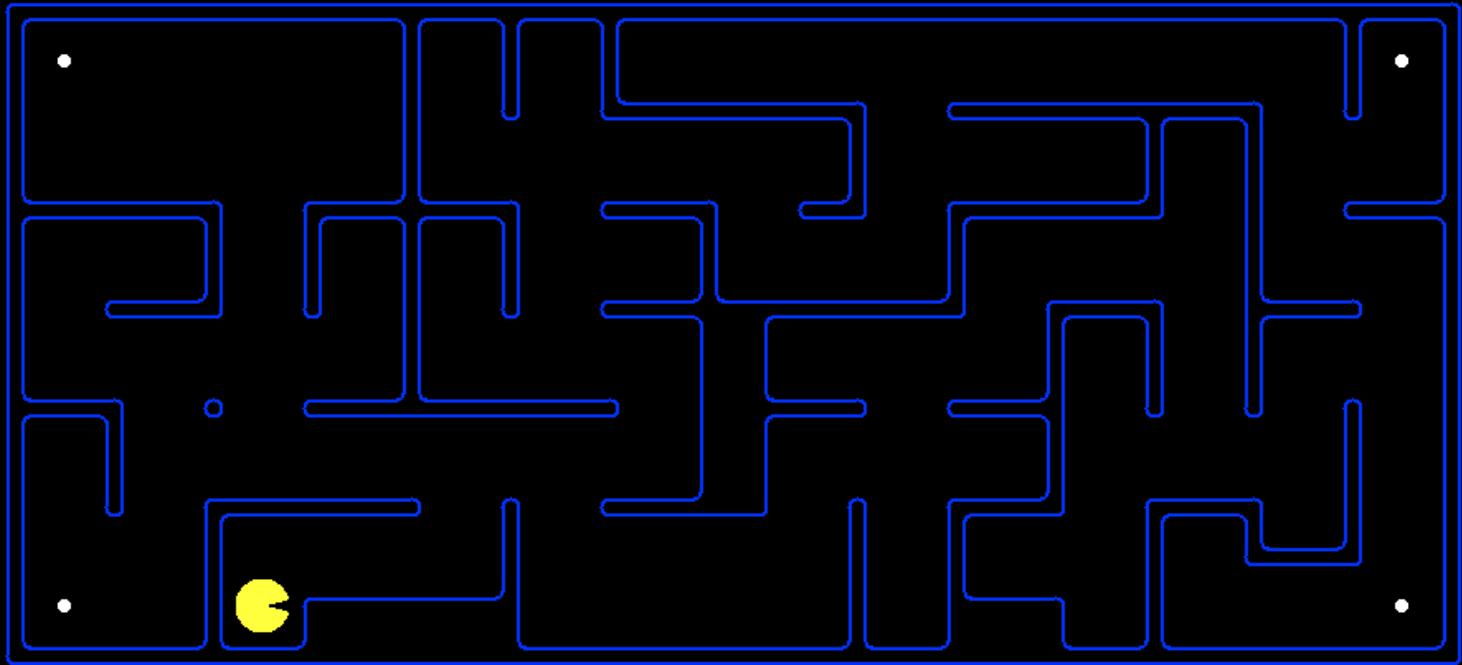
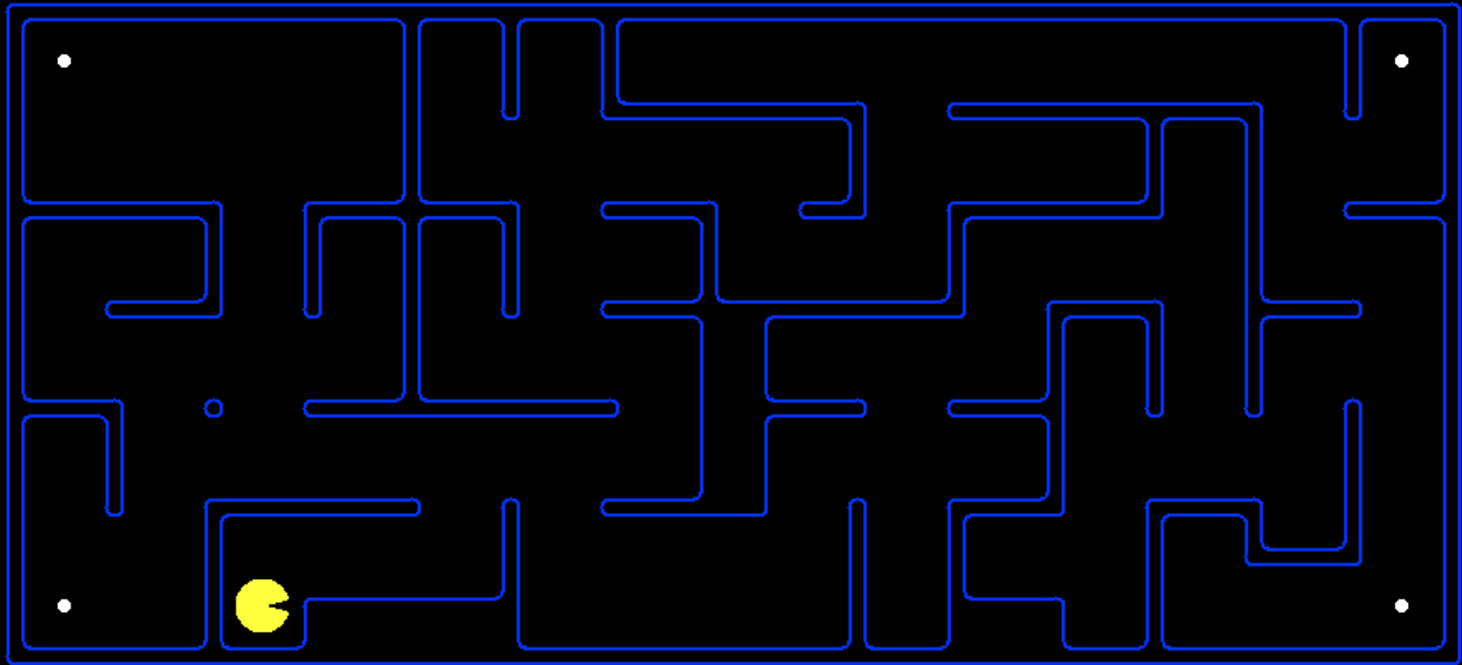


Introduction to Artificial Intelligence

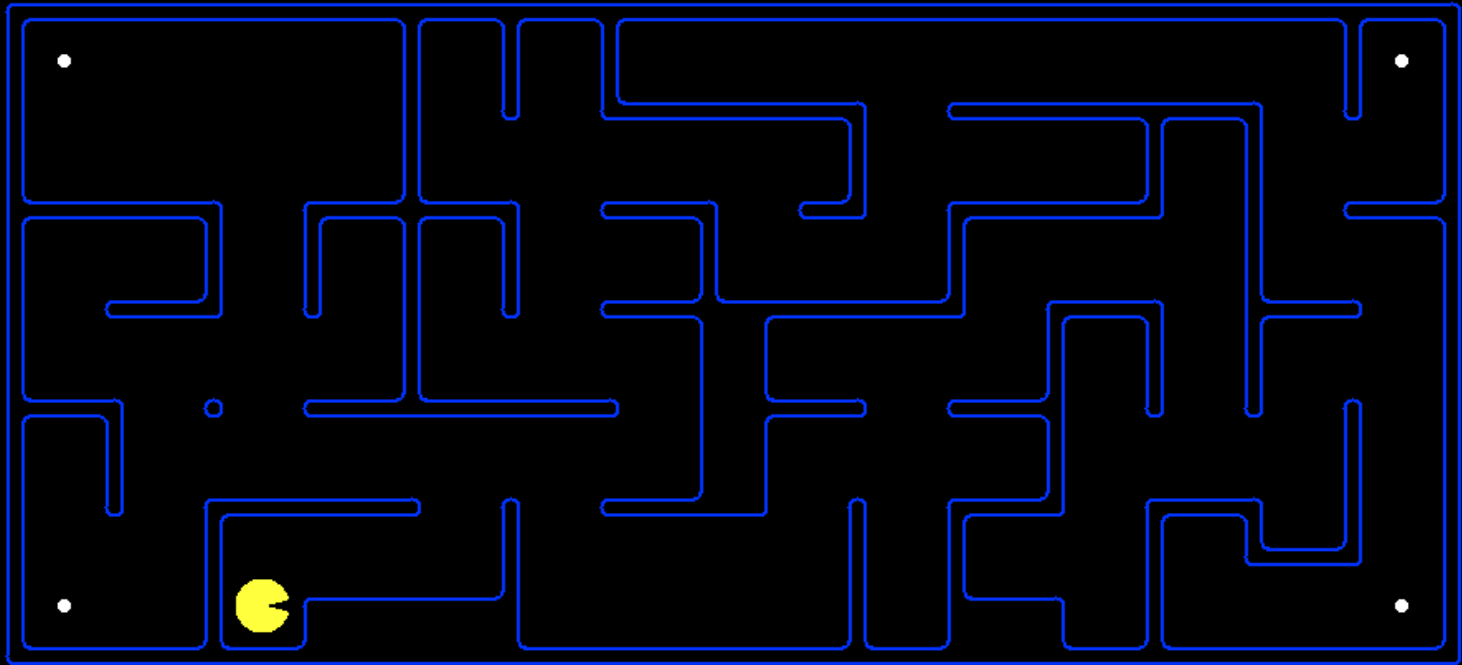
Lecture 3: Constraint satisfaction problems



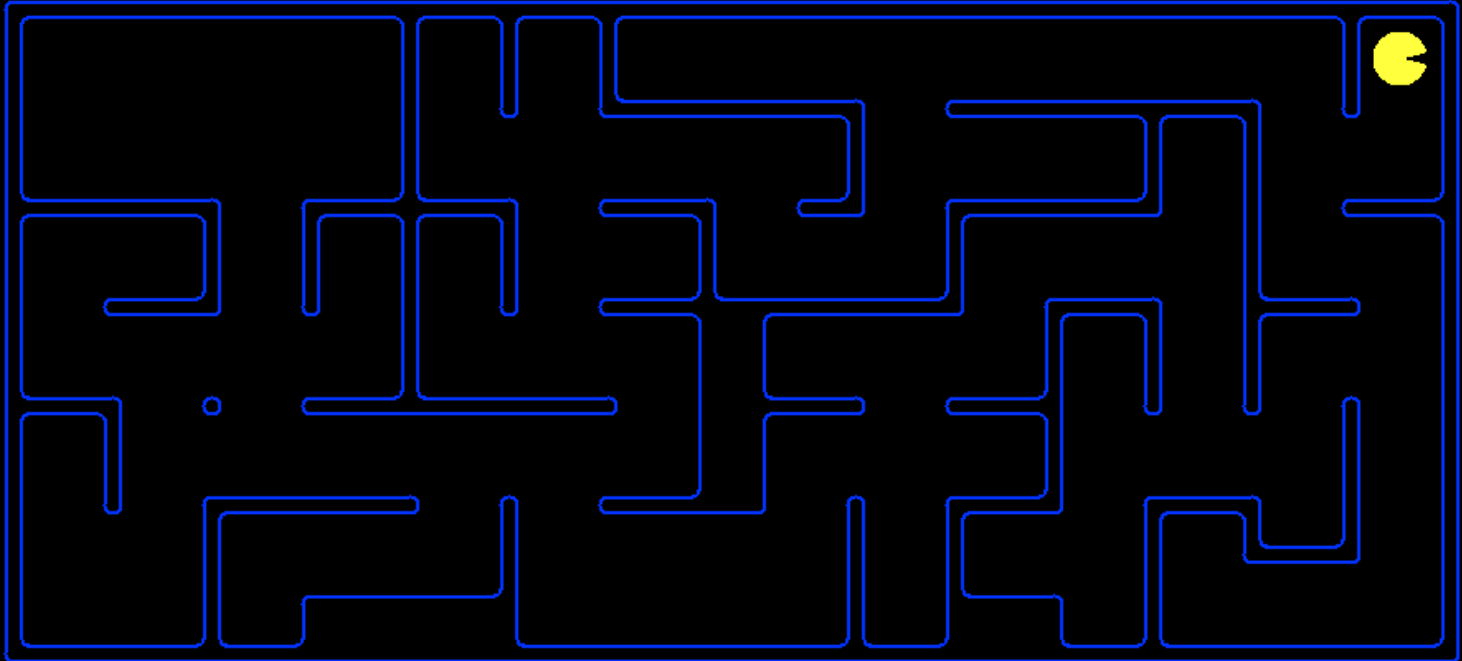
Hmmm, let me think...



(...)



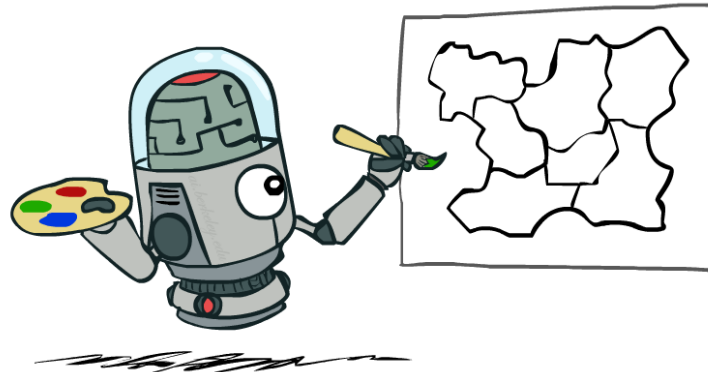
(5 minutes later)



Solution found! [Can we do better?]

Today

- **Constraint satisfaction problems:**
 - Exploiting the representation of a state to accelerate search.
 - Backtracking.
 - Generic heuristics.
- **Logical agents**
 - Propositional logic for reasoning about the world.
 - ... and its connection with CSPs.



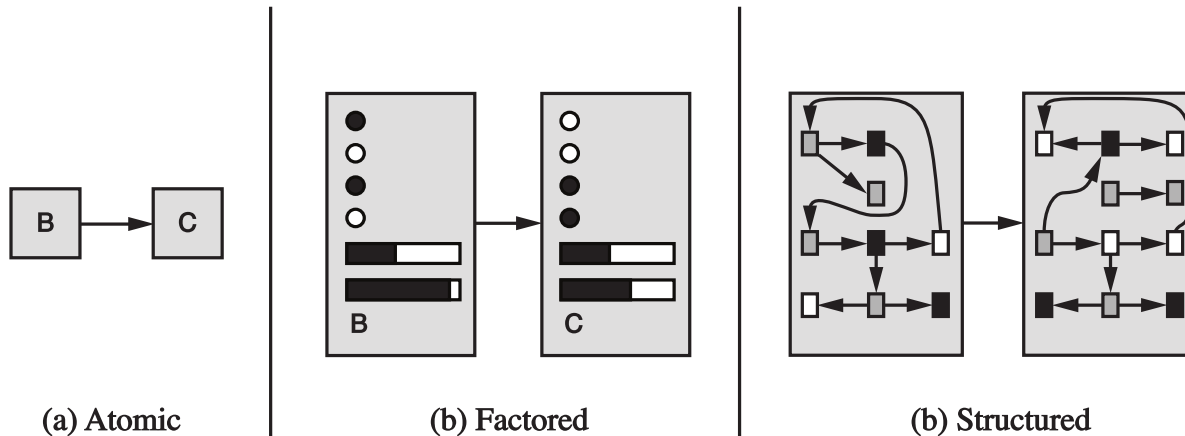
Constraint satisfaction problems

Motivation

In standard search problems:

- States are evaluated by domain-specific heuristics.
- States are tested by a domain-specific function to determine if the goal is achieved.
- From the point of view of the search algorithms however, **states are atomic**.

Instead, if states have **a factored representation**, then the structure of states can be exploited to improve the **efficiency of the search**.



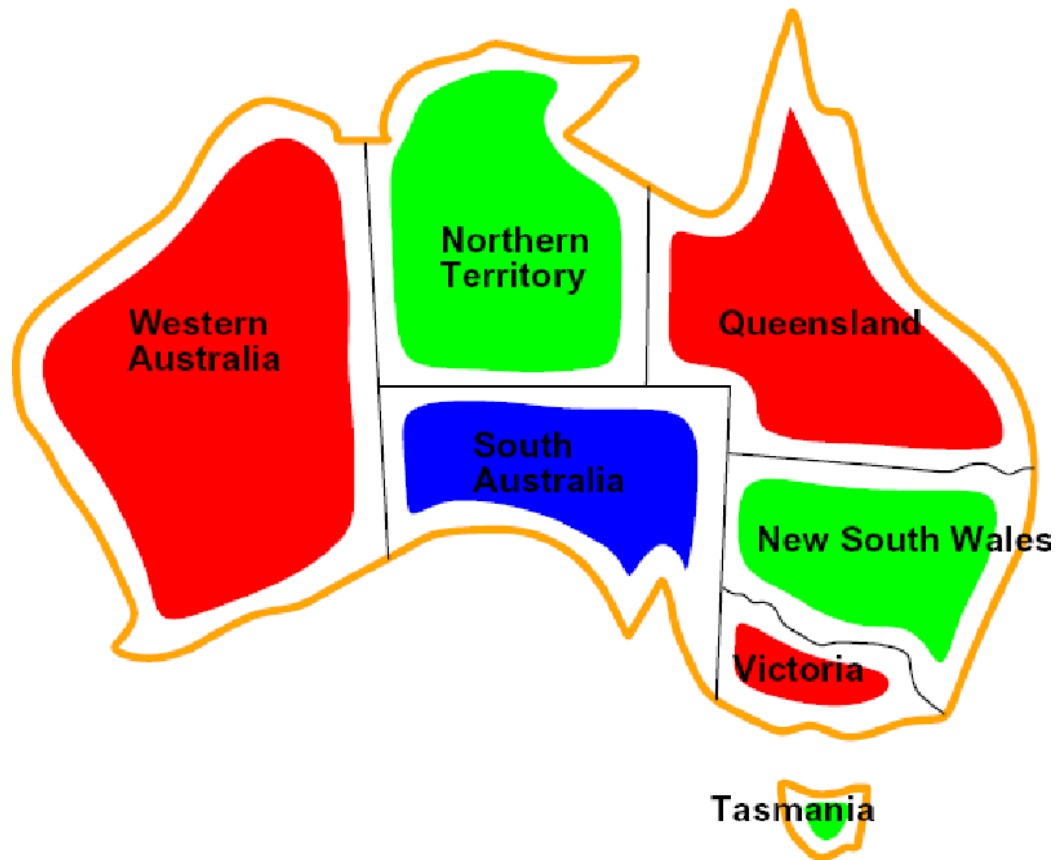
Constraint satisfaction problems

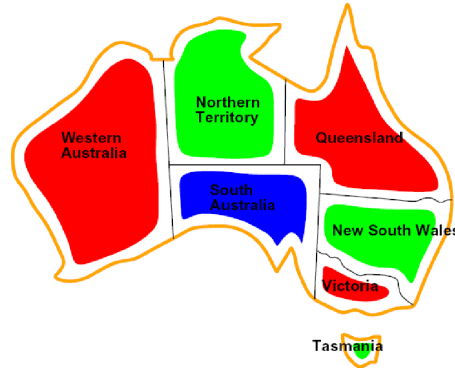
- **Constraint satisfaction problem** algorithms take advantage of factored state representations and use **general-purpose** heuristics to solve complex problems.
- CSPs are specialized to a family of search sub-problems.
- Main idea: eliminate large portions of the search space all at once, by identifying combinations of variable/value that violate constraints.

Formally, a **constraint satisfaction problem** (CSP) consists of three components X , D and C :

- X is a set of **variables**, $\{X_1, \dots, X_n\}$,
- D is a set of **domains**, $\{D_1, \dots, D_n\}$, one for each variable,
- C is a set of **constraints** that specify allowable combinations of values.

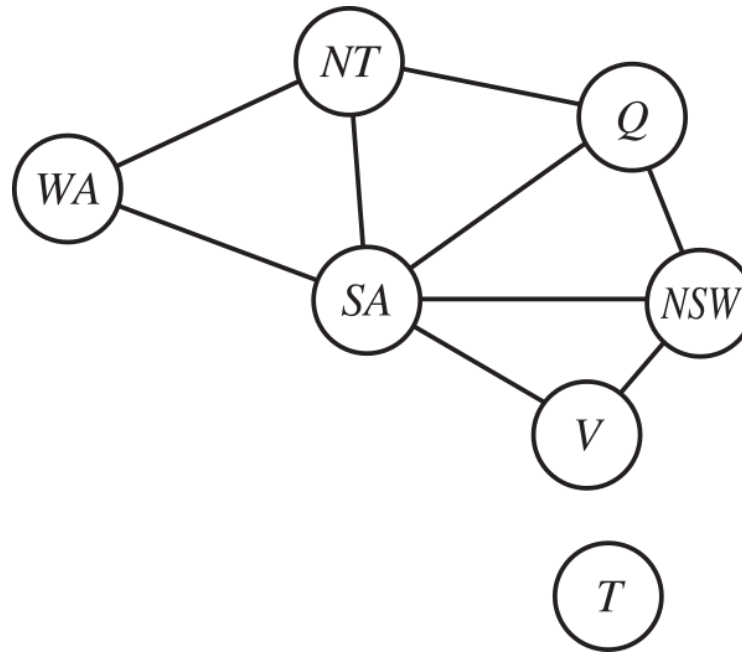
Example: Map coloring





- Variables: $X = \{WA, NT, Q, NSW, V, SA, T\}$
- Domains: $D_i = \{\text{red, green, blue}\}$ for each variable.
- Constraints: $C = \{SA \neq WA, SA \neq NT, SA \neq Q, \dots\}$
 - Implicit: $WA \neq NT$
 - Explicit: $(WA, NT) \in \{\{\text{red, green}\}, \{\text{red, blue}\}, \dots\}$
- Solutions are **assignments** of values to the variables such that constraints are all satisfied.
 - e.g., $\{WA = \text{red}, NT = \text{green}, Q = \text{red}, SA = \text{blue}, NSW = \text{green}, V = \text{red}, T = \text{green}\}$

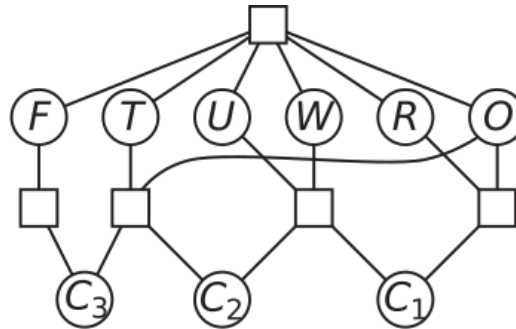
Constraint (hyper)graph



- **Nodes** = variables of the problems
- **Edges** = constraints in the problem involving the variables associated to the end nodes.
- General purpose CSP algorithms **use the graph structure** to speedup search.
 - e.g., Tasmania is an independent subproblem.

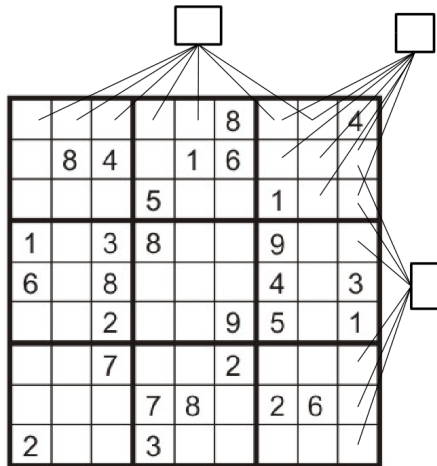
Example: Cryptarithmic

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



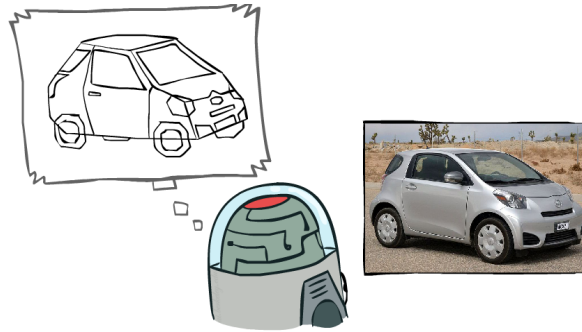
- Variables: $\{T, W, O, F, U, R, C_1, C_2, C_3\}$
- Domains: $D_i = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- Constraints:
 - $\text{alldiff}(T, W, O, F, U, R)$
 - $O + O = R + 10 \times C_1$
 - $C_1 + W + W = U + 10 \times C_2$
 - ...

Example: Sudoku



- Variables: each (open) square
- Domains: $D_i = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- Constraints:
 - 9-way **alldiff** for each column
 - 9-way **alldiff** for each row
 - 9-way **alldiff** for each region

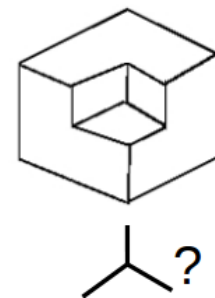
Example: The Waltz algorithm



The Waltz algorithm is a procedure for interpreting 2D line drawings of solid polyhedra as 3D objects. Early example of an AI computation posed as a CSP.

CSP formulation:

- Each intersection is a variable.
- Adjacent intersections impose constraints on each other.
- Solutions are physically realizable 3D objects.



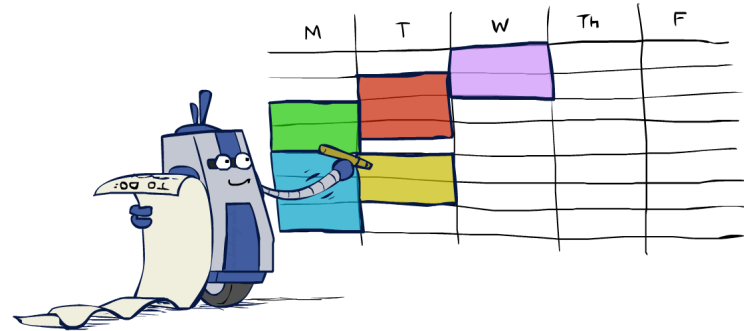
Variations on the CSP formalism

- Discrete variables
 - Finite domains
 - Size d means $O(d^n)$ complete assignments.
 - e.g., boolean CSPs, including the SAT boolean satisfiability problem (NP-complete).
 - Infinite domains
 - e.g., job scheduling, variables are start/end days for for each job.
 - need a constraint language, e.g. $start_1 + 5 \leq start_2$.
 - Solvable for linear constraints, undecidable otherwise.
- Continuous variables
 - e.g., precise start/end times of experiments.
 - Linear constraints solvable in polynomial time by LP methods.

- Varieties of constraints:
 - Unary constraint involve a single variable.
 - Equivalent to reducing the domain, e.g. $SA \neq \text{green}$.
 - Binary constraints involve pairs of variables, e.g. $SA \neq WA$.
 - Higher-order constraints involve 3 or more variables.
- Preferences (soft constraints)
 - e.g., red is better than green.
 - Often representable by a cost for each variable assignment.
 - Results in constraint optimization problems.
 - (We will ignore those for now.)

Real-world examples

- 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
- Circuit layout
- ... and many more



Notice that many real-world problems involve real-valued variables.

Constraint programming

Constraint Programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it.
(Eugene Freuder)



Constraint programming is a programming paradigm in which the user specifies the program as a CSP. The resolution of the problem is left to the computer.

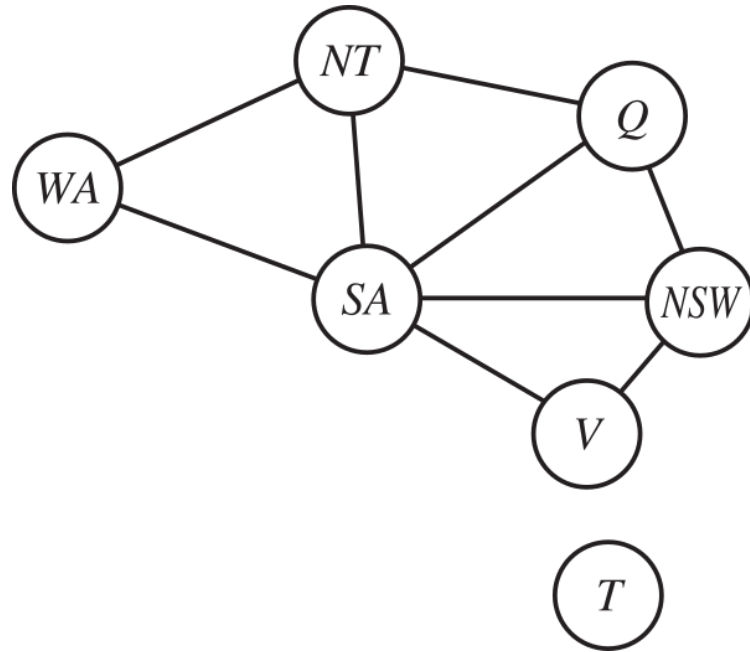
Examples:

- Prolog
- ECLiPSe

Solving CSPs

Standard search formulation

- CSPs can be cast as standard search problems.
 - For which we have solvers, including DFS, BFS or A*.
- States are **partial assignments**:
 - The initial state is the empty assignment {}.
 - Actions: assign a value to an unassigned variable.
 - Goal test: the current assignment is complete and satisfies all constraints.
- This algorithm is **the same** for all CSPs!



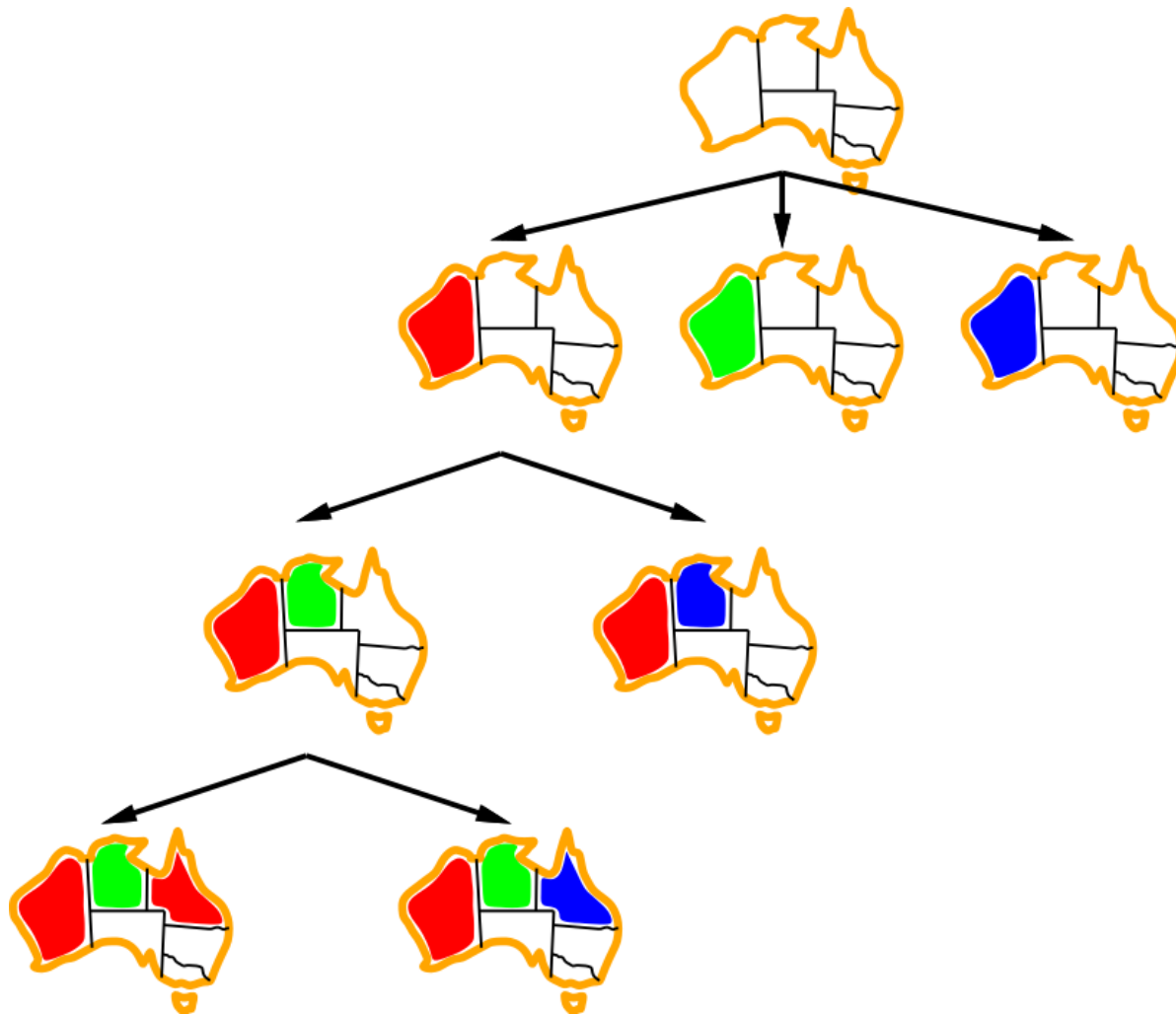
What would BFS or DFS do? What problems does naive search have?

For n variables of domain size d :

- $b = (n - l)d$ at depth l ;
- we generate a tree with $n!d^n$ leaves even if there are only d^n possible assignments!

Backtracking search

- Backtracking search is the basic uninformed algorithm for solving CSPs.
- Idea 1: **One variable at a time:**
 - The naive application of search algorithms ignore a crucial property: variable assignments are **commutative**. Therefore, fix the ordering.
 - $WA = \text{red}$ then $NT = \text{green}$ is the same as $NT = \text{green}$ then $WA = \text{red}$.
 - One only needs to consider assignments to a single variable at each step.
 - $b = d$ and there are d^n leaves.
- Idea 2: **Check constraints as you go:**
 - Consider only values which do not conflict with current partial assignment.
 - Incremental goal test.



```

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  $\leftarrow$  SELECT-UNASSIGNED-VARIABLE(csp)
    for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
        if value is consistent with assignment then
            add { var = value } to assignment
            inferences  $\leftarrow$  INFERENCE(csp, var, value)
            if inferences  $\neq$  failure then
                add inferences to assignment
                result  $\leftarrow$  BACKTRACK(assignment, csp)
                if result  $\neq$  failure then
                    return result
            remove { var = value } and inferences from assignment
    return failure

```

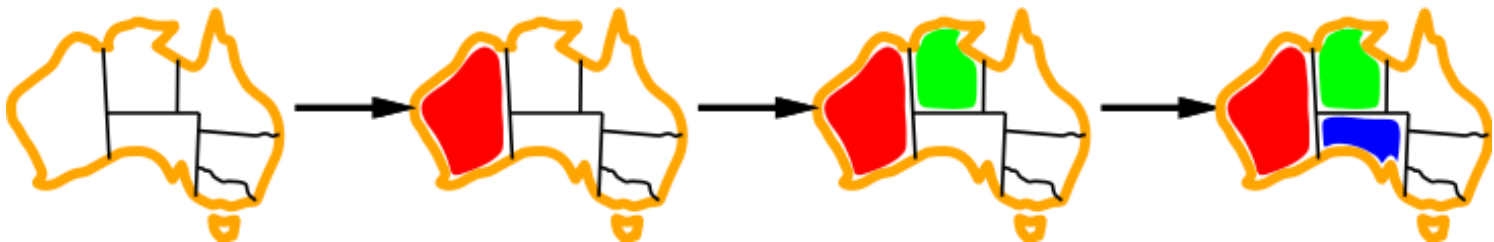
Figure 6.5 A simple backtracking algorithm for constraint satisfaction problems. The algorithm is modeled on the recursive depth-first search of Chapter 3. By varying the functions SELECT-UNASSIGNED-VARIABLE and ORDER-DOMAIN-VALUES, we can implement the general-purpose heuristics discussed in the text. The function INFERENCE can optionally be used to impose arc-, path-, or k -consistency, as desired. If a value choice leads to failure (noticed either by INFERENCE or by BACKTRACK), then value assignments (including those made by INFERENCE) are removed from the current assignment and a new value is tried.

Improving backtracking

- Can we improve backtracking using **general-purpose** ideas, without domain-specific knowledge?
- **Ordering**:
 - Which variable should be assigned next?
 - In what order should its values be tried?
- **Filtering**: can we detect inevitable failure early?
- **Structure**: can we exploit the problem structure?

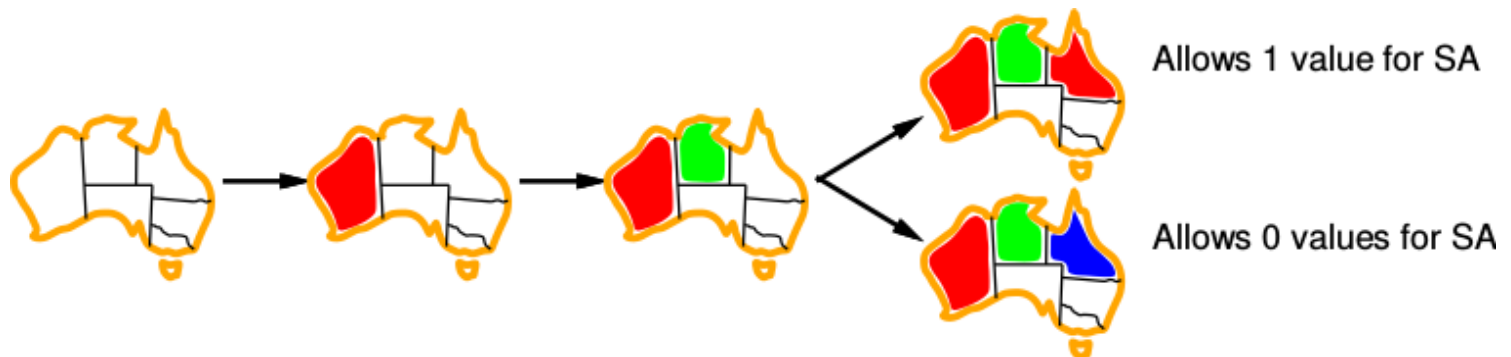
Variable ordering

- **Minimum remaining values:** Choose the variable with the fewest legal values left in its domain.
- Also known as the **fail-first** heuristic.
 - Detecting failures quickly is equivalent to pruning large parts of the search tree.



Value ordering

- **Least constraining value:** Given a choice of variable, choose the **least constraining value**.
- i.e., the value that rules out the fewest values in the remaining variables.



[Q] Why should variable selection be fail-first but value selection be fail-last?

Filtering: Forward checking

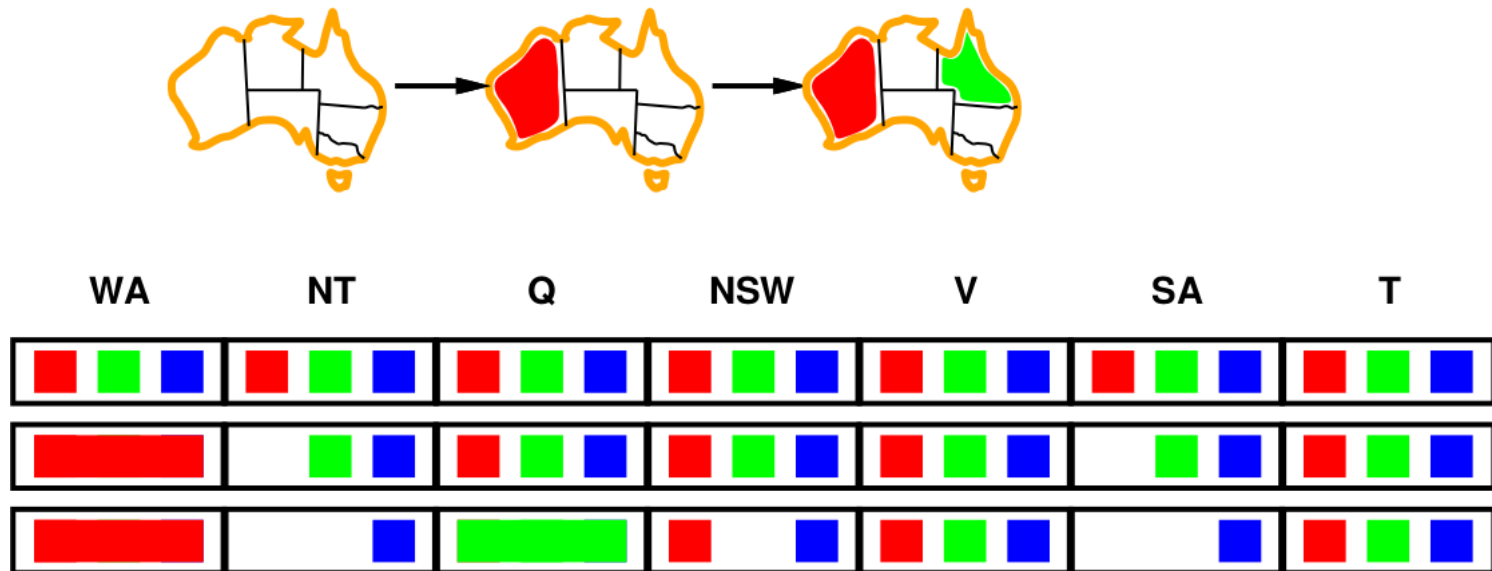
- Keep **track of remaining legal values** for unassigned variables.
 - Whenever a variable X is assigned, and for each unassigned variable Y that is connected to X by a constraint, delete from Y 's domain any value that is inconsistent.
- Terminate search** when any variable has no legal value left.



WA	NT	Q	NSW	V	SA	T
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>
<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>	<div><div></div><div></div><div></div></div>

Filtering: Constraint propagation

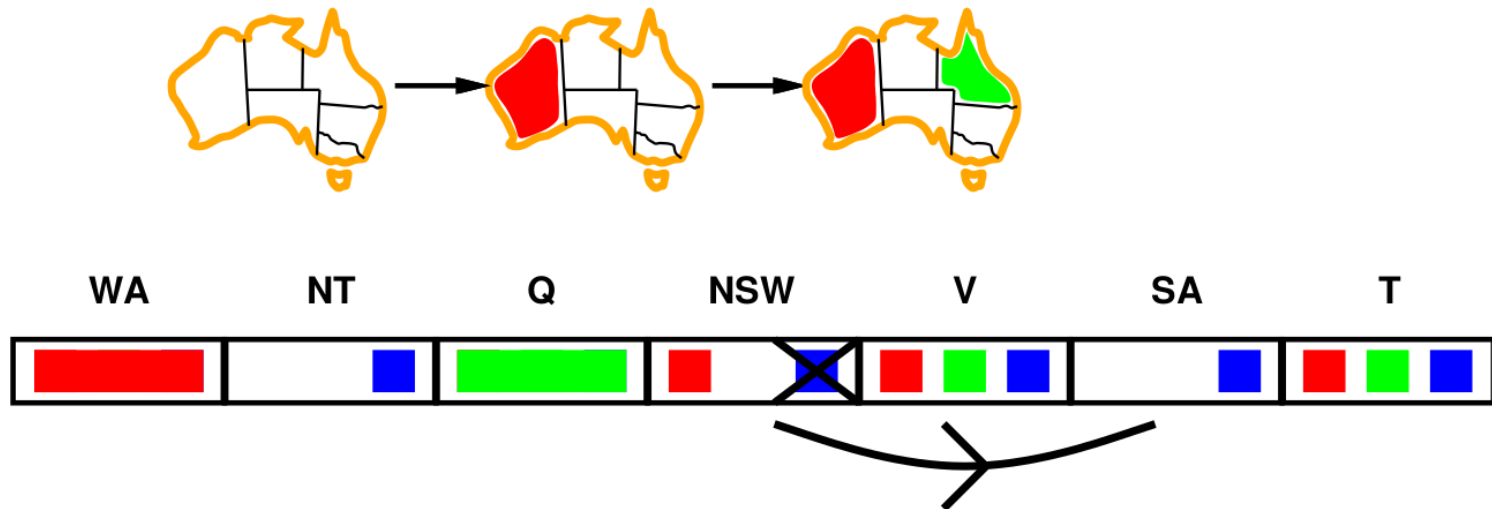
Forward checking propagates information assigned to unassigned variables, but does not provide early detection for all failures:



- *NT* and *SA* cannot both be blue!
- **Constraint propagation** repeatedly enforces constraints locally.

Arc consistency

- An arc $X \rightarrow Y$ is **consistent** if and only if for every value x in the domain of X there is some value y in the domain of Y that satisfies the associated binary constraint.
- Forward checking \Leftrightarrow enforcing consistency of arcs pointing to each new assignment.
- This principle can be generalized to enforce consistency for **all** arcs.



function AC-3(*csp*) **returns** false if an inconsistency is found and 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) \leftarrow 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 \leftarrow *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**

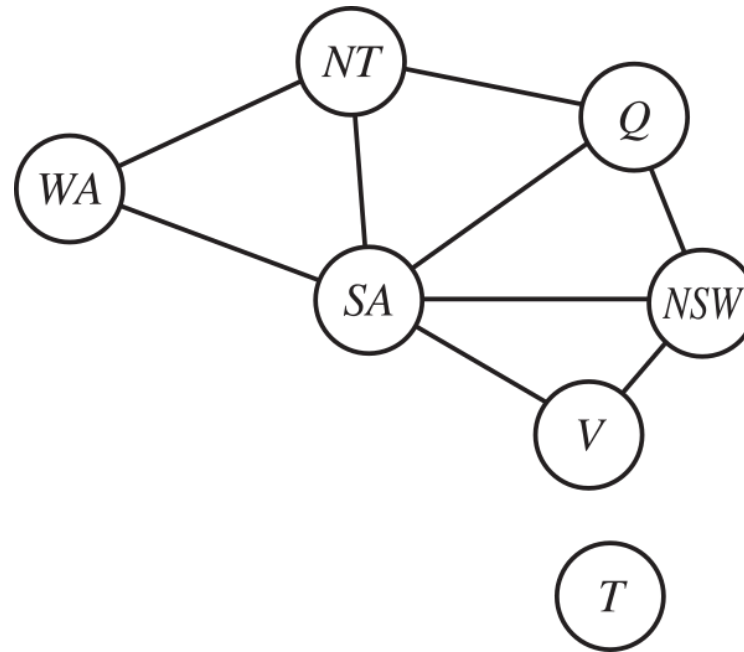
 delete x from D_i

revised \leftarrow *true*

return *revised*

[Q] When in backtracking shall this procedure be called?

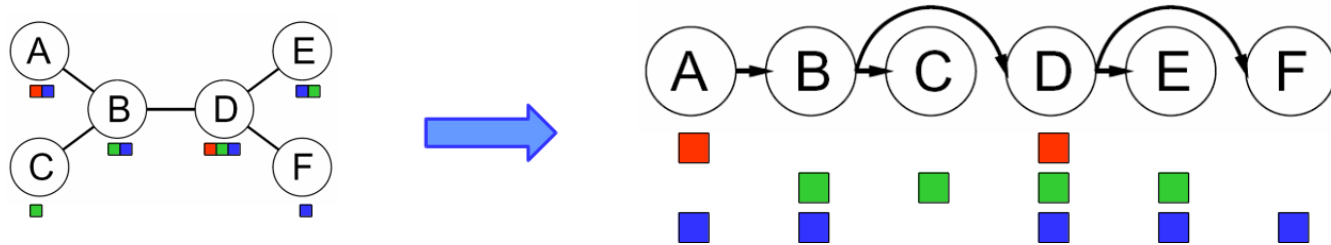
Structure



- Tasmania and mainland are **independent subproblems**.
 - Any solution for the mainland combined with any solution for Tasmania yields a solution for the whole map.
- Independence can be ascertained by finding **connected components** of the constraint graph.

- Time complexity: Assume each subproblem has c variables out of n in total. Then $O\left(\frac{n}{c}d^c\right)$.
 - E.g., $n = 80, d = 2, c = 20$.
 - $2^{80} = 4$ billion years at 10 million nodes/sec.
 - $4 \times 2^{20} = 0.4$ seconds at 10 million nodes/sec.

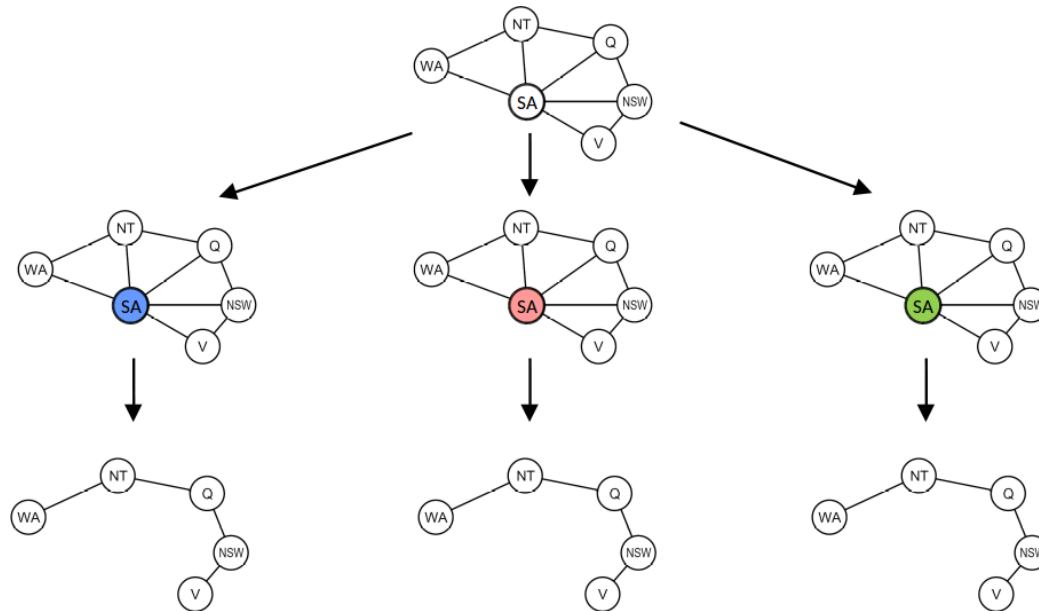
Tree-structured CSPs



- Algorithm for tree-structured CSPs:
 - Order: choose a root variable, order variables so that parents precede children (topological sort).
 - Remove backward:
 - for $i = n$ down to 2 , enforce arc consistency of $\text{parent}(X_i) \rightarrow X_i$.
 - Assign forward:
 - for $i = 1$ to n , assign X_i consistently with its $\text{parent}(X_i)$.
- Time complexity: $O(nd^2)$
 - Compare to general CSPs, where worst-case time is $O(d^n)$.

Nearly tree-structured CSPs

- **Conditioning**: instantiate a variable, prune its neighbors' domains.
- **Cutset conditioning**:
 - Assign (in all ways) a set S of variables such that the remaining constraint graph is a tree.
 - Solve the residual CSPs (tree-structured).
 - If the residual CSP has a solution, return it together with the assignment for S .



Logical agents

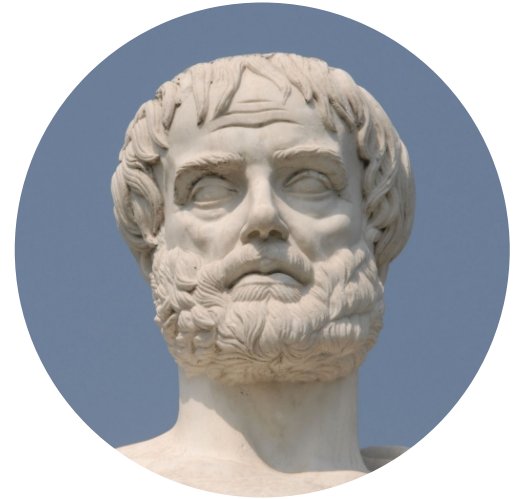
The logicist tradition

- The rational thinking approach to artificial intelligence is concerned with the study of **irrefutable reasoning processes**. It ensures that all actions performed by an agent are formally **provable** from inputs and prior knowledge.
- The Greek philosopher Aristotle was one of the first to attempt to formalize rational thinking. His **syllogisms** provided a pattern for argument structures that always yield correct conclusion when given correct premises.

All men are mortal.

Socrates is a man.

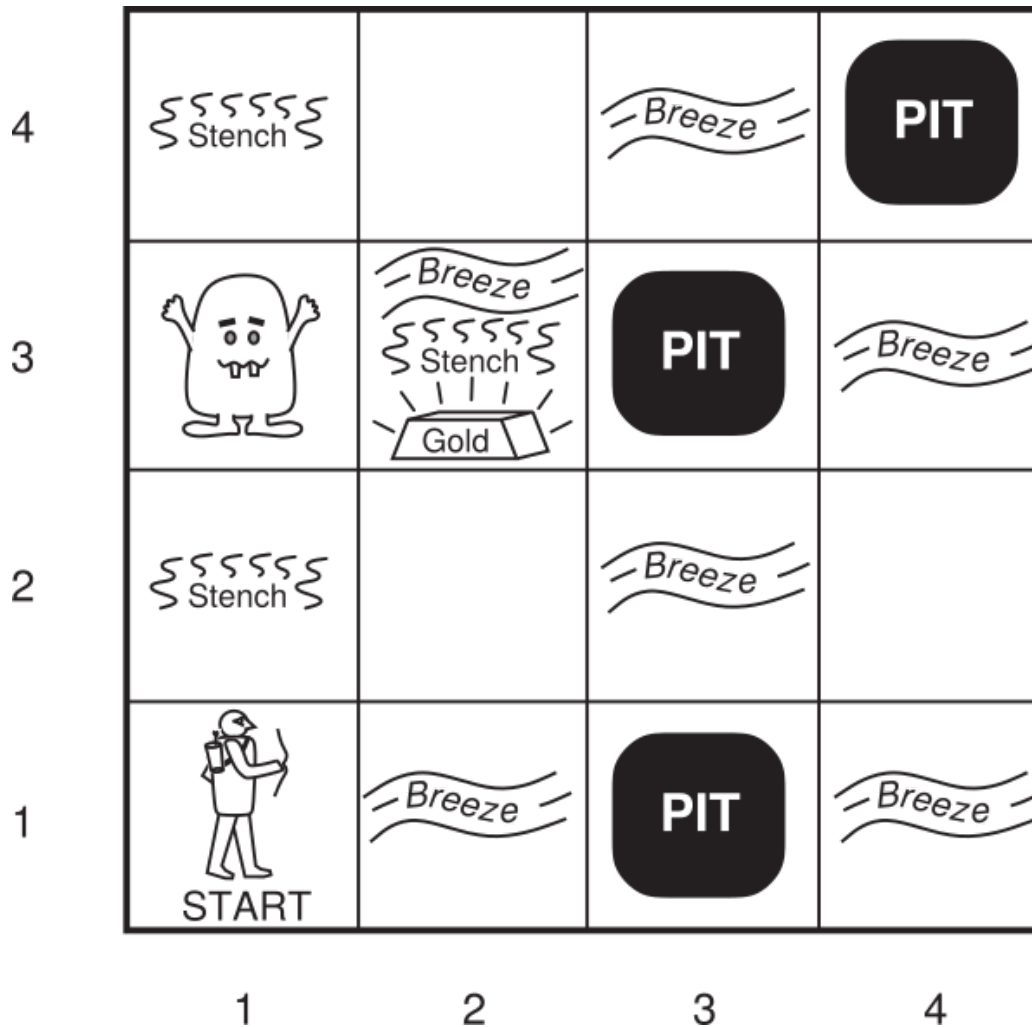
Therefore, Socrates is mortal.



(Aristotle, 384-322 BC)

- Logicians of the 19th century developed a precise notation for statements about all kinds of objects in the world and relationships among them.
- By 1965, programs existed that could, in principle, solve any solvable problem described in logical notation.
- The logicist tradition within AI hopes to build on such programs to create intelligent systems.

The Wumpus world



PEAS description

- **Performance measure:**
 - +1000 for climbing out of the cave with gold;
 - -1000 for falling into a pit or being eaten by the wumpus;
 - -1 per step.
- **Environment:**
 - 4×4 grid of rooms;
 - The agent starts in the lower left square labeled $[1, 1]$, facing right;
 - Locations for gold, the wumpus and pits are chosen randomly from squares other than the start square.
- **Actuators:**
 - Forward, Turn left by 90° or Turn right by 90° .
- **Sensors:**
 - Squares adjacent to wumpus are **smelly**;
 - Squares adjacent to pit are **breezy**;
 - **Glitter** if gold is in the same square;
 - Gold is picked up by reflex, and cannot be dropped.
 - You **bump** if you walk into a wall.
 - The agent program receives the percept $[\text{Stench}, \text{Breeze}, \text{Glitter}, \text{Bump}]$.

Wumpus world characterization

- **Deterministic**: Yes, outcomes are exactly specified.
- **Static**: Yes, Wumpus and pits do not move.
- **Discrete**: Yes.
- **Single-agent**: Yes, Wumpus is essentially a part of the environment.
- **Fully observable**: No, only **local** perception.
 - This is our first example of partial observability.
- **Episodic**: No, what was observed before is very useful.

The agent needs to maintain a model of the world and to update this model upon percepts.

We will use **logical reasoning** to overcome the initial ignorance of the agent.

Exploring the Wumpus world (1)

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2	2,2	3,2	4,2
1,1 A OK	2,1 OK	3,1	4,1

(a)

A = Agent
B = Breeze
G = Glitter, Gold
OK = Safe square
P = Pit
S = Stench
V = Visited
W = Wumpus

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2	2,2 P?	3,2	4,2
1,1 V OK	2,1 A B OK	3,1 P?	4,1

(b)

(a) Percept = [None, None, None, None]

(b) Percept = [None, Breeze, None, None]

Exploring the Wumpus world (2)

1,4	2,4	3,4	4,4
1,3 W!	2,3	3,3	4,3
1,2 A S OK	2,2 OK	3,2	4,2
1,1 V OK	2,1 B V OK	3,1 P!	4,1

(a)

A = Agent
B = Breeze
G = Glitter, Gold
OK = Safe square
P = Pit
S = Stench
V = Visited
W = Wumpus

1,4	2,4 P?	3,4	4,4
1,3 W!	2,3 A S G B	3,3 P?	4,3
1,2 S V OK	2,2 V OK	3,2	4,2
1,1 V OK	2,1 B V OK	3,1 P!	4,1

(b)

(a) Percept = [Stench, None, None, None]

(b) Percept = [Stench, Breeze, Glitter, None]

Logical agents

- Most useful in non-episodic, partially observable environments.
- **Logic (knowledge-based) agents** combine:
 - A **knowledge base (KB)**: a list of facts that are known to the agent.
 - Current **percepts**.
- Hidden aspects of the current state are **inferred** using rules of inference.
- **Logic** provides a good formal language for both
 - Facts encoded as **axioms**.
 - Rules of **inference**.

```
function KB-AGENT(percept) returns an action  
  persistent: KB, a knowledge base  
               t, a counter, initially 0, indicating time  
  
  TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))  
  action ← ASK(KB, MAKE-ACTION-QUERY(t))  
  TELL(KB, MAKE-ACTION-SENTENCE(action, t))  
  t ← t + 1  
  return action
```

Propositional logic

Syntax

The **syntax** of propositional logic defines allowable **sentences**. The syntax of propositional logic is formally defined by the following **grammar**:

$$\begin{aligned} \textit{Sentence} &\rightarrow \textit{AtomicSentence} \mid \textit{ComplexSentence} \\ \textit{AtomicSentence} &\rightarrow \textit{True} \mid \textit{False} \mid P \mid Q \mid R \mid \dots \\ \textit{ComplexSentence} &\rightarrow (\textit{Sentence}) \mid [\textit{Sentence}] \\ &\mid \neg \textit{Sentence} \\ &\mid \textit{Sentence} \wedge \textit{Sentence} \\ &\mid \textit{Sentence} \vee \textit{Sentence} \\ &\mid \textit{Sentence} \Rightarrow \textit{Sentence} \\ &\mid \textit{Sentence} \Leftrightarrow \textit{Sentence} \end{aligned}$$

OPERATOR PRECEDENCE : $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow$

Semantics

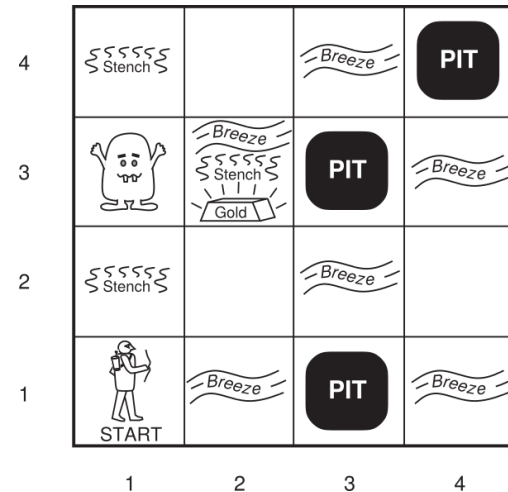
- In propositional logic, a **model** is an assignment of truth values for every proposition symbol.
 - E.g., if the sentences of the knowledge base make use of the symbols P_1 , P_2 and P_3 , then one possible model is $m = \{P_1 = \text{False}, P_2 = \text{True}, P_3 = \text{True}\}$.
- The **semantics** for propositional logic specifies how to (recursively) evaluate the **truth value** of any complex sentence, with respect to a model m , as follows:
 - The truth value of a proposition symbol is specified in m .
 - $\neg P$ is true iff P is false;
 - $P \wedge Q$ is true iff P and Q are true;
 - $P \vee Q$ is true iff either P or Q is true;
 - $P \Rightarrow Q$ is true unless P is true and Q is false;
 - $P \Leftrightarrow Q$ is true iff P and Q are both true or both false.

Wumpus world sentences

- Let $P_{i,j}$ be true if there is a pit in $[i, j]$.
- Let $B_{i,j}$ be true if there is a breeze in $[i, j]$.

Examples:

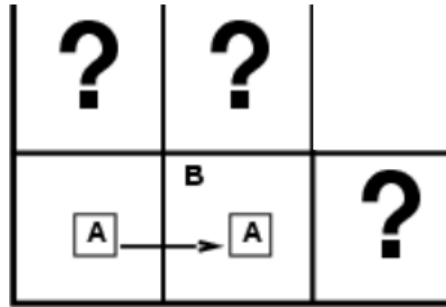
- There is no pit in $[1, 1]$:
 - $R_1 : \neg P_{1,1}$.
- Pits cause breezes in adjacent squares:
 - $R_2 : B_{1,1} \Leftrightarrow (P_{1,2} \vee P_{2,1})$.
 - $R_3 : B_{2,1} \Leftrightarrow (P_{1,1} \vee P_{2,2} \vee P_{3,1})$.
 - These are true in all wumpus worlds.
- Breeze percept for the first two squares, for the specific world we consider:
 - $R_4 : \neg B_{1,1}$.
 - $R_5 : B_{2,1}$.



Entailment

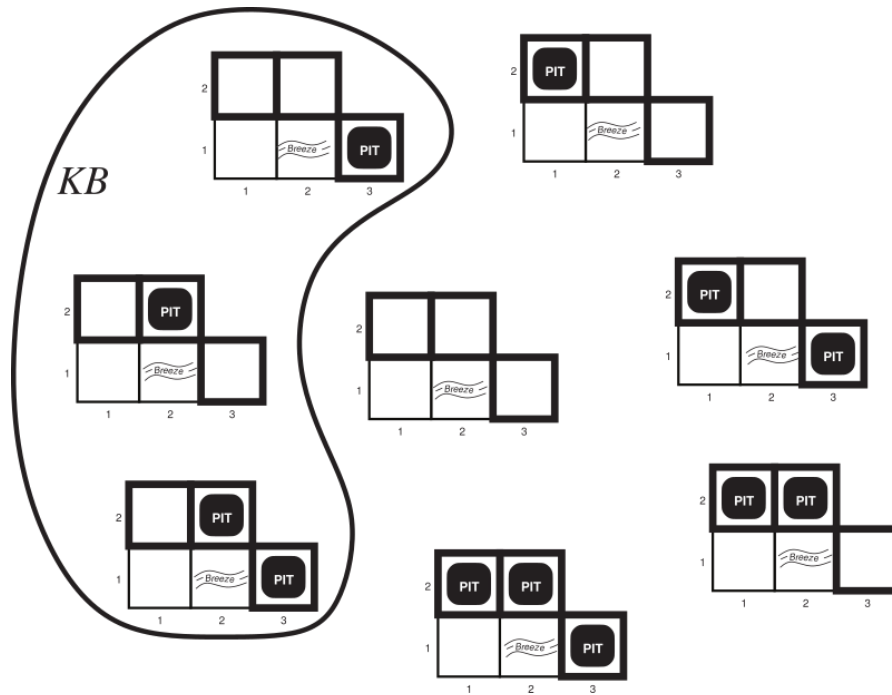
- We say a model m satisfies a sentence α if α is true in m .
 - $M(\alpha)$ is the set of all models that satisfy α .
- $\alpha \models \beta$ iff $M(\alpha) \subseteq M(\beta)$.
 - We say that the sentence α entails the sentence β .
 - β is true in all models where α is true.
 - That is, β follows logically from α .

Wumpus models



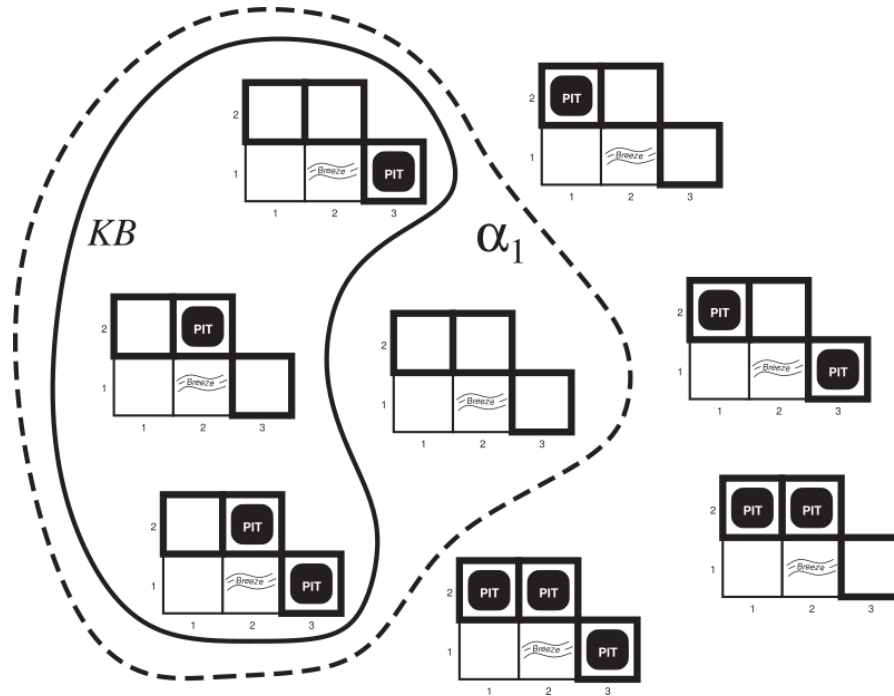
- Let consider possible models for **KB** assuming only pits and a reduced Wumpus world with only 5 squares and pits.
- We consider the situation after:
 - detecting nothing in $[1, 1]$,
 - moving right, sensing breeze in $[2, 1]$.

[Q] How many models are there?

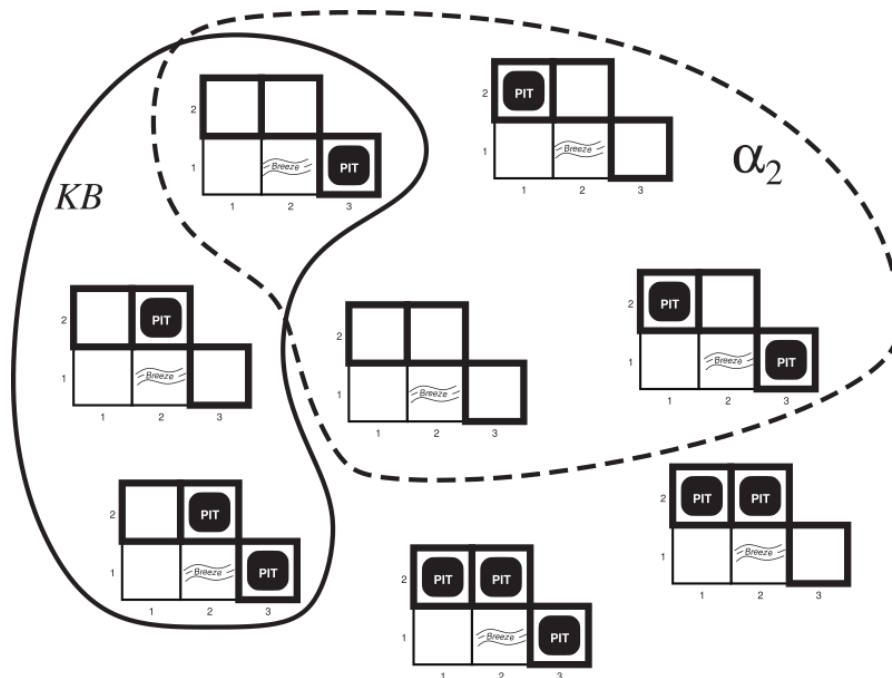


- All 8 possible models in the reduced Wumpus world.
- The knowledge base **KB** contains all possible Wumpus worlds consistent with the observations and the physics of the world.

Entailments



- $\alpha_1 = "[1, 2]$ is safe". Does KB entails α_1 ?
- $KB \models \alpha_1$ since $M(KB) \subseteq M(\alpha_1)$.
 - This proof is called **model checking** because it **enumerates** all possible models to check whether α_1 is true in all models where KB is true.
- Entailment can be used to carry out **logical inference**.



- $\alpha_2 = "[2, 2]$ is safe". Does KB entails α_2 ?
- $KB \not\models \alpha_2$ since $M(KB) \not\subseteq M(\alpha_2)$.
- We **cannot** conclude whether $[2, 2]$ is safe (it may or may not).

Unsatisfiability theorem

$\alpha \models \beta$ iff $(\alpha \wedge \neg\beta)$ is unsatisfiable

- A sentence γ is unsatisfiable iff $M(\gamma) = \{\}$.
 - i.e., there is no assignment of truth values such that γ is true.
- Proving $\alpha \models \beta$ by checking the unsatisfiability of $\alpha \wedge \neg\beta$ corresponds to the proof technique of reductio ad absurdum.
- Checking the satisfiability of a sentence γ can be cast as CSP!
 - More efficient than enumerating all models, but remains NP-complete.
 - Alternatively, propositional satisfiability (SAT) solvers can be used instead of CSPs. These are tailored for this specific problem. Many of them are variants of backtracking.

Limitations

- Representation of **informal** knowledge is difficult.
- Hard to define provable **plausible** reasoning.
- **Combinatorial explosion** (in time and space).
- Logical inference is only a part of intelligence.

Summary

- Constraint satisfaction problems:
 - States are represented by a set of variable/value pairs.
 - Backtracking, a form of depth-first search, is commonly used for solving CSPs.
 - The complexity of solving a CSP is strongly related to the structure of its constraint graph.
- Logical agents:
 - Intelligent agents need knowledge about the world in order to reach good decisions.
 - Logical inference can be used as tool to reason about the world.
 - The inference problem can be cast as the problem of determining the unsatisfiability of a formula.
 - This in turn can be cast as a CSP.

References

- Newell, A., & Simon, H. (1956). The logic theory machine--A complex information processing system. IRE Transactions on information theory, 2(3), 61-79.
- McCarthy, J. (1960). Programs with common sense (pp. 300-307). RLE and MIT computation center.