_j)$

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)
  - $O + O = R + 10 \cdot B_1$
  - $B_1 + W + W = U + 10 \cdot B_2$
  - $B_2 + T + T = O + 10 \cdot B_3$
  - $B_3 = F$
  - $T, F \neq 0$



7 A First Look at an Algorithm for CSPs

General Idea for the Algorithm

```
Function CSPSolver(variables, domains, constraints):
    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 is consistent then: assignments.store(current)
    return assignments
```

- Applicable to all CSPs
  - Recall that search path is irrelevant
  - Find complete and consistent assignment
  - Key Points
    - Once a state is inconsistent, any future assignments will not change this – i.e., can prune whole sub-tree
    - All solutions require |X| = n assignments (i.e., fixed depth)

Which search algorithm should be used?

DFS

Search Tree Size

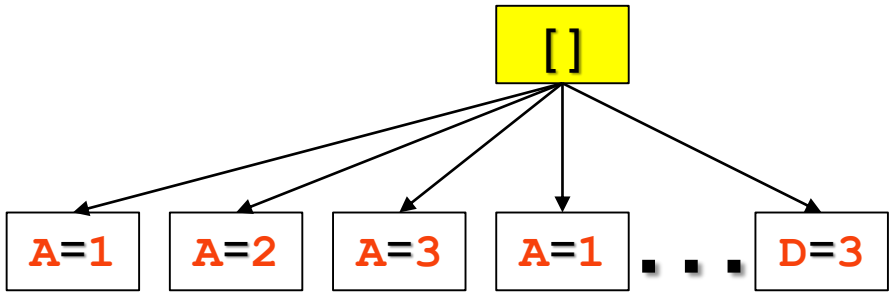
[ ]

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

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

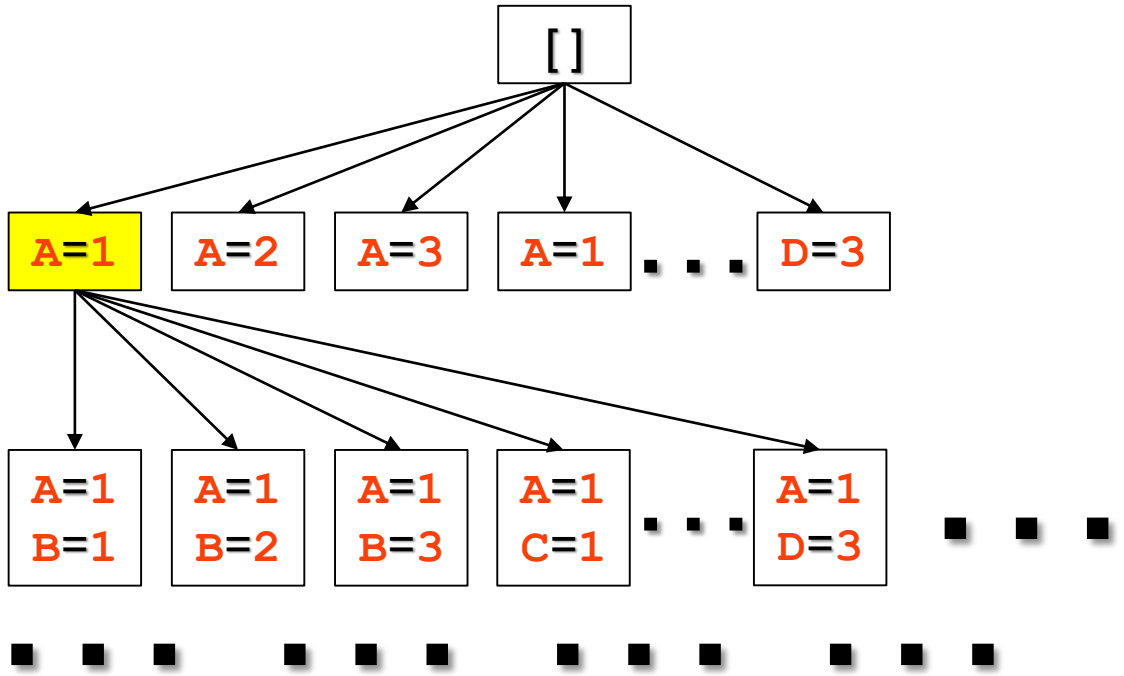


Search Tree Size

- 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

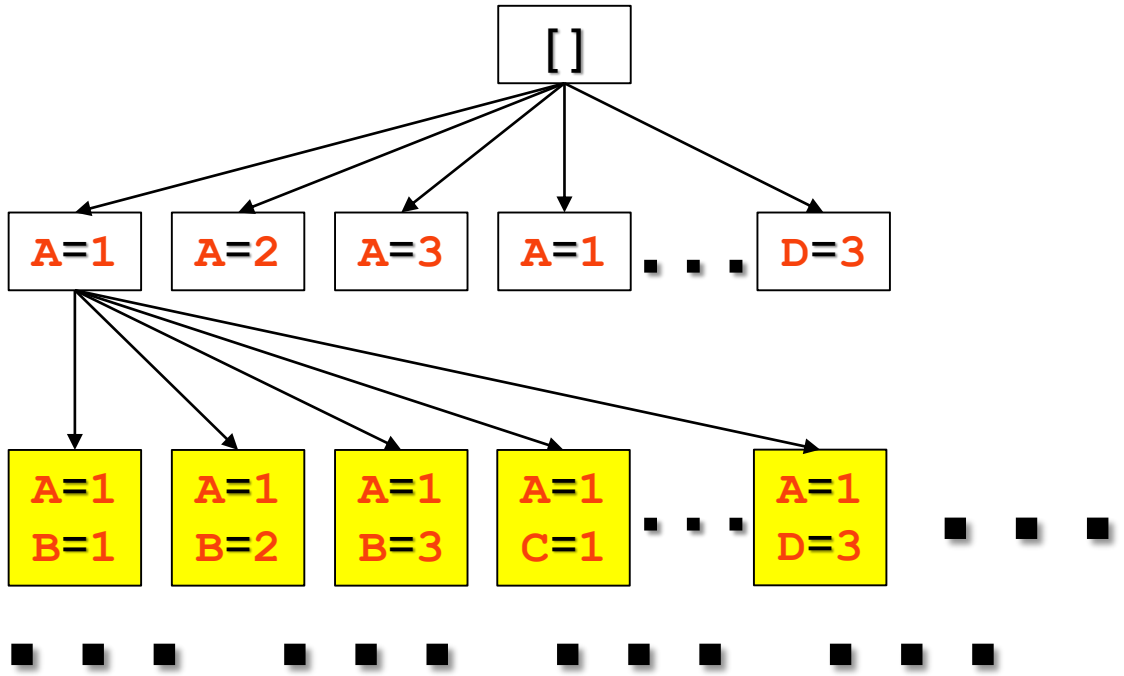
  b at depth 2: 3 variables × 3 values = 9 states



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



Search Tree Size

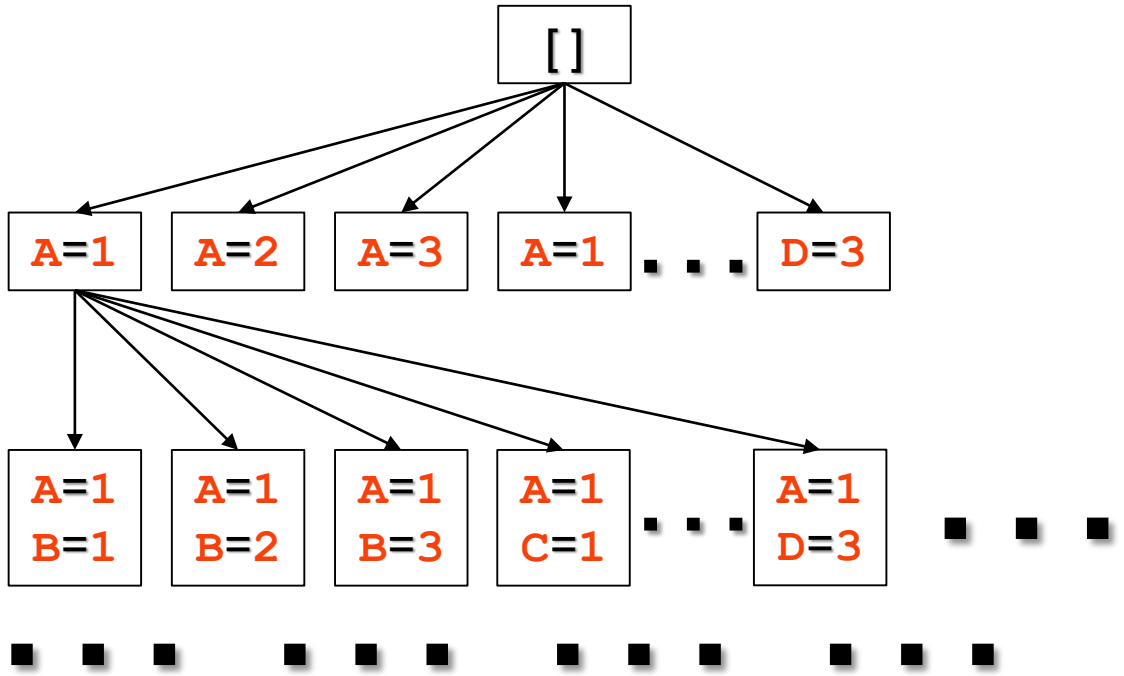
- Example CSP
  - $X = \{A, B, C, D\}$
  - All domains: $d = \{1, 2, 3\}$
  - No constraints
- Analysis

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: $(|X| - \ell) \cdot |d|$ 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 review mechanisms for 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/99329793204