IAI – Introduction to Artificial Intelligence

Graph Search

Each **node in the graph represents a state of the problem**, and can have **zero to any number of children**, each of which is reachable by **following an edge, which represents carrying out an action** within the problem domain.

Solutions are found by starting at the **root node (initial state)** and expanding its children. The collection of expanded but unexplored exterior nodes is called the **frontier or open list**.

States that have been visited and explored are part of the **closed list**.

Nodes from the **frontier** are selected and expanded until a **goal node** is found. The path of edges/actions required to reach that node represent a **solution**.

Graph Search Pseudo Code

```
GRAPH-SEARCH (problem):
      frontier = initial state
      closed list = empty
      loop:
            if the frontier is empty then return failure
            choose a leaf node and remove it from the frontier
            if the node is a goal then return the solution
            add the node to the closed list
            expand the chosen node
            for all children:
                  if it is not in the frontier or closed list
                  then add it to the frontier
```

But how to choose that leaf node? That's where search algorithms are used…

Search Algorithms

Properties of algorithms:

- **Completeness**: is the algorithm guaranteed to find a solution if one exists?
- **Optimality**: does the algorithm find the optimal solution?
- **Time complexity**: how long is needed to complete the search?
- **Space complexity**: how much memory is used to complete the search?

Uninformed Search Algorithms

Depth-First Search (DFS)

- It is **complete** iff the search space is **finite**, and the algorithm backtracks when a **loop** is encountered.
- Algorithm:
  - Generate the first child of the current node.
  - Expand it.
  - Repeat until a leaf is reached.
  - Backtrack to the most recent node with unexplored children and continue.

Breadth-First Search (BFS)

- It is **complete.**
- It is **optimal** IF all actions have the same cost.
- Is it **complex**: if the goal is at level d and the branching factor is b, the complexity is $O(b^d)$.
- Algorithm:
  - Generate all children of the current node
  - Repeat the process children in the order they were generated

Heuristic (Informed/Guided) Search Algorithms

Nodes are selected for expansion based on an **evaluation function**, $f(n)$.

The evaluation functions are a **cost estimate** from the current state to the goal.

The node with the **lowest evaluation** is usually preferred.

The choice of the function $f(n)$ depends on the search strategy.

There are two popular algorithms: **greedy best-first search** and **A* search**.

Greedy Best-First Search

$f(n) = h(n)$

$h(n)$ is a **heuristic function:**

- $h(n)$ = the estimated cost of the **cheapest path from n to a goal state.**
- If n is the goal node, then $h(n) = 0$.
- Example: if the problem is about connected cities, $h(n)$ might be the straight line distance from n to the given state.

A Search*

$f(n) = h(n) + g(n)$

$h(n)$ is a **heuristic function, as explained above**.

$g(n)$ is the cost from the initial state to n.

Admissible and Optimal Heuristics

An **admissible** heuristic **never over-estimates** the distance to the goal, so the optimal distance is always $\geq$ the heuristic value. Example: straight-line distance is always admissible.

$$\text{Admissible: } h(s) \leq h^*(s)$$

Given an admissible heuristic, **A* will find optimal** solutions.

The key challenge: finding a heuristic that is both **admissible** and **informative**.

Creating Good Heuristics

Heuristics are often created by considering a **simplified (relaxed)** version of the problem.

- **Constraints are dropped** to make the problem easier to solve.
- The relaxed problem should **add edges (operators/actions)** to the state space (problem domain).

- An optimal solution to the problem is also a solution to the relaxed problem.
- The **cost of an optimal solution** to a relaxed problem is an **admissible heuristic** for the real problem.

A heuristic must be easy to compute, as is has to be calculated for every node. More on heuristics later.

Programs and Solvers

Using the right data structures and heuristics, programs can be written to solve **any number of problems**.

Intelligence is about solving **new problems**.

Solvers are types of **general program** that can solve an **infinite collection of problems**, not anticipated by the programmer.

There are many types of solver:

- SAT/CSP
- Planning
- Planning with Feedback
- Game Playing
- etc.

Our focus is on **SAT** and **planning**.

Propositional Logic, SAT and CSP

Propositional Logic

Propositional logic is the exact same as it was in ELA:

- Formal language
- Formal semantics
- Proof theory
- Syntactic trees
- Semantics and validity
- Truth valuations (true/false for each symbol)
- Satisfaction (a valuation that makes all formula in a set)

One small quirk: ⊃ is sometimes used in place of → to show implication.

Types of Proof Theory

- Brute force:
  - Check every permutation – requires n^2 checks for a formula with n symbols.
- Axiomatic systems:
  - Based on a few axiom schemas and **one or two rules of inference**.
  - Derivations are often long and unnatural.
- Natural deduction:
  - Based **entirely on rules of inference**.
  - Can be done by hand, but hard to control automatically.
- Resolution:
  - Based on a **single rule of inference** (the resolution rule) that works on clauses only.
  - **A clause is a disjunction of possibly negated atoms.**

Resolution Rule

This rule takes two clauses, one that contains the literal p and one that contains the complement $\neg p$, and produces a third clause, as so:

$$\text{If } p \vee C \text{ and } \neg p \vee D, \text{ then } C \vee D.$$

This is resolution on p, that produces the resolvent $C \vee D$.

Resolution is **refutation complete**, meaning that a set of clauses is **unsatisfiable** if and only if the **empty clause** $\square$ is derivable by repeated application of the resolution rule.

Thus, $A_1, \ldots, A_n \vdash B$ if and only if an empty clause is derivable from the resolution of clauses encoding $A_n, \ldots, A_n$ and $\neg B$.

Unit Resolution

Unit resolution is a specific form of the resolution rule, using only single atoms (e.g. a, $\neg b$, c, etc.).

The aim of unit resolution is to apply the following rhetoric to each clause: **"If we assume that the selected unit is true, what can we tell about the rest of these clauses?"**

There are several outcomes to this:

- The clause is fully resolved, because the selected unit guarantees the whole clause will be true (lines 1, 3, 5)
- The clause is shortened, because you've removed the selected unit but other atoms are needed to decide if the clause will be true or not (lines 2, 6)
- The empty clause □ is reached, because the selected unit means that there is no way that clause can be true (line 4)

	The unit…	**…when applied to….**	**…results in**
1	a	a	Fully resolved
2	a	$\neg a \vee b$	b
3	a	$a \vee b$	Fully resolved
4	$\neg a$	a	□ (empty clause)
5	$\neg a$	$\neg a \vee b$	Fully resolved
6	$\neg a$	$a \vee b$	b

Explained:

1. Simple.
2. We assume a is true, so we know $\neg a$ in the clause won't be true, so we can remove that. The clause could still be satisfied by b though, so we keep it.
3. We assume a is true, and that's enough to satisfy the clause.
4. Simple.
5. We assume $\neg a$ is true, and that's enough to satisfy the clause.
6. We assume $\neg a$ is true, so we know a in the clause won't be true, so we can remove that. The clause could still be satisfied by b though, so we keep it.

SAT Solvers

SAT is the problem of **determining whether a set of clauses is satisfiable**, and if so, determining a satisfying valuation.

Many problems can be mapped into SAT.

SAT is **intractable**, but very large problems can still be solved.

The best SAT algorithms (like **DPLL**) use a **mix of case analysis and resolution**.

SAT is also used in planning, as discussed later.

DPLL Procedure for SAT

(Side note: DPLL is a misnomer. The algorithm is the work of Davis, Logemann and Loveland, so it's actually called the DLL algorithm, but Putnam is often mistakenly credited, resulting in DPLL. It's referred to as DPLL throughout the course, so I'm going with that here.)

DPLL is **sound** and **complete**.

DPLL uses resolution in a restricted form, called **unit resolution**.

- In this form, one parent clause is the unit clause.

Unit resolution is very **efficient** (runs in linear time) but **not complete**.

When the unit resolution gets stuck, DPLL picks an undetermined variable and **splits the problem** in two:

- One where the variable must be true.
- One where the variable must be false.

Pseudo code for DPLL:

```
DPLL( clauses )
      UnitResolution( clauses )
      if there is an empty clause in clauses
            return false
      else if all variables are determined
            return true
      else
            pick a non-determined variable V
            return DPLL( clauses with V ) OR DPLL( clauses  with ¬V )
```

Constraint Satisfaction Problems (CSP)

CSPs are a generalisation of SAT. In CSPs, the main task is to find an assignment of variables that satisfies a set of constraints.

Variables do not need to be Boolean, and constraints do not need to have clauses.

- Other finite domains and constraints are accepted.

CSP problems can be mapped to SAT, but it is easier to use a version of DPLL that has a richer CSP context. It is not easy to express some CSP constraints in SAT.

Planning

A key problem for intelligent autonomous agents is working out **what to do next**.

There are three approaches to this problem:

- **Program based**: specify what to do by hand.
- **Learning based**: learn what to do from experience.
- **Model based**: specify a problem by hand and derive control automatically.

Planning is a model-based approach to autonomy.

Models for Classical Planning

A **planner** accepts a problem and automatically computes the solution.

The problem must encode the following in a **domain-independent** language:

- What operators and sensors are available, and how they work.
- What is the initial situation, and what is the goal.

The simplest case is called **classical planning**:

- The initial state is fully known.
- Operations have deterministic effects.
- No sensing is needed.

A **classical planning model** contains:

- A finite and discrete **state space** S
- A known **initial state**, $s_0 \in S$
- A set of **goal states**, $S_G \subseteq S$
- A deterministic state-**transition function**, $s' = f(a, s)$ for $a \in A(s)$, where $A(s)$ is the set of actions applicable for the state s
- Positive **action costs** $c(a, s)$

A **solution** of a planning problem is a sequence of applicable actions that **map the initial state to one of the goal states**. This means there is a state sequence from the initial state to the goal, such that each successive state can be reached by an applicable action.

An optimal plan does this and **minimises the sum of action costs**.

Models, Languages and Solvers

A planner is a solver for a certain class of models: it takes a model description and computes the corresponding controller.

Models are described in suitable planning languages (Strips, PDDL, PPDDL, etc.) where states represent interpretations over the language.

Strips

A problem in Strips is a tuple $P = \{ F, O, I, G \}$

- F is the set of **all atoms** in the domain (boolean variables)
- O is the set of **all operators** (actions)
- I is the **initial** state, such that $I \subseteq F$
- G is the **goal** state, such that $G \subseteq F$
- Simplified: F is a collection of things that can be true or false; I states which are true at the start, and G states which should be true by the end.

Operators $o \in O$ are represented by:

- The **add** list $Add(a)$
- The **delete** list $Del(a)$
- The **precondition** list $Prec(a)$

From Languages to Models

A Strips **problem** $P = \{ F, O, I, G \}$ determines a **state model** $S(P)$ where

- Each state is a collection of atoms from F
- The initial state is I
- The goal states is/are G
- The actions a for any state s are the operations in O such that $Prec(a)$ are a superset of that state s
  - Simplified: if you're at state s, the application actions $A(s)$ are all the operators in O that have preconditions that have been met (are true) in the current state, s
- The next state is defined by $s' = s - Del(a) + Add(a)$
  - Simplified: if you're at state s, applying the action will take you to state s' by removing any atoms in Del from the current state and adding all atoms in Add.
- The action costs are all 1

A (optimal) **solution of $\boldsymbol{P}$** is a (optimal) **solution of $\boldsymbol{S(P)}$.**

Language extensions like **negation** and **conditional effects** are required for richer models.

PDDL

PDDL = **Planning Domain Description Language**

PDDL specifies **syntax for problems** supporting Strips, variables, types, and more.

PDDL problems are specified in two parts:

- **Domain**: contains action and atom schemas along with argument types
- **Instance**: contains an initial situation, goal and constants (objects) of each type

Domain Format

```
(define (domain domain-name)
      (:requirements :strips :typing)

      (:types typeA typeB typeC)

      (:constants item1 – typeA item2 item3 - typeB)

      (:predicates
            (predicate-name ?var1 – var1type ?var2 – var2type)
            (action-name ?var1 – var1type ?var2 – var2type)
            ...
      )

      (:action action-name
            :parameters (?var1 – var1type ?var2 – var2type)
            :precondition (and
                  (precond-1)
                  (precond-2 ?var1)
                  ...
            )
            :effect (and
                  (new-predicate ?var2)
                  ...
                  (not (
                        old-predicate ?var1
                        ...
                  )
            )
      )

)
```

Instance Format

```
(define (problem problem-name)

      (:domain domain-name)

      (:objects objA objB objC – typeForABC objD – typeForD)

      (:init
            (predicate-1 ?objA)
            (predicate-2)
            ...
      )

      (:goal (and
            (predicate-1 ?objC)
            (predicate-2)
      ))

)
```

Solving Classical Planning Problems

Two of the best approaches to solving planning problems are **heuristic searches** and **SAT**.

These methods can solve problems over huge state spaces, but some problems are inherently hard and so general planners are unlikely to approach the performance of specialised methods.

Solving P by Solving S(P): Path-Finding in Graphs

Search algorithms exploit the correspondence between **classical state models** and **directed graphs**:

- Each graph **node** represents states in the model
- Each **edge** represents an applicable action from one state to another.

When solving planning problems with heuristic search functions, the problem P is solved by **path-finding algorithms** over the **graph** associated with the model $S(P)$.

Two **path-finding algorithm** families are:

- **Uninformed** search (a.k.a. brute force)
  - The goal plays a passive role in the search
  - DFS, BFS, Dijkstra, Iterative Deepening, $h(s) = 0,$ etc.
- **Informed** search (a.k.a. heuristic searching)
  - The goal plays an active role in the search through the heuristic function
  - A*, IDA*, Hill Climbing, Best First, DFS B&B, LRTA*, etc.

Heuristic searching algorithms can find paths in very, very large graphs.

Heuristics for Classical Planning

(See a basic intro to heuristics on page 2.)

General idea of heuristics: find a **relaxed problem** and use its **optimal cost** as a heuristic value.

Examples:

- Sum of Manhattan distances in N-puzzle
- Sum of misplaced blocks in Blocks-World

But, how can suitable relaxations be created automatically for planning?

Delete Relaxation – P^+

Assumption: once an atom is true, it can **always be assumed as true** and used to satisfy preconditions and goals.

- Simplified: when something is achieved, it stays achieved.

This is achieved in Strips by ignoring delete lists – without a delete list, atoms are never "un-set", so once true they will always remain that way.

$h(s) \stackrel{\text{def}}{=} h^*_{P+}(s)$ is an **admissible** heuristic.

- Any solution to P is a solution to P^+.
- In P, any atom may have to be made true more than once, so the real cost will always be equal or higher.

Finding the optimal solution to P^+ is nearly as hard as solving P, so **we can't use the optimal cost of P^+** as the heuristic because it's too intensive to calculate for every state. However, finding just one solution to $P^+(s)$ is easy, even if that solution might not be optimal.

The delete relaxation is used in three different heuristic algorithms: $\boldsymbol{h_{add}}$, $\boldsymbol{h_{max}}$ and $\boldsymbol{h_{FF}}$

Heuristics from P^+: h_{add} and h_{max}

When calculating $h(s)$ for a given state, both of these heuristics **depend on finding the cost of achieving every atom** in the given state. This value is fairly **easy to compute**, because achieved **atoms are permanent** in the P^+ relaxation.

The cost of each atom can be computed as follows: for all atoms in a given state of P^+ and a current state of s:

- The atom p is reachable in **0 steps** if $p \in s$
- The atom p is reachable in $\boldsymbol{i+1}$ **steps** if
  - It cannot be achieved in i steps or less
  - And there is an action a_p that creates p, with preconditions that can be met in i steps or less.

This method **terminates in a number bound by the number of atoms**, when there are no more reachable atoms. If an atom cannot be reached via this method, then there is no plan for that atom in the problem P.

The two algorithms differ only when considering the number to return:

- $\boldsymbol{h_{add}}$ (also called h_{sum}) returns the **cumulative cost of achieving each atom**.
  - This is **not admissible**, because one action may achieve multiple atoms.
- $\boldsymbol{h_{max}}$ returns the cost of **achieving the most expensive atom**.
  - This is **admissible**, but not very informative.

Note: neither of these algorithms actually attempts to solve the relaxed problem! Rather, they offer an estimate of how expensive given states are to achieve.

Heuristics from P^+: h_{FF}

This algorithm is different, as it **actually solves the relaxed problem**. It does not try to find an optimal solution*, but rather a solution that is reasonable. This is often done with the **iterative reachability procedure.**

To determine the heuristic value of a given state s, the algorithm finds a solution from the current state to the given state in the relaxed plan – this solution is often referred to as $\pi(s)$ – and returns the heuristic as the **number of action in the relaxed solution**. This is written as:

$$h(s) = |\pi(s)|$$

This is also called the **relaxed plan h**. It is **not admissible** (the plan found may be non-optimal), but it is **informative**.

* As mentioned above, h_{FF} does not look for an optimal solution, because this would be very computationally expensive. It is slightly easier than solving the real problem, but still far too difficult to use as a heuristic because it has to be calculated at every stage.

Iterative Reachability Procedure

The IRP builds alternative layers of atoms (P_i) and actions (A_i), starting from the current state s.

$$\begin{aligned} P_0 &= \{p \in s\} \\ A_i &= \{a \in O \mid Prec(a) \subseteq P_{0 \to i}\} \\ P_{i+1} &= \{p \in Add(a) \mid a \in A_i, p \notin P_k, k < i+1\} \end{aligned}$$

Simplified: the first atom layer (P_0) contains all the atoms in the current state. Each action layer is all actions applicable with the atoms already achieved. Each subsequent atom later is all atoms achieved by the previous action layer that are not already achieved.

State of the Art Planners

The first heuristic search planners used the graph defined by the state model $S(P)$ and standard algorithms like **Greedy Best-First** or **WA*** and heuristics like $\boldsymbol{h_{add}}$.

Modern planners like **FF** and **LAMA** go beyond this.

They exploit the structure of the heuristic and/or the problem further:

- **Helpful actions**: actions that are the most relevant in a relaxation
- **Landmarks**: using implicit problem sub-goals.

They also use new searching algorithms:

- **Enforced Hill Climbing (EHC)**
- **Multi-Queue Best First Search**

The result is the power to solve **huge problems, very fast**. *Most* of the time.

Enforced Hill Climbing (EHC)

FF's **second phase** is a simple **Greedy Best-First** search guided by $\boldsymbol{h_{FF}}$. This is **complete**, but **slow**.

However, FF's **first phase** is Enforced Hill Climbing (**EHC**) which is **incomplete** but **very fast**, using only helpful actions.

An action is **helpful** in s if it is **applicable** in s and **adds an atom** p that is **not** in s, such that p is a **goal**, or a precondition of an action in the relaxed plan $\pi(s)$.

- Simplified: a helpful action is one that gets you closer to the goal, either by adding an atom that's in the goal, or an atom needed by another action that leads to the goal.

Starting with $s = s_0$ (the initial state), EHC does a **breadth-first search** from s using only helpful actions until a state s' is found such that $h_{FF}(s') < h_{FF}(s)$.

If such a state is found, the process is **repeated** starting with $s = s'$. If not, EHC fails and the second stage is triggered.

Multi-Queue Best First Searching

Standard **best-first** algorithms work with a single queue that is ordered according to the evaluation function.

If there are two or more evaluation functions, it is also possible to have **several queues**, each one ordered by a **different evaluation function**.

Multi-queue best-first searching picks the **best node** in one queue, then the best node in another queue, and so on, **alternating** among queues.

LAMA uses two evaluation functions: one is $f_1 = h_{FF}$, and the other f_2 is given by the number of **unachieved landmarks**.

Landmarks

Landmarks are implicit sub-goals of the problem; formally, they are atoms p that must be true in **all plans**.

For example, in the Blocks problem, $clear(A)$ is a **landmark** in any solution to a problem where A has a misplaced block above it in the initial state.

Finding all landmarks is **computationally hard**, but some landmarks are easy to identify with methods similar to those used to compute heuristics.

An atom p is a landmark in $P^+(s)$ (and hence $P(s)$) **if and only if** heuristics like $h_{max}(s)$ become **infinite** once the actions that add p are removed from the problem.

Therefore, **delete-relaxation landmarks** can be computed in polynomial time. There are more efficient methods than this, such as those used in LAMA.

Planning as SAT

When solving planning problems with SAT, a problem $P = \{ F, O, I, G \}$ with a horizon n is mapped into a **set of clauses** $C(P, n)$ and solved by a **SAT solver** (satz, chaff, etc.).

The theory $C(P, n)$ includes variables $p_0, p_1, \dots, p_n$ and $a_0, a_1, \dots, a_n$ for each $p \in F$ and $a \in O$.

$C(P, n)$ is satisfiable **if and only if** there is a plan with length bounded by n.

Such a plan can be read from a **truth valuation** that satisfies $C(P, n)$.

Theory $C(P, n)$ for Problem $P = \{F, O, I, G\}$

Initial State

p_o for $p \in I$

$\neg p_0$ for $p \in F$ and $p \notin I$

Simplified: in the first round (i.e. round 0), all atoms/propositions that are in the initial state are true; all atoms that are in the domain but **not** in the initial state are false.

Goal

p_n for $p \in G$

Simplified: in the final round (i.e. round n) all atoms that are in the goal state are true.

Actions

For all $i = 0, 1, \dots, n-1$ and each action $a \in O$:

$a_i \rightarrow p_i$ for $p \in Prec(a)$

$a_i \rightarrow p_{i+1}$ for each $p \in Add(a)$

$a_i \rightarrow \neg p_{i+1}$ for each $p \in Del(a)$

Simplified: if action a is performed in round i then it implies that all preconditions must have been true before, all atoms in the Add list must be true in round $(i+1)$, and all atoms in the Del list must be false in round $(i+1)$.

Persistence

For all $i = 0, 1, \dots, n-1$ and each atom $p \in F$, where $O(p^+)$ and $O(p^-)$ stand for the actions that add and delete p:

$p_i \wedge a \in O(p^-) \wedge \neg a_i \rightarrow p_{i+1}$

$\neg p_i \wedge a \in O(p^+) \wedge \neg a_i \rightarrow \neg p_{i+1}$

Simplified: if p is true in this round and a is an action that will delete it, but a is not used this round, then p will stay true in the next round. Similarly, if p is false in this round and a is an action that will add it, but a is not used, then p will stay false in the next round.

Seriality

For each $i = 0, 1, \dots, n-1$, if $a \neq a'$ then $\neg(a_i \wedge a'_i)$.

Simplified: only one distinct action can be applied when moving to the next round.

Performance

This encoding is **simple** (apparently) but **not the best** computationally.

Alternatives can be used, including parallelism (no seriality), NO-Ops, lower bounds, etc.

The best **current SAT planners are very good**, almost rivalling heuristic search planners.

Simple Temporal Networks (STNs)

Constraint problems often involve a **time element**.

For example, the gripper domain can be extended to include the duration of actions.

Basic Temporal Constraints

Actions are annotated with an ***s*** **to indicate the start** and an ***e*** **to indicate the end:**

- $Move_s$ = the **start** of the $Move$ action
- $Move_e$ = the **end** of the $Move$ action

The time elapsed between the start and the end point is:

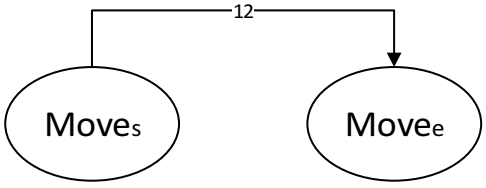
$$Move_e - Move_s$$

If the action $Move$ takes between 7 and 12 units of time, we can say:

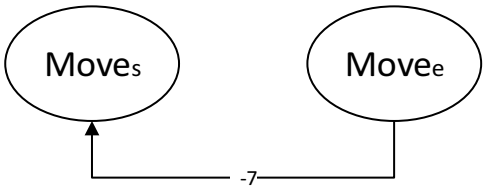
$$7 \leq Move_e - Move_s - \leq 12$$

There are two separate constraints at use here:

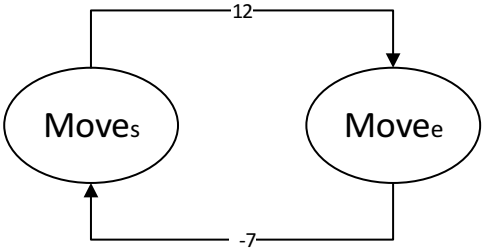
- $Move_e - Move_s \leq 12$, which give one **edge type** representing the upper bound on the time between the two points:



- $Move_s - Move_e \leq 7$, obtained by algebraic rearrangement (multiply everything by -1), gives another edge type representing the lower bound on the time between the two points:



This leads to a simple representation of a time-bound constraint. The graph below tells us that the action $Move$ takes between 12 and 7 units of time.



Now we can make a graph of the plan, which is essentially just a chain of actions from the initial state I to the goal G.

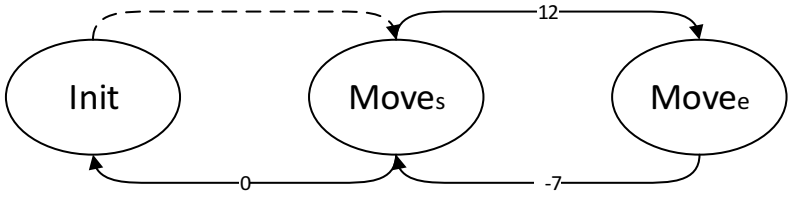
Joining Actions

How are the edges between tasks built?

Consider the time between $Init$ and $Move_s$. $Move$ must start no earlier than $Init$, but any amount of time afterwards. This gives:

$$0 \leq Move_s - Init \leq \infty$$

Unbound (infinite) edges are **displayed as dotted arrows** with no labels, producing this graph:



Path Finding

What is the quickest time in which the plan can be executed?

Finding the quickest solution is easy: find the **shortest path from G to I**, then the (negative) length of this path is the length of the quickest path from I to G.

Consistency

Sometimes it is easy to see by eye whether an end state can be reached in the time available. If this is the case, the STN is **consistent**.

However, in the real world, this is generally not possible, so we need an efficient method to determine consistency.

If an STN contains a **negative cycle** then the constraints are inconsistent.

- A negative cycle is a cycle through the graph where the sum of each edge passed yields a negative result.

Warshall's Algorithm (FC1) can be used to determine whether there is a negative cycle in a weighted graph.

Tightening Constraints

There is a method to determine which constraints can be tightened and how much to tighten them by. This relies on shortest path analysis:

- Find the **length of the shortest path** from $Action_s$ to $Action_e$ (from **start** to **end**) and update the length of the **upper bound** arc between them to that value.
- Find the **length of the shortest path** from $Action_e$ to $Action_s$ (from **end** to **start**) and update the length of the **lower bound** arc between them to that value.

Justifying Tightening

Imagine we have the set of constraints:

- $x_1 - x_2 \leq a_1$
- $x_2 - x_3 \leq a_2$
- $x_{n-1} - x_n \leq a_{n-1}$

Then we can sum them all to create:

- $x_1 - x_n \leq a_1 + a_2 + \cdots + a_{n-1}$

We rename the sum as A, so…

- $x_1 - x_n \leq A$

Imagine we also have the constraint:

- $x_1 - x_n \leq B$

The first set with A represents **finding a short path**, and the second constraint with B represents the **direct path** that already exists. The new constraint $x_1 - x_n \leq A$ is implied by the first set.

If $A < B$ then it is tighter, and we can swap $x_1 - x_n \leq B$ with $x_1 - x_n \leq A$

Checking Constraints

To check **how long a certain activity can take** (and still allow the plan to complete in the required time), we are concerned with the **longest gap** between **starting and ending** that activity.

This can be found by looking for the **shortest path** from the start to the end of that activity, which is normally **not** the direct upper bound arc.

Key Reasoning Steps for STNs

1. **Is the network consistent?** Are there any negative cycles?
2. **What questions can be answered?** (e.g. how long can an activity last? How long will a solution take? Etc.)
3. **Tighten** as many upper bounds as possible.
4. **Tighten** as many lower bounds as possible.
5. **Keep tightening** until no more improvements exist. The resulting graph will still be consistent if it was consistent when tightening started.
6. **Add new constraints** to allow further inference. For example, make the first action start at $t = 0$, or fix the time that elapses between two activities. New activities and/or constraints can be added.
7. **Re-check consistency**, because adding constraints can introduce negative cycles.
8. **Re-tighten** where possible.

Two Player Games

Heuristic search and planning are largely concerned with action selection where an initial state is given and changes follow from the actions of a **single agent**.

How could a goal be planned for in the presence of other agents?

It depends on what the other agents want:

- The other agents may be there to **help** (**cooperative agents**)
- The other agents may be there to **hinder** (**adversarial agents**)
- The other agents may be there for **their own goals** (**not necessarily cooperative or adversarial**)

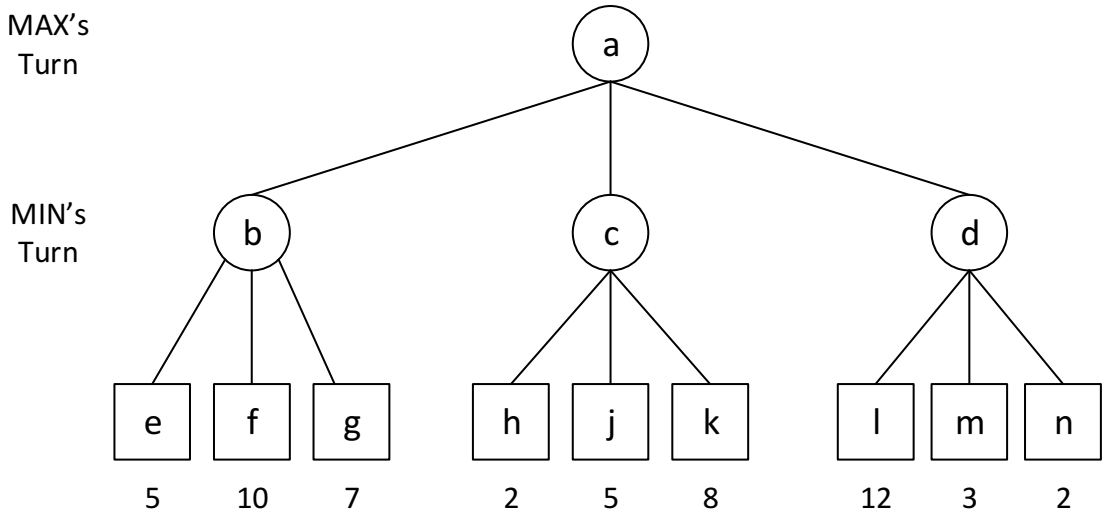
Game Trees

Game trees model for **2-player, adversarial (zero-sum), sequential games**, such as tic-tac-toe, chess, etc. One player seeks to maximise a certain metric, the other seeks to minimise it.

There are **three components**:

- Three types of nodes
  - **MAX nodes** – these represent options for **MAX-Player**
  - **MIN nodes** – these represent options for **MIN-Player**
  - **Terminal nodes** – these represent **final outcomes** and have no options
- Structure
  - The **root node** is either MAX or MIN
  - **Children of MAX** nodes are **MIN** nodes or Terminal nodes
  - **Children of MIN** nodes are **MAX** nodes or Terminal nodes
  - **Terminal nodes**, and only them, have **no children**
- Evaluation function
  - **Values** or **payoffs** associated with terminal nodes

A game tree may look something like this:



Minimax and Maximin Algorithms for Game Trees

Game trees are evaluated from the **bottom up**:

- **Terminal** nodes have a **given value**
- A **MIN** node has a value equal to the **minimum of its children**
- A **MAX** node has a value equal to the **maximum of its children**

These values can be computed by a **depth-first search**.

If the **root node is MAX**:

- The algorithm is called **Maximin**
- The **value of the root** node is the value that **MAX can guarantee**
- The **best initial action** is the one that leads to the child with that value

If the **root node is MIN**:

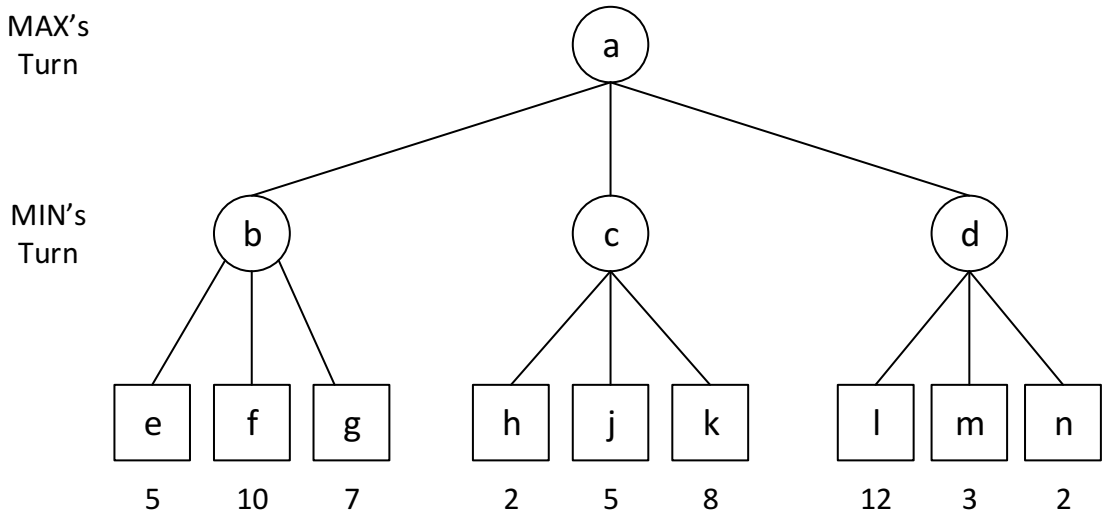
- The algorithm is called **Minimax**
- The **value of the root** node is the value that **MIN can guarantee**
- The **best initial action** is the one that leads to the child with that value

The **time complexity** is $\boldsymbol{O(b^d)}$ where d is the number of levels (depth) and b is the branching factor.

The **space complexity** is $\boldsymbol{O(b \times d)}$.

Example Game Tree 1

Consider the tree below:



What should **MAX-Player** do in this game?

- Max player has three options: b, c or d. MAX-Player knows that MIN-Player will seek to minimise in the next step, so MAX-Player will go to **the option that has the highest minimum terminal node**, which is node b.

What is the **minimum payoff** MAX-Player can ensure for each node?

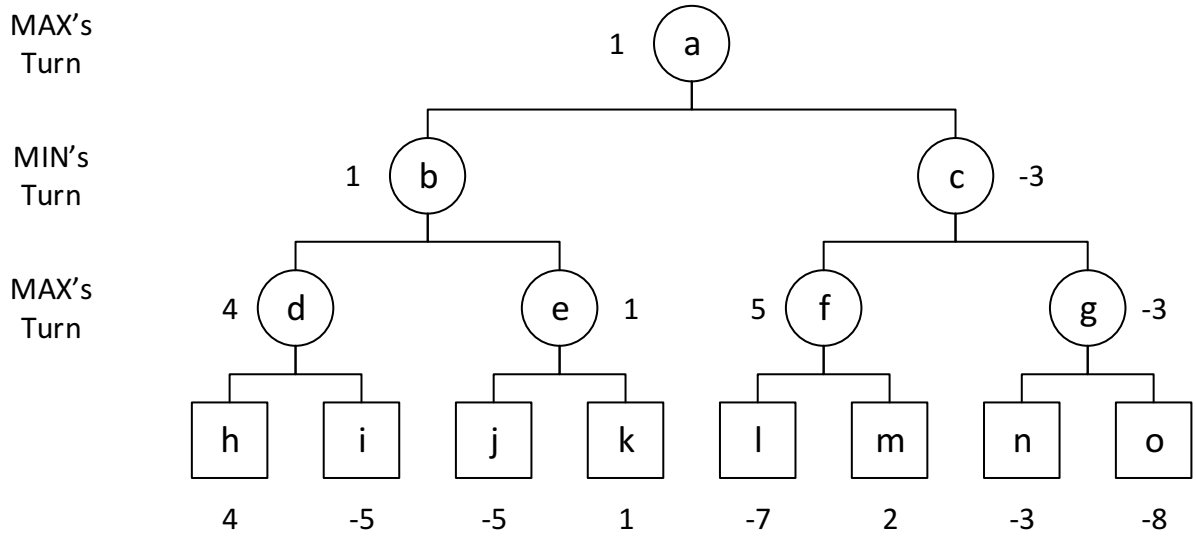
- Moving to node b, MAX-Player ensures a **minimum payoff of 5**.
  Moving to nodes c or d, MAX-Player ensures a **minimum payoff of 2**.

What should **MIN-Player** do if the game reaches b, c or d?

- MIN-Player will move to the node that ensures the lowest **maximum payoff**.

Example Game Tree 2

Consider the tree below:



What should **MAX-Player** do **initially**?

- Max should move to node b.

What's the **minimum payoff** MAX-Player can ensure for **each of the initial moves**?

- Moving to b, the minimum payoff is 1
- Moving to c, the minimum payoff is -3

What's the **minimum payoff** MAX-Player can ensure at the **root of the game**?

- 1

Using and Improving Minimax for Huge Game Trees

Minimax algorithms explore **complete game trees**.

This is possible for simple games like tic-tac-toe, but impossible for games like chess or checkers.

For more complex games, a depth of d is set and the algorithms only look that far; i.e. nodes at depth d are treated as terminal.

The **best first move** in the resulting sub-tree is executed, and the process is repeated after the opponent moves.

The **evaluation function** that sets the value of **terminal nodes** is designed by hand, or learned by the machine.

The quality of AI-driven play depends on the **depth d** used and the **evaluation function** for terminals.

- Gameplay will be poor if the algorithm only looks one or two steps ahead, or the evaluation function identifies bad moves as beneficial, or vice versa.

Programs that are specialised for certain games may incorporate **non-uniform depths** or **non-exhaustive searches**.

Minimax with Alpha-Beta Pruning

Depth-first search performance can be improved substantially.

Two Observations

When some of the children of a MAX node have been evaluated, there is a **lower bound α** on how much MAX can get, even if the other children have not been evaluated yet.

When some of the children of a MIN node have been evaluated, there is an **upper bound β** on how much MIN can get, even if the other children have not been evaluated yet.

- Simplified: if MAX evaluates the first child to be 5, but has a dozen unevaluated children left, it already knows it can force a value of **at least** 5. If MIN does the same, it already knows it can force a value that is **at most** 5.

Two Optimisations

Important: α and β values are **passed down** the tree, but not up.
Important: minimax values are **passed up** the tree, but not down.

At a MIN node, skip the rest of the children in DFS when $\beta \leq \alpha$ (this is called **β-cutoff**).

At a MAX node, skip the rest of the children n DFS when $\beta \leq \alpha$ (this is called **α-cutoff**).

In practise, Alpha-Beta Pruning turns the search complexity from $O(b^d)$ into $O(b^{d/2})$, meaning the **quality** given by a search to depth d can be achieved in the time of a search to depth $d/2$.

Alpha-Beta Pseudo Code

```
ALPHABETA( node, depth, alpha, beta, playerType )
      if depth = 0 or node is terminal
            return value of node

if playerType = MAX
      for each child of node
            alpha = max(alpha, ALPHABETA(child, depth-1, alpha, beta, MIN))
            if beta <= alpha
                  // beta cut-off
                  break
      return alpha

elseif playerType = MIN
      for each child of the node
            beta = min(beta, ALPHABETA(child, depth-1, alpha, beta, MAX))
            if beta <= alpha
                  // alpha cut off
                  Break
      return beta
```

Initial call:

```
ALPHABETA( root, 0, -inf, +inf, MAX )
```