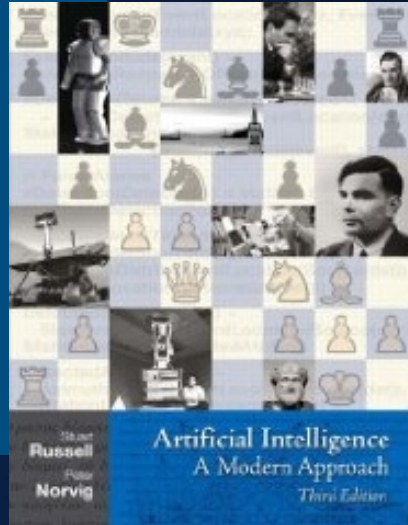


EDWARD TSANG



Foundations of Constraint Satisfaction

Edited by Thom Fruehwirth



AI Fundamentals: Constraints Satisfaction

Maria Simi



Problem solving as search

LESSON 1 - REVIEW OF PROBLEM SOLVING AS SEARCH
IN A STATE SPACE (AIMA CH 3, AI-FCA CH. 3)

Problem solving as search

The dominant approach to solving AI problems is formulating a task as search in a **state space** (in the simplest case).

The paradigm follows these steps:

1. Define a goal (a set of states, a Boolean test function ...)
2. Formulate the task as a search problem:
 - ✓ define a representation for states
 - ✓ define legal actions and transition functions
3. Find a solution (a sequence of actions) by means of a search process
4. Execute the plan

Search is a basic technique in AI

Search happens **inside the agent**: it is a planning stage before acting. It is different from searching in the world, when an agent may have to act in the world and interleave action with planning.

Search is a **general paradigm** underlying much of Artificial Intelligence.

- An agent is usually given only a description of what it should achieve, **not an algorithm to solve it**. The only possibility is to search for a solution.
- Searching can be computationally very inefficient (NP-complete).
- Humans are able to solve specific instances by using their knowledge about the problem. This extra knowledge is called **heuristic knowledge**.

Assumptions in classic “problem solving”

Problem solving agents are **goal driven** agents that work under simplified assumptions (wrt to agent's design):

1. States are treated as **black boxes**, we only need to know their “heuristic value” and whether they are a goal, by applying the Boolean goal function. From the point of view of searching algorithms their internal structure does not matter.
2. The agent has **perfect knowledge** of the state (full accessibility): it knows in which state it is, no uncertainty in sensors.
3. Actions are **deterministic**: you know the consequences of the action; the agent can plan in advance.

On the other hand:

The state space is **generated incrementally**, may not fit in memory, may be infinite. Not exactly the same as searching finite graphs.

Problem formulation and data structure

A problem is defined formally by five components

1. Initial/start state
2. Possible actions in s : $Actions(s)$
3. Transition model: a function $Result: state \times action \rightarrow state$
 $Result(s, a) = s'$, a **successor** state
3. Goal states, defined by a boolean function
 $Goal-Test(s) \rightarrow \{true, false\}$
4. *Path cost* function, that assigns a numeric cost to each path: the sum of the cost of the actions on the path $c(s, a, s')$

Note: 1, 2, 3 implicitly define a graph

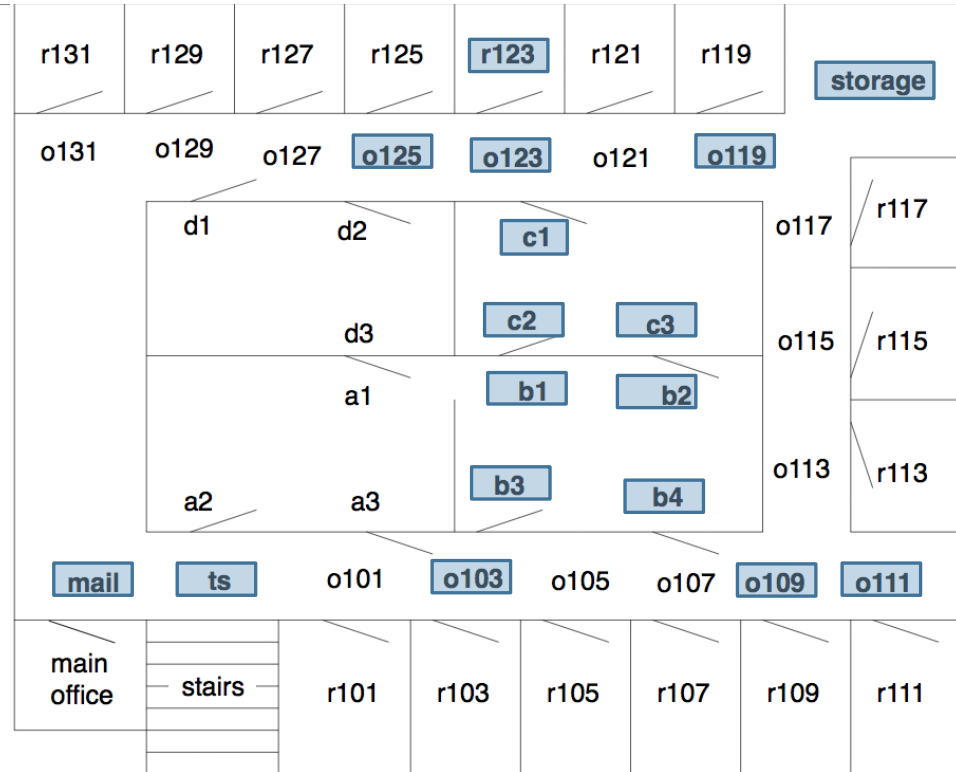
Graphs for searching

- ✓ A (directed) graph consists of a set N of nodes and a set A of arcs (ordered pairs of nodes).
- ✓ Node n_2 is a neighbor/successor of n_1 if there is an arc from n_1 to n_2 . $\langle n_1, n_2 \rangle \in A$.
- ✓ A path is a sequence of nodes $\langle n_0, n_1, \dots, n_k \rangle$ such that $\langle n_{i-1}, n_i \rangle \in A$.
The **length of path** $\langle n_0, n_1, \dots, n_k \rangle$ is k .
- ✓ The **cost of a path** is the sum of the costs of its arcs.
$$cost(\langle n_0, \dots, n_k \rangle) = \sum_{i=1}^k cost(\langle n_{i-1}, n_i \rangle)$$
- ✓ Given a set of start nodes and goal nodes, a **solution** is a path from a start node to a goal node.
- ✓ An **optimal solution** is one with minimum cost.

Path finding for the Delivery Robot

Task: from o103 to r123

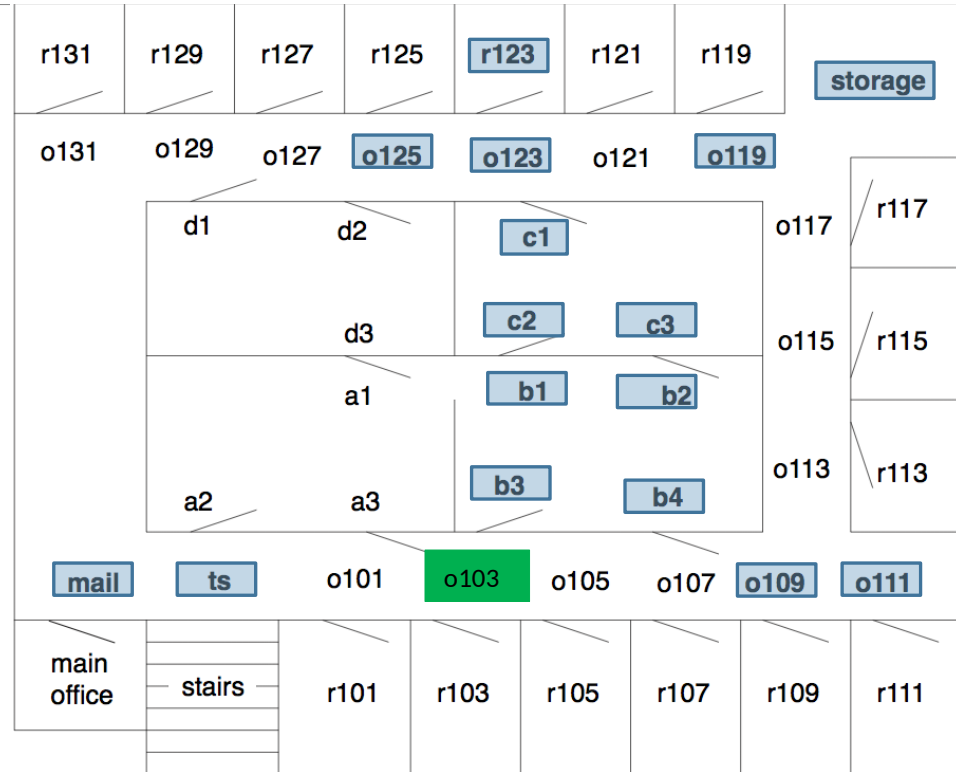
1. States



Path finding for the Delivery Robot

Task: from o103 to r123

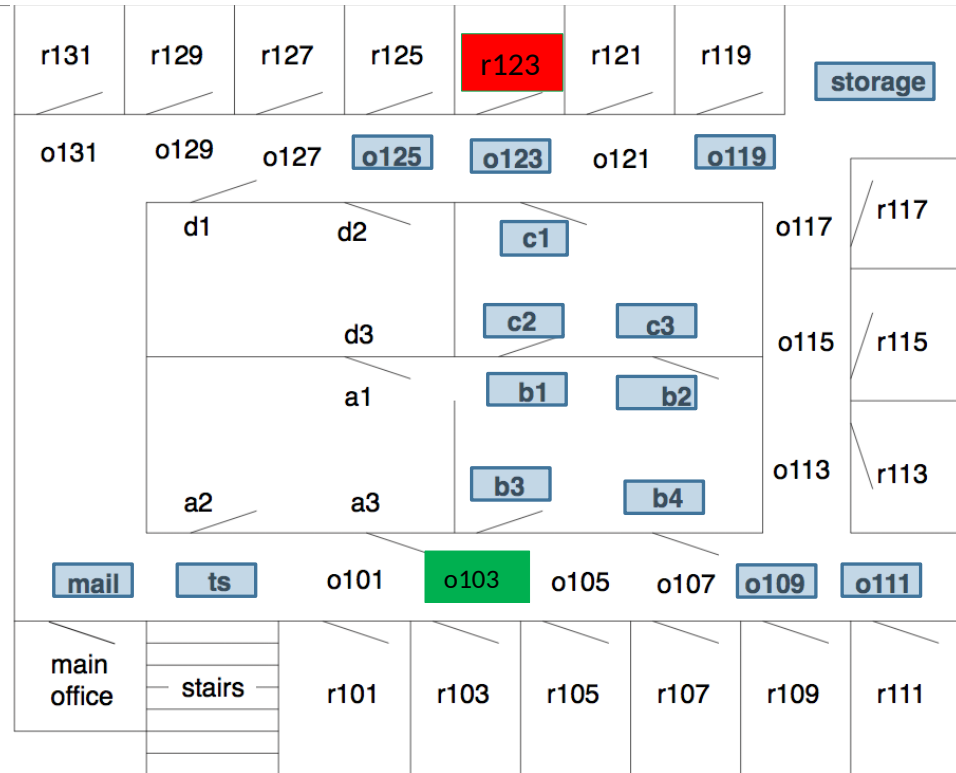
1. States
2. Initial state



Path finding for the Delivery Robot

Task: from o103 to r123

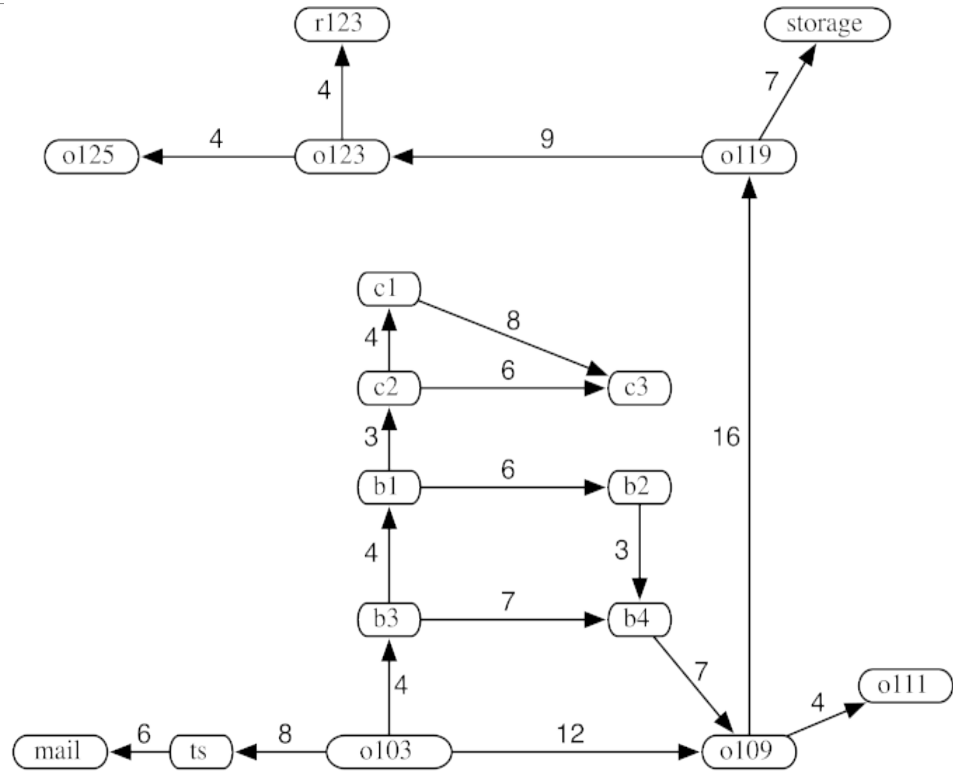
1. States
2. Initial state
3. Goal state
4. Transition model: go to adjacent room



State space for the Delivery Robot

Task: from o103 to r123

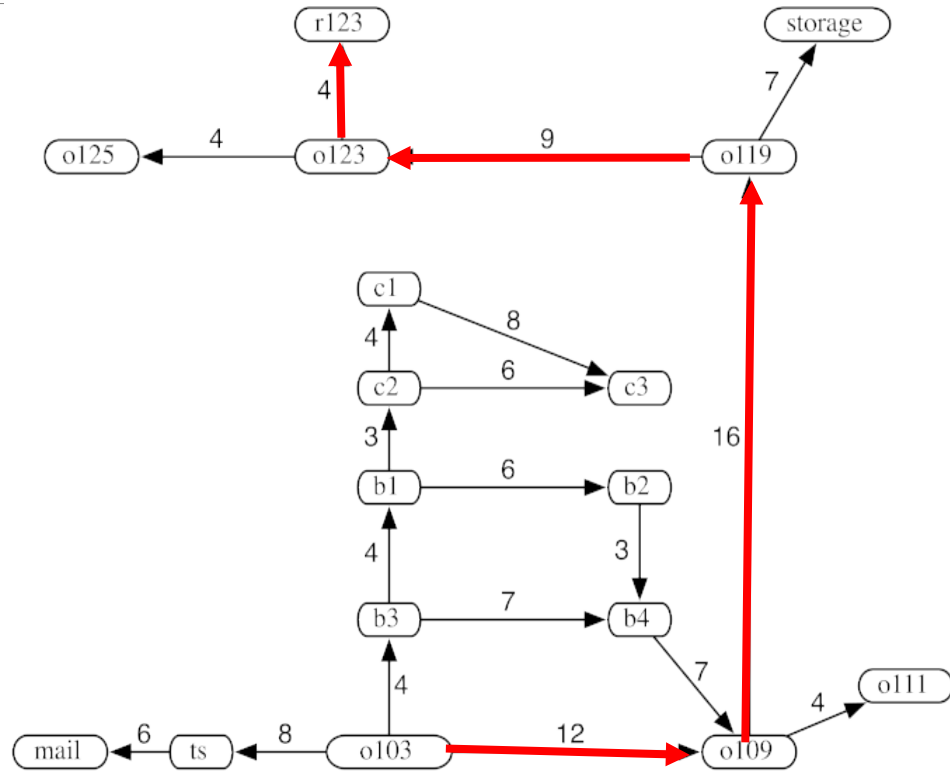
1. States
2. Initial state
3. Goal state
4. State graph



Path finding for the Delivery Robot

Task: from o103 to r123

1. States
2. Initial state
3. Goal state
4. State graph
5. Best solution



Searching algorithms

A **problem** is given as input to a **search algorithm**. A **solution** to a problem is a path (**action sequence**) that leads from the initial state to a goal state.

Solution quality is measured by the **path cost function**: an **optimal solution** has the lowest path cost among all solutions.

Different strategies (**algorithms**) for searching the state space may be characterized by ...

- their time and space **complexity, completeness, optimality** ...
- **Uninformed** search methods vs **informed/heuristic** search methods, which use an **heuristic evaluation function** of the nodes.
- Direction of search (forward or backwards)
- Global vs **local search methods**

Generic search algorithm

Input: a graph,
a set of start nodes,
Boolean procedure $goal(n)$ that tests if n is a goal node.

$frontier := \{\langle s \rangle : s \text{ is a start node}\}$

while $frontier$ is not empty:

select and **remove** path $\langle n_0, \dots, n_k \rangle$ from $frontier$

if $goal(n_k)$

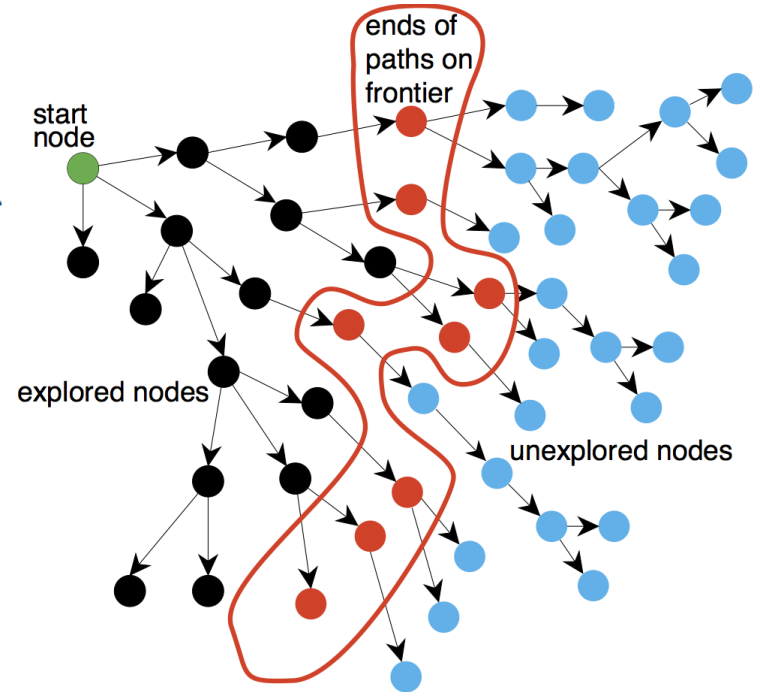
return $\langle n_0, \dots, n_k \rangle$

for every neighbor n of n_k

add $\langle n_0, \dots, n_k, n \rangle$ to $frontier$

end while

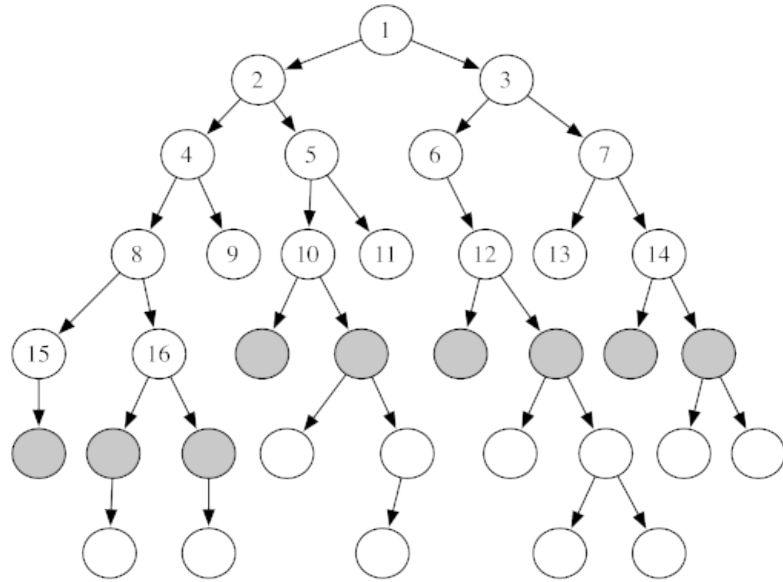
return fail



Breadth

and

Depth search



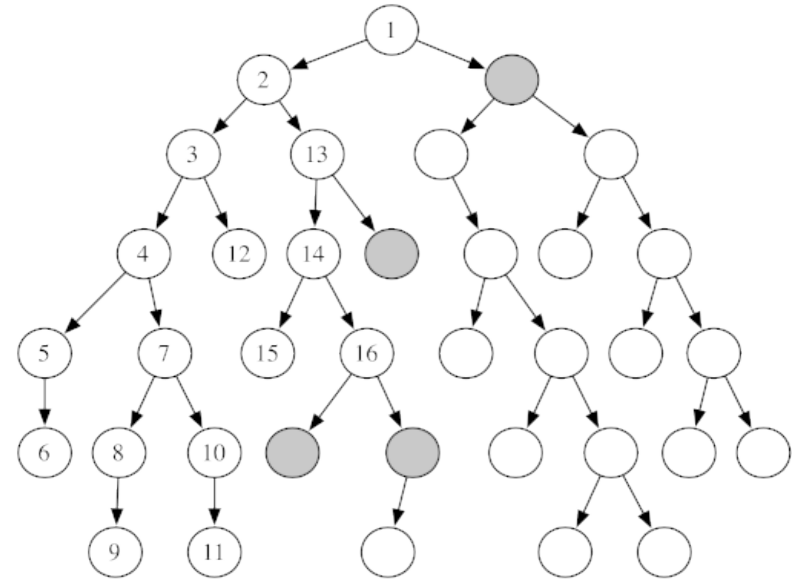
Complete, optimal

Time complexity: b^d

Space complexity: b^d

d
 m

b max num of successors
the depth of the solution
max distance of the solution



Not complete (there may be cycles)

Time complexity: b^d

Space complexity: bm

Depth bounded search and Iterative Deepening

DEPTH BOUNDED/LIMITED SEARCH

- Suppose we know the distance to the solution.
- We could perform depth-first up to a limit without giving up completeness.
- This is **Depth-Limited search** (DL)

ITERATIVE DEEPENING

- What can we do if we are not able to anticipate the distance?
- We try with depth limit 1, then 2, then 3 ... free memory from one iteration to the next!
- This is the idea of **Iterative Deepening search** (ID).
- What about complexity?

Lowest-cost search (Uniform Cost Search)

- At each stage, lowest-cost-first search selects a path on the frontier with lowest cost.
- The frontier is a priority queue ordered by path cost.
- The first path to a goal node found is a least-cost path.
- When arc costs are equal \Rightarrow breadth-first search
- This strategy is **complete** provided the branching factor is finite and there is some $\epsilon > 0$ such that all the arc costs are greater than ϵ .
- It also **optimal** since it guarantees that paths with lower costs are found first.

Heuristic search and heuristic functions

- Idea: don't ignore the goal when selecting paths.
 - ✓ Often there is extra knowledge that can be used to guide the search towards the goal: **heuristics**.
 - ✓ Heuristic is provided by means of a **heuristic function**: $h: N \rightarrow \mathbb{R}$
 - ✓ $h(n)$ is an estimate of the cost of the shortest path from node n to a goal node.
- h needs to be efficient to compute.
- An **admissible heuristic** is a non-negative heuristic function that **under-estimates** the minimum cost of a path from a node n to a goal ($h^*(n)$)
 - ✓ *Admissible heuristic*: $\forall n \ h(n) \leq h^*(n)$.
- **Best-first search** selects the most promising node on the frontier according to the heuristic function.

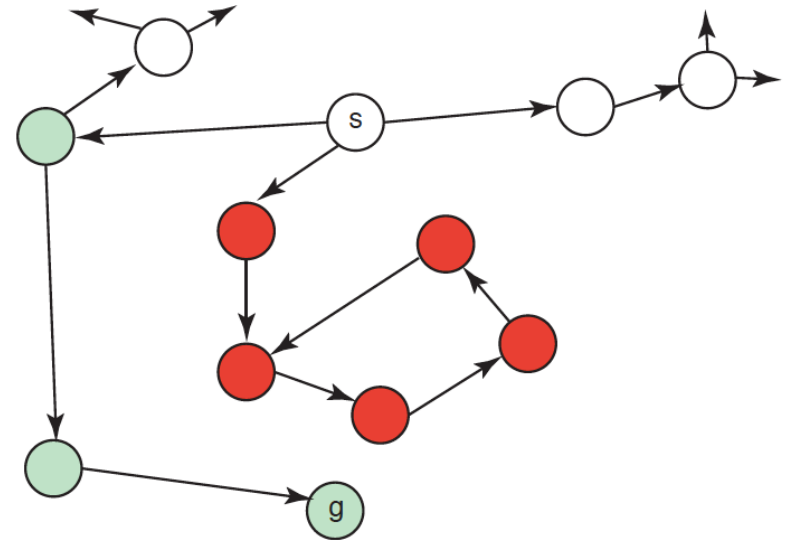
An example of best-first

In this example we use best-first with the heuristic of **straight-line distance** to the goal.

Arcs length is proportional to their cost.

The best-first algorithm is trumped to follow the red path and misses the solution.

Note: this is in fact a **greedy-best-first** strategy (we use only function h)



A* search

Heuristic function:

$$f(n) = g(n) + h(n) \quad h(n) \geq 0$$

$g(n)$ is the cost of path leading to n . ($cost(n)$)

$h(n)$ is an **admissible heuristics**

$f(n)$ estimates the total path cost of going from a start node to a goal via n

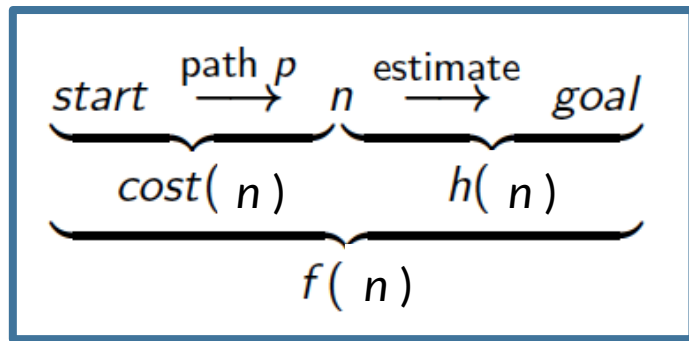
Special cases:

1. Lowest-cost search

➤ $h = 0$

2. Greedy best first

➤ $g = 0$



Properties of A*

A* is **complete** and always finds an **optimal** solution provided:

1. the branching factor is finite
2. arc costs are bounded above zero (there is some $\epsilon > 0$ such that all of the arc costs are greater than ϵ)

Memory requirement are exponential.

Optimizations are possible, when searching graphs.

They operate some sort of **graph pruning**.

This pruning strategies are embedded in the **Graph Search** algorithm of the AIMA book.

Graph pruning optimizations

- **Cycle pruning**

Do not add to the frontier nodes with states already encountered along the path.

Easy to compute.

- **Multiple-path pruning**

Implemented by maintaining an explored set (traditionally called **closed list**) of nodes that are at the end of paths that have been expanded: when n is selected if its state is already in the closed list, it is discarded.

Not so easy to achieve optimality:

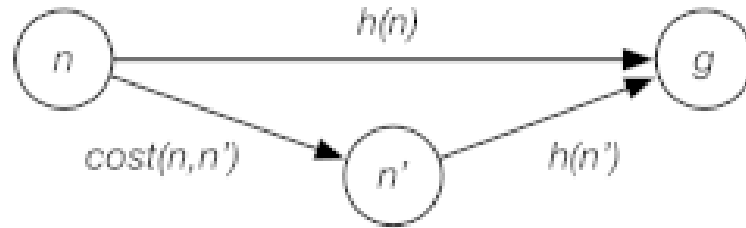
1. Whenever a better path is found to node n , discard or revise longer paths to n . Some bookkeeping is necessary (Graph Search in AIMA)
2. Ensure that the best path to node n is discovered first, so that you can easily discard other paths to n discovered later. This guarantee is provided by the property of **monotonicity** of the heuristic function h .

Consistent heuristics

- Consistency can be guaranteed if the heuristic function satisfies the **monotone restriction**:

$$h(n) \leq \text{cost}(n, n') + h(n') \quad \text{for any arc } \langle n, n' \rangle$$

Triangle inequality



- Consistency is a stronger property wrt admissibility, *consistent* \Rightarrow *admissible*
- With the monotone restriction, the f -values of the paths selected from the frontier are monotonically non-decreasing.

Strategies for saving memory in A*

Memory is a big problem in A* ($O(b^d)$). Two possible strategies:

Doing some extra work:

1. Iterative Deepening A* (IDA*) performs repeated depth-bounded searches, with value of $f(n)$ used as bound.
2. Recursive Best-first, similar to Branch&Bound
3. Simplified Memory-bounded A* (SMA*)

Giving up optimality:

4. Beam search, keeps in the frontier only the best k paths, where k is the beam width.

Summary

Strategy	Selection from Frontier	Path found	Space
Breadth-first	First node added	Fewest arcs	Exponential
Depth-first	Last node added	No	Linear
Iterative deepening	—	Fewest arcs	Linear
Greedy best-first	Minimal $h(p)$	No	Exponential
Lowest-cost-first	Minimal cost (p)	Least cost	Exponential
A^*	Minimal cost $(p) + h(p)$	Least cost	Exponential
IDA*	—	Least cost	Linear

Direction of search

When:

1. the set of goal nodes, $\{n: \text{goal}(n)\}$, is finite and can be generated
2. for any node n the neighbors of n in the **inverse graph**, namely $\{n': \langle n', n \rangle \in A\}$, can be generated

You can search **backward** from the goal.

- **Forward branching factor** and **backward branching factor** determine what is more convenient.
- **Bidirectional search** may be a good idea, it may result in a huge saving ($2b^{d/2} \ll b^d$); provided we make sure the two frontiers meet.
- Two smaller sub-problems instead of a big one (**modularization**).
- Even better if we can identify m states the search must pass through (**islands in island driven search**)

Dynamic programming

Dynamic programming is a general method for optimization that involves **storing partial solutions to problems**, so that a solution that has already been found can be retrieved in a table rather than recomputed.

Dynamic programming for graph searching can be seen as constructing the *perfect* heuristic function in advance so that, by keeping only one element of the frontier, it is guaranteed to find an optimal solution. It is based on a **pre-computed table** of the lowest $cost_to_goal(n)$ value for each node.

The main limitations of dynamic programming are that:

- it only works when the **graph is finite** and the **table small enough** to fit into memory,
- an agent must re-compute a policy for each different goal.

Dynamic programming algorithms are not typical of AI but are used throughout AI.

Constraints Satisfaction Problems

LESSON 1 - AN INTRODUCTION AND PROBLEM
FORMULATION



Searching vs constraint satisfaction

It is often better to describe states in terms of **features** and then to reason in terms of these features. We have called this a **factored representation**.

This representation may be **more natural and efficient** than explicitly enumerating the states. With 10 binary features we can describe $2^{10}=1024$ states.

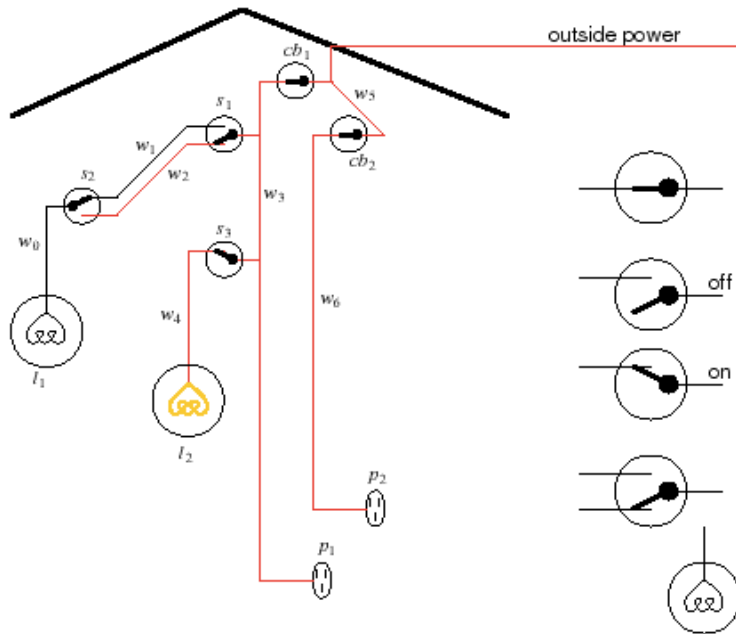
Often these features are not independent and there are **constraints** that specify legal combinations of assignments of values to them.

The mind discovers and **exploits constraints** to solve tasks. For many important problems this strategy can make problem solving more efficient.

Constraint satisfaction is about **generating assignments** that satisfy a set of hard constraints and how to optimize a collection of soft constraints (**preferences**).

A well-established theoretical framework and a set of techniques spanning a large number of applications.

A feature representation for states



Electrical circuit in a house

- each switch is a feature whose position can be on or off.
- each light is a feature that can be lit or no
- each component can work properly or be broken.

A state consists of the position of every switch, the status of every device, and so on, i.e. an assignment of a value to each feature.

For example, a state may be described as switch 1 is on, switch 2 is off, fuse 1 is okay, wire 3 is broken, ... the role of constraints.

What is a constraint satisfaction problem?

A CSP is a problem composed of

- a finite set of **variables**,
- each variable is associated with a **finite domain**
- and a **set of constraints** that restrict the values the variables can simultaneously take.

The task is to assign a value (from the associated domain) to each variable satisfying all the constraints

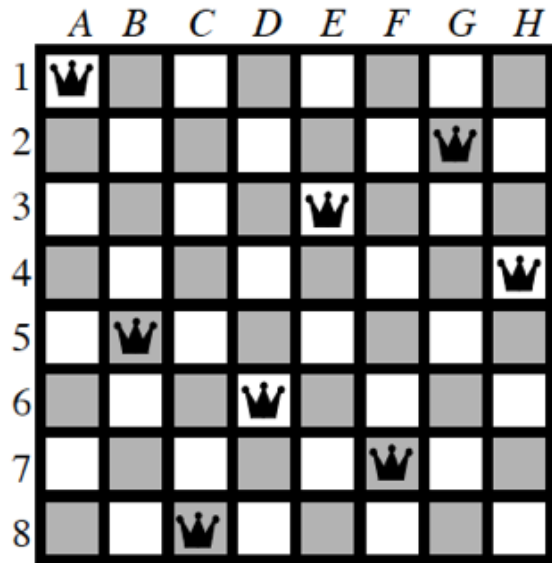
The problem is NP hard in the worst cases but general heuristics exist, and structure can be exploited for efficiency.

Map coloring problem



The task is to color regions on the map with three colors in such a way that no two adjacent regions get the same color.

The N -queens problem



The problem is to place N queens on N different squares on a $N \times N$ chess board, satisfying the constraint that no two queens can threaten each other.

Remind: a queen can threaten other queens on the same row, column or diagonal.

The figure shows a solution to the 8-queens problem

Using constraints to solve problems

1. Informal problem description
2. Formalization of a constraint satisfaction problem by defining the set of variables, their domains and all the relevant constraints
3. Choose a model and an algorithm for solving the problem (or use constraint solving packages)
4. Implementation and actually solving the problem

There are many different ways to formulate the same problem; efficiency may vary

Different models and approaches: systematic search, repair/stochastic approach?

CSP: a formal *definition*

A Constraint Satisfaction Problem consists of three components, X , D , and C

$$CSP = \langle X, D, C \rangle$$

1. X is a set of variables, $\{x_1, \dots, x_n\}$
2. D is a set of domains, $\{D_1, \dots, D_n\}$, one for each variable.

The **domain** of a variable x is a set of all possible values $\{v_1, \dots, v_k\}$ that can be assigned to the variable x

Dom = a function which maps every variable in X to a set of objects of arbitrary type

$Dom(x) = D_x$ is the domain of x , the set of possible values for x

Domains can be numerical (integer or real numbers), boolean, symbolic ...

3. C is a set of constraints that specify allowable combinations of values

CSP: assignment

A **[partial] assignment** of values to a set of variables (also called **compound label**) is a set of pairs:

$$A = \{ \langle x_i, \nu_i \rangle \langle x_j, \nu_j \rangle \dots \}$$

where values are taken from the variable domain: $\nu_i \in D_{x_i}$

A **complete assignment** is an assignment to all the variables of the problem (a possible world)

A complete assignment can be **projected** to a smaller partial assignment by restricting the variables to a subset. We will use this notation for the projection:

Where π is the projection operator of relational algebra

CSP: constraints representation

A **constraint on a set of variables** is a set of possible assignments for those variables

Each constraint C can be represented as a pair $\langle scope, rel \rangle$

- $scope$ is a tuple of variables participating in the constraint: (x_1, x_2, \dots, x_k)
- rel is a relation that defines the allowable combinations of values for those variables, taken from their respective domains

The relation can be represented as:

- an explicit list of all tuples of values that satisfy the constraint, **explicit** relation.
- an implicit relation, an object that supports two operations: (1) testing if a tuple is a member of the relation and (2) enumerating the members of the relation.

We will also use $C_{x_1, x_2, \dots, x_k} = rel$ to denote a constraint on the scope variables x_1, x_2, \dots, x_k : i.e. the constraint $C = \langle (x_1, x_2, \dots, x_k), rel \rangle$

Examples of constraint

Example 1

If x_1 and x_2 both have the domain $\{A, B\}$, the constraint that the two variables must have different values can be written as

$\langle (x_1, x_2), \{(A, B), (B, A)\} \rangle$ by explicit enumeration $C_{x_1, x_2} = \{(A, B), (B, A)\}$

$\langle (x_1, x_2), x_1 \neq x_2 \rangle$ by implicit definition

$$C_{x_1, x_2} = \{(v_1, v_2) \mid v_1 \in D_{x_1}, v_2 \in D_{x_2}, v_1 \neq v_2\}$$

Example 2 (ternary constraint)

$$X = \{a, b, c\} \quad D_a = D_b = D_c = \{1, 2, 3, 4, 5, 6\}$$

$$C_{a,b,c} = \{(1, 2, 3), (1, 4, 3), (4, 5, 6)\} =$$

$$\{ \langle a, 1 \rangle, \langle b, 2 \rangle, \langle c, 3 \rangle \rangle, (\langle a, 1 \rangle \langle b, 4 \rangle \langle c, 3 \rangle), (\langle a, 4 \rangle \langle b, 5 \rangle \langle c, 6 \rangle) \}$$

Constraint satisfaction

Satisfies is a binary relationship between an **assignment** and a **constraint**:

$$\begin{aligned} \text{Satisfies}(\{\langle x_1, \nu_1 \rangle \ \langle x_2, \nu_2 \rangle \ \dots \ \langle x_k, \nu_k \rangle\}, C_{x_1, x_2 \dots x_k}) &\equiv \\ \{\langle x_1, \nu_1 \rangle \ \langle x_2, \nu_2 \rangle \ \dots \ \langle x_k, \nu_k \rangle\} &\in C_{x_1, x_2 \dots x_k} \end{aligned}$$

Example:

$$\text{Satisfies}(\{\langle x_1, A \rangle, \langle x_2, B \rangle\}, \langle (x_1, x_2), \{(A, B), (B, A)\} \rangle)$$

$$\text{Satisfies}(\{\langle x_1, B \rangle, \langle x_2, A \rangle\}, \langle (x_1, x_2), x_1 \neq x_2 \rangle)$$

$$\text{NOT Satisfies}(\{\langle x_1, B \rangle, \langle x_2, B \rangle\}, \langle (x_1, x_2), \{(A, B), (B, A)\} \rangle)$$

CSP: solution

To solve a CSP problem $\langle X, D, C \rangle$, seen as a search problem, we need to define a **state** space and the notion of a **solution**.

A **state** in a CSP is an **assignment** of values to some or all of the variables

Partial/complete assignment:

- A **partial** assignment is one that assigns values to only some of the variables.
- A **complete** assignment is one in which every variable is assigned

Consistent assignment: one that satisfies all the constraints

- *Satisfies*($\{\langle x_1, v_1 \rangle \langle x_2, v_2 \rangle \dots \langle x_k, v_k \rangle\}, C_{x_1, x_2 \dots x_k}$) for any constraint in C

A **solution** to a CSP (a goal state) is a **consistent, complete** assignment.

Types of constraints

The simplest kind of CSP involves variables that have **discrete**, **finite** domains

- Values can be numbers, strings, Booleans (True, False)

When variables are numbers, and the constraints are inequalities we can deal with variables with infinite domains or continuous domains with linear or integer programming (techniques used in Operations research).

According to the number of variables involved constraints can be:

- unary (ex. “x even”)
- binary (ex. “ $x \neq y$ ”)
- higher-order constraints (ex. $x+y = z$)

Absolute/hard vs **soft/preference** constraints

- CSPs with preferences can be solved by optimization methods. These are called Constraint Optimization Problems, or COP.

Problem formalization: examples

Map coloring

The N -queens problem

Scheduling

Car sequencing

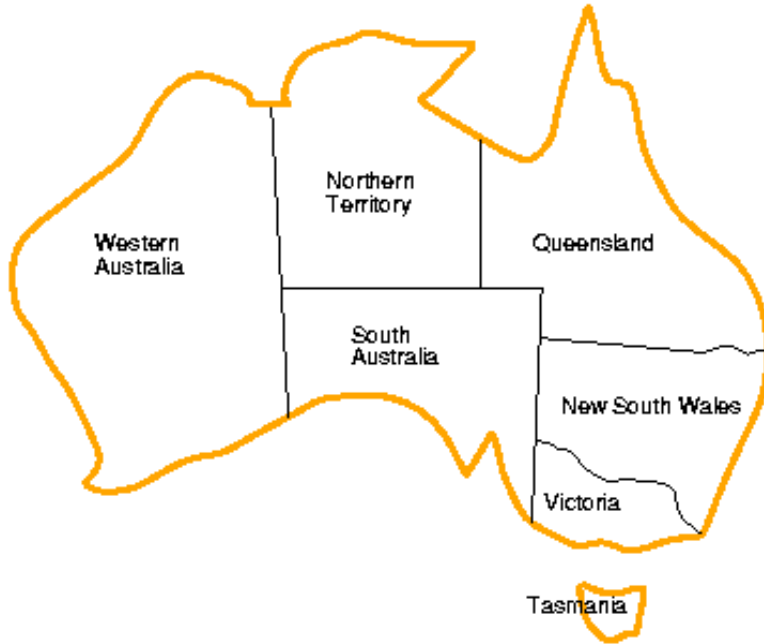
Scene labelling in vision

Temporal reasoning in planning

Subgraph matching in semantic networks

... other

Map coloring



Variables: WA, NT, SA, Q,
NSW, V, T

Domains: {red, green, blue}

Constraints: $WA_{L\bar{0}} NT$, $WA_{L\bar{0}} SA$,
 $NT_{L\bar{0}} Q$, $SA_{L\bar{0}} Q$, $SA_{L\bar{0}} NSW$,
 $SA_{L\bar{0}} V$, $NSW_{L\bar{0}} V$

The 8-queens problem: explicit constraints

A queen for each column means 8 variables instead of 64 of a naïve formulation.

$$X = \{Q_1, Q_2, \dots, Q_8\}$$

$$D_{Q_1} = D_{Q_2} = \dots = D_{Q_8} = \{1, 2, 3, 4, 5, 6, 7, 8\}$$

$$C = \{ \langle (Q_1, Q_2), \quad \text{scope} \\ \{(1, 3), (1, 4), (1, 5), (1, 6), (1, 7), (1, 8), rel \\ (2, 4), (2, 5), (2, 6), (2, 7), (2, 8), (3, 2), (3, 5), (3, 6), (3, 7), (3, 8), \\ (4, 1), (4, 2), (4, 6), (4, 7), (4, 8), (5, 1), (5, 2), (5, 3), (5, 7), (5, 8), \\ (6, 1), (6, 2), (6, 3), (6, 4), (6, 8), (7, 1), (7, 2), (7, 3), (7, 4), (7, 5), \\ (8, 1), (8, 2), (8, 3), (8, 4), (8, 5), (8, 6)\} \rangle, \\ \langle (Q_1, Q_3), \dots \rangle, \langle (Q_1, Q_4), \dots \rangle, \langle (Q_1, Q_5), \dots \rangle \langle (Q_1, Q_6), \dots \rangle \}$$

The 8-queens problem: implicit constraints

A queen for each column.

$$X = \{Q_1, Q_2, \dots, Q_8\}$$

$$D_{Q_1} = D_{Q_2} = \dots = D_{Q_8} = \{1, 2, 3, 4, 5, 6, 7, 8\}$$

1. *not the same row*

$$\forall i, j. Q_i \neq Q_j$$

2. *not the same diagonal: the horizontal distance must be different from the vertical distance*

$$\forall i, j. \text{ if } Q_i = a \wedge Q_j = b \text{ then } |i - j| \neq |a - b|$$

| [Tsang]

Easy to write a function to check for constraint.

	<i>i</i>			<i>j</i>			
<i>a</i>				X			
	O			X			
				X			

Job-Shop scheduling [AIMA]

Scheduling the assembling of a car is a job requiring several tasks; for example installing axles, installing wheels, tightening nuts, put on hubcap, inspect.

- X initial times of the tasks to be performed
- D finite number of times in an interval (minutes)
- C temporal constraints among tasks
 - ✓ *Precedence constraints: i must be completed before j begins*
 $X_i + d_i < X_j$ where d_i is the duration of task i
 - ✓ *Disjunctive constraints between i and j , not overlapping in time (two workers and one tool)*
 $X_i + d_i < X_j$ or $X_j + d_j < X_i$
 - ✓ *Global duration of the planned assembly jobs.*
Limitation on the interval of time considered.

Job-shop scheduling: example

$X = \{AxleF, AxleB, WheelRF, WheelLF, WheelRB, WheelLB, NutsRF, NutsLF, NutsRB, NutsLB, CapRF, CapLF, CapRB, CapLB, Inspect\}$

Precedence constraints:

$$\begin{array}{lll} AxleF + 10 \leq WheelRF & WheelRF + 1 \leq NutsRF & NutsRB + 2 \leq CapRB \\ AxleF + 10 \leq WheelLF & WheelLF + 1 \leq NutsLF & NutsRF + 2 \leq CapRF \\ AxleB + 10 \leq WheelRB & WheelRB + 1 \leq NutsRB & NutsLF + 2 \leq CapLF \\ AxleB + 10 \leq WheelLB & WheelLB + 1 \leq NutsLB & NutsLB + 2 \leq CapLB \end{array}$$





Disjunctive constraint:

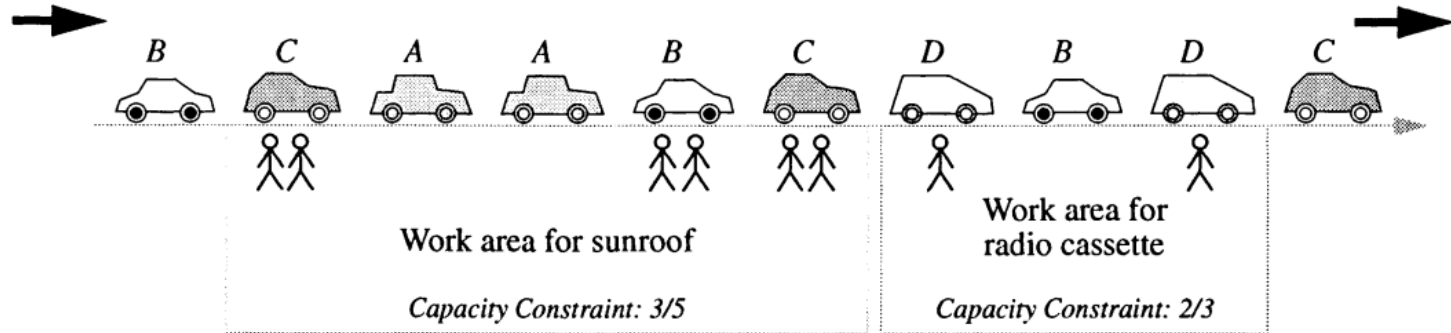
$$(AxleF + 10 \leq AxleB) \text{ or } (AxleB + 10 \leq AxleF)$$

$$X + d_X \leq Inspect \quad D_i = \{1, 2, 3, \dots, 27\} \quad \text{Allowed time: 30 minutes}$$

The car sequencing problem [Tsang]

Production Requirements:

	Model A	Model B	Model C	Model D	
					
Options (✓ = required, × = not):					
Sunroof	×	✓	✓	×	
Radio cassette	✓	×	✓	✓	
Air-conditioning	✓	✓	×	✓	
Anti-rust treatment	×	✓	✓	✓	
Power brakes	✓	×	✓	×	
Number of cars required:	30	30	20	40	Total: 120



The car sequencing problem: description

Cars are placed on conveyor belts which move through different work areas.

Work areas specialize to do a particular job: fitting sunroofs, installing car radios or air-conditioners. Jobs to be done **depend on the model**.

When a car enters a work area, a team of engineers travels with the car while working on it. They must have enough time to finish the job while the car is in their work area.

Each work area has **capacity constraints**, for example if the number of teams fitting sunroofs is 3, they cannot deal with more than 3/5 cars.

A car-sequencing problem is specified by the **production and option requirements** and the **capacity constraints**. The output is a proper sequencing of the types of cars to be produced.

Car sequencing formulation: hint

The car-sequencing problem can be formulated as a CSP in the following way.

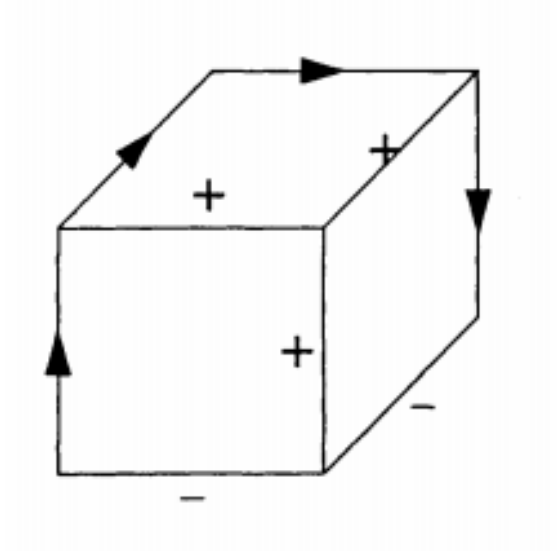
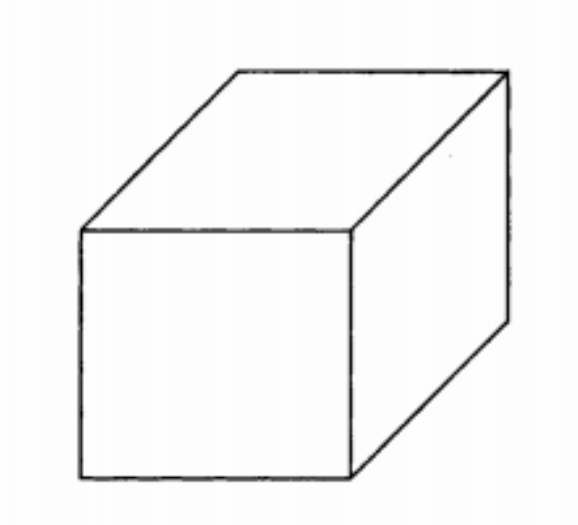
n variables: one for each car position on the conveyor belt

The domain of each variable is the set of car models: {A, B, C, D}

The task is to assign a value (a car model) to each variable (a position in the conveyor belt), satisfying both the production requirements and capacity constraints.

Scene labelling [Tsang]

A problem in computer vision



The scene labelling problem

[Tsang]

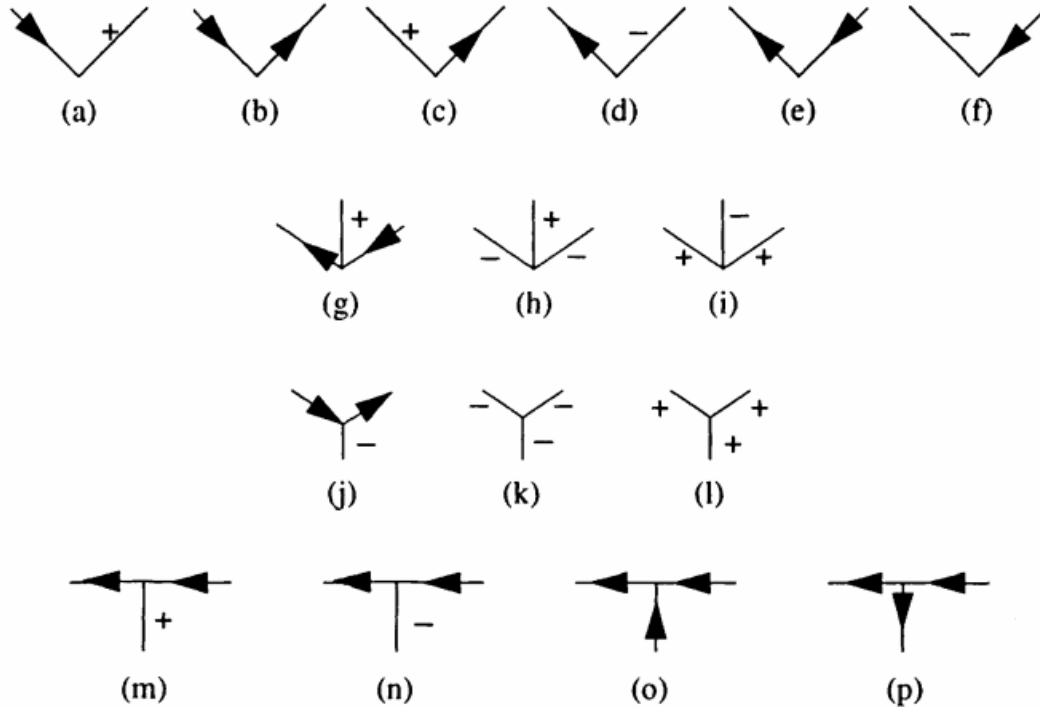
To recognize the objects in the scene, one must first interpret the lines in the drawings. One can categorize the lines in a scene into the following types:

- (1) **convex edges** “+”
- (2) **concave edges** “- ”
- (3) **occluding edges**

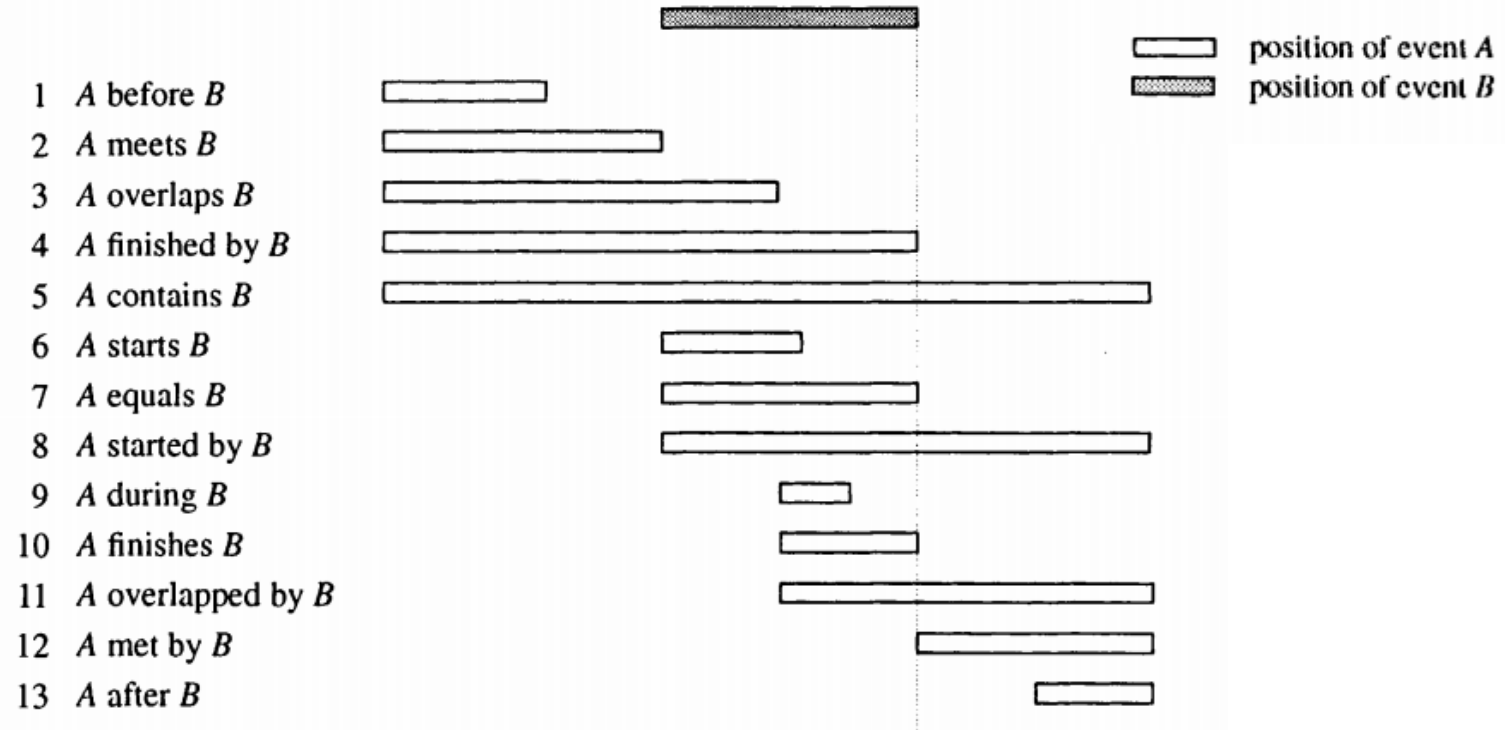
An occluding edge is a convex edge where one of the planes is not seen by the viewer. Occluding edges are marked with arrows according to whether moving in the direction of the arrow the visible plane is to right “→” or the left “←”.

Constraints are dictated by physical constraints.

Legal labels for junctions



Temporal reasoning



CSP solving techniques: an overview

Problem reduction/Inference/Constraint propagation

- Techniques for transforming a CSP into an equivalent one which is easier to solve or recognizable as insoluble (removing values from domains and tightening constraints).

Searching

- Search in the space of labels: enumerate combinations of labels to find solutions.
- How to search efficiently: heuristics, intelligent backtracking ... local search

Exploiting the structure of the problem

- Independent sub-problems, tree-structured constraints, tree-decomposition, exploiting symmetry

Problem characteristics

- Number of solutions required (one / all)
- Problem size (n. of variables and constraints)
- Type of variables and constraints
- Structure of the constraints graph (connectivity, tree form ...)
- Tightness of the problem (measured by the number of solution tuples over the number of all distinct compound labels for all variables)
- Quality of solutions (preference among solutions, COP)
- Partial solutions (if no solution exist, find “best” partial solution)

Conclusions

- We reviewed the basics of the paradigm of problem solving as search,
- We introduced CSP and the language to formally define them.
- Several examples of problems that can be formalized according to this model.
- Types of problems and characteristics that influence the solution.

Next, the techniques for solving CSP:

1. Problem reduction techniques
2. Making the search more efficient
3. Exploiting the problem structure

Your turn

1. Review on the AIMA book what is not clear about problem solving, and heuristic search, A^* in particular.
2. Complete activity Your turn 3

References

Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach* (3rd edition). Pearson Education 2010 [Cap 6 – CSP]

Edward Tsang, *Foundations of Constraints Satisfaction*.

Handbook of Constraint Programming, Edited by F. Rossi, P. van Beek and T. Walsh. Elsevier 2006.