Weighted A^*

Weighted A^* (with duplicate detection and re-opening)

open $:=$ **new** priority queue ordered by ascending $g(\textit{state}(\sigma)) + W * h(\textit{state}(\sigma))$
open.insert(make-root-node(init()))
closed $:= \emptyset$
best-g $:= \emptyset$
while not *open*.empty():
    $\sigma :=$ *open*.pop-min()
    if $\textit{state}(\sigma) \notin \textit{closed}$ **or** $g(\sigma) < \textit{best-g}(\textit{state}(\sigma))$:
        closed $:= \textit{closed} \cup \{\textit{state}(\sigma)\}$
        $\textit{best-g}(\textit{state}(\sigma)) := g(\sigma)$
        if is-goal(state(σ)): **return** extract-solution(σ)
        for each $(a, s') \in$ succ($\textit{state}(\sigma)$):
            $\sigma' :=$ make-node(σ, a, s')
            if $h(\textit{state}(\sigma')) < \infty$: *open*.insert($\sigma'$)
return unsolvable

Weighted A^*: Remarks

The weight $W \in \mathbb{R}_0^+$ is an **algorithm parameter:**

- For $W = 0$, weighted A^* behaves like

Weighted A^*: Remarks

The weight $W \in \mathbb{R}_0^+$ is an **algorithm parameter:**

- For $W = 0$, weighted A^* behaves like uniform-cost search.
- For $W = 1$, weighted A^* behaves like

Weighted A^*: Remarks

The weight $W \in \mathbb{R}_0^+$ is an **algorithm parameter:**

- For $W = 0$, weighted A^* behaves like uniform-cost search.
- For $W = 1$, weighted A^* behaves like A^*.
- For $W \to \infty$, weighted A^* behaves like

Weighted A^*: Remarks

The weight $W \in \mathbb{R}_0^+$ is an **algorithm parameter:**

- For $W = 0$, weighted A^* behaves like uniform-cost search.
- For $W = 1$, weighted A^* behaves like A^*.
- For $W \to \infty$, weighted A^* behaves like greedy best-first search.

Properties:

- For $W > 1$, weighted A^* is bounded suboptimal: if h is admissible, then the solutions returned are at most a factor W more costly than the optimal ones.

Agenda

Hill-Climbing

Hill-Climbing

$\sigma :=$ make-root-node(init())
forever:
  if is-goal(state(σ)):
    return extract-solution(σ)
  $\Sigma' :=$ { make-node(σ, a, s') | $(a, s') \in$ succ(state(σ)) }
  $\sigma :=$ an element of Σ' minimizing h /* (random tie breaking) */

Hill-Climbing

Hill-Climbing

$\sigma :=$ make-root-node(init())
forever:
  if is-goal(state(σ)):
    return extract-solution(σ)
  $\Sigma' := \{$ make-node(σ, a, s') $\mid (a, s') \in$ succ(state(σ)) $\}$
  $\sigma :=$ an element of Σ' minimizing h /* (random tie breaking) */

Remarks:

- Makes sense only if $h(s) > 0$ for $s \notin S^G$.
- Is this complete or optimal?

Enforced Hill-Climbing

Enforced Hill-Climbing: Procedure *improve*

def *improve*(σ_0):
  queue := **new** fifo queue
  queue.push-back(σ_0)
  closed $:= \emptyset$
  while not *queue*.empty():
    $\sigma =$ *queue*.pop-front()
    if *state*(σ) $\notin$ *closed*:
      closed := *closed* $\cup$ {*state*(σ)}
      if h(*state*(σ)) $<$ h(*state*(σ_0)): **return** σ
      for each $(a, s') \in$ succ(*state*(σ)):
        $\sigma' :=$ make-node(σ, a, s')
        queue.push-back(σ')
  fail

$\rightsquigarrow$ Breadth-first search for state with strictly smaller h-value.

Enforced Hill-Climbing, ctd.

Enforced Hill-Climbing

$\sigma :=$ make-root-node(init())
while not is-goal(state(σ)):
        $\sigma :=$ improve(σ)
return extract-solution(σ)

Enforced Hill-Climbing, ctd.

Enforced Hill-Climbing

$\sigma :=$ make-root-node(init())
while not is-goal(state(σ)):
    $\sigma :=$ improve(σ)
return extract-solution(σ)

Remarks:

- Makes sense only if $h(s) > 0$ for $s \notin S^G$.
- Is this optimal?

Agenda

AI Planning for Autonomy

3. Introduction to Planning

How to Describe Arbitrary Search Problems

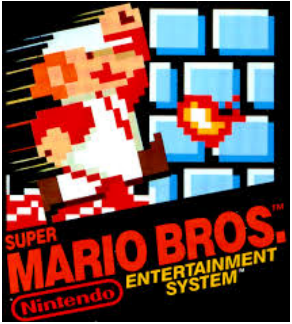
Chris Ewin & Tim Miller

THE UNIVERSITY OF MELBOURNE

With slides by Nir Lipovetsky

Beating Kasparov is great . . .

Beating Kasparov is great . . . but how to play Mario?



- You (and your brother/sister/little nephew) are better than Deep Blue at **everything** - except playing Chess.
- Is that (artificial) 'Intelligence'?

→ How to build machines that automatically solve **new** problems?

Planning: Motivation

How to develop systems or 'agents'
that can make decisions on their own?

Autonomous Behavior in AI

The key problem is to select the action to do next. This is the so-called control problem. Three approaches to this problem:

- **Programming-based:** Specify control by hand
- **Learning-based:** Learn control from experience
- **Model-based:** Specify problem by hand, derive control automatically

$\rightarrow$ Approaches not orthogonal; successes and limitations in each . . .

$\rightarrow$ Different models yield different types of controllers . . .

Programming-Based Approach

→ Control specified by programmer, e.g.:

- If Mario finds no danger, then run...
- If danger appears and Mario is big, jump and kill ...
- ...

- Advantage: domain-knowledge easy to express
- Disadvantage: cannot deal with situations not anticipated by programmer

Learning-Based Approach

→ Learns a controller from experience or through simulation:

- **Unsupervised** (Reinforcement Learning):
  - penalize Mario each time that 'dies'
  - reward agent each time oponent 'dies' and level is finished, . . .
- **Supervised** (Classification)
  - learn to classify actions into good or bad from info provided by teacher
- **Evolutionary**:
  - from pool of possible controllers: try them out, select the ones that do best, and mutate and recombine for a number of iterations, keeping best

- Advantage: does not require much knowledge in principle
- Disadvantage: in practice, hard to know which features to learn, and is slow

General Problem Solving

Ambition: Write one program that can solve all problems.

$\rightarrow$ Write $X \in \{algorithms\}$: for all $Y \in \{\text{'}problems'\}$: X'solves'Y

$\rightarrow$ What is a 'problem'? What does it mean to 'solve' it?

Ambition 2.0: Write one program that can solve a large class of problems

Ambition 3.0: Write one program that can solve a large class of problems effectively

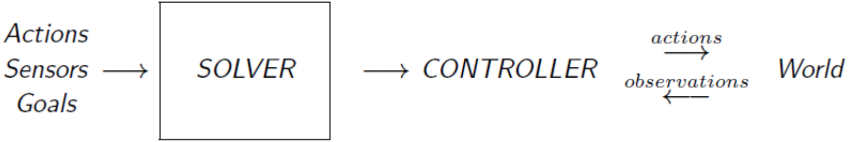
(some new problem) $\rightsquigarrow$ (describe problem $\rightarrow$ use off-the-shelf solver) $\rightsquigarrow$ (solution competitive with a human-made specialized program)

$\rightarrow$ Beat humans at coming up with clever solution methods!
(Link: GPS started on 1959)

Model-Based Approach / General Problem Solving

→ specify model for problem: actions, initial situation, goals, and sensors

→ let a solver compute controller automatically



Model-Based Approach / General Problem Solving

→ Advantage:

- Powerful: In some applications generality is absolutely necessary
- Quick: Rapid prototyping. 10s lines of problem description vs. 1000s lines of C++ code. (Language generation!)
- Flexible & Clear: Adapt/maintain the description.
- Intelligent & domain-independent: Determines automatically how to solve a complex problem effectively! (The ultimate goal, no?!)

→ Disadvantage: need a model; computationally intractable

- Efficiency loss: Without any domain-specific knowledge about Chess, you don't beat Kasparov . . .

→ Trade-off between 'automatic and general' vs. 'manualwork but effective'

Model-based approach to intelligent behavior called **Planning** in AI

How to make fully automatic algorithms effective?

Agenda

Basic State Model: Classical Planning

Ambition:

Write one program that can solve all classical search problems.

State Model:

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

$\rightarrow$ A **solution** is a sequence of applicable actions that maps s_0 into S_G, and it is **optimal** if it minimizes **sum of action costs** (e.g., # of steps)

$\rightarrow$ Different **models** and **controllers** obtained by relaxing assumptions in **blue** . . .

Uncertainty but No Feedback: Conformant Planning

- finite and discrete state space S
- a set of possible initial state $S_0 \in S$
- a set $S_G \subseteq S$ of goal states
- actions $A(s) \subseteq A$ applicable in each $s \in S$
- a non-deterministic transition function $F(a, s) \subseteq S$ for $a \in A(s)$
- uniform action costs $c(a, s)$

$\rightarrow$ A **solution** is still an **action sequence** but must achieve the goal for **any possible initial state and transition**

$\rightarrow$ More complex than **classical planning**, verifying that a plan is **conformant** intractable in the worst case; but special case of **planning with partial observability**

Planning with Markov Decision Processes

MDPs are **fully observable, probabilistic** state models:

- a state space S
- initial state $s_0 \in S$
- a set $G \subseteq S$ of goal states
- actions $A(s) \subseteq A$ applicable in each state $s \in S$
- transition probabilities $P_a(s'|s)$ for $s \in S$ and $a \in A(s)$
- action costs $c(a, s) > 0$

$\rightarrow$ **Solutions** are **functions (policies)** mapping states into actions

$\rightarrow$ **Optimal** solutions minimize **expected cost** to goal

Partially Observable MDPs (POMDPs)

POMDPs are **partially observable, probabilistic** state models:

- states $s \in S$
- actions $A(s) \subseteq A$
- transition probabilities $P_a(s'|s)$ for $s \in S$ and $a \in A(s)$
- initial belief state b_0
- final belief state b_f
- sensor model given by probabilities $P_a(o|s)$, $o \in Obs$

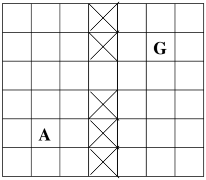
$\rightarrow$ **Belief states** are probability distributions over S

$\rightarrow$ **Solutions** are policies that map belief states into actions

$\rightarrow$ **Optimal** policies minimize **expected** cost to go from b_0 to G

Example

Agent **A** must reach **G**, moving one cell at a time in **known** map



- If actions deterministic and initial location known, planning problem is **classical**
- If actions stochastic and location observable, problem is an **MDP**
- If actions stochastic and location partially observable, problem is a **POMDP**

Different combinations of uncertainty and feedback: three problems, three models

Agenda

Models, Languages, and Solvers

- A planner is a solver over a class of models; it takes a model description, and computes the corresponding controller

$$\textit{Model} \Longrightarrow \boxed{\textit{Planner}} \Longrightarrow \textit{Controller}$$

- Many models, many solution forms: uncertainty, feedback, costs, . . .
- Models described in suitable planning languages (Strips, PDDL, PPDDL, . . .) where states represent interpretations over the language.

A Basic Language for Classical Planning: Strips

- A **problem** in STRIPS is a tuple $P = \langle F, O, I, G \rangle$:
  - F stands for set of all atoms (boolean vars)
  - O stands for set of all operators (actions)
  - $I \subseteq F$ stands for initial situation
  - $G \subseteq F$ stands for goal situation
- Operators $o \in O$ **represented** by
  - the Add list $Add(o) \subseteq F$
  - the Delete list $Del(o) \subseteq F$
  - the Precondition list $Pre(o) \subseteq F$

From Language to Models (STRIPS Semantics)

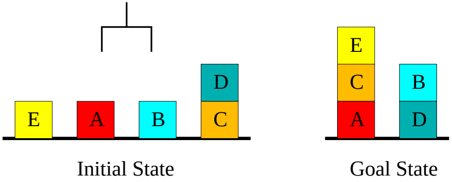
A STRIPS problem $P = \langle F, O, I, G \rangle$ determines **state model** $\mathcal{S}(P)$ where

- the states $s \in S$ are collections of atoms from F. $S = 2^F$
- the initial state s_0 is I
- the goal states s are such that $G \subseteq s$
- the actions a in $A(s)$ are ops in O s.t. $Prec(a) \subseteq s$
- the next state is $s' = s - Del(a) + Add(a)$
- action costs $c(a, s)$ are all 1

$\rightarrow$ (Optimal) **Solution** of P is (optimal) **solution** of $\mathcal{S}(P)$

$\rightarrow$ Slight language extensions often convenient: **negation**, **conditional effects**, **non-boolean variables**; some required for describing richer models (costs, probabilities, ...).

(Oh no it's) The Blocksworld



- Propositions: $on(x, y)$, $onTable(x)$, $clear(x)$, $holding(x)$, $armEmpty()$.
- Initial state: $\{onTable(E), clear(E), \dots, onTable(C), on(D, C), clear(D), armEmpty()\}$.
- Goal: $\{on(E, C), on(C, A), on(B, D)\}$.
- Actions: $stack(x, y)$, $unstack(x, y)$, $putdown(x)$, $pickup(x)$.
- $stack(x, y)$?